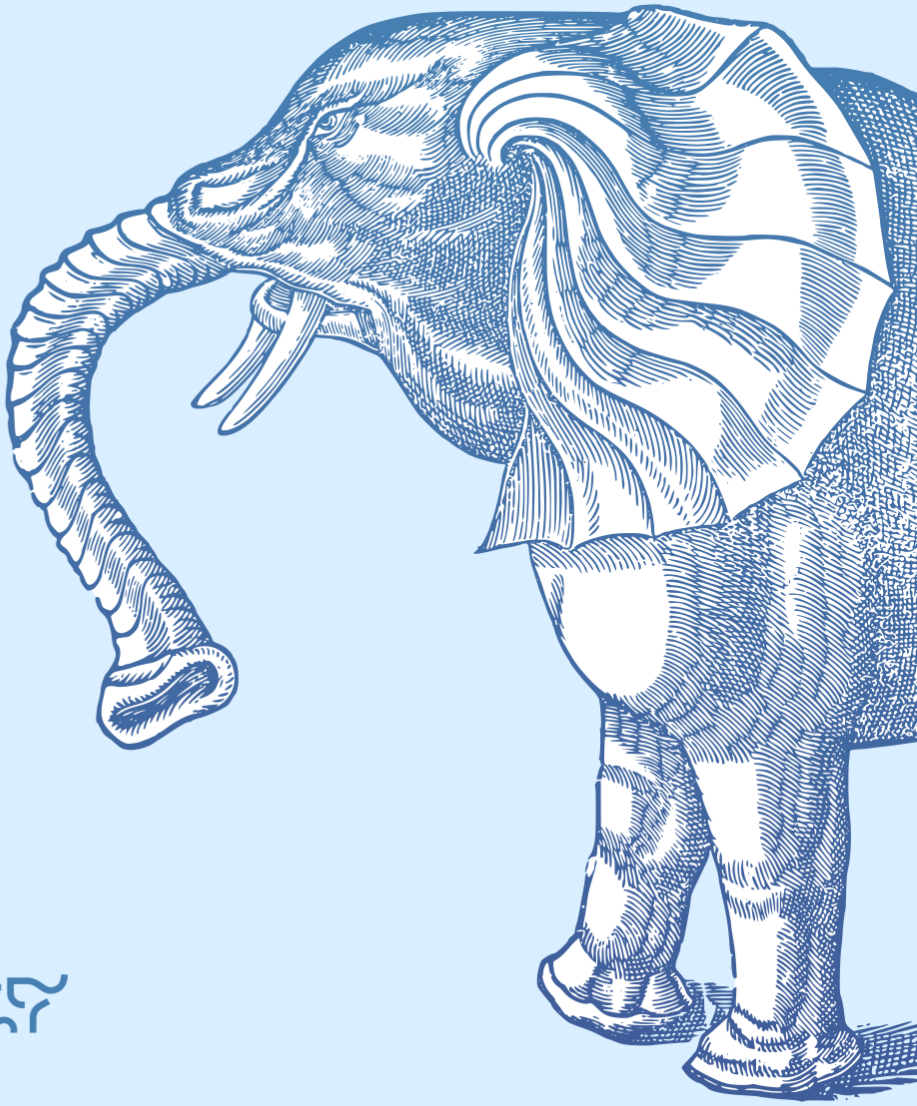
Егор Рогов

PostgreSQL 18 изнутри



ДМК ПРЕСС
ИЗДАТЕЛЬСТВО

Компания Postgres Professional

Егор Рогов

PostgreSQL 18 изнутри

ДМК издательство

Москва, 2026

УДК 004.65
ББК 32.972.134
Р59

Рогов Е. В.

Р59 PostgreSQL 18 изнутри. — М.: ДМК Пресс, 2026. — 682 с.

ISBN 978-5-93700-499-4

В книге рассматривается внутреннее устройство СУБД PostgreSQL: детали реализации многоверсионности и изоляции на основе снимков данных, включая процедуру очистки неактуальных версий строк; буферный кеш и журнал предзаписи; использование блокировок различных уровней; планирование и выполнение SQL-запросов; принципы расширяемости и особенности имеющихся индексных методов доступа. Большое внимание уделяется возможностям, предоставляемым для самостоятельного изучения механизмов функционирования PostgreSQL.

В настоящем издании учтены замечания читателей и исправлены опечатки, а также отражены изменения, произошедшие в версии PostgreSQL 18.

Сайт книги: https://postgrespro.ru/education/books/internals.

Для администраторов и программистов.

УДК 004.65
ББК 32.972.134

На обложке использован фрагмент гравюры из книги *The History of Four-footed Beasts and Serpents* Эдварда Топсела, изданной в 1658 году в Лондоне.

ISBN 978-5-93700-499-4

© Текст, оформление, ООО «ППГ», 2026
© Издание, ДМК Пресс, 2026

Оглавление

О книге 17
Глава 1. Введение 23

Часть I. Изоляция и многоверсионность **45**
Глава 2. Изоляция 47
Глава 3. Страницы и версии строк 75
Глава 4. Снимки данных 97
Глава 5. Самоочистка и hot-обновления 111
Глава 6. Очистка и заморозка 125
Глава 7. Автоочистка 138
Глава 8. Перестроение таблиц и индексов 170

Часть II. Буферный кеш и журнал **181**
Глава 9. Буферный кеш 183
Глава 10. Журнал предзаписи 211
Глава 11. Режимы журнала 234

Часть III. Блокировки **253**
Глава 12. Блокировки отношений 255
Глава 13. Блокировки строк 268
Глава 14. Блокировки разных объектов 293
Глава 15. Блокировки в памяти 305

Часть IV. Выполнение запросов **317**
Глава 16. Этапы выполнения запросов 319
Глава 17. Статистика 346
Глава 18. Табличные методы доступа 371
Глава 19. Индексные методы доступа 394
Глава 20. Индексное сканирование 414
Глава 21. Вложенный цикл 438

Глава 22. Хеширование 457
Глава 23. Сортировка и слияние 483

Часть V. Типы индексов **509**
Глава 24. Хеш-индекс 511
Глава 25. B-дерево 523
Глава 26. Индекс GiST 555
Глава 27. Индекс SP-GiST 589
Глава 28. Индекс GIN 612
Глава 29. Индекс BRIN 642

Заключение 670
Предметный указатель 671

Содержание

О книге **17**

Глава 1. Введение **23**

1.1. Организация данных 23
Базы данных 23
Системный каталог 24
Схемы 25
Табличные пространства 26
Отношения 27
Слои и файлы 28
Страницы 33
TOAST 33
1.2. Процессы и память 39
1.3. Клиенты и клиент-серверный протокол 41

Часть I. Изоляция и многоверсионность **45**

Глава 2. Изоляция **47**

2.1. Согласованность 47
2.2. Уровни изоляции и аномалии в стандарте SQL 49
Потерянное обновление 50
Грязное чтение и Read Uncommitted 50
Неповторяющееся чтение и Read Committed 51
Фантомное чтение и Repeatable Read 51
Отсутствие аномалий и Serializable 52
Почему именно эти аномалии? 52
2.3. Уровни изоляции в PostgreSQL 54
Read Committed 55
Repeatable Read 63
Serializable 70
2.4. Какой уровень изоляции использовать? 73

Глава 3. Страницы и версии строк **75**
3.1. Структура страниц . . . 75
Заголовок страницы . . . 75
Специальная область . . . 76
Версии строк . . . 76
Указатели на версии строк . . . 77
Свободное место . . . 78
3.2. Структура версий строк . . . 78
3.3. Выполнение операций над версиями строк . . . 80
Вставка . . . 81
Фиксация . . . 85
Удаление . . . 87
Отмена . . . 88
Обновление . . . 88
3.4. Индексы . . . 89
3.5. TOAST . . . 90
3.6. Виртуальные транзакции . . . 91
3.7. Вложенные транзакции . . . 92
Точки сохранения . . . 92
Ошибки и атомарность операций . . . 94

Глава 4. Снимки данных **97**
4.1. Что такое снимок данных . . . 97
4.2. Видимость версий строк в снимке . . . 98
4.3. Из чего состоит снимок . . . 99
4.4. Видимость собственных изменений . . . 104
4.5. Горизонт транзакции . . . 105
4.6. Снимок данных для системного каталога . . . 108
4.7. Экспорт снимка данных . . . 109

Глава 5. Самоочистка и hot-обновления **111**
5.1. Самоочистка . . . 111
5.2. Hot-обновления . . . 115
5.3. Самоочистка при hot-обновлениях . . . 119
5.4. Разрыв hot-цепочки . . . 121
5.5. Самоочистка индексов . . . 123

Глава 6. Очистка и заморозка **125**
6.1. Очистка 125
6.2. Еще раз о горизонте базы данных 128
6.3. Этапы выполнения очистки 131
Сканирование таблицы 131
Очистка индексов 132
Очистка таблицы 133
Усечение таблицы 133
6.4. Заморозка 134
Переполнение счетчика транзакций 134
Заморозка версий и правила видимости 135
6.5. Анализ 137

Глава 7. Автоочистка **138**
7.1. Автоматическая очистка и анализ 138
Устройство автоочистки 138
Какие таблицы требуют очистки 140
Какие таблицы требуют анализа 143
Автоочистка в действии 143
7.2. Заморозка 148
7.3. Управление заморозкой 150
Минимальный возраст для заморозки 151
Возраст для агрессивной заморозки 153
Возраст для аварийного срабатывания автоочистки 155
Возраст для приоритетного режима заморозки 157
7.4. Нетерпеливое сканирование 158
7.5. Заморозка при загрузке 161
7.6. Регулирование нагрузки и мониторинг 163
Ручная очистка 163
Автоочистка 167

Глава 8. Перестроение таблиц и индексов **170**
8.1. Полная очистка 170
Необходимость 170
Оценка плотности информации 171
Заморозка 175
8.2. Другие способы перестроения 176
Аналоги полной очистки 176

Перестроение без долгих блокировок 177
8.3. Профилактика 178
Читающие запросы 178
Обновление данных 178

Часть II. Буферный кеш и журнал **181**

Глава 9. Буферный кеш **183**
9.1. Ввод-вывод и кеширование 183
Общие соображения 183
Текущее положение дел 185
9.2. Устройство буферного кеша 188
9.3. Попадание в кеш 190
9.4. Промах кеша 195
Поиск буфера и вытеснение 196
9.5. Массовое вытеснение 198
9.6. Настройка размера 202
9.7. Прогрев кеша 206
9.8. Локальный кеш 209

Глава 10. Журнал предзаписи **211**
10.1. Журналирование 211
10.2. Устройство журнала 213
Логическая структура 213
Физическая структура 216
10.3. Контрольная точка 218
10.4. Восстановление 223
10.5. Фоновая запись 226
10.6. Настройка 227
Настройка контрольной точки 227
Настройка фоновой записи 230
Мониторинг 230

Глава 11. Режимы журнала **234**
11.1. Производительность 234
11.2. Надежность 238
Кеширование 239

Повреждение данных 240
Неатомарность записи 242
11.3. Уровни журнала 246
Minimal 247
Replica 249
Logical 251

Часть III. Блокировки **253**

Глава 12. Блокировки отношений **255**
12.1. Общие сведения о блокировках 255
12.2. Тяжелые блокировки 258
12.3. Блокировки номеров транзакций 259
12.4. Блокировки отношений 261
12.5. Очередь ожидания 264

Глава 13. Блокировки строк **268**
13.1. Устройство 268
13.2. Режимы блокировки строки 269
Исключительные режимы 269
Разделяемые режимы 271
13.3. Мультитранзакции 272
13.4. Очередь ожидания 274
Исключительные режимы 274
Разделяемые режимы 281
13.5. Блокировка без ожидания 284
13.6. Взаимоблокировки 286
Взаимоблокировка при обновлении строк 288
Взаимоблокировка двух команд UPDATE 289

Глава 14. Блокировки разных объектов **293**
14.1. Блокировки неотношений 293
14.2. Блокировки расширения отношения 295
14.3. Блокировки страниц 296
14.4. Рекомендательные блокировки 296
14.5. Предикатные блокировки 298

Глава 15. Блокировки в памяти **305**
15.1. Спин-блокировки 305
15.2. Легкие блокировки 306
15.3. Примеры 306
Буферный кеш 306
Буферы журнала предзаписи 308
Кеш SLRU 309
15.4. Мониторинг ожиданий 310
15.5. Семплирование 312

Часть IV. Выполнение запросов **317**

Глава 16. Этапы выполнения запросов **319**
16.1. Демонстрационная база данных 319
16.2. Простой протокол запросов 322
Разбор 322
Трансформация 324
Планирование 326
Исполнение 337
16.3. Расширенный протокол запросов 340
Подготовка 340
Привязка параметров 341
Планирование и исполнение 342
Получение результатов 344

Глава 17. Статистика **346**
17.1. Базовая статистика 346
17.2. Неопределенные значения 350
17.3. Уникальные значения 351
17.4. Наиболее частые значения 353
17.5. Гистограмма 356
17.6. Статистика для нескалярных типов данных 360
17.7. Средний размер поля 361
17.8. Корреляция 361
17.9. Статистика по выражению 362
Расширенная статистика по выражению 363

Статистика для индекса по выражению 364
17.10. Многовариантная статистика 365
Функциональные зависимости между столбцами 365
Многовариантное число различных значений 367
Многовариантные списки частых значений 369

Глава 18. Табличные методы доступа **371**
18.1. Подключаемые движки хранения 371
18.2. Последовательное сканирование 374
Оценка стоимости 375
18.3. Параллельные планы выполнения 379
18.4. Параллельное последовательное сканирование 380
Оценка стоимости 380
18.5. Ограничения параллельного выполнения 384
Количество рабочих процессов 384
Нераспараллеливаемые запросы 388
Ограниченно распараллеливаемые запросы 389

Глава 19. Индексные методы доступа **394**
19.1. Индексы и расширяемость 394
19.2. Классы и семейства операторов 397
Класс операторов 397
Семейство операторов 402
19.3. Интерфейс механизма индексирования 404
Свойства метода доступа 405
Свойства индекса 409
Свойства столбцов 410

Глава 20. Индексное сканирование **414**
20.1. Простое индексное сканирование 414
Оценка стоимости 415
Хороший случай: высокая корреляция 416
Плохой случай: низкая корреляция 419
20.2. Сканирование только индекса 422
Include-индексы 425
20.3. Сканирование по битовой карте 427
Точность карты 428

Действия с битовыми картами 429
Оценка стоимости 430
20.4. Параллельные версии индексного сканирования 434
20.5. Сравнение методов доступа 436

Глава 21. Вложенный цикл **438**
21.1. Виды и способы соединений 438
21.2. Соединение вложенным циклом 440
Декартово произведение 440
Параметризованное соединение 444
Кеширование (мемоизация) строк 448
Внешние соединения 451
Анти- и полусоединения 452
Неэквисоединения 454
Параллельный режим 455

Глава 22. Хеширование **457**
22.1. Соединение хешированием 457
Однопроходное соединение хешированием 457
Двухпроходное соединение хешированием 463
Динамические корректировки плана 466
Соединение хешированием в параллельных планах 470
Параллельное однопроходное хеш-соединение 471
Параллельное двухпроходное хеш-соединение 473
Модификации 476
22.2. Группировка и уникальные значения 478

Глава 23. Сортировка и слияние **483**
23.1. Соединение слиянием 483
Слияние отсортированных наборов 483
Параллельный режим 487
Модификации 488
23.2. Сортировка 489
Быстрая сортировка 491
Частичная пирамидальная сортировка 492
Внешняя сортировка 494
Инкрементальная сортировка 498

Параллельный режим 500
23.3. Группировка и уникальные значения 502
23.4. Сравнение способов соединения 505

Часть V. Типы индексов **509**

Глава 24. Хеш-индекс **511**
24.1. Общий принцип 511
24.2. Страничная организация 512
24.3. Класс операторов 519
24.4. Свойства 520
Свойства уровня метода доступа 520
Свойства уровня индекса 521
Свойства уровня столбца 522

Глава 25. B-дерево **523**
25.1. Общий принцип 523
25.2. Поиск и вставка 524
Поиск по равенству 524
Поиск по неравенству 526
Поиск по диапазону 527
Вставка 527
25.3. Страничная организация 529
Компактное хранение дубликатов 533
Компактное хранение внутренних индексных записей 535
25.4. Сортировка и составные индексы 536
Порядок столбцов 536
Оптимизация пропуска 539
Направление сортировки 544
25.5. Класс операторов 546
Семантика сравнения 546
25.6. Свойства 552
Свойства уровня метода доступа 552
Свойства уровня индекса 553
Свойства уровня столбца 553

Глава 26. Индекс GiST **555**
26.1. Общий принцип 555

26.2. R-дерево для точек 557
Страничная организация 560
Класс операторов 561
Поиск вхождения в область 563
Поиск ближайших соседей 565
Вставка 570
Ограничение исключения 571
Свойства 574
26.3. RD-дерево для полнотекстового поиска 577
Про полнотекстовый поиск 577
Индексация tsvector 578
Свойства 586
26.4. Другие типы данных 586

Глава 27. Индекс SP-GiST **589**
27.1. Общий принцип 589
27.2. Дерево квадрантов для точек 591
Класс операторов 592
Страничная организация 596
Поиск 597
Вставка 598
Свойства 601
27.3. K-мерные деревья для точек 603
27.4. Префиксное дерево для строк 605
Класс операторов 606
Поиск 607
Вставка 608
Свойства 610
27.5. Другие типы данных 611

Глава 28. Индекс GIN **612**
28.1. Общий принцип 612
28.2. Индекс для полнотекстового поиска 613
Страничная организация 615
Класс операторов 617
Поиск 619
Частые и редкие лексемы 620

Вставка 624
Ограничение выборки 626
Свойства 627
Ограничения GIN и RUM-индекс 629
28.3. Индекс для триграмм 630
28.4. Индекс для массивов 632
28.5. Индекс для JSON 636
Класс операторов jsonb_ops 636
Класс операторов jsonb_path_ops 639
28.6. Другие типы данных 641

Глава 29. Индекс BRIN **642**
29.1. Общий принцип 642
29.2. Пример 643
29.3. Страничная организация 645
29.4. Поиск 647
29.5. Обновление сводной информации 648
Вставка значений 648
Обобщение зоны 649
29.6. Диапазоны значений (minmax) 650
Выбор столбцов для индексирования 651
Размер зоны и эффективность поиска 652
Свойства 656
29.7. Мультидиапазоны значений (minmax-multi) 659
29.8. Охватывающие значения (inclusion) 662
29.9. Фильтры Блума (bloom) 665

Заключение **670**

Предметный указатель **671**

О книге

> — До чего же это все просто! — воскликнул Шпунтик. — А я где-то читал, что писателю нужен какой-то вымысел, замысел...
>
> — Э, замысел! — нетерпеливо перебил его Смекайло. — Это только в книгах так пишется, что нужен замысел, а попробуй задумай что-нибудь, когда все уже и без тебя задумано! Что ни возьми — все уже было.
>
> Николай Носов, *Приключения Незнайки и его друзей*

Для кого эта книга

Эта книга для тех, кого не устраивает работа с базой данных как с черным ящиком. Если вы любознательны, не довольствуетесь авторитетными советами и хотите во всем разобраться сами — нам по пути.

Я ориентируюсь на читателей, имеющих определенный опыт использования PostgreSQL и хотя бы в общих чертах представляющих себе, что к чему. Для совсем новичков текст будет тяжеловат. Например, я ни слова не скажу о том, как устанавливать сервер, вводить команды в psql или изменять конфигурационные параметры.

Надеюсь, что книга будет полезной и тем, кто хорошо знаком с устройством другой СУБД, но переходит на PostgreSQL и хочет разобраться в отличиях. Несколько лет назад такая книга сэкономила бы мне много времени. Именно поэтому я ее в конце концов и написал.

Чего нет в книге

Эта книга — не сборник рецептов. На все случаи жизни готовых решений не напасешься, а понимание внутренней механики сложной системы дает

возможность критически переосмысливать чужой опыт и делать свои собственные выводы. Поэтому я и объясняю такие подробности устройства, знание которых на первый взгляд не имеет практического смысла.

Но эта книга и не учебник. Она углубляется в одни области (более интересные мне самому) и обходит стороной другие. Если вы изучаете SQL, обратите внимание на учебник Евгения Моргунова *PostgreSQL. Основы языка SQL*[1], а необходимый теоретический фундамент даст книга Бориса Новикова *Основы технологий баз данных*[2].

Называться справочником эта книга тоже не претендует. Я старался быть точным, но у меня не было цели заменить книгой документацию, поэтому я легко опускал непринципиальные, на мой взгляд, подробности. В любой непонятной ситуации читайте документацию.

Еще эта книга не учит разрабатывать ядро PostgreSQL. Я не предполагаю у читателя знания языка C и ориентируюсь на администраторов и прикладных разработчиков. Хотя и ссылаюсь постоянно на исходный код, из которого можно узнать столько подробностей, сколько душе угодно, и даже больше.

Что в книге есть

Во вводной главе без особых деталей я даю основные понятия, на которые опирается все дальнейшее повествование. Я предполагаю, что вы не почерпнете из этой главы практически ничего нового, но все-таки включаю ее для полноты картины. К тому же она может пригодиться тем, кто переходит с других СУБД.

Первая часть книги посвящена вопросам согласованности и изоляции, которые я сперва рассматриваю с позиции пользователя (какие уровни изоляции существуют и чем это грозит), а затем с точки зрения внутреннего устройства. Для этого мне приходится погрузиться в детали реализации многоверсионности и изоляции на основе снимков данных. Особенно много внимания требует процедура очистки неактуальных версий строк.

[1] postgrespro.ru/education/books/sqlprimer.
[2] postgrespro.ru/education/books/dbtech.

Во второй части я рассматриваю буферный кеш и механизм, позволяющий восстанавливать согласованность после сбоев, — журнал предзаписи.

В третьей части детально разбирается устройство и использование блокировок разных уровней: легких блокировок для оперативной памяти, тяжелых блокировок для отношений, блокировок табличных строк.

Четвертая часть объясняет, как сервер планирует и выполняет SQL-запросы. Я рассказываю, какие есть способы доступа к данным, какие применяются методы соединения и как используется статистическая информация.

В пятой части обсуждение индексов, сводившееся ранее к B-деревьям, добирается и до остальных методов доступа. Сначала я рассматриваю общие принципы расширяемости, устанавливающие границы между ядром системы индексирования, индексными методами доступа и типами данных (что требует введения понятия классов операторов), а затем подробно останавливаюсь на особенностях каждого из имеющихся методов.

В состав PostgreSQL входит масса «интроспективных» расширений, которые не нужны для обычной работы, но дают возможность заглянуть во внутреннюю жизнь сервера. В книге используются многие из них. Кроме того, что эти расширения позволяют лучше изучить устройство сервера, они могут облегчить диагностику в сложных случаях.

Обозначения

Я пытался писать книгу так, чтобы ее можно было читать последовательно, страница за страницей. Но всю правду не получается раскрыть сразу, и к одной и той же теме приходится возвращаться несколько раз. Если бы я каждый раз писал «это будет рассмотрено позже», книга сильно увеличилась бы в размере, поэтому в таких случаях я ставлю на полях номер страницы, на которой тема развивается дальше. Такой же номер, ведущий назад, отсылает к месту в книге, где уже что-то говорилось о предмете обсуждения. *с. 19*

Текст книги и все примеры актуальны для PostgreSQL 18. Некоторые абзацы имеют на полях отметку о номере версии. Это означает, что сказанное справедливо для версий PostgreSQL, начиная с указанной, а более ранние версии либо вовсе не имели описанной возможности, либо были устроены v. 18

как-то иначе. Такие пометки могут оказаться полезными для тех, кто еще не обновил систему до последнего выпуска.

Также на полях указываются значения по умолчанию для обсуждаемых параметров. Сами параметры (как обычные, так и параметры хранения) выделены курсивом: *work_mem*.

4MB

В сносках я постоянно ссылаюсь на первоисточники. Их несколько, и на первом месте стоит кладезь полезной информации — документация[1]. Являясь органичной частью проекта, она всегда поддерживается в актуальном состоянии самими разработчиками. Но главный первоисточник — безусловно, исходный код[2]. Удивительно, на какое количество вопросов можно найти ответы просто в комментариях и файлах README, даже не владея языком C. Реже я ссылаюсь на записи коммитфеста[3]: в переписках pgsql-hackers всегда можно проследить историю изменений и понять логику принятых разработчиками решений, но ценой чтения огромного массива обсуждений.

> Лирические отступления и замечания, которые могут увести в сторону от основной мысли, но которые я не удержался и вставил в книгу, выделены так, чтобы их можно было пропустить.

Конечно, в книге много фрагментов кода, в основном на языке SQL. Код показан с приглашением `=>`; если необходимо, то следом за ним приведен и ответ сервера:

```
=> SELECT clock_timestamp();
        clock_timestamp
-------------------------------
 2026-04-05 20:51:34.767686+03
(1 row)
```

Если аккуратно повторять все приведенные команды в PostgreSQL 18, должен получиться такой же результат (конечно, с точностью до номеров транзакций и прочих несущественных деталей). Во всяком случае, весь код в книге — результат выполнения скрипта, содержащего ровно эти команды.

Когда требуется показать одновременную работу нескольких транзакций, код, выполняющийся в другом сеансе, выделен отступом и отчеркиванием:

[1] postgrespro.ru/docs/postgresql/18/index.
[2] git.postgresql.org/gitweb/?p=postgresql.git;a=summary.
[3] commitfest.postgresql.org.

```
=> SHOW server_version;
 server_version
----------------
 18.3
(1 row)
```

Чтобы повторить такие команды (а это полезно для самообразования, как и любые эксперименты), удобно открыть два терминала с psql.

Отдельные команды и названия различных объектов базы данных (таких как таблицы и столбцы, функции, расширения) выделены в тексте моноширинным шрифтом: `UPDATE`, `pg_class`.

Вызовы утилит из операционной системы показаны с приглашением, оканчивающимся на $:

```
postgres$ whoami
postgres
```

Я использую Linux, но без какой-либо специфики; достаточно будет самого базового понимания.

Благодарности

Книгу невозможно написать в одиночку, и это отличный повод сказать спасибо хорошим людям.

Я благодарен Павлу Лузанову, который в нужный момент предложил мне заняться чем-то действительно стоящим.

Я признателен компании Postgres Professional за возможность работать над этой книгой не только в свободное время. Но компания — это люди, и я хочу сказать отдельное спасибо Олегу Бартунову за неиссякаемую энергию и идеи и Ивану Панченко за всестороннюю поддержку и LaTeX.

Спасибо моим товарищам по образовательному отделу за творческую атмосферу и дискуссии, в ходе которых формировался материал учебных курсов и способ его подачи, что нашло свое отражение и в книге. Спасибо Павлу Толмачеву за внимательную вычитку черновиков.

Многие главы книги впервые были опубликованы в виде статей на Хабре[1], и я благодарен читателям за замечания и отклики. Они показали нужность этой работы, позволили разглядеть белые пятна в моих знаниях и сделать текст лучше.

Спасибо Людмиле Мантровой, проделавшей огромную работу над языком книги. Если вы не спотыкаетесь на каждом втором предложении, это ее заслуга.

В книге я не называю имен, но за каждой функцией и возможностью, про которые я пишу, стоит многолетний труд вполне конкретных людей. Я восхищаюсь разработчиками PostgreSQL, и мне особенно приятно, что многих из них я имею честь называть коллегами.

[1] habr.com/ru/company/postgrespro/blog.

1

Введение

1.1. Организация данных

Базы данных

PostgreSQL — программа, которая относится к классу систем управления базами данных. Когда эта программа выполняется, мы называем ее *сервером* PostgreSQL, или *экземпляром* сервера.

Данные, которыми управляет PostgreSQL, хранятся в базах данных[1]. Один экземпляр PostgreSQL одновременно работает с несколькими базами, которые вместе называются *кластером баз данных*.

Чтобы кластер можно было использовать, его необходимо *инициализировать*[2] (создать). Каталог, в котором размещаются все файлы, относящиеся к кластеру, обычно называют словом PGDATA, по имени переменной окружения, указывающей на этот каталог.

> Пакетные установки могут вводить свои «слои абстракции» над штатным механизмом PostgreSQL, явно прописывая все необходимые утилитам параметры. Сервер баз данных представляется в таком случае в виде службы операционной системы, и непосредственно с переменной PGDATA можно никогда не встретиться. Но сам термин устоялся, и я буду им пользоваться.

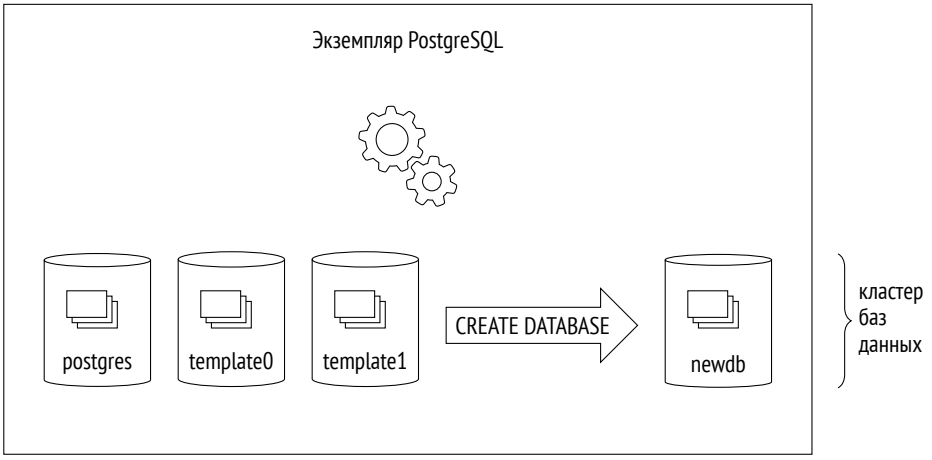
[1] postgrespro.ru/docs/postgresql/18/managing-databases.
[2] postgrespro.ru/docs/postgresql/18/app-initdb.

При инициализации в PGDATA создаются три одинаковые базы данных:

template0 используется, например, для восстановления из логической ре-
с. 631 зервной копии или для создания базы в другой кодировке и никогда не должна меняться;

template1 служит шаблоном для всех остальных баз данных, которые может создать пользователь в этом кластере;

postgres представляет собой обычную базу данных, которую можно использовать по своему усмотрению.



Системный каталог

Метаинформация обо всех объектах кластера (таких как таблицы, индексы, типы данных или функции) хранится в таблицах, относящихся к *системному каталогу*[1]. В каждой базе данных имеется собственный набор таблиц (и представлений), описывающих объекты этой конкретной базы. Существует также несколько таблиц системного каталога, общих для всего кластера, которые не принадлежат какой-либо определенной базе данных (формально используется фиктивная база с нулевым идентификатором), но доступны в любой из них.

[1] postgrespro.ru/docs/postgresql/18/catalogs.

К системному каталогу можно обращаться с помощью обычных запросов SQL, а изменения в нем происходят при выполнении команд DDL. Клиент psql располагает целым рядом специальных команд для просмотра системного каталога.

Все имена таблиц системного каталога имеют префикс `pg_`, например `pg_database`. Имена столбцов начинаются с трехбуквенного кода, который обычно соответствует имени таблицы, например `datname`.

Во всех таблицах системного каталога столбец с первичным ключом называется `oid` и имеет одноименный тип `oid` (object identifier) — целое 32-битное число.

> По сути, механизм идентификаторов объектов `oid` — аналог последовательностей, появившийся в PostgreSQL задолго до самих последовательностей. Его особенность в том, что уникальные номера используются в разных таблицах системного каталога, хотя выдаются с помощью единого счетчика. Когда общее число выданных номеров выходит за диапазон значений, счетчик обнуляется. Уникальность значений в конкретной таблице гарантируется тем, что очередное значение `oid` проверяется с помощью уникального индекса и, если оно уже используется в этой таблице, счетчик увеличивается, а проверка повторяется[1].

Схемы

Схемы[2] представляют собой пространства имен для всех объектов, хранящихся в базе данных. Кроме пользовательских схем, имеется несколько специальных служебных:

public используется по умолчанию для пользовательских объектов, если не выполнены иные настройки;

pg_catalog используется для таблиц системного каталога;

information_schema дает альтернативное представление системного каталога, регламентируемое стандартом SQL;

pg_toast используется для объектов, относящихся к TOAST; *с. 33*

[1] backend/catalog/catalog.c, функция GetNewOidWithIndex.
[2] postgrespro.ru/docs/postgresql/18/ddl-schemas.

pg_temp объединяет временные таблицы (хотя временные таблицы разных пользователей создаются в разных схемах `pg_temp_N`, каждый обращается к своим объектам, используя имя `pg_temp`).

Схемы существуют внутри базы данных, и все объекты базы принадлежат каким-либо схемам.

Если при обращении к объекту схема не указана явно, выбирается первая подходящая схема из перечисленных в *пути поиска*. Путь формируется на основе значения параметра *search_path*, к которому, в частности, неявно добавляются схемы `pg_catalog` и (при необходимости) `pg_temp`. Это позволяет иметь в разных схемах объекты с одинаковыми именами.

Табличные пространства

В отличие от логического распределения объектов по базам данных и схемам, *табличные пространства* определяют физическое расположение данных. Фактически табличное пространство — это каталог файловой системы. Например, табличные пространства можно использовать, чтобы разместить архивные данные на медленных носителях, а данные, с которыми идет активная работа, — на быстрых.

Одно и то же табличное пространство может использоваться разными базами данных, а одна база данных может хранить данные в нескольких табличных пространствах. То есть логическая и физическая структуры не зависят друг от друга.

У каждой базы данных есть так называемое *табличное пространство по умолчанию*, в котором создаются все объекты, если явно не указать иное. В этом же табличном пространстве хранятся и объекты системного каталога.

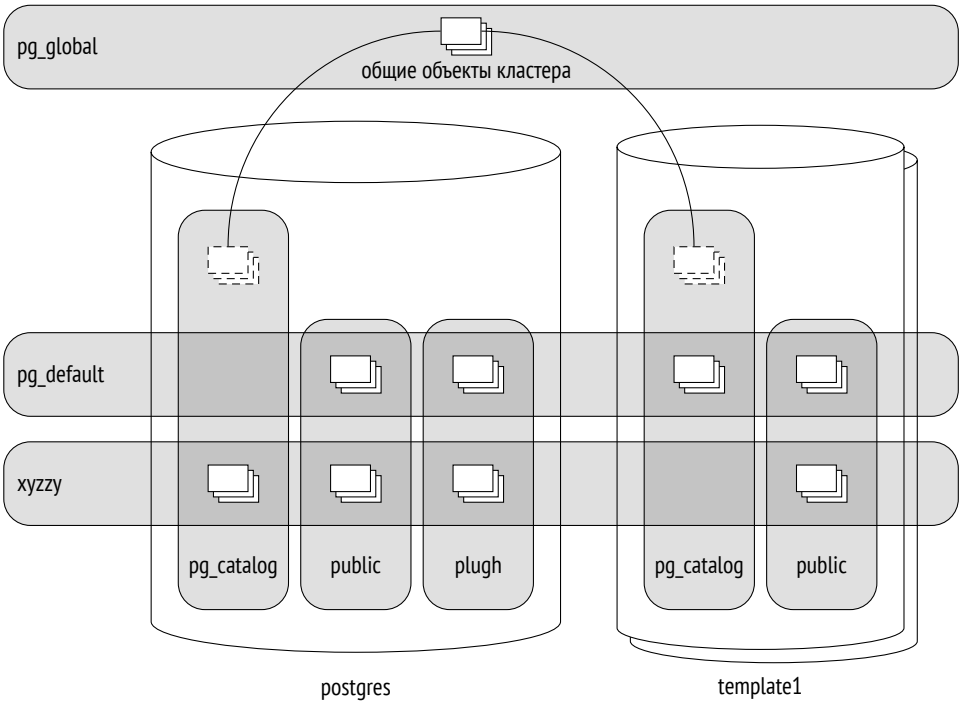
При инициализации кластера создаются два табличных пространства:

pg_default располагается в каталоге PGDATA/base и используется как табличное пространство по умолчанию, если явно не выбрать для этой цели другое пространство;

pg_global располагается в каталоге PGDATA/global и хранит общие для всего кластера объекты системного каталога.

При создании пользовательского табличного пространства указывается произвольный каталог; PostgreSQL создает на него символьную ссылку в каталоге PGDATA/pg_tblspc. Вообще, все пути PostgreSQL относительные и отсчитываются от каталога данных PGDATA. Благодаря этому можно переносить PGDATA на другое место (конечно, предварительно остановив сервер).

Рисунок сводит воедино базы данных, схемы и табличные пространства. Здесь база данных `postgres` использует табличное пространство по умолчанию `xyzzy`, а база `template1` — `pg_default`. На пересечении табличных пространств и схем находятся различные объекты базы данных:



Отношения

Самые важные объекты базы данных — *таблицы* и *индексы* — при всех своих различиях схожи в том, что состоят из строк. Для таблиц это очевидно, но и

с. 523 в B-деревьях узлы состоят из строк, содержащих индексированные значения и ссылки на другие узлы или на табличные строки.

Есть еще некоторое количество объектов, устроенных похожим образом: *последовательности* (по сути однострочные таблицы), *материализованные представления* (по сути таблицы, которые помнят запрос). А еще есть обычные *представления*, которые сами по себе не хранят данные, но во всех остальных смыслах похожи на таблицы.

Все эти объекты в PostgreSQL называются общим словом *отношение* (relation).

> Слово, на мой взгляд, выбрано неудачно, поскольку смешивает таблицы базы данных и (настоящие) отношения реляционной теории. Дает о себе знать университетское происхождение проекта и склонность его основателя, Майкла Стоунбрейкера, во всем видеть отношения. В одной из работ он даже ввел понятие «упорядоченного отношения» (ordered relation) для таблицы, в которой порядок строк задается индексом.
>
> Таблица системного каталога для отношений изначально называлась `pg_relation`, но довольно скоро, на волне увлечения объектной ориентированностью, ее переименовали в привычный нам сейчас `pg_class`. Однако столбцы этой таблицы по-прежнему имеют префикс `rel`.

Слои и файлы

Информация, связанная с отношением, организована в несколько *слоев*[1] (forks) разных типов, каждый из них содержит определенный вид данных.

Изначально слой представлен единственным *файлом*. Имя файла состоит из числового идентификатора (`oid`), к которому может быть добавлен суффикс, соответствующий типу слоя.

Файл постепенно растет, и когда его размер достигает 1 Гбайта, создается следующий файл этого же слоя (такие файлы иногда называют *сегментами*). Порядковый номер сегмента добавляется в конец имени файла.

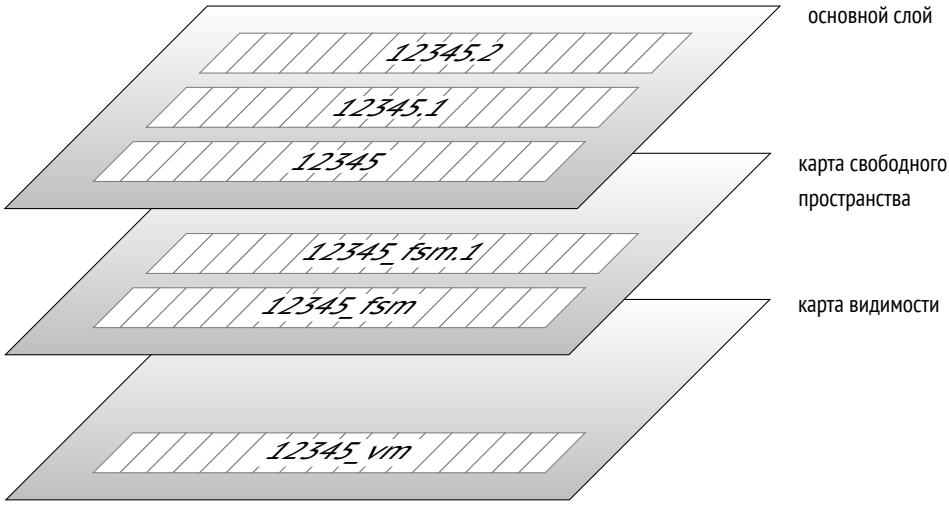
Ограничение размера файла в 1 Гбайт возникло исторически для поддержки различных файловых систем, некоторые из которых не умеют работать

[1] postgrespro.ru/docs/postgresql/18/storage-file-layout.

с файлами большого размера. Ограничение можно изменить при сборке PostgreSQL (`./configure --with-segsize`).

Таким образом, одному отношению на диске может соответствовать несколько файлов. Даже для небольшой таблицы без индексов их будет минимум три, по числу обязательных слоев.

Внутри каталога каждого табличного пространства (кроме `pg_global`) создаются подкаталоги для отдельных баз данных. Все файлы объектов, принадлежащих одному табличному пространству и одной базе данных, будут помещены в один подкаталог. Это необходимо учитывать, потому что файловые системы обычно не очень хорошо работают с большим количеством файлов в каталоге.



Имеется несколько стандартных типов слоев.

Основной слой (main fork) — это собственно данные: те самые табличные или индексные строки. Основной слой существует для любых отношений (кроме представлений, которые не содержат данных).

Имена файлов основного слоя состоят только из числового идентификатора (равного значению столбца `relfilenode` таблицы `pg_class`).

Посмотрим для примера путь к файлу таблицы, созданной в табличном пространстве `pg_default`:

```
=> CREATE UNLOGGED TABLE t(
  a integer,
  b numeric,
  c text,
  d json
);
=> INSERT INTO t VALUES (1, 2.0, 'foo', '{}');
=> SELECT pg_relation_filepath('t');
 pg_relation_filepath
----------------------
 base/16384/16385
(1 row)
```

Каталог base соответствует табличному пространству pg_default, следующий подкаталог — базе данных, и уже в нем находится интересующий нас файл:

```
=> SELECT oid FROM pg_database WHERE datname = 'internals';
  oid
-------
 16384
(1 row)
=> SELECT relfilenode FROM pg_class WHERE relname = 't';
 relfilenode
-------------
       16385
(1 row)
```

Вот соответствующий файл в файловой системе:

```
=> SELECT size
FROM pg_stat_file('/usr/local/pgsql/data/base/16384/16385');
 size
------
 8192
(1 row)
```

Слой инициализации (init fork)[1] существует только для нежурналируемых таблиц (созданных с указанием UNLOGGED) и их индексов. Такие объекты

[1] postgrespro.ru/docs/postgresql/18/storage-init.

ничем не отличаются от обычных, кроме того, что действия с ними не записываются в журнал предзаписи. За счет этого работа с ними происходит быстрее, но в случае сбоя невозможно восстановить данные в согласованном состоянии. Поэтому при восстановлении PostgreSQL просто удаляет все слои таких объектов и записывает слой инициализации на место основного слоя. В результате получается «пустышка». *с. 211*

Таблица `t` создана нежурналируемой, поэтому у нее есть слой инициализации. Он имеет такое же имя, как и основной слой, но с суффиксом _init:

```
=> SELECT size
FROM pg_stat_file('/usr/local/pgsql/data/base/16384/16385_init');
 size
------
    0
(1 row)
```

Карта свободного пространства (free space map)[1] — слой, в котором отслеживается примерный объем свободного места внутри страниц. Этот объем постоянно меняется: при добавлении новых версий строк уменьшается, при очистке — увеличивается. Карта свободного пространства используется при вставке новых версий строк, чтобы быстро найти подходящую страницу, на которую поместятся добавляемые данные.

Файлы, относящиеся к карте свободного пространства, имеют суффикс _fsm. Но появляются они не сразу, а только при необходимости. Самый простой способ добиться этого — выполнить очистку таблицы: *с. 133*

```
=> VACUUM t;
=> SELECT size
FROM pg_stat_file('/usr/local/pgsql/data/base/16384/16385_fsm');
 size
-------
 24576
(1 row)
```

[1] postgrespro.ru/docs/postgresql/18/storage-fsm; backend/storage/freespace/README.

Для ускорения поиска карта свободного пространства организована как дерево и занимает минимум три страницы (отсюда и размер файла слоя для почти пустой таблицы).

Карта свободного пространства существует не только для таблиц, но и для индексов. Но поскольку индексную строку нельзя добавить на произвольную страницу (например, для B-дерева место вставки определяется порядком сортировки), отслеживаются только те страницы, которые были полностью очищены и могут быть снова задействованы в индексной структуре.

Карта видимости (visibility map)[1] — слой, который позволяет быстро определить, требует ли страница очистки или заморозки. Для этого на каждую табличную страницу в этом слое отведено два бита.

Одним битом отмечены страницы, которые содержат только актуальные версии строк. Процесс очистки пропускает такие страницы, поскольку в них нечего очищать. Кроме того, когда транзакция пытается прочитать строку из такой страницы, можно не проверять ее видимость, а это позволяет использовать сканирование только индекса.

с. 131

с. 422

Второй бит отмечает страницы, на которых все версии строк заморожены. Эту часть слоя я буду называть *картой заморозки*.

с. 134

Файлы карты видимости имеют суффикс _vm. Обычно они самые небольшие по размеру:

```
=> SELECT size
FROM pg_stat_file('/usr/local/pgsql/data/base/16384/16385_vm');
 size
------
 8192
(1 row)
```

Карта видимости существует для таблиц, но не для индексов.

с. 89

[1] postgrespro.ru/docs/postgresql/18/storage-vm.

Страницы

Для удобства организации ввода-вывода файлы логически поделены на *страницы* (или *блоки*) — это минимальный объем данных, который считывается или записывается. Соответственно, и многие внутренние алгоритмы PostgreSQL ориентированы на работу со страницами. *с. 75*

Обычно страница имеет размер 8 Кбайт. Его можно поменять в некоторых пределах (до 32 Кбайт), но только при сборке (`./configure --with-blocksize`), и, как правило, никто этим не занимается. Собранный и запущенный экземпляр может работать со страницами только одного размера — создать табличные пространства с разноразмерными страницами нельзя.

Независимо от того, к какому слою принадлежат файлы, они используются сервером примерно одинаково. Страницы сначала помещаются в буферный кеш (где их могут читать и изменять процессы), а затем при необходимости вытесняются обратно на диск. *с. 183*

TOAST

Каждая строка должна помещаться целиком на одну страницу: нет способа «продолжить» строку на следующей странице. Для длинных строк используется технология, названная TOAST[1] (The Oversized Attributes Storage Technique).

TOAST подразумевает несколько стратегий. Длинные значения атрибутов можно отправить в отдельную служебную таблицу, предварительно нарезав на небольшие фрагменты-«тосты». Другой вариант — сжать длинное значение так, чтобы строка все-таки поместилась на одну страницу. А можно и то и другое: сначала сжать, а уже потом нарезать и отправить.

Если основная таблица содержит потенциально длинные атрибуты, для нее сразу же создается отдельная toast-таблица (одна для всех атрибутов). Например, если в таблице есть столбец типа `numeric` или `text`, toast-таблица будет создана, даже если в таком столбце никогда не будут храниться длинные значения.

[1] postgrespro.ru/docs/postgresql/18/storage-toast; include/access/heaptoast.h.

Для индексов технология TOAST предлагает только сжатие; вынесение атрибутов в отдельную таблицу не поддерживается. Это накладывает ограничение на размер индексируемых ключей (как это ограничение реализуется,
с. 397 зависит от класса операторов).

Обычно стратегии определяются типами столбцов. Посмотреть стратегии проще всего командой \d+ в psql, но для сокращения вывода я использую запрос к системному каталогу:

```
\d+  => SELECT attname, atttypid::regtype,
       CASE attstorage
         WHEN 'p' THEN 'plain'
         WHEN 'e' THEN 'external'
         WHEN 'm' THEN 'main'
         WHEN 'x' THEN 'extended'
       END AS storage
     FROM pg_attribute
     WHERE attrelid = 't'::regclass AND attnum > 0;
 attname | atttypid | storage
---------+----------+----------
 a       | integer  | plain
 b       | numeric  | main
 c       | text     | extended
 d       | json     | extended
(4 rows)
```

Стратегии состоят в следующем:

plain — TOAST не используется (стратегия применяется для заведомо «коротких» типов данных, как integer);

extended — допускается как сжатие, так и хранение в отдельной toast-таблице;

external — длинные значения хранятся в toast-таблице несжатыми;

main — длинные значения в первую очередь сжимаются, а в toast-таблицу попадают, только если сжатие не помогло.

В общих чертах алгоритм выглядит следующим образом[1]. PostgreSQL стремится к тому, чтобы на странице помещалось хотя бы четыре строки. Поэтому, если размер строки превышает четвертую часть страницы за вычетом

[1] backend/access/heap/heaptoast.c.

заголовка (для страницы стандартного размера это около 2000 байт), к части значений необходимо применить TOAST. Действуем в порядке, описанном ниже, и прекращаем, как только длина строки перестает превышать пороговое значение.

1. Сначала перебираем атрибуты со стратегиями external и extended, двигаясь от самых длинных к более коротким. Extended-атрибуты сжимаются, и если после этого значение (само по себе, без учета других атрибутов) превосходит четверть страницы, оно сразу же отправляется в toast-таблицу. External-атрибуты обрабатываются так же, но не сжимаются.
2. Если после первого прохода строка все еще не помещается, по одному отправляем в toast-таблицу оставшиеся атрибуты со стратегиями external и extended.
3. Если и это не помогло, пытаемся сжать атрибуты со стратегией main, оставляя их при этом в табличной странице.
4. Если строка все равно недостаточно коротка, отправляем в toast-таблицу main-атрибуты.

Пороговое значение составляет все те же 2000 байт, но может переопределяться параметром хранения *toast_tuple_target* на уровне таблицы.

Иногда может оказаться полезным изменить стратегию для некоторых столбцов. Если заранее известно, что данные в столбце не сжимаются (например, в столбце хранятся JPEG-изображения), можно установить для него стратегию external — это позволит сэкономить на бесполезных попытках сжатия. Изменить стратегию можно как при создании таблицы (предложение STORAGE команды CREATE TABLE), так и впоследствии: v. 16

```
=> ALTER TABLE t ALTER COLUMN d SET STORAGE external;
```

Повторив запрос, получим:

```
 attname | atttypid | storage
---------+----------+----------
 a       | integer  | plain
 b       | numeric  | main
 c       | text     | extended
 d       | json     | external
(4 rows)
```

v. 15 pglz

Для сжатия доступны два алгоритма: традиционный PGLZ и более современный LZ4 (оба являются разновидностями алгоритма словарного сжатия Лемпеля—Зива). По умолчанию сжатие выполняется алгоритмом, указанным в параметре *default_toast_compression*; значение можно установить для отдельных столбцов в предложении COMPRESSION. Эталонные тесты показывают, что при сходном с PGLZ уровне сжатия LZ4 тратит меньше ресурсов процессора.

Toast-таблицы располагаются в отдельной схеме pg_toast, не входящей в путь поиска, и поэтому обычно не видны. Для временных таблиц используется схема pg_toast_temp_*N* аналогично обычной pg_temp_*N*.

Конечно, при желании всегда можно подглядеть за внутренней механикой процесса. Скажем, в таблице t есть три потенциально длинных атрибута, поэтому toast-таблица обязана быть. Вот она:

```
=> SELECT relnamespace::regnamespace, relname
FROM pg_class WHERE oid = (
  SELECT reltoastrelid FROM pg_class WHERE relname = 't'
);
 relnamespace |    relname
--------------+----------------
 pg_toast     | pg_toast_16385
(1 row)
```

```
=> \d+ pg_toast.pg_toast_16385
TOAST table "pg_toast.pg_toast_16385"
   Column   |  Type   | Storage
------------+---------+---------
 chunk_id   | oid     | plain
 chunk_seq  | integer | plain
 chunk_data | bytea   | plain
Owning table: "public.t"
Indexes:
    "pg_toast_16385_index" PRIMARY KEY, btree (chunk_id, chunk_seq)
Access method: heap
```

Логично, что для «тостов», на которые нарезается строка, применяется стратегия plain: TOAST второго уровня не существует.

Вместе с toast-таблицей в той же схеме создается и индекс, который *всегда* используется для доступа к фрагментам значений. Имя индекса видно в выводе команды, но его можно найти и запросом:

```
=> SELECT indexrelid::regclass FROM pg_index
WHERE indrelid = (
  SELECT oid FROM pg_class WHERE relname = 'pg_toast_16385'
);
          indexrelid
-------------------------------
 pg_toast.pg_toast_16385_index
(1 row)

=> \d pg_toast.pg_toast_16385_index
Unlogged index "pg_toast.pg_toast_16385_index"
  Column   |  Type   | Key? | Definition
-----------+---------+------+------------
 chunk_id  | oid     | yes  | chunk_id
 chunk_seq | integer | yes  | chunk_seq
primary key, btree, for table "pg_toast.pg_toast_16385"
```

Таким образом, toast-таблица увеличивает минимальное число файлов, «обслуживающих» таблицу, до восьми: три слоя основной таблицы, три слоя toast-таблицы и два слоя toast-индекса.

Столбец c использует стратегию extended, значения в нем будут сжиматься:

```
=> UPDATE t SET c = repeat('A',5000);
=> SELECT * FROM pg_toast.pg_toast_16385;
 chunk_id | chunk_seq | chunk_data
----------+-----------+------------
(0 rows)
```

В toast-таблице ничего нет: повторяющиеся символы сжались алгоритмом PGLZ, и после этого значение поместилось в обычной табличной странице.

А теперь составим значение из случайных символов:

```
=> UPDATE t SET c = (
  SELECT string_agg( chr(trunc(65+random()*26)::integer), '')
  FROM generate_series(1,5000)
)
RETURNING left(c,10) || '...' || right(c,10);
        ?column?
-------------------------
 TGSXWZTOBN...SJWTGKUXDT
(1 row)
UPDATE 1
```

Такую последовательность сжать не получается. В табличную строку записывается идентификатор chunk_id:

```
=> SELECT pg_column_toast_chunk_id(c) -- начиная с v.17
FROM t;
 pg_column_toast_chunk_id
--------------------------
                    16390
(1 row)
```

А само значение попадает в toast-таблицу:

```
=> SELECT chunk_id,
  chunk_seq,
  length(chunk_data),
  left(encode(chunk_data,'escape')::text, 10) || '...' ||
  right(encode(chunk_data,'escape')::text, 10)
FROM pg_toast.pg_toast_16385;
 chunk_id | chunk_seq | length |        ?column?
----------+-----------+--------+------------------------
    16390 |         0 |   1996 | TGSXWZTOBN...LGRUNTKNNC
    16390 |         1 |   1996 | AFBHZKOFQG...KMKVNXBKWL
    16390 |         2 |   1008 | VEVQVJGFYA...SJWTGKUXDT
(3 rows)
```

Видно, что данные нарезаны на фрагменты. Размер фрагментов выбирается так, чтобы на странице toast-таблицы помещалось четыре строки. От версии к версии это значение может немного меняться в зависимости от размера заголовка страницы.

При обращении к «длинному» атрибуту PostgreSQL автоматически, прозрачно для приложения, восстанавливает исходное значение и возвращает его клиенту. Если же такие атрибуты не участвуют в запросе, то toast-таблица не читается. Это одна из причин не использовать «звездочку» в производственном коде.

v. 13 Если клиент запрашивает начальную часть длинного значения, то будут прочитаны только необходимые фрагменты, в том числе и в случае, когда значение хранится в сжатом виде.

Тем не менее и на сжатие с нарезкой, и на последующее восстановление тратится довольно много ресурсов. Поэтому хранить объемные данные в PostgreSQL — не лучшая идея, особенно если они активно используются

и при этом для них не требуется транзакционная логика (как пример: отсканированные оригиналы бухгалтерских документов). Потенциально более выгодная альтернатива — хранить такие данные в файловой системе, а в базе данных держать только имена соответствующих файлов. Правда, тогда СУБД не сможет обеспечивать согласованность данных.

Затраты на распаковку значений, обработанных TOAST, позволяет оценить команда `EXPLAIN` с параметрами `analyze` и `serialize`. v. 17

1.2. Процессы и память

Экземпляр сервера PostgreSQL состоит из нескольких взаимодействующих процессов.

В первую очередь при старте сервера запускается процесс postgres, традиционно называемый postmaster. Postmaster запускает все остальные процессы (в Unix-подобных системах для этого используется системный вызов `fork`) и «присматривает» за ними — если какой-нибудь процесс завершится аварийно, postmaster перезапустит его (или весь сервер, если сочтет, что процесс мог повредить общие данные).

Процессная модель применяется в PostgreSQL с самого начала проекта из-за своей простоты, и все это время не прекращаются дискуссии о переходе к использованию потоков.

У текущей модели есть ряд недостатков: статически выделяемая общая память не позволяет на лету менять размер таких структур, как буферный кеш; параллельные алгоритмы сложны в реализации и менее эффективны, чем могли бы быть; сеансы жестко привязаны к процессам. Использование потоков выглядит многообещающе, хотя и чревато сложностями с изолированностью, совместимостью с операционными системами, управлением ресурсами. Не говоря уже о том, что переход потребует радикальных изменений в коде и годы работы. Пока побеждает консервативный взгляд, и в ближайшее время никаких изменений не предвидится.

Работу сервера обеспечивает ряд служебных процессов. Основные из них:

startup восстанавливает систему после сбоев; *с. 223*

autovacuum очищает таблицы и индексы от неактуальных данных; *с. 138*

с. 236 **wal writer** записывает на диск журнальные записи;

с. 219 **checkpointer** выполняет контрольную точку;

с. 226 **writer** записывает грязные страницы на диск;

wal sender передает журнальные записи на реплику;

wal receiver принимает журнальные записи на реплике.

Некоторые из этих процессов завершаются после выполнения своей задачи, другие работают постоянно в фоновом режиме, а какие-то могут быть отключены.

> Каждый процесс управляется конфигурационными параметрами, иногда десятками параметров. Чтобы осмысленно выполнять настройку сервера, нужно хорошо представлять его внутреннее устройство. Но из общих соображений можно выбрать лишь начальные адекватные значения параметров, которые затем потребуется уточнять, получая обратную связь от мониторинга.

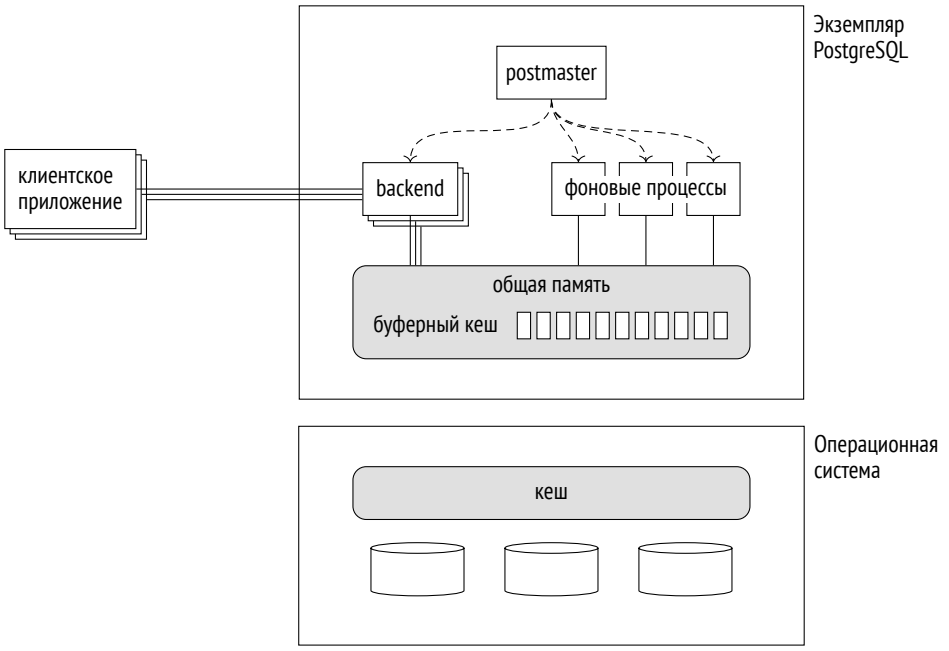
Чтобы процессы могли обмениваться информацией, postmaster выделяет *общую память*, которая доступна всем процессам.

Из-за того, что диски (особенно HDD, но и SSD тоже) работают значитель-
с. 183 но медленнее, чем оперативная память, применяется кеширование: в общей области оперативной памяти отводится место под недавно прочитанные страницы в надежде, что они еще не раз понадобятся и можно будет сэкономить на повторном обращении к диску. Измененные данные также записываются на диск не сразу, а через некоторое время.

Буферный кеш занимает бóльшую часть общей памяти. В ней же располагаются и другие буферы, которые используются сервером для ускорения работы с диском.

Свой кеш имеется и у операционной системы. PostgreSQL не использует (почти) прямой ввод-вывод в обход механизмов операционной системы, поэтому кеширование получается двойным.

При сбое (например, при аварийном отключении питания или крахе операционной системы) содержимое оперативной памяти, включая буферный кеш, пропадает. На диске остаются файлы, страницы которых записаны



в разные моменты времени. Чтобы иметь возможность восстановить согласованность данных, в процессе работы PostgreSQL ведет *журнал предзаписи* (WAL), позволяющий при необходимости выполнить потерянные операции повторно. *с. 211*

1.3. Клиенты и клиент-серверный протокол

Еще одна задача процесса postmaster — слушать входящие соединения. При появлении нового клиента postmaster порождает для него *обслуживающий процесс*[1] (backend). Клиент устанавливает соединение и начинает *сеанс* общения со своим серверным процессом. Сеанс продолжается до отключения клиента или разрыва связи.

[1] backend/tcop/postgres.c, функция PostgresMain.

Для каждого клиента на сервере порождается собственный обслуживающий процесс. Большое количество подключений может вызывать проблемы:

- Каждому процессу требуется оперативная память для хранения кеша системного каталога, подготовленных запросов, промежуточных результатов при выполнении запросов и других данных. Чем больше соединений открыто, тем больше должно быть доступной памяти.

с. 340
с. 338

- Если соединения выполняются часто, а сеансы при этом короткие (то есть клиент выполняет один небольшой запрос и отключается), непозволительно много ресурсов будет тратиться на установление соединения, порождение нового процесса и ненужное заполнение локальных кешей.

- Чем больше запущено процессов, тем больше времени требуется на просмотр их списка, а эта операция выполняется очень часто. В результате с увеличением числа клиентов производительность может падать.

с. 99

В таких случаях используют *пул соединений*, чтобы ограничить число обслуживающих процессов. PostgreSQL не имеет встроенного пула соединений, поэтому приходится применять сторонние решения: менеджеры пулов, встроенные в сервер приложений, или внешние программы (такие как PgBouncer[1] или Odyssey[2]). При этом, как правило, один процесс на сервере поочередно выполняет транзакции разных клиентов. Это накладывает определенные ограничения на разработку приложений, позволяя использовать средства, которые локализованы в пределах транзакции, но не сеанса.

Чтобы клиент и сервер понимали друг друга, они должны использовать один и тот же протокол взаимодействия[3]. Обычно для реализации протокола используют штатную библиотеку libpq, хотя встречаются и независимые реализации.

Говоря в самых общих чертах, протокол позволяет клиенту подключиться к серверу и выполнять SQL-запросы.

[1] pgbouncer.org.
[2] yandex.ru/dev/odyssey.
[3] postgrespro.ru/docs/postgresql/18/protocol.

Подключение всегда выполняется под определенной ролью (пользователем) и к конкретной базе данных. Несмотря на то что сервер работает с кластером баз данных, для использования в приложении сразу нескольких баз придется выполнить отдельное подключение к каждой из них. При подключении выполняется *аутентификация*: обслуживающий процесс удостоверяется, что пользователь является тем, за кого себя выдает (например, спросив пароль), и проверяет полномочия пользователя на подключение к серверу и к выбранной базе данных.

SQL-запросы передаются обслуживающему процессу в текстовом виде. Процесс разбирает текст, оптимизирует запрос, выполняет его и возвращает результат клиенту.

Часть I

ИЗОЛЯЦИЯ И МНОГОВЕРСИОННОСТЬ

2

Изоляция

2.1. Согласованность

Важная особенность реляционных СУБД — обеспечение *согласованности* (consistency), то есть *корректности* данных.

Известно, что на уровне базы данных можно создавать *ограничения целостности* (integrity constraints), такие как `NOT NULL` или `UNIQUE`. СУБД следит за тем, чтобы данные никогда не нарушали эти ограничения, то есть оставались целостными.

Если бы все ограничения были сформулированы на уровне базы данных, согласованность была бы гарантирована. Но некоторые условия слишком сложны для этого, например охватывают сразу несколько таблиц. И даже если ограничение в принципе можно было бы определить в базе данных, но из каких-то соображений оно не определено, это не означает, что его можно нарушать.

Итак, согласованность строже, чем целостность, но что конкретно под ней понимается, СУБД не знает. Если приложение нарушит согласованность, не нарушая целостности, у СУБД не будет способа узнать об этом. Получается, что гарантом согласованности выступает приложение, и остается верить, что оно написано корректно и никогда не ошибается.

Но если приложение выполняет только корректные последовательности операторов, в чем тогда роль СУБД?

Во-первых, корректная последовательность операторов может временно нарушать согласованность данных, и это, как ни странно, нормально.

Заезженный, но понятный пример состоит в переводе средств с одного счета на другой. Правило согласованности может звучать так: *перевод никогда не меняет общей суммы денег на счетах*. Такое правило довольно трудно (хотя и возможно) записать на SQL в виде ограничения целостности, так что пусть оно существует на уровне приложения и остается невидимым для СУБД. Перевод состоит из двух операций: первая уменьшает средства на одном счете, вторая — увеличивает на другом. Первая операция нарушает согласованность данных, вторая — восстанавливает.

Если первая операция выполнится, а вторая — по причине какого-то сбоя — нет, то согласованность нарушится. А это недопустимо. Ценой неимоверных усилий такие ситуации можно обрабатывать на уровне приложения, но, к счастью, это не требуется. Задачу полностью решает СУБД, если знает, что две операции составляют неделимое целое, то есть *транзакцию*.

Но есть и второй, более тонкий момент. Транзакции, абсолютно правильные сами по себе, при одновременном выполнении могут начать работать некорректно. Это происходит из-за того, что перемешивается порядок выполнения операций разных транзакций. Если бы СУБД сначала выполняла все операции одной транзакции, а только потом — все операции другой, такой проблемы не возникало бы, но без распараллеливания работы производительность была бы невообразимо низкой.

> Действительно одновременно транзакции работают в системах, где это позволяет аппаратура: многоядерный процессор, дисковый массив. Но все те же рассуждения справедливы и для сервера, который выполняет команды последовательно в режиме разделения времени. Иногда для обобщения используют термин *конкурентное выполнение*.

Ситуации, когда корректные транзакции некорректно работают вместе, называются *аномалиями* одновременного выполнения.

Простой пример: если приложение хочет получить из базы согласованные данные, то оно как минимум не должно видеть изменения других незафиксированных транзакций. Иначе (если какая-либо транзакция будет отменена) можно увидеть состояние, в котором база данных никогда не находилась. Такая аномалия называется *грязным чтением*. Есть множество других, более сложных аномалий.

Роль СУБД состоит в том, чтобы выполнять транзакции параллельно и при этом гарантировать, что результат такого одновременного выполнения будет совпадать с результатом одного из возможных последовательных выполнений. Иными словами — *изолировать* транзакции друг от друга, устранив любые возможные аномалии.

Таким образом, транзакцией называется множество операций, которые переводят базу данных из одного корректного состояния в другое корректное состояние (*согласованность*) при условии, что транзакция выполнена полностью (*атомарность*) и без помех со стороны других транзакций (*изоляция*). Это определение объединяет требования, стоящие за первыми тремя буквами акронима ACID: Atomicity, Consistency, Isolation. Они настолько тесно связаны друг с другом, что рассматривать их по отдельности просто нет смысла. На самом деле сложно отделить и требование долговечности (Durability), ведь при крахе системы в ней остаются изменения незафиксированных транзакций, а с ними приходится что-то делать, чтобы восстановить согласованность данных. *с. 211*

Получается, что СУБД помогает приложению поддерживать согласованность, учитывая состав транзакции, но не имея при этом понятия о подразумеваемых правилах согласованности.

К сожалению, реализация полной изоляции — технически сложная задача, сопряженная с уменьшением производительности системы. Поэтому на практике почти всегда применяется ослабленная изоляция, которая предотвращает некоторые, но не все аномалии. А это означает, что часть работы по обеспечению согласованности данных ложится на приложение. Именно поэтому очень важно понимать, какой уровень изоляции используется в системе, какие гарантии он дает, а какие — нет, и как в таких условиях писать корректный код.

2.2. Уровни изоляции и аномалии в стандарте SQL

Стандарт SQL описывает четыре уровня изоляции[1]. Эти уровни определяются перечислением аномалий, которые допускаются или не допускаются

[1] postgrespro.ru/docs/postgresql/18/transaction-iso.

при одновременном выполнении транзакций. Поэтому разговор об уровнях придется начать с аномалий.

Стоит иметь в виду, что стандарт — некое теоретическое построение, которое влияет на практику, но с которым практика в то же время сильно расходится. Поэтому все примеры здесь умозрительные. Они будут использовать операции над счетами клиентов: это довольно наглядно, хотя, надо признать, не имеет ни малейшего отношения к тому, как банковские операции устроены в действительности.

Интересно, что со стандартом SQL расходится и настоящая теория баз данных[1], развившаяся уже после того, как стандарт был принят, а практика успела уйти вперед.

Потерянное обновление

Аномалия *потерянного обновления* (lost update) возникает, когда две транзакции читают одну и ту же строку таблицы, затем одна из них обновляет эту строку, после чего вторая обновляет эту же строку, не учитывая изменений, сделанных первой транзакцией.

Например, две транзакции собираются увеличить сумму на одном и том же счете на 100 ₽. Первая транзакция читает текущее значение (1000 ₽), затем вторая транзакция читает то же самое значение. Первая увеличивает сумму (получается 1100 ₽) и записывает в базу это новое значение. Вторая поступает так же: получает те же 1100 ₽ и записывает их. В результате клиент потерял 100 ₽.

Потерянное обновление не допускается стандартом ни на одном уровне изоляции.

Грязное чтение и Read Uncommitted

Аномалия *грязного чтения* (dirty read) возникает, когда транзакция читает еще не зафиксированные изменения, сделанные другой транзакцией.

[1] postgrespro.ru/education/books/dbtech.

Например, первая транзакция переводит 100 ₽ на пустой счет клиента, но не фиксирует изменение. Другая транзакция читает состояние счета (обновленное, но не зафиксированное) и позволяет клиенту снять наличные — несмотря на то, что первая транзакция прерывается и отменяет свои изменения, так что никаких денег на счете клиента нет.

Грязное чтение допускается стандартом на уровне Read Uncommitted.

Неповторяющееся чтение и Read Committed

Аномалия *неповторяющегося чтения* (non-repeatable read) возникает, когда транзакция читает одну и ту же строку два раза, а в промежутке между чтениями вторая транзакция изменяет (или удаляет) эту строку и фиксирует изменения. Тогда первая транзакция получит разные результаты.

Например, пусть правило согласованности *запрещает отрицательные суммы на счетах клиентов*. Первая транзакция собирается уменьшить сумму на счете на 100 ₽. Она проверяет текущее значение, получает 1000 ₽ и решает, что уменьшение возможно. В это время вторая транзакция уменьшает сумму на счете до нуля и фиксирует изменения. Если бы теперь первая транзакция повторно проверила сумму, она получила бы 0 ₽ (но она уже приняла решение уменьшить значение, и счет «уходит в минус»).

Неповторяющееся чтение допускается стандартом на уровнях Read Uncommitted и Read Committed.

Фантомное чтение и Repeatable Read

Аномалия *фантомного чтения* (phantom read) возникает, когда одна транзакция два раза читает набор строк по одинаковому условию, а в промежутке между чтениями другая транзакция добавляет строки, удовлетворяющие этому условию, и фиксирует изменения. Тогда первая транзакция получит разные наборы строк.

Например, пусть правило согласованности *запрещает клиенту иметь более трех счетов*. Первая транзакция собирается открыть новый счет, проверяет их текущее количество (скажем, два) и решает, что открытие возможно. В это

время вторая транзакция тоже открывает клиенту новый счет и фиксирует изменения. Если бы теперь первая транзакция перепроверила количество, она получила бы три (но она уже выполняет открытие еще одного счета, и у клиента их оказывается четыре).

Фантомное чтение допускается стандартом на уровнях Read Uncommitted, Read Committed и Repeatable Read.

Отсутствие аномалий и Serializable

Стандарт определяет и уровень, на котором не допускаются никакие аномалии, — Serializable. И это совсем не то же самое, что запрет на потерянное обновление и на грязное, неповторяющееся и фантомное чтение. Дело в том, что существует значительно больше известных аномалий, чем перечислено в стандарте, и еще неизвестное число пока неизвестных.

Уровень Serializable должен предотвращать *любые* аномалии. Это означает, что на таком уровне разработчику приложения не надо думать об изоляции. Если транзакции выполняют корректные последовательности операторов, работая в одиночку, данные останутся согласованными и при одновременной работе этих транзакций.

В качестве иллюстрации приведу известную всем таблицу из стандарта, к которой для ясности добавлен последний столбец:

	потерянные изменения	грязное чтение	неповторяющееся чтение	фантомное чтение	другие аномалии
Read Uncommitted	–	да	да	да	да
Read Committed	–	–	да	да	да
Repeatable Read	–	–	–	да	да
Serializable	–	–	–	–	–

Почему именно эти аномалии?

Почему из множества возможных аномалий в стандарте перечислены только несколько, и почему именно такие?

Достоверно этого, видимо, никто не знает. Но не исключено, что о других аномалиях во времена принятия первых версий стандарта просто не задумывались, поскольку теория заметно отставала от практики.

Кроме того, предполагалось, что изоляция должна быть построена на блокировках. Идея широко применявшегося *протокола двухфазного блокирования* (2PL) состоит в том, что в процессе выполнения транзакция блокирует затронутые строки, а при завершении — освобождает блокировки. Сильно упрощая: чем больше блокировок захватывает транзакция, тем лучше она изолирована от других транзакций. Но и тем сильнее страдает производительность системы, поскольку вместо совместной работы транзакции начинают выстраиваться в очередь за одними и теми же строками.

Как мне представляется, разница между стандартными уровнями изоляции в значительной степени объясняется как раз количеством необходимых для их реализации блокировок.

Если транзакция блокирует изменяемые строки для изменений, но не для чтения, получаем уровень Read Uncommitted с возможностью прочитать незафиксированные данные.

Если изменяемые строки блокируются и для чтения, и для изменений, получаем уровень Read Committed: незафиксированные данные прочитать нельзя, но при повторном обращении к строке можно получить другое значение (неповторяющееся чтение).

Если для всех операций блокируются и читаемые, и изменяемые строки, получаем уровень Repeatable Read: повторное чтение строки будет выдавать то же значение.

Но с Serializable проблема: невозможно заблокировать строку, которой еще нет. Из-за этого остается возможность фантомного чтения: другая транзакция может добавить строку, попадающую под условия выполненного ранее запроса, и эта строка окажется в повторной выборке.

Поэтому для полной изоляции обычных блокировок не хватает — нужно блокировать не строки, а условия (предикаты). Такие *предикатные* блокировки были предложены еще в 1976 году при работе над System R, но их практическая применимость ограничена достаточно простыми условиями, для которых понятно, конфликтуют ли два разных предиката. До реализации *с. 298*

предикатных блокировок в задуманном виде дело, насколько мне известно, не дошло ни в одной системе.

2.3. Уровни изоляции в PostgreSQL

Со временем на смену блокировочным протоколам управления транзакциями пришел *протокол изоляции на основе снимков* (Snapshot Isolation, SI). Его идея состоит в том, что каждая транзакция работает с согласованным снимком данных на определенный момент времени. В снимок попадают все актуальные изменения, зафиксированные до момента его создания.

Изоляция на основе снимков позволяет обходиться минимумом блокиро-
с. 268 вок. Фактически блокируется только повторное изменение одной и той же строки. Все остальные операции могут выполняться одновременно: пишущие транзакции никогда не блокируют читающие транзакции, а читающие вообще никогда никого не блокируют.

В PostgreSQL реализован *многоверсионный* вариант протокола SI. Многоверсионность подразумевает, что в СУБД в один момент времени могут сосуществовать несколько версий одной и той же строки. Это позволяет включать в снимок подходящую версию, а не обрывать транзакции, пытающиеся прочитать устаревшие данные.

За счет использования снимков данных изоляция в PostgreSQL отличается от той, что требует стандарт, и в целом она строже. Грязное чтение не допускается по определению. Формально в PostgreSQL можно указать уровень Read Uncommitted, но работать он будет точно так же, как Read Committed, поэтому дальше я вообще не буду говорить про этот уровень. Уровень Repeatable Read
с. 161 не допускает не только неповторяющегося, но и фантомного чтения (хотя и не обеспечивает полную изоляцию). Правда, на уровне Read Committed можно *в ряде случаев* потерять изменения.

	потерянные изменения	грязное чтение	неповторяющееся чтение	фантомное чтение	другие аномалии
Read Committed	да	–	да	да	да
Repeatable Read	–	–	–	–	да
Serializable	–	–	–	–	–

Перед тем как исследовать внутреннее устройство изоляции, давайте подробно рассмотрим каждый из трех уровней с точки зрения пользователя. с. 97

Для этого создадим таблицу счетов. У Алисы и Боба по 1000 ₽, но у Боба открыто два счета:

```
=> CREATE TABLE accounts(
  id integer PRIMARY KEY GENERATED BY DEFAULT AS IDENTITY,
  client text,
  amount numeric
);
=> INSERT INTO accounts VALUES
  (1, 'alice', 1000.00),
  (2, 'bob', 100.00),
  (3, 'bob', 900.00);
```

Read Committed

Отсутствие грязного чтения. Легко убедиться в том, что грязные данные прочитать невозможно. Начнем транзакцию. По умолчанию она использует уровень изоляции Read Committed[1]:

```
=> BEGIN;
=> SHOW transaction_isolation;
 transaction_isolation
-----------------------
 read committed
(1 row)
```

Говоря точнее, умолчательный уровень задается параметром, который при необходимости можно изменить:

```
=> SHOW default_transaction_isolation;
 default_transaction_isolation
-------------------------------
 read committed
(1 row)
```

Итак, в открытой транзакции снимаем средства со счета, но не фиксируем изменения. Свои собственные изменения транзакция всегда видит:

[1] postgrespro.ru/docs/postgresql/18/transaction-iso#XACT-READ-COMMITTED.

```
=> UPDATE accounts SET amount = amount - 200 WHERE id = 1;
=> SELECT * FROM accounts WHERE client = 'alice';
 id | client | amount
----+--------+--------
  1 | alice  | 800.00
(1 row)
```

Во втором сеансе начинаем еще одну транзакцию с тем же уровнем Read Committed:

```
=> BEGIN;
=> SELECT * FROM accounts WHERE client = 'alice';
 id | client | amount
----+--------+---------
  1 | alice  | 1000.00
(1 row)
```

Как и ожидалось, другая транзакция не видит незафиксированные изменения — грязное чтение не допускается.

Неповторяющееся чтение. Пусть теперь первая транзакция зафиксирует изменения, а вторая повторно выполнит тот же самый запрос:

```
=> COMMIT;
```

```
=> SELECT * FROM accounts WHERE client = 'alice';
 id | client | amount
----+--------+--------
  1 | alice  | 800.00
(1 row)
=> COMMIT;
```

Запрос получает уже новые данные — это и есть аномалия *неповторяющегося чтения*, которая допускается на уровне Read Committed.

Практический вывод: в транзакции нельзя принимать решения на основании данных, прочитанных предыдущим оператором, ведь за время между выполнением операторов все может измениться. Вот пример, вариации которого встречаются в прикладном коде так часто, что он является классическим антипаттерном:

```
IF (SELECT amount FROM accounts WHERE id = 1) >= 1000 THEN
  UPDATE accounts SET amount = amount - 1000 WHERE id = 1;
END IF;
```

За время, которое проходит между проверкой и обновлением, другие транзакции могут как угодно изменить состояние счета, так что такая «проверка» ни от чего не спасает. Удобно представлять себе, что между операторами одной транзакции могут «вклиниться» произвольные операторы других транзакций, например вот так:

```
IF (SELECT amount FROM accounts WHERE id = 1) >= 1000 THEN

    UPDATE accounts SET amount = amount - 200 WHERE id = 1;
    COMMIT;

  UPDATE accounts SET amount = amount - 1000 WHERE id = 1;
END IF;
```

Если, переставив операторы, можно все испортить, значит, код написан некорректно. Не стоит обманывать себя, что с таким стечением обстоятельств мы не столкнемся: если неприятность может случиться, она произойдет обязательно. А вот воспроизводить и, следовательно, исправлять такие ошибки очень сложно.

Как написать код корректно? Есть несколько возможностей:

- Заменить процедурный код декларативным.

  Например, в данном случае проверка легко превращается в ограничение целостности:

  ```
  ALTER TABLE accounts ADD CHECK amount >= 0;
  ```

  Теперь никакие проверки в коде не нужны: достаточно просто выполнить действие и при необходимости обработать исключение, которое возникнет в случае попытки нарушения целостности.

- Использовать один оператор SQL.

  Проблемы с согласованностью возникают из-за того, что в промежутке между операторами может завершиться другая транзакция, и видимые данные изменятся. Если оператор один, то и промежутков никаких нет.

  В PostgreSQL достаточно средств, чтобы одним SQL-оператором решать сложные задачи. Отмечу общие табличные выражения (CTE), в которых в том числе можно использовать операторы `INSERT`, `UPDATE`, `DELETE`, `MERGE`, а также оператор `INSERT ON CONFLICT`, который атомарно реализует логику «вставить, а если строка уже есть, то обновить».

v. 15

- Задействовать пользовательские блокировки.

  Последнее средство — вручную установить исключительную блокировку на все нужные строки (`SELECT FOR UPDATE`) или вообще на всю таблицу (`LOCK TABLE`). Это всегда работает, но сводит на нет преимущества многоверсионности: вместо одновременного выполнения часть операций будет выполняться последовательно.

с. 268
с. 261

Несогласованное чтение. Однако не все так просто. Реализация PostgreSQL такова, что допускает другие, менее известные аномалии, которые не регламентируются стандартом.

Допустим, первая транзакция начала перевод средств с одного счета Боба на другой:

```
=> BEGIN;
=> UPDATE accounts SET amount = amount - 100 WHERE id = 2;
```

В это время другая транзакция подсчитывает баланс Боба, причем подсчет выполняется в цикле по всем счетам Боба. Фактически транзакция начинает с первого счета (и, разумеется, видит прежнее состояние):

```
=> BEGIN;
=> SELECT amount FROM accounts WHERE id = 2;
 amount
--------
 100.00
(1 row)
```

В этот момент первая транзакция успешно завершается:

```
=> UPDATE accounts SET amount = amount + 100 WHERE id = 3;
=> COMMIT;
```

А другая читает состояние второго счета (и видит уже новое значение):

```
=> SELECT amount FROM accounts WHERE id = 3;
 amount
---------
 1000.00
(1 row)
=> COMMIT;
```

В итоге вторая транзакция получила в сумме 1100 ₽, то есть прочитала некорректные данные. Такая аномалия называется *несогласованным чтением* (read skew).

Как избежать этой аномалии, оставаясь на уровне Read Committed? Конечно, использовать один оператор. Например, так:

```
SELECT sum(amount) FROM accounts WHERE client = 'bob';
```

До сих пор я утверждал, что видимость данных может поменяться только между операторами, но так ли это очевидно? А если запрос выполняется долго, может ли он увидеть часть данных в одном состоянии, а часть — уже в другом?

Проверим. Удобный способ для этого — вставить в оператор искусственную задержку, вызвав функцию pg_sleep. Первая строка будет прочитана сразу, а вторая — через две секунды:

```
=> SELECT amount, pg_sleep(2) -- 2 секунды
FROM accounts WHERE client = 'bob';
```

Пока эта конструкция выполняется, в другой транзакции переводим средства обратно:

```
=> BEGIN;
=> UPDATE accounts SET amount = amount + 100 WHERE id = 2;
=> UPDATE accounts SET amount = amount - 100 WHERE id = 3;
=> COMMIT;
```

Результат показывает, что оператор видит данные в таком состоянии, в каком они находились на момент начала его выполнения, что, безусловно, правильно:

```
 amount  | pg_sleep
---------+----------
    0.00 |
 1000.00 |
(2 rows)
```

Но и тут не все так просто. Если в запросе вызывается *изменчивая* функция (с категорией изменчивости VOLATILE) и в этой функции выполняется другой запрос, то этот вложенный запрос будет видеть данные, не согласованные с данными основного запроса.

Проверим состояние счетов Боба, используя функцию:

```
=> CREATE FUNCTION get_amount(id integer) RETURNS numeric
AS $$
  SELECT amount FROM accounts a WHERE a.id = get_amount.id;
$$ VOLATILE LANGUAGE sql;
=> SELECT get_amount(id), pg_sleep(2)
FROM accounts WHERE client = 'bob';
```

И снова переведем деньги между счетами, пока запрос с задержкой выполняется:

```
=> BEGIN;
=> UPDATE accounts SET amount = amount + 100 WHERE id = 2;
=> UPDATE accounts SET amount = amount - 100 WHERE id = 3;
=> COMMIT;
```

В этом случае получим несогласованные данные — 100 ₽ пропали:

```
 get_amount | pg_sleep
------------+----------
     100.00 |
     800.00 |
(2 rows)
```

Подчеркну, что такой эффект возможен только на уровне изоляции Read Committed и только с категорией изменчивости VOLATILE. Беда в том, что по умолчанию используется именно этот уровень изоляции и именно эта

категория изменчивости, так что остается признать — грабли лежат очень удачно.

Несогласованное чтение вместо потерянного обновления. Аномалию несогласованного чтения в рамках одного оператора можно — несколько неожиданным образом — получить и при обновлении.

Посмотрим, что происходит при попытке изменения одной и той же строки двумя транзакциями. Сейчас у Боба 1000 ₽ на двух счетах:

```
=> SELECT * FROM accounts WHERE client = 'bob';
 id | client | amount
----+--------+--------
  2 | bob    | 200.00
  3 | bob    | 800.00
(2 rows)
```

Начинаем транзакцию, которая уменьшает баланс Боба:

```
=> BEGIN;
=> UPDATE accounts SET amount = amount - 100 WHERE id = 3;
```

В это же время другая транзакция начисляет проценты на все счета с общим балансом, равным или превышающим 1000 ₽:

```
=> UPDATE accounts SET amount = amount * 1.01
WHERE client IN (
  SELECT client
  FROM accounts
  GROUP BY client
  HAVING sum(amount) >= 1000
);
```

Выполнение оператора UPDATE состоит из двух частей. Сначала фактически выполняется оператор SELECT, который отбирает для обновления строки, соответствующие условию. Поскольку изменение первой транзакции не зафиксировано, вторая транзакция не может его видеть, и оно никак не влияет на выбор строк для начисления процентов. Таким образом, счета Боба попадают под условие, и после выполнения обновления его баланс должен увеличиться на 10 ₽.

На втором этапе выполнения выбранные строки обновляются одна за другой. Вторая транзакция вынуждена подождать, поскольку строка id = 3 в настоящий момент изменяется первой транзакцией и поэтому заблокирована.

Между тем первая транзакция фиксирует изменения:

```
=> COMMIT;
=> SELECT * FROM accounts WHERE client = 'bob';
 id | client |  amount
----+--------+----------
  2 | bob    | 202.0000
  3 | bob    | 707.0000
(2 rows)
```

Да, с одной стороны, команда UPDATE не должна видеть изменений второй транзакции. Но с другой — она не должна потерять изменения, зафиксированные во второй транзакции.

с. 279 После снятия блокировки оператор UPDATE *перечитывает* строку, которую пытается обновить (но только ее одну!). В результате получается, что Бобу начислено 9 ₽, исходя из суммы 900 ₽. Но если бы у Боба было 900 ₽, его счета вообще не должны были попасть в выборку.

Таким образом, транзакция прочитала некорректные данные: часть строк — на один момент времени, часть — на другой. Взамен потерянного обновления мы снова получаем аномалию несогласованного чтения.

Потерянное обновление. Впрочем, хитрость с перечитыванием заблокированной строки не спасает от потери изменений, если обновление происходит не в одном операторе SQL.

с. 50 Вот пример, который уже был. Приложение читает и запоминает (вне базы данных) текущий баланс счета Алисы:

```
=> BEGIN;
=> SELECT amount FROM accounts WHERE id = 1;
 amount
--------
 800.00
(1 row)
```

В это время другая транзакция действует так же:

```
=> BEGIN;
=> SELECT amount FROM accounts WHERE id = 1;
 amount
--------
 800.00
(1 row)
```

Первая транзакция увеличивает запомненное ранее значение на 100 ₽ и записывает в базу:

```
=> UPDATE accounts SET amount = 800.00 + 100 WHERE id = 1
RETURNING amount;
 amount
--------
 900.00
(1 row)
UPDATE 1
=> COMMIT;
```

И вторая транзакция тоже:

```
=> UPDATE accounts SET amount = 800.00 + 100 WHERE id = 1
RETURNING amount;
 amount
--------
 900.00
(1 row)
UPDATE 1
=> COMMIT;
```

К сожалению, Алиса недосчиталась 100 ₽. СУБД ничего не знает о том, что запомненное значение 800 ₽ как-то связано с `accounts.amount`, и допускает аномалию потерянного изменения. На уровне изоляции Read Committed такой код некорректен.

Repeatable Read

Отсутствие неповторяющегося и фантомного чтения. Само название уровня изоляции Repeatable Read[1] говорит о повторяемости чтения. Проверим это,

[1] postgrespro.ru/docs/postgresql/18/transaction-iso#XACT-REPEATABLE-READ.

а заодно убедимся и в отсутствии фантомов. Для этого в первой транзакции вернем счета Боба в прежнее состояние и создадим новый счет для Чарли:

```
=> BEGIN;
=> UPDATE accounts SET amount = 200.00 WHERE id = 2;
=> UPDATE accounts SET amount = 800.00 WHERE id = 3;
=> INSERT INTO accounts VALUES
  (4, 'charlie', 100.00);
=> SELECT * FROM accounts ORDER BY id;
 id | client  | amount
----+---------+--------
  1 | alice   | 900.00
  2 | bob     | 200.00
  3 | bob     | 800.00
  4 | charlie | 100.00
(4 rows)
```

Во втором сеансе начнем транзакцию с уровнем Repeatable Read, указав его в команде BEGIN (уровень первой транзакции не важен):

```
=> BEGIN ISOLATION LEVEL REPEATABLE READ;
=> SELECT * FROM accounts ORDER BY id;
 id | client |  amount
----+--------+----------
  1 | alice  |   900.00
  2 | bob    | 202.0000
  3 | bob    | 707.0000
(3 rows)
```

Теперь первая транзакция фиксирует изменения, а вторая повторно выполняет тот же самый запрос:

```
=> COMMIT;
```

```
=> SELECT * FROM accounts ORDER BY id;
 id | client |  amount
----+--------+----------
  1 | alice  |   900.00
  2 | bob    | 202.0000
  3 | bob    | 707.0000
(3 rows)
=> COMMIT;
```

Вторая транзакция продолжает видеть ровно те же данные, что и в начале: не видно ни изменений в существующих строках, ни новых строк. На таком уровне можно не беспокоиться о том, что между двумя операторами что-то поменяется.

Ошибка сериализации вместо потерянных изменений. Как мы уже видели, на уровне изоляции Read Committed при обновлении одной и той же строки двумя транзакциями может возникнуть аномалия несогласованного чтения. Это происходит из-за того, что ожидающая транзакция перечитывает заблокированную строку и видит ее на один момент времени, а остальные строки выборки — на другой. *с. 61*

На уровне Repeatable Read такая аномалия не допускается, но, если она все-таки возникает, сделать уже ничего нельзя — поэтому транзакция обрывается с ошибкой сериализации. Проверим, повторив тот же сценарий с процентами:

```
=> SELECT * FROM accounts WHERE client = 'bob';
 id | client | amount
----+--------+--------
  2 | bob    | 200.00
  3 | bob    | 800.00
(2 rows)

=> BEGIN;
=> UPDATE accounts SET amount = amount - 100.00 WHERE id = 3;
```

```
=> BEGIN ISOLATION LEVEL REPEATABLE READ;
=> UPDATE accounts SET amount = amount * 1.01
WHERE client IN (
  SELECT client
  FROM accounts
  GROUP BY client
  HAVING sum(amount) >= 1000
);
```

```
=> COMMIT;
```

```
    ERROR:  could not serialize access due to concurrent update
    => ROLLBACK;
```

Данные остались согласованными:

```
=> SELECT * FROM accounts WHERE client = 'bob';
 id | client | amount
----+--------+--------
  2 | bob    | 200.00
  3 | bob    | 700.00
(2 rows)
```

Такая же ошибка будет и в случае любого другого конкурентного изменения строки, даже если оно не затрагивает интересующие нас столбцы.

Можно проверить, что и сценарий с приложениями, обновляющими баланс на основе запомненного значения, приводит к той же ошибке:

```
=> BEGIN ISOLATION LEVEL REPEATABLE READ;
=> SELECT amount FROM accounts WHERE id = 1;
 amount
--------
 900.00
(1 row)
```

```
    => BEGIN ISOLATION LEVEL REPEATABLE READ;
    => SELECT amount FROM accounts WHERE id = 1;
     amount
    --------
     900.00
    (1 row)
```

```
=> UPDATE accounts SET amount = 900.00 + 100.00 WHERE id = 1
RETURNING amount;
 amount
---------
 1000.00
(1 row)
UPDATE 1
=> COMMIT;
```

```
=> UPDATE accounts SET amount = 900.00 + 100.00 WHERE id = 1
RETURNING amount;
ERROR:  could not serialize access due to concurrent update
=> ROLLBACK;
```

Практический вывод: если приложение использует уровень изоляции Repeatable Read для пишущих транзакций, оно должно быть готово повторять транзакции, завершившиеся ошибкой сериализации. Для только читающих транзакций такой исход невозможен.

Несогласованная запись. Итак, в PostgreSQL на уровне изоляции Repeatable Read предотвращаются все аномалии, описанные в стандарте. Но не все вообще — науке до сих пор неизвестно, сколько их существует. Однако доказан важный факт: изоляция на основе снимков оставляет возможными *ровно две* аномалии, сколько бы их ни было всего.

Первая из этих аномалий — *несогласованная запись* (write skew).

Пусть действует такое правило согласованности: *допускаются отрицательные суммы на отдельных счетах клиента, если общая сумма на всех счетах этого клиента остается неотрицательной*.

Первая транзакция получает сумму на счетах Боба:

```
=> BEGIN ISOLATION LEVEL REPEATABLE READ;
=> SELECT sum(amount) FROM accounts WHERE client = 'bob';
  sum
--------
 900.00
(1 row)
```

Вторая транзакция получает ту же сумму:

```
=> BEGIN ISOLATION LEVEL REPEATABLE READ;
=> SELECT sum(amount) FROM accounts WHERE client = 'bob';
  sum
--------
 900.00
(1 row)
```

Первая транзакция справедливо полагает, что сумму одного из счетов можно уменьшить на 600 ₽:

```
=> UPDATE accounts SET amount = amount - 600.00 WHERE id = 2;
```

И вторая транзакция приходит к такому же выводу. Но уменьшает другой счет:

```
    => UPDATE accounts SET amount = amount - 600.00 WHERE id = 3;
    => COMMIT;
```

```
=> COMMIT;
=> SELECT * FROM accounts WHERE client = 'bob';
 id | client | amount
----+--------+---------
  2 | bob    | -400.00
  3 | bob    |  100.00
(2 rows)
```

У нас получилось увести баланс Боба в минус, хотя поодиночке каждая из транзакций отработала бы корректно.

Аномалия только читающей транзакции. Аномалия *только читающей транзакции* — вторая и последняя из аномалий, возможных на уровне Repeatable Read. Чтобы продемонстрировать ее, потребуются три транзакции, две из которых будут изменять данные, а третья — только читать.

Но сначала восстановим состояние счетов Боба:

```
=> UPDATE accounts SET amount = 900.00 WHERE id = 2;
=> SELECT * FROM accounts WHERE client = 'bob';
```

```
 id | client | amount
----+--------+--------
  3 | bob    | 100.00
  2 | bob    | 900.00
(2 rows)
```

Первая транзакция начисляет Бобу проценты на сумму средств на всех счетах. Проценты зачисляются на один из его счетов:

```
=> BEGIN ISOLATION LEVEL REPEATABLE READ; -- 1
=> UPDATE accounts SET amount = amount + (
  SELECT sum(amount) FROM accounts WHERE client = 'bob'
) * 0.01
WHERE id = 2;
```

Затем другая транзакция снимает деньги с другого счета Боба и фиксирует свои изменения:

```
=> BEGIN ISOLATION LEVEL REPEATABLE READ; -- 2
=> UPDATE accounts SET amount = amount - 100.00 WHERE id = 3;
=> COMMIT;
```

Если в этот момент первая транзакция будет зафиксирована, никакой аномалии не возникнет: мы могли бы считать, что сначала выполнена первая транзакция, а затем вторая (но не наоборот, потому что первая транзакция увидела состояние счета id = 3 до того, как этот счет был изменен второй транзакцией).

Но представим, что в этот момент третья (только читающая) транзакция запрашивает суммы на счетах Боба. Она может видеть изменения второй транзакции (которая уже была зафиксирована), но не первой (которая еще не была зафиксирована):

```
=> SELECT * FROM accounts WHERE client = 'bob'; -- 3
 id | client | amount
----+--------+--------
  2 | bob    | 900.00
  3 | bob    |   0.00
(2 rows)
```

С другой стороны, мы уже установили выше, что вторую транзакцию следует считать начавшейся после первой. Последствия аномалии становятся понятными после завершения первой транзакции:

```
=> COMMIT;
```

У Боба на счетах всего 900 ₽ (что показала только читающая транзакция), а проценты оказались начисленными на все 1000 ₽:

```
=> SELECT * FROM accounts WHERE client = 'bob';
 id | client |  amount
----+--------+----------
  2 | bob    | 910.0000
  3 | bob    |     0.00
(2 rows)
```

Serializable

На уровне Serializable[1] предотвращаются все возможные аномалии. Фактически Serializable реализован как надстройка над изоляцией на основе снимков данных. Те аномалии, которые не возникают при Repeatable Read (такие как грязное, неповторяемое, фантомное чтение), не возникают и на уровне Serializable. А те две аномалии, которые возникают (несогласованная запись и аномалия только читающей транзакции), специальным образом обнаруживаются, и в этом случае транзакция обрывается: возникает уже знакомая нам ошибка сериализации.

с. 67 **Отсутствие аномалий.** Можно убедиться, что сценарий с аномалией несогласованной записи приведет к ошибке сериализации:

```
=> BEGIN ISOLATION LEVEL SERIALIZABLE;
=> SELECT sum(amount) FROM accounts WHERE client = 'bob';
   sum
----------
 910.0000
(1 row)
```

```
=> BEGIN ISOLATION LEVEL SERIALIZABLE;
=> SELECT sum(amount) FROM accounts WHERE client = 'bob';
   sum
----------
 910.0000
(1 row)
```

```
=> UPDATE accounts SET amount = amount - 600.00 WHERE id = 2;
```

[1] postgrespro.ru/docs/postgresql/18/transaction-iso#XACT-SERIALIZABLE.

```
    => UPDATE accounts SET amount = amount - 600.00 WHERE id = 3;
    => COMMIT;
    COMMIT
```

```
=> COMMIT;
ERROR:  could not serialize access due to read/write dependencies among
transactions
DETAIL:  Reason code: Canceled on identification as a pivot, during
commit attempt.
HINT:  The transaction might succeed if retried.
```

К такой же ошибке приведет и сценарий аномалии только читающей транзакции.

Откладывание читающей транзакции. Чтобы только читающая транзакция не приводила к аномалии и не могла пострадать от нее, PostgreSQL предлагает интересный механизм: такая транзакция может быть отложена до тех пор, пока ее выполнение не станет безопасным. Это единственный случай, когда оператор SELECT может быть заблокирован обновлениями строк.

Посмотрим на примере сценария аномалии только читающей транзакции:

```
=> UPDATE accounts SET amount = 900.00 WHERE id = 2;
=> UPDATE accounts SET amount = 100.00 WHERE id = 3;
=> SELECT * FROM accounts WHERE client = 'bob' ORDER BY id;
 id | client | amount
----+--------+--------
  2 | bob    | 900.00
  3 | bob    | 100.00
(2 rows)
```

```
=> BEGIN ISOLATION LEVEL SERIALIZABLE; -- 1
=> UPDATE accounts SET amount = amount + (
  SELECT sum(amount) FROM accounts WHERE client = 'bob'
) * 0.01
WHERE id = 2;
```

```
=> BEGIN ISOLATION LEVEL SERIALIZABLE; -- 2
=> UPDATE accounts SET amount = amount - 100.00 WHERE id = 3;
=> COMMIT;
```

Третью транзакцию явно объявляем только читающий (READ ONLY) и откладываемой (DEFERRABLE):

```
=> BEGIN ISOLATION LEVEL SERIALIZABLE READ ONLY DEFERRABLE; -- 3
=> SELECT * FROM accounts WHERE client = 'alice';
```

При попытке выполнить запрос транзакция блокируется, потому что иначе она приведет к аномалии.

И только после того, как первая транзакция будет зафиксирована, третья продолжит выполнение:

```
=> COMMIT;
```

```
 id | client | amount
----+--------+---------
  1 | alice  | 1000.00
(1 row)
=> SELECT * FROM accounts WHERE client = 'bob';
 id | client |  amount
----+--------+----------
  2 | bob    | 910.0000
  3 | bob    |     0.00
(2 rows)
=> COMMIT;
```

Таким образом, приложение, использующее уровень изоляции Serializable, должно повторять транзакции, завершившиеся ошибкой сериализации. (Точно так же следует поступать и на уровне Repeatable Read, если не ограничиваться только читающими транзакциями.)

Уровень Serializable дает простоту программирования, но цена за нее — накладные расходы на обнаружение возможных аномалий и обрыв некоторой доли транзакций. Снизить накладные расходы можно, явно обозначая только читающие транзакции как READ ONLY. Но главный вопрос, конечно, в том,

насколько велика доля оборванных транзакций, ведь их придется выполнять повторно. Если бы обрывались только те транзакции, которые действительно несовместимо пересекаются по данным, все было бы неплохо. Но такая реализация неизбежно оказалась бы слишком ресурсоемкой, поскольку пришлось бы отслеживать операции с каждой строкой.

В действительности реализация такова, что допускает ложноположительные срабатывания: будут обрываться и некоторые совершенно нормальные транзакции, которым «просто не повезло». Везение зависит от многих причин, например от наличия подходящих индексов или доступного объема оперативной памяти, и поведение сложно предсказать заранее. *с. 298*

Если используется уровень изоляции Serializable, он должен применяться для всех транзакций приложения. При смешении транзакций разного уровня изоляции уровень Serializable будет (без всяких предупреждений) вести себя как Repeatable Read. Поэтому при использовании Serializable имеет смысл изменить значение параметра *default_transaction_isolation*, хотя, конечно, это не помешает указать неправильный уровень явно. read committed

Есть и другие ограничения реализации, например запросы на уровне Serializable не будут работать на репликах. И хотя работа над улучшением функциональности не прекращается, имеющиеся ограничения и накладные расходы снижают привлекательность такого уровня изоляции. v. 12

2.4. Какой уровень изоляции использовать?

Уровень изоляции Read Committed используется в PostgreSQL по умолчанию, и, по всей видимости, именно этот уровень применяется в абсолютном большинстве приложений. Он удобен тем, что на нем обрыв транзакции возможен только в случае сбоя, но для предотвращения несогласованности обрыв не применяется. Иными словами, ошибка сериализации возникнуть не может, и о повторении транзакций заботиться не надо.

Обратная сторона медали — большое число возможных аномалий, подробно рассмотренных выше. Разработчик вынужден постоянно иметь их в виду и писать код так, чтобы не допускать их появления. Если не получается

сформулировать нужные действия в одном SQL-операторе, приходится прибегать к явной установке блокировок. Самое неприятное то, что код сложно тестировать на наличие ошибок, связанных с получением несогласованных данных, а сами ошибки могут возникать непредсказуемым и невоспроизводимым образом и поэтому сложны в исправлении.

Уровень изоляции Repeatable Read снимает часть проблем несогласованности, но, увы, не все. Поэтому приходится не только помнить об оставшихся аномалиях, но и изменять приложение так, чтобы оно корректно обрабатывало ошибки сериализации. Это, конечно, неудобно. Но для только читающих транзакций этот уровень прекрасно дополняет Read Committed и полезен, например, для построения отчетов, использующих несколько SQL-запросов.

Наконец, уровень Serializable позволяет вообще не заботиться о несогласованности, что значительно упрощает написание кода. Единственное, что требуется от приложения, — уметь повторять любую транзакцию при получении ошибки сериализации. Но доля прерываемых транзакций и дополнительные накладные расходы могут существенно снизить пропускную способность. Также следует учитывать, что уровень Serializable не применим на репликах и что его нельзя смешивать с другими уровнями изоляции.

3

Страницы и версии строк

3.1. Структура страниц

Каждая страница имеет внутреннюю разметку и, как правило, содержит следующие разделы [1]:

- заголовок;
- массив указателей на версии строк;
- свободное пространство;
- версии строк;
- специальную область.

Заголовок страницы

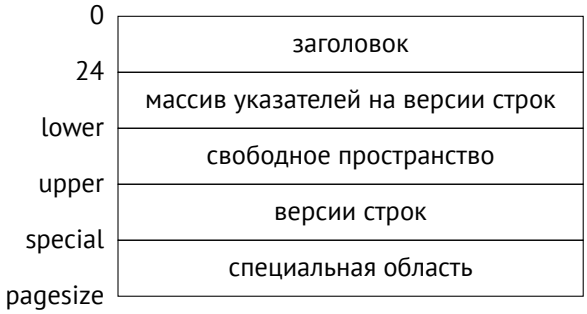
Заголовок страницы располагается в младших адресах и имеет фиксированный размер. Он хранит различную информацию о странице, такую как контрольная сумма, а также размеры всех остальных областей. *с. 127*

Размеры легко получить с помощью расширения `pageinspect`[2]. Заглянем в самую первую страницу таблицы (нумерация начинается с нуля):

[1] postgrespro.ru/docs/postgresql/18/storage-page-layout; include/storage/bufpage.h.

[2] postgrespro.ru/docs/postgresql/18/pageinspect.

```
=> CREATE EXTENSION pageinspect;
=> SELECT lower, upper, special, pagesize
FROM page_header(get_raw_page('accounts',0));
 lower | upper | special | pagesize
-------+-------+---------+----------
   152 |  6904 |    8192 |     8192
(1 row)
```



Специальная область

Специальная область расположена в противоположном конце страницы, в старших адресах. Она используется некоторыми типами индексов для хранения вспомогательной информации. В остальных случаях, в том числе в табличных страницах, эта область имеет нулевой размер.

В целом индексные страницы устроены более разнообразно, и их содержимое зависит от конкретного типа индекса. И даже у одного типа индекса бывают разные виды страниц: например, у B-дерева есть нулевая страница
с. 529 метаданных с особой структурой и «обычные» страницы, организованные как табличные.

Версии строк

Перед специальной областью располагаются *строки* (rows) — те самые данные, которые хранятся в базе, с добавлением некоторой служебной информации.

В случае таблиц мы говорим не просто о строках, а о *версиях строк* (row versions, tuples), поскольку многоверсионность предполагает существование нескольких версий одной и той же строки. На индексы многоверсионность не распространяется; вместо этого индексы ссылаются на все возможные табличные версии строк, среди которых по правилам видимости выбираются подходящие.

> Термин tuple заимствован из реляционной теории и переводится как *кортеж*. Это еще одно наследие академического прошлого PostgreSQL. Чтобы не смешивать теорию с устройством СУБД, я буду использовать перевод *версия строки* (иногда, если это не вызывает неоднозначностей, заменяя более коротким словом *строка*).

Указатели на версии строк

Массив указателей на версии строк служит оглавлением страницы. Он располагается сразу за заголовком.

Индексные строки должны как-то ссылаться на версии строк в таблице. Для этого используются шестибайтные *идентификаторы версий строк* (tuple id, tid). Идентификатор состоит из номера страницы в файле основного слоя и должен еще содержать какое-то указание на версию строки в этой странице. *с. 29*

В качестве такого указания можно было бы использовать смещение относительно начала страницы. Но тогда версию строки нельзя было бы перемещать внутри страницы, не сломав ссылки из индексов. А это привело бы к фрагментации места внутри страниц и другим неприятным последствиям.

Поэтому используется косвенная адресация: идентификатор версии ссылается на номер указателя, а уже указатель — на текущую позицию версии строки в странице. При перемещении версии строки ее идентификатор не меняется; достаточно изменить указатель, который находится на той же странице.

Каждый указатель занимает ровно четыре байта и содержит:

- смещение версии строки относительно начала страницы;
- длину версии строки;
- несколько битов, определяющих статус версии строки.

Свободное место

Между указателями и версиями строк может оставаться *свободное место* (ко-
с. 31 торое и отмечено в карте свободного пространства). Никакой фрагментации внутри страницы не бывает, все свободное место всегда представлено одним фрагментом[1].

3.2. Структура версий строк

Версия строки состоит из заголовка, за которым следуют собственно данные. Заголовок версии содержит множество полей, среди которых:

xmin, xmax — номера транзакций, которые отличают данную версию от других версий той же строки;

infomask — ряд информационных битов, определяющих свойства версии;

ctid — ссылка на следующую, более новую версию той же строки;

битовая карта неопределенных значений — массив битов, отмечающих столбцы, которые допускают неопределенные значения (`NULL`).

В результате заголовок получается довольно большой — минимум 23 байта на каждую версию строки, а обычно больше из-за карты неопределенных значений и из-за обязательного выравнивания начала данных. Для «узкой» таблицы объем служебных данных вполне может превышать объем полезной информации.

Формат данных на диске полностью совпадает с представлением данных в оперативной памяти. Страница вместе с версиями строк читается в буферный кеш «как есть», без каких бы то ни было преобразований. Поэтому файлы данных с одной платформы оказываются несовместимыми с другими платформами[2].

[1] backend/storage/page/bufpage.c, функция PageRepairFragmentation.
[2] include/access/htup_details.h.

Одна из причин несовместимости — порядок следования байтов. Например, в архитектуре x86 принят порядок от младших разрядов к старшим (little-endian), z/Architecture использует обратный порядок (big-endian), а в ARM порядок переключаемый.

Несовместимость вызывается также выравниванием данных по границам машинных слов, которое требуется многим архитектурам. Например, в 32-битной системе архитектуры x86 целые числа (тип `integer`, занимает четыре байта) будут выровнены по границе четырехбайтных слов, как и числа с плавающей точкой двойной точности (тип `double precision`, восемь байт). А в 64-битной системе значения `double` будут выровнены по границе восьмибайтных слов.

Из-за выравнивания размер табличной строки зависит от порядка расположения полей. Обычно этот эффект не сильно заметен, но в некоторых случаях он может привести к существенному увеличению размера. Вот пример:

```
=> CREATE TABLE padding(
  b1 boolean,
  f1 float,
  b2 boolean,
  f2 float
);
=> INSERT INTO padding VALUES (true,1.0,false,2.0);
=> SELECT lp_len FROM heap_page_items(get_raw_page('padding', 0));
 lp_len
--------
     56
(1 row)
```

Я использовал функцию `heap_page_items` расширения `pageinspect`, которая позволяет получить информацию об указателях и версиях строк.

> Словом heap (куча) в PostgreSQL обозначаются таблицы. Это еще одно не вполне очевидное употребление термина, намекающее на сходство механизма выделения места под версии строк с динамическим распределением оперативной памяти. Конечно, некоторую аналогию усмотреть можно, но для управления таблицей используются совсем другие алгоритмы. Можно считать, что слово употребляется в смысле «все свалено в кучу», подчеркивая отличие от упорядоченных индексов.

Строка занимает 56 байт. Из них 24 байта уходит на заголовок, столбцы типа `float` занимают по 8 байт, `boolean` — по 1 байту. В сумме имеем 42, а 14 байт

пропадают из-за выравнивания float по границе восьмибайтных слов. Перестроив таблицу, можно использовать место более эффективно:

```
=> DROP TABLE padding;
=> CREATE TABLE padding(
  f1 float,
  f2 float,
  b1 boolean,
  b2 boolean
);
```

Надо, однако, учитывать, что выравниваются и версии строк внутри страницы. Как и предполагалось, строки стали занимать 42 байта, но 6 байт все-таки пропадают между строками:

```
=> INSERT INTO padding VALUES (1.0,2.0,true,false);
=> SELECT lp_len, 8192 - lp_off AS allocated, lp_off
FROM heap_page_items(get_raw_page('padding', 0));
 lp_len | allocated | lp_off
--------+-----------+--------
     42 |        48 |   8144
(1 row)
```

Еще одна возможная микрооптимизация — перенести в начало таблицы все столбцы фиксированного размера, не допускающие неопределенных значений. Доступ к таким полям будет более эффективным благодаря возможности закешировать смещение поля от начала версии строки[1].

3.3. Выполнение операций над версиями строк

Чтобы разные версии одной и той же строки можно было различить, каждая из версий имеет две отметки, определяющие ее «время действия», — xmin и xmax. Но используется не время как таковое, а постоянно увеличивающийся
с. 134 счетчик номеров транзакций.

Когда строка создается, значение xmin устанавливается равным номеру транзакции, выполнившей команду INSERT.

[1] backend/access/common/heaptuple.c, функция heap_deform_tuple.

Когда строка удаляется, значению xmax текущей версии присваивается номер транзакции, выполнившей команду DELETE.

Команду UPDATE можно в некотором приближении рассматривать как две операции: DELETE и INSERT. В текущей версии строки устанавливается значение xmax, равное номеру транзакции, выполнившей UPDATE. Затем создается новая версия той же строки; значение xmin у нее совпадает со значением xmax предыдущей версии. Точно так же и команда MERGE «распадается» на элементарные вставки и удаления. v. 15

Дальше я подробно покажу, как выполняются разные операции с версиями строк на низком уровне[1].

Для экспериментов понадобится таблица с двумя столбцами и индексом по одному из них:

```
=> CREATE TABLE t(
  id integer GENERATED ALWAYS AS IDENTITY,
  s text
);
=> CREATE INDEX ON t(s);
```

Вставка

Вставляем одну строку, предварительно начав транзакцию:

```
=> BEGIN;
=> INSERT INTO t(s) VALUES ('FOO');
```

Вот номер нашей текущей транзакции:

```
=> --  txid_current() до v.13
SELECT pg_current_xact_id();
 pg_current_xact_id
--------------------
                795
(1 row)
```

[1] backend/access/transam/README.

> Для понятия транзакции в PostgreSQL часто используется сокращение xact, которое можно найти и в именах функций, и в коде. Соответственно, номер транзакции обозначается как xact id, txid или даже просто xid. Эти названия еще не раз нам встретятся.

Заглянем в содержимое страницы. Функция `heap_page_items` дает нам всю необходимую информацию, но показывает данные «как есть», в формате, сложном для восприятия:

```
=> SELECT *
FROM heap_page_items(get_raw_page('t',0))
\gx
-[ RECORD 1 ]-------------------
lp          | 1
lp_off      | 8160
lp_flags    | 1
lp_len      | 32
t_xmin      | 795
t_xmax      | 0
t_field3    | 0
t_ctid      | (0,1)
t_infomask2 | 2
t_infomask  | 2050
t_hoff      | 24
t_bits      |
t_oid       |
t_data      | \x0100000009464f4f
```

Чтобы разобраться, оставим только часть информации и расшифруем некоторые столбцы:

```
=> SELECT '(0,'||lp||')' AS ctid,
     CASE lp_flags
       WHEN 0 THEN 'unused'
       WHEN 1 THEN 'normal'
       WHEN 2 THEN 'redirect to '||lp_off
       WHEN 3 THEN 'dead'
     END AS state,
     t_xmin as xmin,
     t_xmax as xmax,
     (t_infomask & 256) > 0  AS xmin_committed,
     (t_infomask & 512) > 0  AS xmin_aborted,
     (t_infomask & 1024) > 0 AS xmax_committed,
     (t_infomask & 2048) > 0 AS xmax_aborted
FROM heap_page_items(get_raw_page('t',0))
\gx
```

```
-[ RECORD 1 ]--+-------
ctid           | (0,1)
state          | normal
xmin           | 795
xmax           | 0
xmin_committed | f
xmin_aborted   | f
xmax_committed | f
xmax_aborted   | t
```

Вот что здесь сделано:

- Указатель lp приведен к обычному виду идентификатора версии строки: (номер страницы, номер указателя).
- Расшифровано состояние указателя lp_flags. Здесь он имеет значение normal — это значит, что указатель действительно ссылается на версию строки.
- Из всех информационных битов пока выделены только две пары. Биты xmin_committed и xmin_aborted показывают, зафиксирована ли и отменена ли транзакция с номером xmin. Аналогичную информацию о транзакции xmax дают биты xmax_committed и xmax_aborted.

  В расширении pageinspect есть функция heap_tuple_infomask_flags, расшифровывающая все информационные биты, но я буду выводить только те, что нужны в данный момент, и в более компактном виде. v. 13

Итак, при вставке строки в табличной странице появился указатель с номером 1, ссылающийся на первую и пока единственную версию строки.

Поле xmin в версии строки заполнено номером текущей транзакции. Транзакция еще активна, поэтому биты xmin_committed и xmin_aborted не установлены.

Поле xmax заполнено фиктивным номером 0, поскольку данная версия строки не удалена и является актуальной. Транзакции не будут обращать внимания на этот номер, поскольку установлен бит xmax_aborted.

Может показаться странным, что бит оборванной транзакции установлен у транзакции, которой не было. Но с точки зрения изоляции это одно и то же: отмененная транзакция не оставляет следов и, следовательно, не существовала.

Такой запрос нам еще не раз понадобится, поэтому я оберну его в функцию. А заодно для компактности вывода уберу столбцы информационных битов, приписав статус транзакции к ее номеру:

```
=> CREATE FUNCTION heap_page(relname text, pageno integer)
RETURNS TABLE(ctid tid, state text, xmin text, xmax text)
AS $$
SELECT (pageno,lp)::text::tid AS ctid,
     CASE lp_flags
       WHEN 0 THEN 'unused'
       WHEN 1 THEN 'normal'
       WHEN 2 THEN 'redirect to '||lp_off
       WHEN 3 THEN 'dead'
     END AS state,
     t_xmin || CASE
       WHEN (t_infomask & 256) > 0 THEN ' c'
       WHEN (t_infomask & 512) > 0 THEN ' a'
       ELSE ''
     END AS xmin,
     t_xmax || CASE
       WHEN (t_infomask & 1024) > 0 THEN ' c'
       WHEN (t_infomask & 2048) > 0 THEN ' a'
       ELSE ''
     END AS xmax
FROM heap_page_items(get_raw_page(relname,pageno))
ORDER BY lp;
$$ LANGUAGE sql;
```

В таком виде уже значительно понятнее, что творится в заголовке версии:

```
=> SELECT * FROM heap_page('t',0);
 ctid  | state  | xmin | xmax
-------+--------+------+------
 (0,1) | normal | 795  | 0 a
(1 row)
```

Похожую, но существенно менее детальную информацию можно получить и из самой таблицы, используя псевдостолбцы `xmin` и `xmax`:

```
=> SELECT xmin, xmax, * FROM t;
 xmin | xmax | id |  s
------+------+----+-----
  795 |    0 |  1 | FOO
(1 row)
```

Фиксация

При успешном завершении транзакции нужно запомнить ее статус — отметить, что она *зафиксирована*. Для этого используется структура, называемая clog[1] (commit log). Это не таблица системного каталога, а специальные файлы в каталоге PGDATA/pg_xact.

На несколько файлов clog разбит исключительно для удобства. Работа с этими файлами ведется постранично, через буферы в общей памяти сервера. Размер кеша clog настраивается параметром *transaction_buffers*, а по умолчанию составляет $\frac{1}{512}$ размера буферного кеша.

с. 309
v. 17

В clog, как и в заголовке версии строки, для каждой транзакции отведено два бита: `committed` и `aborted`.

Итак, при фиксации транзакции в clog выставляется бит `committed` для данной транзакции. Когда какая-либо другая транзакция обратится к нашей табличной странице, ей придется ответить на вопрос: завершилась ли транзакция с номером `xmin`?

- Если нет, то созданная версия строки не должна быть видна.

  Чтобы проверить, выполняется ли транзакция, просматривается еще одна структура, которая располагается в общей памяти экземпляра и называется ProcArray. В ней находится список всех активных процессов, и для каждого указан номер его текущей (активной) транзакции.

- Если завершилась, то фиксацией или отменой? При отмене версия строки тоже не должна быть видна.

  Для этой проверки как раз и нужна структура clog. Но хотя последние страницы clog сохраняются в буферах в оперативной памяти, все

[1] include/access/clog.h;
backend/access/transam/clog.c.

равно такую проверку накладно выполнять каждый раз. Поэтому выясненный однажды статус транзакции записывается в заголовок версии строки в информационные биты `xmin_committed` и `xmin_aborted`; их еще называют *битами-подсказками* (hint bits). Если один из этих битов установлен, то состояние транзакции `xmin` считается известным, и следующей транзакции уже не придется обращаться ни к clog, ни к ProcArray.

Почему эти биты не устанавливаются той транзакцией, которая вставляет строку? Дело в том, что в это время транзакция еще не знает, завершится ли она успешно. А в момент фиксации уже не понятно, какие именно строки в каких именно страницах были изменены. Таких страниц может оказаться много, и запоминать их невыгодно. К тому же часть страниц может быть вытеснена из буферного кеша на диск; читать их заново, чтобы изменить биты, означало бы существенно замедлить фиксацию.

Обратная сторона экономии состоит в том, что любая транзакция (даже выполняющая простое чтение — `SELECT`) может начать выставлять информационные биты в страницах данных. Конечно, страницы в буферном кеше при этом становятся грязными.

Зафиксируем наконец вставку строки, с которой мы начали транзакцию:

```
=> COMMIT;
```

В странице ничего не изменилось (но мы знаем, что статус транзакции уже записан в clog):

```
=> SELECT * FROM heap_page('t',0);
 ctid  | state  | xmin | xmax
-------+--------+------+------
 (0,1) | normal | 795  | 0 a
(1 row)
```

Теперь транзакция, первой обратившаяся к странице («нормальным» образом, не с помощью `pageinspect`), должна будет определить статус транзакции `xmin` и записать его в информационные биты:

```
=> SELECT * FROM t;
 id |  s
----+-----
  1 | FOO
(1 row)
=> SELECT * FROM heap_page('t',0);
 ctid  | state  | xmin  | xmax
-------+--------+-------+------
 (0,1) | normal | 795 c | 0 a
(1 row)
```

Удаление

При удалении строки в поле xmax актуальной версии записывается номер удаляющей транзакции, а бит xmax_aborted сбрасывается.

> Это же значение xmax, соответствующее активной транзакции, выступает в качестве блокировки строки. Если другая транзакция собирается обновить или удалить эту строку, она будет вынуждена дождаться завершения транзакции xmax. *с. 268*

Удалим строку:

```
=> BEGIN;
=> DELETE FROM t;
=> SELECT pg_current_xact_id();
 pg_current_xact_id
--------------------
                796
(1 row)
```

Номер транзакции записался в поле xmax, но информационные биты еще не установлены:

```
=> SELECT * FROM heap_page('t',0);
 ctid  | state  | xmin  | xmax
-------+--------+-------+------
 (0,1) | normal | 795 c | 796
(1 row)
```

Отмена

Отмена изменений работает аналогично фиксации и выполняется так же быстро, только в clog вместо бита committed выставляется бит aborted. Хоть команда и называется ROLLBACK, отката изменений не происходит: все, что транзакция успела изменить в страницах данных, остается на месте.

```
=> ROLLBACK;
=> SELECT * FROM heap_page('t',0);
 ctid  | state  | xmin  | xmax
-------+--------+-------+------
 (0,1) | normal | 795 c | 796
(1 row)
```

При обращении к странице проверяется статус, и в версии строки устанавливается бит-подсказка xmax_aborted. Сам номер xmax при этом остается в странице, но смотреть на него уже никто не будет:

```
=> SELECT * FROM t;
 id |  s
----+-----
  1 | FOO
(1 row)
=> SELECT * FROM heap_page('t',0);
 ctid  | state  | xmin  | xmax
-------+--------+-------+-------
 (0,1) | normal | 795 c | 796 a
(1 row)
```

Обновление

Обновление работает так, как будто сначала выполнилось удаление текущей версии строки, а затем — вставка новой:

```
=> BEGIN;
=> UPDATE t SET s = 'BAR';
=> SELECT pg_current_xact_id();
```

```
 pg_current_xact_id
--------------------
                797
(1 row)
```

Запрос выдает одну строку (новую версию):

```
=> SELECT * FROM t;
 id |  s
----+-----
  1 | BAR
(1 row)
```

Но в странице хранятся обе версии:

```
=> SELECT * FROM heap_page('t',0);
 ctid  | state  | xmin  | xmax
-------+--------+-------+------
 (0,1) | normal | 795 c | 797
 (0,2) | normal | 797   | 0 a
(2 rows)
```

Удаленная ранее версия помечена номером текущей транзакции в поле `xmax`. Это значение записано поверх старого, поскольку предыдущая транзакция была отменена. А бит `xmax_aborted` сброшен, так как статус текущей транзакции еще неизвестен.

Ну и завершим транзакцию.

```
=> COMMIT;
```

3.4. Индексы

В индексах любого типа никогда не бывает версий строк, каждая строка представлена ровно одним экземпляром. Иными словами, в заголовке индексной строки не бывает полей `xmin` и `xmax`. Ссылки из индекса ведут на все табличные версии строк. Транзакции требуется заглянуть в таблицу, чтобы разобраться, какая из версий ей видна (если только страница не отмечена в карте видимости). *с. 115*

Для удобства создадим простую функцию, показывающую с помощью расширения pageinspect все индексные записи на странице (в пределах стра-
с. 523 ницы B-дерева записи располагаются в виде плоского списка):

```
=> CREATE FUNCTION index_page(relname text, pageno integer)
RETURNS TABLE(itemoffset smallint, htid tid)
AS $$
SELECT itemoffset,
       htid -- ctid до v.13
FROM bt_page_items(relname,pageno);
$$ LANGUAGE sql;
```

Обнаруживаем на странице указатели на обе версии табличной строки, как на актуальную, так и на старую:

```
=> SELECT * FROM index_page('t_s_idx',1);
 itemoffset | htid
------------+-------
          1 | (0,2)
          2 | (0,1)
(2 rows)
```

Поскольку BAR < FOO, указатель на вторую версию строки стоит в индексе на первом месте.

3.5. TOAST

с. 33 Toast-таблица является, по сути, обычной таблицей, и для нее поддерживается собственная версионность: версии «тостов» не связаны с версиями строк основной таблицы. Но внутренняя работа с toast-таблицей построена так, что строки никогда не обновляются, а только добавляются и удаляются, так что версионность в данном случае несколько вырожденная.

Когда данные меняются, в основной таблице всегда создается новая версия строки. Но если обновление не затрагивает «длинное» значение, хранящееся в TOAST, то новая версия строки будет ссылаться на прежнее значение в toast-таблице. И только когда обновление поменяет «длинное» значение, будут созданы и новая версия строки в основной таблице, и новые «тосты».

3.6. Виртуальные транзакции

PostgreSQL использует оптимизацию, позволяющую «экономить» номера транзакций.

Если транзакция только читает данные, то она никак не влияет на видимость версий строк. Поэтому вначале обслуживающий процесс выдает транзакции *виртуальный номер*[1] (virtual xid). Номер состоит из идентификатора обслуживающего процесса и последовательного числа. Выдача этого номера не требует синхронизации между всеми процессами и поэтому выполняется очень быстро. Настоящего номера у транзакции еще нет:

с. 259

```
=> BEGIN;
=> -- txid_current_if_assigned() до v.13
SELECT pg_current_xact_id_if_assigned();
 pg_current_xact_id_if_assigned
--------------------------------

(1 row)
```

В разные моменты времени в системе вполне могут оказаться виртуальные транзакции с номерами, которые уже использовались. И это нормально, поскольку виртуальные номера существуют только в оперативной памяти, пока транзакция активна; они никогда не записываются в страницы данных и не попадают на диск.

Если же транзакция начинает менять данные, ей выдается настоящий, уникальный номер транзакции:

```
=> UPDATE accounts
SET amount = amount - 1.00;
=> SELECT pg_current_xact_id_if_assigned();
 pg_current_xact_id_if_assigned
--------------------------------
                            799
(1 row)
=> COMMIT;
```

[1] postgrespro.ru/docs/postgresql/18/transaction-id; backend/access/transam/xact.c.

3.7. Вложенные транзакции

Точки сохранения

В SQL определены *точки сохранения* (savepoint), которые позволяют отменить часть операций транзакции, не прерывая ее полностью. Но это не укладывается в приведенную выше схему, поскольку статус транзакции один на все изменения, а физически никакие данные не откатываются.

Чтобы реализовать такой функционал, транзакция с точкой сохранения разбивается на несколько *вложенных транзакций* [1] (subtransaction), статусом которых можно управлять отдельно.

Вложенные транзакции имеют свой собственный номер (больший, чем номер основной транзакции). Статус вложенных транзакций записывается в clog обычным образом, но зафиксированные вложенные транзакции одновременно отмечаются двумя битами, committed и aborted. Финальный статус зависит от статуса основной транзакции: если она отменена, то и все вложенные транзакции считаются отмененными.

Информация о вложенности транзакций хранится в файлах внутри каталога PGDATA/pg_subtrans, а обращение к страницам этих файлов происходит че-
с. 309 рез SLRU-кеш. Его размер настраивается параметром *subtransaction_buffers*, а по умолчанию составляет $\frac{1}{512}$ размера буферного кеша.

> Не путайте вложенные транзакции с автономными. Автономные транзакции никак не зависят друг от друга, в отличие от вложенных. Автономных транзакций в обычном PostgreSQL нет, и, пожалуй, к лучшему: по делу они нужны очень и очень редко, а их наличие в других СУБД провоцирует злоупотребления, от которых потом все страдают.

Опустошим таблицу, начнем транзакцию и вставим строку:

```
=> TRUNCATE TABLE t;
=> BEGIN;
=> INSERT INTO t(s) VALUES ('FOO');
=> SELECT pg_current_xact_id();
```

[1] postgrespro.ru/docs/postgresql/18/subxacts;
backend/access/transam/subtrans.c.

```
 pg_current_xact_id
--------------------
                801
(1 row)
```

Теперь поставим точку сохранения и вставим еще одну строку:

```
=> SAVEPOINT sp;
=> INSERT INTO t(s) VALUES ('XYZ');
=> SELECT pg_current_xact_id();
 pg_current_xact_id
--------------------
                801
(1 row)
```

Заметьте, что функция pg_current_xact_id выдает номер основной, а не вложенной транзакции.

```
=> SELECT *
FROM heap_page('t',0) p
  LEFT JOIN t ON p.ctid = t.ctid;
 ctid  | state  | xmin | xmax | id |  s
-------+--------+------+------+----+-----
 (0,1) | normal | 801  | 0 a  |  2 | FOO
 (0,2) | normal | 802  | 0 a  |  3 | XYZ
(2 rows)
```

Откатимся к точке сохранения и вставим третью строку:

```
=> ROLLBACK TO sp;
=> INSERT INTO t(s) VALUES ('BAR');
=> SELECT *
FROM heap_page('t',0) p
  LEFT JOIN t ON p.ctid = t.ctid;
 ctid  | state  | xmin | xmax | id |  s
-------+--------+------+------+----+-----
 (0,1) | normal | 801  | 0 a  |  2 | FOO
 (0,2) | normal | 802  | 0 a  |    |
 (0,3) | normal | 803  | 0 a  |  4 | BAR
(3 rows)
```

В странице мы продолжаем видеть строку, добавленную отмененной вложенной транзакцией.

Фиксируем изменения:

```
=> COMMIT;
=> SELECT * FROM t;
 id |  s
----+-----
  2 | FOO
  4 | BAR
(2 rows)
=> SELECT * FROM heap_page('t',0);
 ctid  | state  | xmin  | xmax
-------+--------+-------+------
 (0,1) | normal | 801 c | 0 a
 (0,2) | normal | 802 a | 0 a
 (0,3) | normal | 803 c | 0 a
(3 rows)
```

Теперь хорошо видно, что каждая вложенная транзакция имеет собственный статус.

Вложенные транзакции нельзя использовать в SQL напрямую, то есть нельзя начать новую транзакцию, не завершив текущую:

```
=> BEGIN;
BEGIN
=> BEGIN;
WARNING:  there is already a transaction in progress
BEGIN
=> COMMIT;
COMMIT
=> COMMIT;
WARNING:  there is no transaction in progress
COMMIT
```

Этот механизм задействуется неявно: при использовании точек сохранения, при обработке исключений PL/pgSQL и в некоторых других, более экзотических случаях.

Ошибки и атомарность операций

Что случится, если в процессе выполнения операции произойдет ошибка?

Например:

```
=> BEGIN;
=> SELECT * FROM t;
 id |  s
----+-----
  2 | FOO
  4 | BAR
(2 rows)
=> UPDATE t SET s = repeat('X', 1/(id-4));
ERROR:  division by zero
```

После сбоя транзакция считается прерванной, и больше ни одна операция в ней не допускается:

```
=> SELECT * FROM t;
ERROR:  current transaction is aborted, commands ignored until end of
transaction block
```

И даже если попытаться зафиксировать изменения, PostgreSQL сообщит об отмене:

```
=> COMMIT;
ROLLBACK
```

Почему нельзя продолжить выполнение транзакции после сбоя? Поскольку уже выполненные действия никогда не откатываются, мы получили бы доступ к части изменений, выполненных до ошибки, — была бы нарушена атомарность даже не транзакции, а оператора.

Как в нашем примере, где оператор до ошибки успел обновить одну строку из двух:

```
=> SELECT * FROM heap_page('t',0);
 ctid  | state  | xmin  | xmax
-------+--------+-------+------
 (0,1) | normal | 801 c | 804
 (0,2) | normal | 802 a | 0 a
 (0,3) | normal | 803 c | 0 a
 (0,4) | normal | 804   | 0 a
(4 rows)
```

К слову, в psql имеется режим, который все-таки позволяет продолжать работу транзакции после сбоя так, как будто действия ошибочного оператора откатываются:

```
=> \set ON_ERROR_ROLLBACK on
=> BEGIN;
=> UPDATE t SET s = repeat('X', 1/(id-4));
ERROR:  division by zero
=> SELECT * FROM t;
 id |  s
----+-----
  2 | FOO
  4 | BAR
(2 rows)
=> COMMIT;
COMMIT
```

Нетрудно догадаться, что в таком режиме psql фактически ставит перед каждой командой неявную точку сохранения, а в случае сбоя инициирует откат к ней. Такой режим не используется по умолчанию, поскольку установка точек сохранения (даже без отката к ним) сопряжена с существенными накладными расходами.

Ч

Снимки данных

4.1. Что такое снимок данных

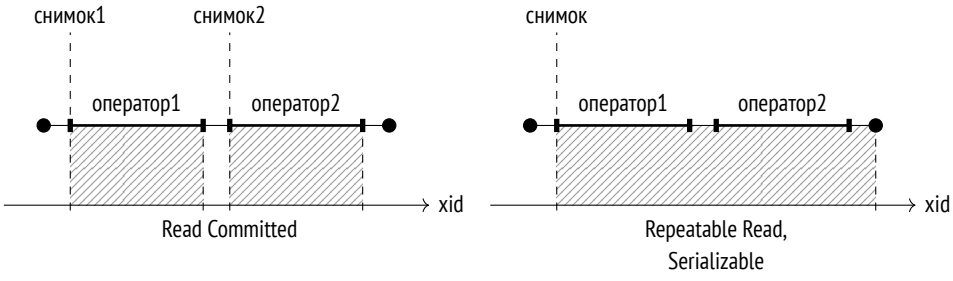
Физически в страницах данных могут находиться несколько версий одной и той же строки, хотя каждая транзакция должна видеть максимум одну из них. Все вместе версии разных строк, наблюдаемые транзакцией, образуют *снимок данных* (snapshot). Снимок обеспечивает согласованную в ACID-смысле картину данных на определенный момент времени и содержит только самые актуальные данные, зафиксированные к моменту его создания.

с. 54

Чтобы обеспечить изоляцию, каждая транзакция работает со своим собственным снимком. При этом разные транзакции видят разные, но тем не менее согласованные (на разные моменты времени) данные.

На уровне изоляции Read Committed снимок создается в начале каждого оператора транзакции и остается активным все время работы этого оператора.

На уровнях Repeatable Read и Serializable снимок создается один раз в начале первого оператора транзакции и остается активным до самого конца транзакции.



4.2. Видимость версий строк в снимке

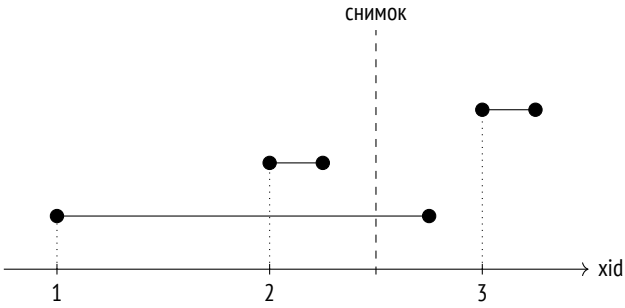
Конечно, снимок данных не является физической копией всех необходимых версий строк. Фактически снимок задается несколькими числами, а видимость версий строк в снимке определяется правилами.

Будет данная версия строки видна в снимке или нет — зависит от полей `xmin` и `xmax` ее заголовка (то есть от номеров создавшей и удалившей транзакций) и от соответствующих этим полям информационных битов. Интервалы `xmin`–`xmax` не пересекаются, поэтому каждая строка представлена в любом снимке максимум одной своей версией.

Точные правила видимости [1] довольно сложны и учитывают множество различных ситуаций и крайних случаев. Упрощая, можно сказать, что версия строки видна, когда в снимке видны изменения, сделанные транзакцией `xmin`, и не видны изменения, сделанные транзакцией `xmax` (иными словами, если версия строки уже появилась, но еще не удалена).

В свою очередь, изменения транзакции видны в снимке, если эта транзакция была зафиксирована до момента создания снимка. И еще, в качестве исключения из общего правила, транзакция видит в снимке свои собственные еще не зафиксированные изменения. Изменения оборванных транзакций, разумеется, ни в одном снимке не видны.

Вот простой пример. Транзакции изображены в виде отрезков (от момента начала до момента фиксации):



[1] backend/access/heap/heapam_visibility.c.

Здесь:

- изменения транзакции 2 будут видны, потому что она завершилась до создания снимка;
- изменения транзакции 1 не будут видны, потому что она была активна на момент создания снимка;
- изменения транзакции 3 не будут видны, потому что она началась позже создания снимка (не важно, закончилась она или нет).

4.3. Из чего состоит снимок

К сожалению, PostgreSQL видит картину совсем не так [1], как было показано на рисунке. Дело в том, что системе неизвестно, когда транзакции были зафиксированы. Известно только, когда они начинались (этот момент определяется номером транзакции), а вот факт завершения нигде не записывается.

Время фиксации отслеживается [2] при включенном параметре *track_commit_timestamp*, но оно никак не участвует в проверке видимости (хотя может быть полезным, например, для сторонних репликационных решений). off

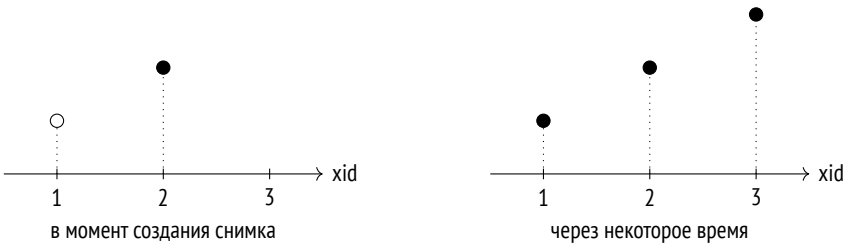
Кроме того, PostgreSQL всегда сохраняет время фиксации и время отмены транзакций в соответствующих журнальных записях, но использует эту информацию только при восстановлении до целевой точки. *с. 211*

Узнать можно лишь *текущий* статус транзакций. Эта информация есть в общей памяти сервера в структуре ProcArray, которая содержит список всех активных сеансов и их транзакций. Постфактум же невозможно выяснить, была ли какая-то транзакция активна в момент создания снимка.

Поэтому для получения снимка недостаточно сохранить момент его создания: требуется также запомнить, в каком статусе находились транзакции на этот момент. Без информации о статусах впоследствии невозможно будет понять, какие версии строк должны быть видны в снимке, а какие — нет.

[1] include/utils/snapshot.h.
backend/utils/time/snapmgr.c.
[2] backend/access/transam/commit_ts.c.

Сравните, какая информация доступна системе в момент создания снимка и некоторое время спустя (белым кружком обозначена активная транзакция, а черными — завершенные):



Если не запомнить, что во время создания снимка первая транзакция еще выполнялась, а третьей не существовало, их невозможно будет отличить от зафиксированной на тот момент второй транзакции и исключить из рассмотрения.

По этой причине в PostgreSQL нельзя создать снимок, показывающий согласованные данные по состоянию на некоторый момент в прошлом, даже если все необходимые для этого версии строк существуют в табличных страницах. Соответственно, невозможно реализовать и ретроспективные (темпоральные, flashback) запросы.

> Интересно, что такая функциональность была заявлена как одна из целей Postgres и была реализована с самого начала, но ее убрали из СУБД, когда проект был взят на поддержку сообществом[1].

Итак, снимок данных состоит из нескольких значений, которые запоминаются в момент его создания[2]:

Нижняя граница снимка `xmin`, в качестве которой выступает номер самой старой активной транзакции.

с. 134

Все транзакции с меньшими номерами либо зафиксированы, и тогда их изменения видны в снимке, либо отменены, и тогда изменения игнорируются.

[1] *Джозеф Хеллерштейн*. Postgres в ретроспективе // habr.com/ru/company/postgrespro/blog/438890.

[2] backend/storage/ipc/procarray.c, функция GetSnapshotData.

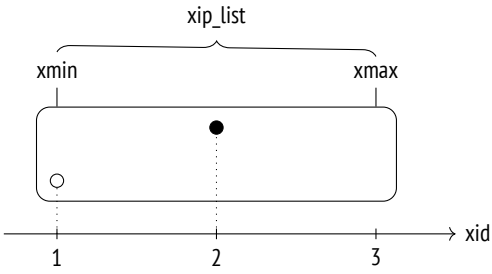
Верхняя граница снимка `xmax`, в качестве которой берется значение, на единицу большее номера последней зафиксированной транзакции. Верхняя граница определяет момент времени, в который был сделан снимок.

Все транзакции с номерами, большими или равными `xmax`, не завершены или не существуют, и поэтому изменения таких транзакций точно не видны.

Список активных транзакций `xip_list` (xid-in-progress list), в который попадают номера всех активных транзакций (меньшие `xmax`), за исключением виртуальных, которые никак не влияют на видимость. *с. 91*

Также в снимке сохраняются еще несколько параметров, но они пока не так важны.

Графически можно представить снимок как прямоугольник, охватывающий транзакции от `xmin` до `xmax`:



Чтобы посмотреть, как видимость определяется снимком, воспроизведем ситуацию по показанному выше сценарию на таблице `accounts`.

```
=> TRUNCATE TABLE accounts;
```

Первая транзакция вставляет в таблицу первую строку и остается активной:

```
=> BEGIN;
=> INSERT INTO accounts VALUES (1, 'alice', 1000.00);
=> SELECT pg_current_xact_id();
 pg_current_xact_id
--------------------
                809
(1 row)
```

Вторая транзакция вставляет вторую строку и сразу завершается:

```
=> BEGIN;
=> INSERT INTO accounts VALUES (2, 'bob', 100.00);
=> SELECT pg_current_xact_id();
 pg_current_xact_id
--------------------
                810
(1 row)
=> COMMIT;
```

В этот момент (в другом сеансе) создаем новый снимок. Для этого достаточно выполнить любой запрос, но мы сразу посмотрим на снимок с помощью специальной функции:

```
=> BEGIN ISOLATION LEVEL REPEATABLE READ;
=> -- txid_current_snapshot() до v.13
SELECT pg_current_snapshot();
 pg_current_snapshot
---------------------
 809:811:809
(1 row)
```

Функция выводит через двоеточие три составляющих снимка: поля `xmin`, `xmax` и список `xip_list` (состоящий в данном случае из одного номера).

После того как снимок создан, завершаем первую транзакцию:

```
=> COMMIT;
```

Третья транзакция выполняется после создания снимка и меняет вторую строку. Появляется новая версия:

```
=> BEGIN;
=> UPDATE accounts SET amount = amount + 100 WHERE id = 2;
=> SELECT pg_current_xact_id();
 pg_current_xact_id
--------------------
                811
(1 row)
=> COMMIT;
```

В созданном снимке видна только одна версия:

```
=> SELECT ctid, * FROM accounts;
 ctid  | id | client | amount
-------+----+--------+--------
 (0,2) |  2 | bob    | 100.00
(1 row)
```

Хотя в таблице их три:

```
=> SELECT * FROM heap_page('accounts',0);
 ctid  | state  | xmin  | xmax
-------+--------+-------+-------
 (0,1) | normal | 809 c | 0 a
 (0,2) | normal | 810 c | 811 c
 (0,3) | normal | 811 c | 0 a
(3 rows)
```

Как PostgreSQL понимает, какие версии показывать? По сформулированным выше правилам в снимке видны изменения только следующих зафиксированных транзакций:

- с номерами $\texttt{xid} < \texttt{xmin}$ — безусловно (например, транзакция, создавшая таблицу accounts);
- с номерами $\texttt{xmin} \leqslant \texttt{xid} < \texttt{xmax}$ — за исключением попавших в список xip_list.

Первая строка (0,1) не видна, так как номер создавшей ее транзакции входит в список активных транзакций (хотя и попадает при этом в диапазон снимка).

Последняя версия второй строки (0,3) не видна, поскольку создана транзакцией с номером, выходящим за верхнюю границу снимка.

Зато видна первая версия второй строки (0,2): номер создавшей ее транзакции попадает в диапазон снимка, но не входит в список активных транзакций (вставка видна), и при этом номер удалившей транзакции выходит за верхнюю границу снимка (удаление не видно).

```
=> COMMIT;
```

4.4. Видимость собственных изменений

Картину несколько усложняет определение видимости собственных изменений транзакции, поскольку возможна ситуация, при которой должна быть видна только часть из них. Например, курсор, открытый в определенный момент, ни при каком уровне изоляции не может видеть последующие изменения.

Для этого в заголовке версии строки есть специальное поле (которое отображается в псевдостолбцах cmin и cmax), показывающее порядковый номер операции внутри транзакции. Столбец cmin представляет номер операции вставки, а cmax — операции удаления, но для экономии места в заголовке строки это на самом деле одно поле, а не два разных. Считается, что вставка и удаление той же строки в одной транзакции выполняются редко. (Если это все-таки происходит, в то же самое поле записывается специальный «комбономер», для которого обслуживающий процесс запоминает реальные cmin и cmax[1].)

В качестве примера начнем транзакцию и вставим в таблицу строку:

```
=> BEGIN;
=> INSERT INTO accounts VALUES (3, 'charlie', 100.00);
=> SELECT pg_current_xact_id();
 pg_current_xact_id
--------------------
                812
(1 row)
```

Теперь откроем курсор для запроса, возвращающего число строк в таблице:

```
=> DECLARE c CURSOR FOR SELECT count(*) FROM accounts;
```

И после этого вставим еще одну строку:

```
=> INSERT INTO accounts VALUES (4, 'charlie', 200.00);
```

Выведем содержимое таблицы, добавив столбец cmin для строк, вставленных нашей транзакцией (для других строк это значение не имеет смысла):

[1] backend/utils/time/combocid.c.

```
=> SELECT xmin, CASE WHEN xmin = 812 THEN cmin END cmin, *
FROM accounts;
 xmin | cmin | id | client  | amount
------+------+----+---------+---------
  809 |      |  1 | alice   | 1000.00
  811 |      |  2 | bob     |  200.00
  812 |    0 |  3 | charlie |  100.00
  812 |    1 |  4 | charlie |  200.00
(4 rows)
```

Запрос курсора обнаружит три строки, а не четыре — строка, добавленная после открытия курсора, не попадет в снимок данных, поскольку в нем учитываются только версии строк с cmin < 1:

```
=> FETCH c;
 count
-------
     3
(1 row)
```

Разумеется, этот номер cmin также запоминается в снимке, но вывести его средствами SQL не получится.

4.5. Горизонт транзакции

Нижняя граница снимка (номер xmin самой ранней транзакции, активной на момент его создания) имеет важный смысл — она определяет *горизонт транзакции*, работающей с этим снимком.

Горизонт транзакции без активного снимка (например, транзакции с уровнем изоляции Read Committed между выполнениями операторов) определяется ее собственным номером, если он ей присвоен.

Все транзакции, находящиеся за горизонтом (то есть транзакции с номерами xid < xmin), уже гарантированно зафиксированы. А это значит, что за своим горизонтом транзакция всегда видит только актуальные версии строк.

Название навеяно, конечно, понятием *горизонта событий* в физике.

PostgreSQL знает текущий горизонт транзакций всех процессов; собственный горизонт транзакция может увидеть в таблице `pg_stat_activity`:

```
=> BEGIN;
=> SELECT backend_xmin FROM pg_stat_activity
WHERE pid = pg_backend_pid();
 backend_xmin
--------------
          812
(1 row)
```

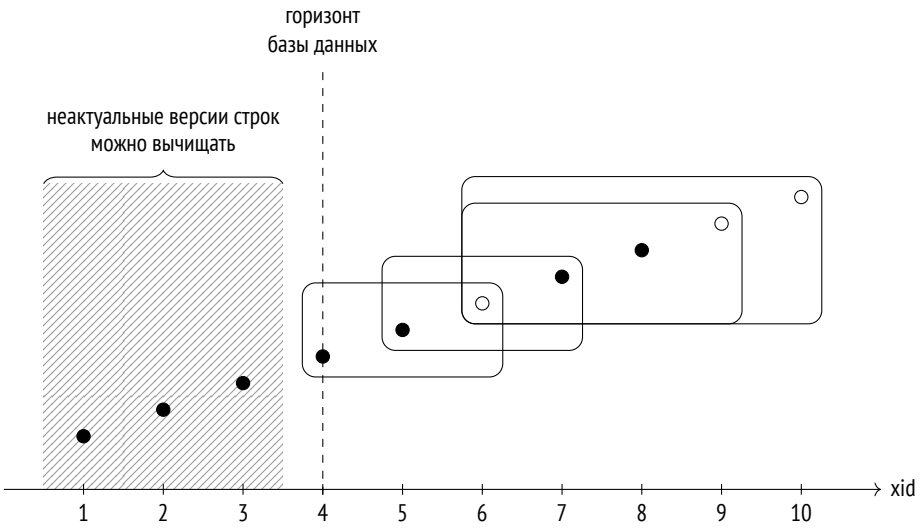
Виртуальные транзакции хоть и не имеют настоящего номера, но используют снимки точно так же, как и обычные транзакции, и поэтому обладают собственным горизонтом. Исключение составляют виртуальные транзакции без активного снимка: для них понятие горизонта не имеет смысла, и они полностью «прозрачны» для системы с точки зрения снимков данных и видимости (несмотря на то что в `pg_stat_activity.backend_xmin` может остаться номер от старого снимка).

Похожим образом можно определить и *горизонт базы данных*. Для этого надо взять горизонты всех транзакций, работающих с этой базой, и среди них найти наиболее «дальний», имеющий самый старый `xmin`[1]. Это и будет тот горизонт, за которым неактуальные версии строк в этой базе данных уже никогда не будут видны ни одной транзакции. *Такие версии строк могут быть безопасно удалены очисткой* — именно поэтому понятие горизонта так важно с практической точки зрения.

Сделаем выводы из сказанного:

- если транзакция с уровнем изоляции Repeatable Read или Serializable (не важно, настоящая или виртуальная) выполняется долго, она тем самым удерживает горизонт базы данных и препятствует очистке;
- настоящая транзакция с уровнем изоляции Read Committed точно так же удерживает горизонт базы данных, выполняет ли она оператор или просто бездействует (находится в состоянии idle in transaction);
- виртуальная транзакция с уровнем изоляции Read Committed удерживает горизонт только в процессе выполнения операторов.

[1] backend/storage/ipc/procarray.c, функция ComputeXidHorizons.



При этом на всю базу данных есть только один горизонт, и если какая-то транзакция его удерживает — она не дает очищать данные внутри этого горизонта, даже те, к которым не обращалась.

> Для общекластерных таблиц системного каталога используется отдельный горизонт, учитывающий транзакции во всех базах данных. А для временных таблиц, наоборот, не нужно учитывать никакие транзакции, кроме выполняющихся в текущем процессе.

Продолжим пример. Транзакция в первом сеансе до сих пор выполняется и продолжает удерживать горизонт, что можно проверить, увеличив счетчик транзакций:

```
=> SELECT pg_current_xact_id();
 pg_current_xact_id
--------------------
                813
(1 row)
```

```
=> SELECT backend_xmin FROM pg_stat_activity
WHERE pid = pg_backend_pid();
 backend_xmin
--------------
          812
(1 row)
```

И только после завершения транзакции горизонт продвигается вперед, позволяя очистке удалить неактуальные версии строк:

```
=> COMMIT;
=> SELECT backend_xmin FROM pg_stat_activity
WHERE pid = pg_backend_pid();
 backend_xmin
--------------
          814
(1 row)
```

с. 178 В идеале не следует совмещать активные обновления (порождающие версии строк) с долгими транзакциями, поскольку это будет приводить к раздуванию таблиц и индексов.

4.6. Снимок данных для системного каталога

Хотя системный каталог и представлен обычными таблицами, при обращении к ним нельзя задействовать тот же снимок данных, что используется транзакцией или оператором. Снимок должен быть достаточно «свежим», чтобы включать все последние изменения, ведь иначе транзакция могла бы увидеть уже неактуальное определение столбцов таблицы или пропустить созданное ограничение целостности.

Вот простой пример:

```
=> BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;
=> SELECT 1; -- создан снимок для транзакции


    => ALTER TABLE accounts ALTER amount SET NOT NULL;


=> INSERT INTO accounts(client, amount) VALUES ('alice', NULL);
ERROR:  null value in column "amount" of relation "accounts" violates
not-null constraint
DETAIL:  Failing row contains (1, alice, null).
=> ROLLBACK;
```

Команда INSERT «увидела» ограничение целостности, которое появилось уже после того, как был создан снимок данных транзакции. Может показаться, что такое поведение нарушает изоляцию, но если бы транзакция успела обратиться к таблице accounts, то команда ALTER TABLE была бы заблокирована до окончания этой транзакции. *с. 261*

В целом система ведет себя так, как будто для каждого обращения к системному каталогу создается новый снимок. Но реализация [1], конечно, более сложна, поскольку постоянное создание новых снимков негативно сказалось бы на производительности; кроме того, многие объекты системного каталога кешируются, и это тоже надо учитывать.

4.7. Экспорт снимка данных

Бывают ситуации, когда несколько параллельных транзакций должны гарантированно видеть одну и ту же картину данных. В качестве примера можно привести утилиту pg_dump, умеющую работать в параллельном режиме: все рабочие процессы должны видеть базу данных в одном и том же состоянии, чтобы резервная копия получилась согласованной.

Разумеется, нельзя полагаться на то, что картины данных совпадут просто потому, что транзакции запущены «одновременно». Такие гарантии дает механизм экспорта и импорта снимка.

Функция pg_export_snapshot возвращает идентификатор снимка, который может быть передан в другую транзакцию (внешними по отношению к СУБД средствами):

```
=> BEGIN ISOLATION LEVEL REPEATABLE READ;
=> SELECT count(*) FROM accounts;
 count
-------
     4
(1 row)
```

[1] backend/utils/time/snapmgr.c, функция GetCatalogSnapshot.

```
=> SELECT pg_export_snapshot();
 pg_export_snapshot
---------------------
 00000011-00000008-1
(1 row)
```

Перед тем как выполнить первый оператор, другая транзакция может импортировать снимок с помощью команды SET TRANSACTION SNAPSHOT. Предварительно надо установить уровень изоляции Repeatable Read или Serializable, потому что на уровне Read Committed операторы будут использовать собственные снимки:

```
=> DELETE FROM accounts;
=> BEGIN ISOLATION LEVEL REPEATABLE READ;
=> SET TRANSACTION SNAPSHOT '00000011-00000008-1';
```

Теперь вторая транзакция будет работать со снимком первой и, соответственно, видеть четыре строки (а не ноль):

```
=> SELECT count(*) FROM accounts;
 count
-------
     4
(1 row)
```

Разумеется, изменения, сделанные первой транзакцией после экспорта снимка, не будут видны второй транзакции (и наоборот): действуют обычные правила видимости.

Время жизни экспортированного снимка совпадает со временем жизни экспортирующей транзакции.

```
=> COMMIT;
```

```
=> COMMIT;
```

5

Самоочистка и hot-обновления

5.1. Самоочистка

При обращении к табличной странице — как при обновлении, так и при обычном чтении — может выполняться быстрая внутристраничная *самоочистка*[1]. Это происходит в одном из трех случаев:

- Ранее выполненное обновление (`UPDATE`) не обнаружило достаточно места, чтобы разместить новую версию строки на этой же странице. Такая ситуация запоминается в заголовке страницы.
- Табличная страница заполнена больше, чем на 90 %.
- Табличная страница заполнена больше, чем на значение параметра хранения *fillfactor*.

  PostgreSQL вставляет (`INSERT`) новую строку на страницу, только если эта страница заполнена менее, чем на *fillfactor* процентов. Остальное место приберегается для новых версий строк, которые получаются в результате обновлений (`UPDATE`). Со значением по умолчанию место в таблицах не резервируется.

100

Самоочистка убирает со страницы версии строк, не видимые ни в одном снимке (находящиеся за горизонтом базы данных). Она никогда не выходит за одну табличную страницу, зато выполняется очень быстро. Указатели на вычищенные версии строк не освобождаются, так как на них могут вести

с. 106

[1] backend/access/heap/pruneheap.c, функция heap_page_prune_opt.

ссылки из индексов, а индекс — это уже другая страница. Из этих же соображений не обновляются ни карта видимости, ни карта свободного пространства (получается, что освобожденное место используется для обновлений, а не для вставок).

Тот факт, что страница может очищаться при чтении, означает, что оператор SELECT может вызвать изменение страниц. Это еще один такой случай
с. 86 в дополнение к отложенному изменению битов-подсказок.

Посмотрим на примере, как работает самоочистка. Создадим таблицу с двумя столбцами и индексы по каждому из них:

```
=> CREATE TABLE hot(id integer, s char(2000)) WITH (fillfactor = 75);
=> CREATE INDEX hot_id ON hot(id);
=> CREATE INDEX hot_s ON hot(s);
```

Если в столбце s хранить только латинские буквы, каждая версия строки будет иметь фиксированный размер 2004 байта, не считая 24 байт заголовка. Параметр хранения *fillfactor* установлен в 75 %. Значит, места на странице будет хватать на четыре версии строки, но вставить получится только три.

Теперь вставим одну строку и несколько раз изменим ее:

```
=> INSERT INTO hot VALUES (1, 'A');
=> UPDATE hot SET s = 'B';
=> UPDATE hot SET s = 'C';
=> UPDATE hot SET s = 'D';
```

В странице сейчас четыре версии строки:

```
=> SELECT * FROM heap_page('hot',0);
 ctid  | state  | xmin  | xmax
-------+--------+-------+-------
 (0,1) | normal | 820 c | 821 c
 (0,2) | normal | 821 c | 822 c
 (0,3) | normal | 822 c | 823
 (0,4) | normal | 823   | 0 a
(4 rows)
```

Как и ожидалось, мы только что вышли за *fillfactor*. На это указывает разни-
с. 75 ца между значениями pagesize и upper — она превышает порог в 75 % от размера страницы, составляющий 6144 байта:

```
=> SELECT upper, pagesize FROM page_header(get_raw_page('hot',0));
 upper | pagesize
-------+----------
    64 |     8192
(1 row)
```

При следующем обращении к странице срабатывает самоочистка, которая вычищает все неактуальные версии. Теперь освободившегося места хватает на новую версию (0,5):

```
=> UPDATE hot SET s = 'E';
=> SELECT * FROM heap_page('hot',0);
 ctid  | state  | xmin  | xmax
-------+--------+-------+------
 (0,1) | dead   |       |
 (0,2) | dead   |       |
 (0,3) | dead   |       |
 (0,4) | normal | 823 c | 824
 (0,5) | normal | 824   | 0 a
(5 rows)
```

Версии строк, оставшиеся после очистки, физически сдвигаются в сторону старших адресов страницы так, чтобы все свободное место было представлено одним непрерывным фрагментом. Соответствующим образом изменяются и указатели. Благодаря этому не возникает проблем с фрагментацией свободного места в странице.

Указатели на удаленные версии строк пока нельзя освободить, поскольку на них ведут ссылки с индексной страницы; статус таких указателей меняется с normal на dead («мертвая» версия). Заглянем в первую страницу индекса hot_s (нулевая занята метаинформацией): *с. 529*

```
=> SELECT * FROM index_page('hot_s',1);
 itemoffset | htid
------------+-------
          1 | (0,1)
          2 | (0,2)
          3 | (0,3)
          4 | (0,4)
          5 | (0,5)
(5 rows)
```

Ту же картину мы увидим и в другом индексе:

```
=> SELECT * FROM index_page('hot_id',1);
 itemoffset | htid
------------+-------
          1 | (0,1)
          2 | (0,2)
          3 | (0,3)
          4 | (0,4)
          5 | (0,5)
(5 rows)
```

При индексном доступе сервер может получить (0,1), (0,2) или (0,3) в качестве идентификатора версии строки. Тогда он попробует прочитать соответствующую версию строки из табличной страницы, но обнаружит, что указатель имеет статус dead, то есть версия уже не существует и должна игнорироваться. А заодно изменит статус указателя и в индексной странице, чтобы повторно не обращаться к таблице[1].

v. 13 Расширим функцию для просмотра индексной страницы, добавив признак «мертвого» указателя:

```
=> DROP FUNCTION index_page(text, integer);
=> CREATE FUNCTION index_page(relname text, pageno integer)
RETURNS TABLE(itemoffset smallint, htid tid, dead boolean)
AS $$
SELECT itemoffset,
       htid,
       dead -- начиная с v.13
FROM bt_page_items(relname,pageno);
$$ LANGUAGE sql;
=> SELECT * FROM index_page('hot_id',1);
 itemoffset | htid  | dead
------------+-------+------
          1 | (0,1) | f
          2 | (0,2) | f
          3 | (0,3) | f
          4 | (0,4) | f
          5 | (0,5) | f
(5 rows)
```

Пока все указатели на индексной странице активны. Но при первом же обращении к таблице по индексу статус указателей меняется:

[1] backend/access/index/indexam.c, функция index_fetch_heap.

```
=> EXPLAIN (analyze, costs off, timing off, summary off, buffers off)
SELECT * FROM hot WHERE id = 1;
                        QUERY PLAN
-----------------------------------------------------------
 Index Scan using hot_id on hot (actual rows=1.00 loops=1)
   Index Cond: (id = 1)
   Index Searches: 1
(3 rows)

=> SELECT * FROM index_page('hot_id',1);
 itemoffset | htid  | dead
------------+-------+------
          1 | (0,1) | t
          2 | (0,2) | t
          3 | (0,3) | t
          4 | (0,4) | t
          5 | (0,5) | f
(5 rows)
```

Хотя табличная строка, на которую ссылается четвертый указатель, еще не вычищена и имеет статус normal, она уже ушла за горизонт базы данных. Поэтому и четвертый указатель в индексе помечен как мертвый.

5.2. Hot-обновления

Держать в индексе ссылки на все версии каждой строки неэффективно.

Во-первых, при любом изменении строки приходится обновлять *все* индексы, созданные для таблицы: раз появилась новая версия, необходимо иметь на нее ссылки из каждого индекса, даже если изменились поля, в этот индекс не входящие.

Во-вторых, в индексах накапливаются ссылки на исторические версии строк, которые потом приходится вычищать вместе с самими версиями. *с. 125*

Естественно, чем больше индексов создано на таблице, тем с большими сложностями приходится сталкиваться.

Однако если в результате обновления изменились только столбцы, не входящие *ни в один* индекс, то нет смысла создавать в индексах дополнительную запись, содержащую те же самые значения.

С появлением лишних индексных записей борется оптимизация, называемая *hot-обновлением*[1] (Heap-Only Tuple update). Такое обновление возможно в трех случаях:

- на таблице не создан ни один индекс (в этом случае любое обновление будет hot-обновлением);
- на таблице есть индексы, но обновляемые столбцы не используются ни в одном из них;
- обновляемые столбцы используются в индексе, но их значения не изменились.

Считается, что столбец используется в индексе, если он хоть как-то упоминается в команде `CREATE INDEX`: либо индекс построен по самому столбцу или по выражению, содержащему этот столбец, либо столбец входит в include-индекс как неключевой. Исключение составляют лишь индексы BRIN: в них нет ссылок на табличные строки, и поэтому они никак не мешают hot-обновлениям.

v. 16 *с. 642*

При hot-обновлении в индексной странице присутствует лишь одна запись, и она ссылается на самую первую версию табличной строки. Все остальные версии той же строки внутри табличной страницы связаны в цепочку указателями `ctid` в заголовках версий.

Версии, на которые нет ссылок из индекса, маркируются битом Heap-Only Tuple («только табличная версия строки»). Если же версия входит в цепочку, она помечается битом Heap Hot Updated.

Если при сканировании индекса сервер попадает в табличную страницу и обнаруживает версию с флагом Heap Hot Updated, он понимает, что не надо останавливаться, и проходит дальше по всей цепочке обновлений. Разумеется, для всех прочитанных таким образом версий строк проверяется видимость, прежде чем клиент получит результат.

Чтобы посмотреть на работу hot-обновления, удалим один индекс и опустошим таблицу.

[1] backend/access/heap/README.HOT.

```
=> DROP INDEX hot_s;
=> TRUNCATE TABLE hot;
```

Для удобства пересоздадим функцию heap_page, дополнив вывод тремя полями — ctid и двумя битами, относящимися к hot-обновлению:

```
=> DROP FUNCTION heap_page(text,integer);
=> CREATE FUNCTION heap_page(
  relname text,
  pageno integer
)
RETURNS TABLE(
  ctid tid,
  state text,
  xmin text,
  xmax text,
  hhu text,
  hot text,
  t_ctid tid
)
AS $$
SELECT (pageno,lp)::text::tid AS ctid,
       CASE lp_flags
         WHEN 0 THEN 'unused'
         WHEN 1 THEN 'normal'
         WHEN 2 THEN 'redirect to '||lp_off
         WHEN 3 THEN 'dead'
       END AS state,
       t_xmin || CASE
         -- f пригодится позже
         WHEN (t_infomask & 256+512) = 256+512 THEN ' f'
         WHEN (t_infomask & 256) > 0 THEN ' c'
         WHEN (t_infomask & 512) > 0 THEN ' a'
         ELSE ''
       END AS xmin,
       t_xmax || CASE
         WHEN (t_infomask & 1024) > 0 THEN ' c'
         WHEN (t_infomask & 2048) > 0 THEN ' a'
         ELSE ''
       END AS xmax,
       CASE WHEN (t_infomask2 & 16384) > 0 THEN 't' END AS hhu,
       CASE WHEN (t_infomask2 & 32768) > 0 THEN 't' END AS hot,
       t_ctid
FROM heap_page_items(get_raw_page(relname,pageno))
ORDER BY lp;
$$ LANGUAGE sql;
```

Повторяем вставку и обновление строки:

```
=> INSERT INTO hot VALUES (1, 'A');
=> UPDATE hot SET s = 'B';
```

В странице появилась hot-цепочка изменений:

- флаг Heap Hot Updated (hhu) говорит о том, что надо идти дальше по цепочке ctid;
- флаг Heap Only Tuple (hot) показывает, что на данную версию строки нет ссылок из индексов.

```
=> SELECT * FROM heap_page('hot',0);
 ctid  | state  | xmin  | xmax | hhu | hot | t_ctid
-------+--------+-------+------+-----+-----+--------
 (0,1) | normal | 831 c | 832  | t   |     | (0,2)
 (0,2) | normal | 832   | 0 a  |     | t   | (0,2)
(2 rows)
```

При дальнейших изменениях цепочка будет расти — но только в пределах страницы:

```
=> UPDATE hot SET s = 'C';
=> UPDATE hot SET s = 'D';
=> SELECT * FROM heap_page('hot',0);
 ctid  | state  | xmin  | xmax  | hhu | hot | t_ctid
-------+--------+-------+-------+-----+-----+--------
 (0,1) | normal | 831 c | 832 c | t   |     | (0,2)
 (0,2) | normal | 832 c | 833 c | t   | t   | (0,3)
 (0,3) | normal | 833 c | 834   | t   | t   | (0,4)
 (0,4) | normal | 834   | 0 a   |     | t   | (0,4)
(4 rows)
```

При этом в индексе обнаруживаем одну-единственную ссылку на «голову» цепочки:

```
=> SELECT * FROM index_page('hot_id',1);
 itemoffset | htid  | dead
------------+-------+------
          1 | (0,1) | f
(1 row)
```

Цепочка hot-обновлений не выходит за рамки страницы, поэтому обход всей цепочки никогда не требует обращения к другим страницам и не ухудшает производительность.

Тип обновлений отслеживается как часть статистики использования таблицы. В данном случае из семи операций UPDATE три были hot-обновлениями:

```
=> SELECT n_tup_upd, n_tup_hot_upd
FROM pg_stat_all_tables
WHERE relid = 'hot'::regclass;
 n_tup_upd | n_tup_hot_upd
-----------+---------------
         7 |             3
(1 row)
```

5.3. Самоочистка при hot-обновлениях

Частный, но важный случай представляет собой самоочистка при hot-обновлениях.

В начатом примере уже превышен порог *fillfactor*, так что следующее обновление должно привести к самоочистке. Но на этот раз в странице находится цепочка обновлений. «Голова» этой hot-цепочки всегда должна оставаться на своем месте, поскольку на нее ссылается индекс, а остальные указатели могут быть освобождены, ведь на них точно нет ссылок извне.

Чтобы не трогать «голову», применяется двойная адресация: указатель, на который ссылается индекс, — в данном случае (0,1) — получает статус redirect и ведет на версию строки, с которой в данный момент начинается цепочка:

```
=> UPDATE hot SET s = 'E';
=> SELECT * FROM heap_page('hot',0);
 ctid  |     state     | xmin  | xmax | hhu | hot | t_ctid
-------+---------------+-------+------+-----+-----+--------
 (0,1) | redirect to 4 |       |      |     |     |
 (0,2) | normal        | 835   | 0 a  |     | t   | (0,2)
 (0,3) | unused        |       |      |     |     |
 (0,4) | normal        | 834 c | 835  | t   | t   | (0,2)
(4 rows)
```



Здесь версии (0,1), (0,2) и (0,3) были вычищены, «головной» указатель 1 остался перенаправлять доступ, а указатели 2 и 3 были освобождены (получили статус unused), поскольку на эти версии гарантированно не было ссылок из индексов. Новая версия строки была записана на освободившееся место под номером (0,2).

Выполним обновление еще несколько раз:

```
=> UPDATE hot SET s = 'F';
```

```
=> UPDATE hot SET s = 'G';
```

Hot-цепочка выросла:

```
=> SELECT * FROM heap_page('hot',0);
 ctid  |     state     | xmin  | xmax  | hhu | hot | t_ctid
-------+---------------+-------+-------+-----+-----+--------
 (0,1) | redirect to 4 |       |       |     |     |
 (0,2) | normal        | 835 c | 836 c | t   | t   | (0,3)
 (0,3) | normal        | 836 c | 837   | t   | t   | (0,5)
 (0,4) | normal        | 834 c | 835 c | t   | t   | (0,2)
 (0,5) | normal        | 837   | 0 a   |     | t   | (0,5)
(5 rows)
```

Следующее обновление снова вызывает самоочистку:

```
=> UPDATE hot SET s = 'H';
=> SELECT * FROM heap_page('hot',0);
 ctid  |     state     | xmin  | xmax | hhu | hot | t_ctid
-------+---------------+-------+------+-----+-----+--------
 (0,1) | redirect to 5 |       |      |     |     |
 (0,2) | normal        | 838   | 0 a  |     | t   | (0,2)
 (0,3) | unused        |       |      |     |     |
 (0,4) | unused        |       |      |     |     |
 (0,5) | normal        | 837 c | 838  | t   | t   | (0,2)
(5 rows)
```

Часть версий вычищена, а указатель на «голову» соответствующим образом сдвинут. Если продолжать обновления, в этом (удачно подобранном) примере все версии строк так и будут «крутиться» в пределах одной страницы.

5.4. Разрыв hot-цепочки

Если на странице не хватит свободного места, чтобы разместить новую версию строки, цепочка прервется. На версию строки, размещенную на другой странице, придется сделать отдельную ссылку из индекса.

Чтобы получить такую ситуацию, начнем параллельную транзакцию и построим в ней снимок данных, который не даст вычищать версии строк:

```
=> BEGIN ISOLATION LEVEL REPEATABLE READ;
=> SELECT 1;
```

Теперь выполним обновления в первом сеансе:

```
=> UPDATE hot SET s = 'I';
=> UPDATE hot SET s = 'J';
=> UPDATE hot SET s = 'K';
```

Пока все версии помещаются на одной странице:

```
=> SELECT * FROM heap_page('hot',0);
 ctid  |     state     | xmin  | xmax  | hhu | hot | t_ctid
-------+---------------+-------+-------+-----+-----+--------
 (0,1) | redirect to 2 |       |       |     |     |       )
 (0,2) | normal        | 838 c | 839 c | t   | t   | (0,3) )
 (0,3) | normal        | 839 c | 840 c | t   | t   | (0,4) )
 (0,4) | normal        | 840 c | 841   | t   | t   | (0,5) )
 (0,5) | normal        | 841   | 0 a   |     | t   | (0,5) ↺
(5 rows)
```

При следующем обновлении места на странице уже не хватит, а самоочистка не сможет ничего освободить:

```
=> UPDATE hot SET s = 'L';
```

```
=> COMMIT; -- снимок больше не нужен
```

```
=> SELECT * FROM heap_page('hot',0);
 ctid  |     state     | xmin  | xmax  | hhu | hot | t_ctid
-------+---------------+-------+-------+-----+-----+--------
 (0,1) | redirect to 2 |       |       |     |     |
 (0,2) | normal        | 838 c | 839 c | t   | t   | (0,3)
 (0,3) | normal        | 839 c | 840 c | t   | t   | (0,4)
 (0,4) | normal        | 840 c | 841 c | t   | t   | (0,5)
 (0,5) | normal        | 841 c | 842   |     | t   | (1,1)
(5 rows)
```

В версии (0,5) видим ссылку (1,1), ведущую на страницу 1:

```
=> SELECT * FROM heap_page('hot',1);
 ctid  | state  | xmin | xmax | hhu | hot | t_ctid
-------+--------+------+------+-----+-----+--------
 (1,1) | normal | 842  | 0 a  |     |     | (1,1)
(1 row)
```

Впрочем, эта ссылка не используется, поскольку в версии (0,5) не установлен бит Heap Hot Updated. А на версию (1,1) можно попасть из индекса, в котором теперь две ссылки. Каждая из них ведет на начало своей hot-цепочки:

```
=> SELECT * FROM index_page('hot_id',1);
 itemoffset | htid  | dead
------------+-------+------
          1 | (0,1) | f
          2 | (1,1) | f
(2 rows)
```

v. 16 Разрывы hot-цепочек отслеживаются в статистике отдельно:

```
=> SELECT n_tup_upd, n_tup_hot_upd, n_tup_newpage_upd
FROM pg_stat_all_tables
WHERE relid = 'hot'::regclass;
 n_tup_upd | n_tup_hot_upd | n_tup_newpage_upd
-----------+---------------+-------------------
        15 |            10 |                 1
(1 row)
```

Здесь из пятнадцати операций `UPDATE` одиннадцать были hot-обновлениями: десяти хватило места на странице, а одной нет, что привело к разрыву цепочки.

При частых обновлениях столбцов, не входящих в индексы, может иметь смысл уменьшить параметр *fillfactor*, чтобы зарезервировать на странице достаточно места для обновлений. Конечно, надо учитывать, что чем ниже *fillfactor*, тем больше остается незанятого места на странице, и, соответственно, физический размер таблицы увеличивается.

5.5. Самоочистка индексов

Я говорил о том, что самоочистка работает с одной табличной страницей и не трогает индексы. Но и для индексов есть собственная самоочистка[1], которая тоже работает с одной (но уже индексной) страницей.

Такая очистка происходит, когда при вставке строки в B-дерево на индексной странице не оказывается места и ее требуется *расщепить* (split), перераспределив данные между двумя новыми страницами. Проблема в том, что при удалении записей две индексные страницы уже не «склеиваются» в одну. Это приводит к раздуванию индекса, а однажды выросший размер может не уменьшиться даже при удалении существенной части данных. Но если самоочистке удается убрать часть индексных записей, момент расщепления может быть отложен.

Есть два вида записей, которые самоочистка может удалить из индекса.

В первую очередь удаляются записи, помеченные ранее как мертвые[2]. Как я уже говорил, такая пометка ставится при индексном доступе, если оказывается, что индексная запись ссылается на версию строки, которая не видна ни в одном снимке или уже не существует.

Если заведомо мертвых записей не нашлось, проверяются строки, ссылающиеся на разные версии одной и той же табличной строки[3]. Из-за многоверсионности при обновлениях может возникать большое количество неактуальных версий строк, которые, скорее всего, достаточно быстро уйдут за горизонт базы данных. Hot-обновления смягчают этот эффект, но работают не v. 14

[1] postgrespro.ru/docs/postgresql/18/btree#BTREE-DELETION.
[2] backend/access/nbtree/README, раздел Simple deletion.
[3] backend/access/nbtree/README, раздел Bottom-Up deletion; include/access/tableam.h.

всегда: если обновляемый столбец входит в какой-нибудь индекс, ссылки на все версии строк появляются и во всех остальных индексах. Перед расщеплением страницы имеет смысл поискать в ней записи, еще не помеченные как мертвые, но уже допускающие очистку. Для этого надо проверить видимость соответствующих версий строк, что требует обращения к табличной странице. Поэтому рассматриваются не все, а только «перспективные» индексные записи, появившиеся как копия существующих для нужд многоверсионности. Заплатить цену проверки видимости таких записей оказывается выгоднее, чем допустить лишнее расщепление индексной страницы.

6

Очистка и заморозка

6.1. Очистка

Самоочистка выполняется быстро, но освобождает только часть места. Она работает в пределах одной табличной страницы и не затрагивает индексы (или наоборот: в пределах одной индексной и не затрагивает таблицу).

Основная, *обычная очистка*[1] выполняется командой `VACUUM`[2]. Она обрабатывает таблицу полностью, вычищая не только ненужные версии строк, но и ссылки на них из всех индексов.

Очистка работает параллельно с другой активностью в системе. Таблица и индексы при этом могут использоваться обычным образом и для чтения, и для изменения (однако одновременное выполнение таких команд, как `CREATE INDEX`, `ALTER TABLE`, и некоторых других будет невозможно). *с. 261*

Чтобы не просматривать лишние страницы, используется карта видимости. *с. 32* Отмеченные в ней страницы пропускаются, поскольку содержат только актуальные версии строк, а вот в остальных могут быть версии, которые можно вычистить. Очистка обновляет карту видимости, если после обработки на странице остаются только версии строк за горизонтом базы данных.

Очистка обновляет и карту свободного пространства, отражая появившееся в страницах свободное место.

[1] postgrespro.ru/docs/postgresql/18/routine-vacuuming.
[2] postgrespro.ru/docs/postgresql/18/sql-vacuum; backend/commands/vacuum.c.

Создадим таблицу и индекс:

```
=> CREATE TABLE vac(
  id integer,
  s char(100)
) WITH (autovacuum_enabled = off);
=> CREATE INDEX vac_s ON vac(s);
```

Здесь с помощью параметра хранения *autovacuum_enabled* отключается автоматическая очистка, чтобы — исключительно для целей эксперимента — точно управлять моментом очистки.

Добавляем строку и выполняем пару обновлений:

```
=> INSERT INTO vac(id,s) VALUES (1,'A');
=> UPDATE vac SET s = 'B';
=> UPDATE vac SET s = 'C';
```

Сейчас в таблице три версии строки:

```
=> SELECT * FROM heap_page('vac',0);
 ctid  | state  | xmin  | xmax  | hhu | hot | t_ctid
-------+--------+-------+-------+-----+-----+--------
 (0,1) | normal | 845 c | 846 c |     |     | (0,2)
 (0,2) | normal | 846 c | 847   |     |     | (0,3)
 (0,3) | normal | 847   | 0 a   |     |     | (0,3)
(3 rows)
```

На каждую из них ведет ссылка из индекса:

```
=> SELECT * FROM index_page('vac_s',1);
 itemoffset | htid  | dead
------------+-------+------
          1 | (0,1) | f
          2 | (0,2) | f
          3 | (0,3) | f
(3 rows)
```

После очистки мертвые версии строк пропадают, и остается только одна, актуальная:

```
=> VACUUM vac;
```

```
=> SELECT * FROM heap_page('vac',0);
 ctid  | state  | xmin  | xmax | hhu | hot | t_ctid
-------+--------+-------+------+-----+-----+--------
 (0,1) | unused |       |      |     |     |
 (0,2) | unused |       |      |     |     |
 (0,3) | normal | 847 c | 0 a  |     |     | (0,3)
(3 rows)
```

При этом два первых указателя, ссылавшиеся на вычищенные версии, получили статус unused, а не dead, как было бы при самоочистке, поскольку на них больше нет ссылок из индекса:

```
=> SELECT * FROM index_page('vac_s',1);
 itemoffset | htid  | dead
------------+-------+------
          1 | (0,3) | f
(1 row)
```

Указатели в статусе unused считаются свободными и могут быть задействованы для новых версий строк.

Страница теперь отмечена в карте видимости, куда мы можем заглянуть с помощью расширения pg_visibility:

```
=> CREATE EXTENSION pg_visibility;
=> SELECT all_visible
FROM pg_visibility_map('vac',0);
 all_visible
-------------
 t
(1 row)
```

Признак того, что все версии строк на странице видны во всех снимках данных, проставляется и в заголовке табличной страницы:

```
=> SELECT flags & 4 > 0 AS all_visible
FROM page_header(get_raw_page('vac',0));
 all_visible
-------------
 t
(1 row)
```

6.2. Еще раз о горизонте базы данных

Определяя, какие версии строк считать мертвыми, очистка ориентируется на горизонт базы данных. Это настолько важное понятие, что к нему следует вернуться еще раз.

Повторим предыдущий опыт с начала:

```
=> TRUNCATE vac;
=> INSERT INTO vac(id,s) VALUES (1,'A');
=> UPDATE vac SET s = 'B';
```

Но перед тем как снова обновить строку, начнем вторую транзакцию, удер-
с. 106 живающую горизонт базы данных (подойдет практически любая транзакция, кроме виртуальной с уровнем изоляции Read Committed). Например, пусть транзакция поменяет что-нибудь в *другой* таблице.

```
=> BEGIN;
=> INSERT INTO accounts(id, client, amount)
  VALUES (1, 'alice', 1000.00);
```

```
=> UPDATE vac SET s = 'C';
```

Сейчас в таблице три версии строки, а в индексе — три ссылки. Что изменится после очистки?

```
=> VACUUM vac;
=> SELECT * FROM heap_page('vac',0);
 ctid  | state  | xmin  | xmax  | hhu | hot | t_ctid
-------+--------+-------+-------+-----+-----+--------
 (0,1) | unused |       |       |     |     |
 (0,2) | normal | 851 c | 853 c |     |     | (0,3)
 (0,3) | normal | 853 c | 0 a   |     |     | (0,3)
(3 rows)
=> SELECT * FROM index_page('vac_s',1);
 itemoffset | htid  | dead
------------+-------+------
          1 | (0,2) | f
          2 | (0,3) | f
(2 rows)
```

Если в прошлый раз в таблице осталась только одна версия строки, то теперь их две: очистка решила, что версия (0,2) еще не может быть удалена. Причина, конечно, в горизонте базы данных, который в нашем примере определяется незавершенной транзакцией:

```
=> SELECT backend_xmin FROM pg_stat_activity
WHERE pid = pg_backend_pid();
 backend_xmin
--------------
          852
(1 row)
```

Можно попросить очистку рассказать о том, что происходит, указав предложение VERBOSE:

```
=> VACUUM VERBOSE vac;
INFO:  vacuuming "internals.public.vac"
INFO:  finished vacuuming "internals.public.vac": index scans: 0
pages: 0 removed, 1 remain, 1 scanned (100.00% of total), 0 eagerly
scanned
tuples: 0 removed, 2 remain, 1 are dead but not yet removable
removable cutoff: 852, which was 2 XIDs old when operation ended
frozen: 0 pages from table (0.00% of total) had 0 tuples frozen
visibility map: 0 pages set all-visible, 0 pages set all-frozen (0 were
all-visible)
index scan not needed: 0 pages from table (0.00% of total) had 0 dead
item identifiers removed
avg read rate: 0.000 MB/s, avg write rate: 0.000 MB/s
buffer usage: 14 hits, 0 reads, 0 dirtied
WAL usage: 0 records, 0 full page images, 0 bytes, 0 buffers full
system usage: CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s
VACUUM
```

Здесь видно, что:

- в таблице не обнаружено версий строк, которые можно было бы удалить (`tuples: 0 removed`);
- найдено две версии, которые удалять нельзя (`2 remain`);
- одна из неудаляемых строк мертвая (`1 are dead but not yet removable`), другая — актуальная;
- текущий горизонт очистки (`removable cutoff`) совпадает с горизонтом открытой транзакции.

Завершение открытой транзакции приводит к тому, что горизонт базы данных продвигается и позволяет продолжить очистку:

```
    => COMMIT;
```

```
=> VACUUM VERBOSE vac;
INFO:  vacuuming "internals.public.vac"
INFO:  finished vacuuming "internals.public.vac": index scans: 1
pages: 0 removed, 1 remain, 1 scanned (100.00% of total), 0 eagerly
scanned
tuples: 1 removed, 1 remain, 0 are dead but not yet removable
removable cutoff: 854, which was 0 XIDs old when operation ended
new relfrozenxid: 853, which is 2 XIDs ahead of previous value
frozen: 0 pages from table (0.00% of total) had 0 tuples frozen
visibility map: 1 pages set all-visible, 0 pages set all-frozen (0 were
all-visible)
index scan needed: 1 pages from table (100.00% of total) had 1 dead
item identifiers removed
index "vac_s": pages: 2 in total, 0 newly deleted, 0 currently deleted,
0 reusable
avg read rate: 0.000 MB/s, avg write rate: 95.274 MB/s
buffer usage: 27 hits, 0 reads, 1 dirtied
WAL usage: 6 records, 1 full page images, 8981 bytes, 0 buffers full
system usage: CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s
VACUUM
```

Очистка обнаружила ушедшую за изменившийся горизонт мертвую версию и удалила ее; в странице осталась только последняя, актуальная версия:

```
=> SELECT * FROM heap_page('vac',0);
 ctid  | state  | xmin  | xmax | hhu | hot | t_ctid
-------+--------+-------+------+-----+-----+--------
 (0,1) | unused |       |      |     |     |
 (0,2) | unused |       |      |     |     |
 (0,3) | normal | 853 c | 0 a  |     |     | (0,3)
(3 rows)
```

В индексе также только одна запись:

```
=> SELECT * FROM index_page('vac_s',1);
 itemoffset | htid  | dead
------------+-------+------
          1 | (0,3) | f
(1 row)
```

6.3. Этапы выполнения очистки

Работа очистки выглядит несложной, но это кажущаяся простота. Ведь и таблицу, и индексы надо обрабатывать одновременно, не блокируя при этом работу остальных процессов. Для этого очистка каждой таблицы выполняется в несколько этапов[1].

Вначале таблица сканируется в поисках мертвых версий строк; найденные версии вычищаются сперва из индекса, а затем из таблицы. Этот процесс может повторяться, если мертвых версий оказалось слишком много. В конце работы таблица по возможности усекается.

Дальше мы поговорим об этих этапах подробнее.

Сканирование таблицы

v. 17

Все начинается со *сканирования таблицы*[2] с учетом карты видимости: отмеченные в карте страницы пропускаются, поскольку гарантированно содержат только актуальные версии строк. Идентификаторы (`tid`) тех версий строк из прочитанных страниц, что выходят за горизонт и больше не нужны, записываются в специальное хранилище в памяти. Сразу удалять эти версии строк нельзя, поскольку на них могут ссылаться индексы.

Хранилище представляет собой префиксное дерево[3]. Ключом в нем служат номера страниц, а в узлах хранятся битовые карты версий строк. Структура ограничена в размерах значением параметра *maintenance_work_mem*; память под нее выделяется по мере необходимости.

с. 605
с. 427
64MB

> По сравнению с обычным массивом, который использовался до PostgreSQL 17, префиксное дерево хранит идентификаторы на порядок компактнее, а его общий размер может превышать 1 Гбайт — максимальный размер фрагмента, выделяемого в памяти за один раз.

[1] backend/access/heap/vacuumlazy.c, функция heap_vacuum_rel.
[2] backend/access/heap/vacuumlazy.c, функция lazy_scan_heap.
[3] backend/access/common/tidstore.c.

Очистка индексов

Дальше возможно одно из двух: либо сканирование доходит до конца таблицы, либо заканчивается память, выделенная под идентификаторы. В любом из этих случаев начинается *этап очистки индексов*[1]. Чтобы очистить индексные страницы от записей, ведущих на запомненные версии строк, *каждый* из индексов, созданных для таблицы, сканируется *полностью*.

> Индекс позволяет быстро найти версию строки, если известен ключ индексирования, но нет способа быстро найти индексную строку по идентификатору табличной версии. Работа в этом направлении для B-деревьев пока не привела к успеху[2].

v. 13 512kB Для каждого индекса, размер которого превышает значение параметра *min_parallel_index_scan_size*, может запускаться отдельный рабочий процесс; в этом случае очистка нескольких индексов будет происходить в параллельном режиме. Желаемое количество процессов очистки можно указать в команде `VACUUM (parallel N)`. В реальности их будет запущено не больше, чем имеется свободных фоновых процессов[3] и подходящих индексов (поскольку один индекс не может обрабатываться несколькими процессами одновременно).

На этапе очистки индексов выполняется работа по поддержанию карты свободного пространства в актуальном состоянии и вычислению статистики работы очистки. Но этот этап будет пропущен, если строки только добавляются (но не удаляются и не изменяются), поскольку в таком случае в таблице не будет ни одной мертвой версии. Тогда очистка индексов выполнится принудительно один раз в конце (это называется *«уборкой» индексов*[4]).

Итак, после этого этапа в индексах уже нет ссылок на ненужные версии строк, но сами версии еще присутствуют в таблице. Это нормальная ситуация: при индексном доступе на них уже невозможно попасть, а при последовательном сканировании таблицы мертвые версии будут отметены проверкой видимости.

[1] backend/access/heap/vacuumlazy.c, функция lazy_vacuum_all_indexes.
[2] commitfest.postgresql.org/21/1802.
[3] postgrespro.ru/docs/postgresql/18/bgworker.
[4] backend/access/heap/vacuumlazy.c, функция lazy_cleanup_all_indexes; backend/access/nbtree/nbtree.c, функция btvacuumcleanup.

Очистка таблицы

Затем начинается *этап очистки таблицы*[1]. Таблица снова сканируется: это необходимо, чтобы вычистить из страниц запомненные ранее версии строк и освободить указатели. Теперь, когда ссылок из индексов уже нет, это можно сделать безопасно.

Освобожденное при очистке место записывается в карту свободного пространства, а страницы, на которых остались только актуальные версии, видимые во всех снимках, отмечаются в карте видимости.

Если на этапе сканирования таблица не была прочитана полностью, то хранилище `tid` очищается, и все повторяется с того момента, на котором сканирование таблицы остановилось в прошлый раз.

Усечение таблицы

После очистки в табличных страницах появляется свободное место; иногда может посчастливиться освободить страницу целиком. Если в конце файла образовалось некоторое количество пустых страниц, очистка может «откусить» хвостовую часть файла и вернуть место операционной системе. Это происходит на заключительном *этапе усечения таблицы*[2].

Усечение требует кратковременной исключительной блокировки таблицы. *с. 261*
Попытки получить блокировку продолжаются не дольше пяти секунд, чтобы не останавливать надолго работу других процессов.

Из-за необходимости блокировать работу с таблицей усечение выполняется, только если пустых страниц достаточно много: как минимум $\frac{1}{16}$ часть файла или, для больших таблиц, 1000 страниц (эти пороговые значения не настраиваются).

Есть несколько способов полностью отключить усечение, которые могут пригодиться, если блокировка все равно вызывает проблемы:

- вызвать команду `VACUUM` с указанием `truncate off`; v. 12

[1] backend/access/heap/vacuumlazy.c, функция lazy_vacuum_heap.
[2] backend/access/heap/vacuumlazy.c, функция lazy_truncate_heap.

v. 12 • сбросить параметры хранения *vacuum_truncate* и *toast.vacuum_truncate*;

v. 18 • сбросить конфигурационный параметр *vacuum_truncate*.

6.4. Заморозка

Переполнение счетчика транзакций

Под номер транзакции в PostgreSQL выделено 32 бита. Четыре миллиарда выглядит довольно большим числом, но при активной работе оно может быть израсходовано достаточно быстро. Например, при нагрузке 1000 транзакций в секунду (не считая виртуальных) это произойдет всего через полтора месяца непрерывной работы.

После исчерпания разрядности счетчик должен будет обнулиться и пойти по следующему кругу (по-английски это называется словом wraparound). Но ведь считать, что транзакция с меньшим номером началась раньше транзакции с бóльшим номером, можно только в случае постоянно возрастающих номеров. Поэтому понятно, что просто так обнулить счетчик и начать нумерацию заново нельзя.

Почему под номер транзакции не выделено 64 бита — ведь это полностью исключило бы проблему? Дело в том, что в заголовке каждой версии строки
с. 75 хранятся два номера транзакций — `xmin` и `xmax`. Заголовок и так достаточно большой (минимум 24 байта с учетом выравнивания), а увеличение разрядности добавило бы еще 8 байт.

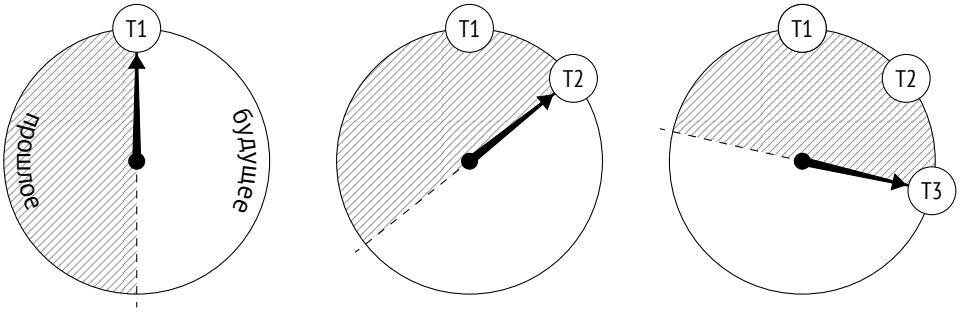
> 64-битные номера транзакций, добавляющие к обычному номеру 32-битную «эпоху», тоже используются[1], но только для служебных целей и никогда не попадают в страницы данных.

Чтобы корректно обрабатывать переполнение счетчика, надо сравнивать не номера транзакций, а их возраст (выраженный в количестве транзакций, начатых с момента появления данной транзакции). Таким образом, вместо понятий «меньше» и «больше» используются «предшествует» («старше») и «следует за» («младше»).

[1] include/access/transam.h, тип FullTransactionId.

В коде это естественным образом реализуется 32-битной арифметикой: надо найти разницу 32-битных номеров транзакций и сравнить полученный результат с нулем[1].

Графически можно представить, что номера транзакций закольцованы. Тогда для любой транзакции половина номеров против часовой стрелки будет старше (в прошлом), а половина по часовой стрелке — младше (в будущем).



Однако в такой закольцованной схеме возникает неприятная ситуация. Старая транзакция (T1) находится для новых транзакций в далеком прошлом. Но через некоторое время для очередной транзакции она окажется в той половине круга, которая относится к будущему. Если бы это было так, то привело бы к катастрофическим последствиям — изменения, сделанные транзакцией T1 давным-давно, оказались бы невидимыми для новых транзакций.

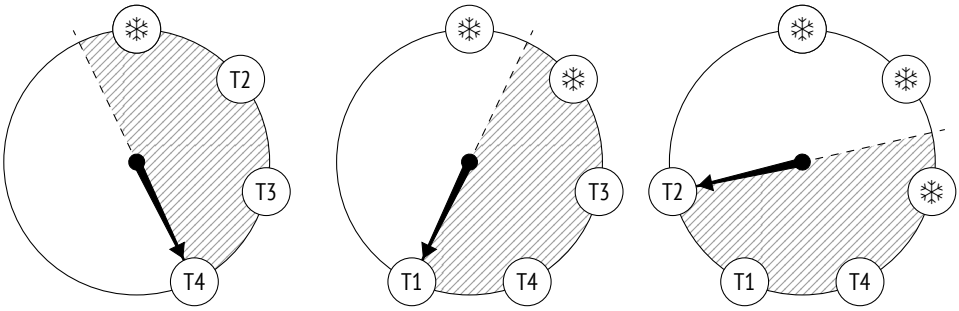
Заморозка версий и правила видимости

Чтобы не допустить таких «путешествий во времени», процесс очистки (помимо освобождения места в страницах) выполняет еще одну задачу[2]. Он ищет версии строк, находящиеся за горизонтом базы данных (видимые во всех снимках), и специальным образом их помечает — *замораживает*. *с. 274*

Замороженная версия строки видна во всех снимках данных без оглядки на номер транзакции `xmin`, поэтому такой номер может быть безопасно использован заново.

[1] backend/access/transam/transam.c, функция TransactionIdPrecedes.
[2] postgrespro.ru/docs/postgresql/18/routine-vacuuming#VACUUM-FOR-WRAPAROUND.

Можно считать, что в замороженных версиях строк номер xmin заменяется на условную «минус бесконечность» (показанную на рисунке в виде снежинки) — признак того, что создавшая версию транзакция завершилась так давно, что конкретный номер уже не важен. Хотя на самом деле xmin не меняется, а в качестве признака заморозки выставляются одновременно два бита-подсказки — бит фиксации и бит отмены.



Многие источники (включая документацию) упоминают специальный номер транзакции FrozenTransactionId = 2. Это и есть та самая «минус бесконечность», на которую заменялся xmin до версии 9.4. Используемые сейчас биты-подсказки позволяют сохранить в версии строки исходный номер транзакции, что удобно для поддержки и отладки. Однако транзакции с номером 2 еще могут встретиться в страницах данных старых систем, даже обновленных до последних версий.

Номер транзакции xmax никак не участвует в заморозке. Он фигурирует только в неактуальных версиях строк, а когда такие версии перестанут быть видимы в снимках (то есть номер уйдет за горизонт базы данных), они будут удалены очисткой.

Вручную заморозка вызывается командой VACUUM FREEZE. В нашем примере она замораживает единственную версию строки:

```
=> VACUUM FREEZE vac;
=> SELECT * FROM heap_page('vac',0);
 ctid  | state  | xmin  | xmax | hhu | hot | t_ctid
-------+--------+-------+------+-----+-----+--------
 (0,1) | unused |       |      |     |     |
 (0,2) | unused |       |      |     |     |
 (0,3) | normal | 853 f | 0 a  |     |     | (0,3)
(3 rows)
```

Страница теперь отмечена не только в карте видимости, но и в карте заморозки:

```
=> SELECT all_visible, all_frozen
FROM pg_visibility_map('vac',0);
 all_visible | all_frozen
-------------+------------
 t           | t
(1 row)
```

6.5. Анализ

В дополнение к очистке есть еще одна задача, никак с ней формально не связанная: *анализ*[1], или, иными словами, сбор статистической информации. Бóльшая часть статистики предназначается для планировщика запросов: количество строк (pg_class.reltuples) и страниц (pg_class.relpages) в отношениях, распределение данных по столбцам таблиц и другие сведения. Такие значения, как количество страниц, отмеченных в карте видимости (pg_class.relallvisible) и в карте заморозки (pg_class.relallfrozen), не используются планировщиком, но нужны для управления самим анализом и очисткой.

с. 346

Вручную анализ можно выполнять сам по себе командой ANALYZE[2], но можно и совмещать с очисткой: VACUUM ANALYZE. Правда, при этом очистка и анализ выполняются последовательно — никакой экономии не происходит.

> Исторически вариант VACUUM ANALYZE возник первым, в версии 6.1, а отдельная команда ANALYZE появилась только в версии 7.2. В более ранних версиях статистика обновлялась скриптом на TCL.

[1] postgrespro.ru/docs/postgresql/18/routine-vacuuming#VACUUM-FOR-STATISTICS.
[2] backend/commands/analyze.c.

7

Автоочистка

7.1. Автоматическая очистка и анализ

Когда горизонт базы данных регулярно продвигается вперед, обычная очистка должна справляться со своей работой. Вопрос в том, как часто ее следует вызывать.

Если очищать изменяющуюся таблицу слишком редко, она вырастет в размерах больше, чем хотелось бы. Кроме того, в ней может накопиться слишком много изменений, и тогда очередной очистке потребуется совершить несколько проходов по индексам.

Если очищать таблицу слишком часто, то вместо полезной работы сервер будет постоянно заниматься обслуживанием.

К тому же нагрузка может изменяться со временем, так что запуск очистки по какому-либо фиксированному расписанию в любом случае не годится: чем чаще обновляется таблица, тем чаще ее надо и очищать.

Задача решается *автоматической очисткой*[1] — механизмом, который подстраивает частоту срабатывания очистки и анализа под скорость изменения таблиц.

Устройство автоочистки

При включенной автоматической очистке (конфигурационный параметр `on` *autovacuum*) в системе всегда присутствует процесс autovacuum launcher,

[1] postgrespro.ru/docs/postgresql/18/routine-vacuuming#AUTOVACUUM.

ответственный за расписание. Он поддерживает список баз данных, в которых есть какая-либо активность, опираясь на статистику их использования. Такая статистика собирается при установленном параметре *track_counts*. Не следует выключать эти параметры, иначе автоочистка не будет работать.

on

Раз в *autovacuum_naptime* единиц времени процесс autovacuum launcher запускает (как обычно, с помощью postmaster) рабочий процесс autovacuum worker[1] для каждой «активной» базы данных из списка. Соответственно, если в кластере используется несколько баз данных (*N* штук), то за интервал *autovacuum_naptime* будет запущено *N* рабочих процессов. Но при этом общее количество одновременно выполняющихся рабочих процессов ограничено значением параметра *autovacuum_max_workers*.

1min

3

Ограничение количества рабочих процессов можно менять на лету, но только в пределах *autovacuum_worker_slots*. А изменение количества слотов уже требует перезапуска сервера, поскольку резервирует место в его разделяемой памяти.

v. 18

16

> Рабочие процессы автоочистки очень напоминают фоновые рабочие процессы, но появились задолго до того, как был создан этот механизм заданий общего вида. Реализацию процессов автоочистки не стали менять, поэтому они не задействуют слоты из *max_worker_processes*.

с. 379

Запустившись, рабочий процесс подключается к указанной ему базе данных и начинает с того, что строит два списка:

- список всех таблиц, материализованных представлений и toast-таблиц, требующих очистки;
- список всех таблиц и материализованных представлений, требующих анализа (toast-таблицы не анализируются, потому что обращение к ним всегда происходит по индексу).

Затем рабочий процесс по очереди очищает или анализирует отобранные объекты (или выполняет и то, и другое), после чего завершается.

[1] backend/postmaster/autovacuum.c.

Очистка каждой таблицы происходит так же, как при запуске команды VACUUM вручную. Но есть некоторые отличия:

- Очистка, запускаемая вручную, использует для накопления идентификаторов версий фрагмент памяти размером *maintenance_work_mem*.

  При автоматической очистке одновременно выполняются несколько рабочих процессов, и оперативная память нужна каждому. Поэтому для
-1 автоочистки предусмотрен отдельный параметр *autovacuum_work_mem*. Значение по умолчанию, при котором используется обычное ограничение *maintenance_work_mem*, скорее всего, следует скорректировать, если конфигурация предполагает большое количество рабочих процессов (*autovacuum_max_workers*).

- Очистка, запускаемая вручную, может обрабатывать несколько индексов на одной таблице параллельно, а автоматическая очистка — нет, поскольку это могло бы привести к слишком большому числу одновременно запущенных процессов.

Если процесс не успеет выполнить всю намеченную работу за интервал времени *autovacuum_naptime*, процесс autovacuum launcher пошлет в ту же базу данных еще один рабочий процесс, и они будут работать вместе. Второй процесс построит свои собственные списки объектов для очистки и анализа и пойдет по ним. Параллельно будут обрабатываться *разные* таблицы; на уровне одной таблицы параллелизма нет.

Какие таблицы требуют очистки

Автоочистку можно отключить на уровне отдельных таблиц, хотя сложно придумать причину, по которой это было бы необходимо. За отключение отвечают два параметра хранения: один для простых таблиц, другой — для toast-таблиц:

- *autovacuum_enabled*;
- *toast.autovacuum_enabled*.

с. 153 Обычно же автоочистка срабатывает при накоплении либо неактуальных версий, либо новых строк.

Неактуальные версии строк. Примерное число неактуальных версий постоянно отслеживается и доступно в таблице `pg_stat_all_tables`.

Считается, что очистка необходима, если количество неактуальных, мертвых версий строк превышает пороговое значение, определяемое тремя параметрами:

- *autovacuum_vacuum_threshold* — абсолютное значение (в штуках); 50
- *autovacuum_vacuum_scale_factor* — доля строк в таблице; 0.2
- *autovacuum_vacuum_max_threshold* — дополнительный порог для очень больших таблиц. 100 млн, v. 18

Формула такова: очистка мертвых версий требуется, если

$$\texttt{pg_stat_all_tables.n_dead_tup} > \textit{autovacuum_vacuum_threshold} + \textit{autovacuum_vacuum_scale_factor} \times \texttt{pg_class.reltuples}$$

или

$$\texttt{pg_stat_all_tables.n_dead_tup} > \textit{autovacuum_vacuum_max_threshold}.$$ v. 18

В первой части формулы для больших таблиц (а именно с ними и связаны возможные проблемы) важен параметр *autovacuum_vacuum_scale_factor*. Значение по умолчанию 20 % представляется сильно завышенным; скорее всего, его потребуется существенно уменьшить. Если и после этого пороговое значение слишком велико для очень больших таблиц, его можно ограничить сверху параметром из второй части формулы.

Если не удается подобрать универсальные значения параметров, их можно переопределить для отдельных таблиц с помощью одноименных параметров хранения (*autovacuum_vacuum_threshold*, *autovacuum_vacuum_scale_factor*, *autovacuum_vacuum_max_threshold*). Аналогичные параметры хранения с префиксом *toast* определяют настройки для toast-таблиц.

v. 13 **Вставленные строки.** Для хранилищ данных типична ситуация, когда строки добавляются, но никогда не удаляются и не изменяются. В этом случае в таблице не будет мертвых версий и предыдущее условие срабатывания не будет выполняться. Однако и такие таблицы необходимо очищать, чтобы заблаго-
с. 422 временно замораживать версии строк и обновлять карту видимости (и тем самым разрешать сканирование только индекса). Поэтому добавляется еще одно условие.

Очистка будет выполняться, если число строк, вставленных с момента последней очистки, превышает пороговое значение, определяемое парой параметров:

1000 • *autovacuum_vacuum_insert_threshold*;

0.2 • *autovacuum_vacuum_insert_scale_factor*.

Как и число неактуальных версий, количество вставленных строк отслеживается как часть статистики использования таблицы.

Формула такова:

$$\texttt{pg_stat_all_tables.n_ins_since_vacuum} > \textit{autovacuum_vacuum_insert_threshold} + \textit{autovacuum_vacuum_insert_scale_factor} \times \texttt{pg_class.reltuples} \times$$
v. 18 $$\left(1 - \texttt{pg_class.relallfrozen} \,/\, \texttt{pg_class.relpages}\right).$$

Если с первым слагаемым все понятно — это просто абсолютное значение, — то ко второму необходим комментарий. В нем вычисляется доля не от всех, а только от *незамороженных* строк; соответствующая часть таблицы оценивается отношением числа полностью замороженных страниц (`relallfrozen`) к их общему количеству (`relpages`). Дело в том, что с ростом таблицы строк в ней становится все больше и больше, и без коэффициента очистка выполнялась бы все реже и реже. А вот количество незамороженных строк периодически уменьшается, поэтому частота срабатываний не увеличивается — что хорошо.

Как и в предыдущем случае, мы можем переопределить значения параметров для отдельных таблиц с помощью одноименных параметров хранения (*autovacuum_vacuum_insert_threshold* и *autovacuum_vacuum_insert_scale_factor*) и их toast-версий.

Какие таблицы требуют анализа

С автоанализом дело обстоит проще, чем с очисткой, потому что учитываются только измененные строки.

Считается, что анализа требуют те таблицы, в которых число измененных строк (с момента прошлого анализа) превышает пороговое значение, заданное парой параметров:

- *autovacuum_analyze_threshold*; 50
- *autovacuum_analyze_scale_factor*. 0.1

Формула срабатывания автоанализа:

$$\texttt{pg_stat_all_tables.n_mod_since_analyze} > \textit{autovacuum_analyze_threshold} + \textit{autovacuum_analyze_scale_factor} \times \texttt{pg_class.reltuples}.$$

Настройки автоанализа также можно переопределить на уровне таблиц с помощью одноименных параметров хранения (*autovacuum_analyze_threshold* и *autovacuum_analyze_scale_factor*). Поскольку toast-таблицы не анализируются, для них соответствующих параметров нет.

Автоочистка в действии

Начнем с написания функции, возвращающей текущее значение параметра:

```
=> CREATE FUNCTION p(param text, c pg_class) RETURNS float
AS $$
  SELECT coalesce(
    -- если параметр хранения задан, то берем его
    (SELECT option_value
     FROM   pg_options_to_table(c.reloptions)
     WHERE  option_name = CASE -- для toast-таблиц имя отличается
              WHEN c.relkind = 't' THEN 'toast.' ELSE ''
            END || param
    ),
    -- иначе берем значение конфигурационного параметра
    current_setting(param)
  )::float;
$$ LANGUAGE sql;
```

Мы используем эту функцию в представлениях, которые покажут нам, какие таблицы в данный момент нуждаются в очистке и анализе[1]. Представления пригодятся в примерах, а заодно формализуют все сказанное выше в этом разделе.

Так может выглядеть представление для очистки:

```
=> CREATE VIEW need_vacuum AS
WITH c AS (
  SELECT c.oid,
    greatest(c.reltuples, 0) AS reltuples,
    CASE
      WHEN c.relpages > 0 THEN
        1 - c.relallfrozen::float / c.relpages
      ELSE 1
    END AS pct_unfrozen,
    p('autovacuum_vacuum_threshold', c) AS threshold,
    p('autovacuum_vacuum_scale_factor', c) AS scale_factor,
    p('autovacuum_vacuum_max_threshold', c) AS max_threshold,
    p('autovacuum_vacuum_insert_threshold', c) AS ins_threshold,
    p('autovacuum_vacuum_insert_scale_factor', c) AS ins_scale_factor
  FROM pg_class c
  WHERE c.relkind IN ('r','m','t')
),
r AS (
  SELECT st.schemaname || '.' || st.relname AS tablename,
    st.n_dead_tup AS dead_tup,
    least(
      c.threshold + c.scale_factor * c.reltuples,
      c.max_threshold
    ) AS max_dead_tup,
    st.n_ins_since_vacuum AS ins_tup,
    c.ins_threshold + c.ins_scale_factor * c.reltuples * c.pct_unfrozen
      AS max_ins_tup,
    st.last_autovacuum
  FROM pg_stat_all_tables st
    JOIN c ON c.oid = st.relid
)
SELECT *,
  CASE
    WHEN dead_tup > max_dead_tup OR ins_tup > max_ins_tup
      THEN 'Пора!'
  END AS is_it_time
FROM r;
```

[1] backend/postmaster/autovacuum.c, функция relation_needs_vacanalyze.

Столбец max_dead_tup показывает пороговое значение срабатывания автоочистки из-за мертвых версий строк, а столбец max_ins_tup — из-за вставки новых строк.

А вот аналогичное представление для анализа:

```
=> CREATE VIEW need_analyze AS
WITH c AS (
  SELECT c.oid,
    greatest(c.reltuples, 0) AS reltuples,
    p('autovacuum_analyze_threshold', c) AS threshold,
    p('autovacuum_analyze_scale_factor', c) AS scale_factor
  FROM pg_class c
  WHERE c.relkind IN ('r','m')
),
r AS (
  SELECT st.schemaname || '.' || st.relname AS tablename,
    st.last_autoanalyze,
    st.n_mod_since_analyze AS mod_tup,
    c.threshold + c.scale_factor * c.reltuples AS max_mod_tup
  FROM pg_stat_all_tables st
    JOIN c ON c.oid = st.relid
)
SELECT *,
  CASE
    WHEN mod_tup > max_mod_tup THEN 'Пора!'
  END AS is_it_time
FROM r;
```

Столбец max_mod_tup показывает пороговое значение срабатывания автоанализа.

Чтобы не ждать долго автоочистку, для экспериментов будем запускать процесс каждую секунду:

```
=> ALTER SYSTEM SET autovacuum_naptime = '1s';
=> SELECT pg_reload_conf();
```

Опустошим таблицу vac и вставим в нее тысячу строк. Напомню, что автоочистка отключена на уровне таблицы.

```
=> TRUNCATE TABLE vac;
=> INSERT INTO vac(id,s)
  SELECT id, 'A' FROM generate_series(1,1000) id;
```

Вот что покажет наше представление для очистки:

```
=> SELECT * FROM need_vacuum WHERE tablename = 'public.vac' \gx
-[ RECORD 1 ]---+-----------
tablename       | public.vac
dead_tup        | 0
max_dead_tup    | 50
ins_tup         | 1000
max_ins_tup     | 1000
last_autovacuum |
is_it_time      |
```

Пороговое значение `max_dead_tup` = 50, хотя по приведенной выше формуле должно быть $50 + 0{,}2 \times 1000 = 250$. Дело в том, что еще нет статистики по таблице, поскольку команда `INSERT` сама по себе ее не обновляет:

```
=> SELECT reltuples FROM pg_class WHERE relname = 'vac';
 reltuples
-----------
        -1
(1 row)
```

v. 14 Специальная константа −1 для `pg_class.reltuples` позволяет отличать таблицу без статистики от пустой, но проанализированной таблицы, для которой этот столбец будет равен нулю. Для расчетов отрицательное значение приводится к нулю, что и дает $50 + 0{,}2 \times 0 = 50$.

Значение `max_ins_tup` = 1000 отличается от расчетного числа 1200 по той же причине.

Заглянем в представление для анализа:

```
=> SELECT * FROM need_analyze WHERE tablename = 'public.vac' \gx
-[ RECORD 1 ]----+-----------
tablename        | public.vac
last_autoanalyze |
mod_tup          | 1006
max_mod_tup      | 50
is_it_time       | Пора!
```

Мы изменили тысячу строк (в данном случае они были добавлены), и это больше порога, который пока равен 50 с учетом неизвестного размера таблицы. Значит, автоанализ должен сработать.

Включим его:

```
=> ALTER TABLE vac SET (autovacuum_enabled = on);
```

После небольшой паузы таблица оказывается проанализированной, пороговое значение исправляется на корректные 150 строк:

```
=> SELECT reltuples FROM pg_class WHERE relname = 'vac';
 reltuples
-----------
      1000
(1 row)
=> SELECT * FROM need_analyze WHERE tablename = 'public.vac' \gx
-[ RECORD 1 ]----+------------------------------
tablename        | public.vac
last_autoanalyze | 2026-04-05 20:52:06.591521+03
mod_tup          | 0
max_mod_tup      | 150
is_it_time       |
```

Вернемся к автоочистке:

```
=> SELECT * FROM need_vacuum WHERE tablename = 'public.vac' \gx
-[ RECORD 1 ]---+-----------
tablename       | public.vac
dead_tup        | 0
max_dead_tup    | 250
ins_tup         | 1000
max_ins_tup     | 1200
last_autovacuum |
is_it_time      |
```

Значения `max_dead_tup` и `max_ins_tup` тоже исправились с учетом выясненного в процессе анализа размера таблицы.

На данный момент очистка может состояться в одном из двух случаев:

- при появлении более 250 мертвых версий строк;
- при вставке в таблицу более 200 строк. v. 13

Снова отключим автоочистку и обновим 251 строку — на одну больше, чем пороговое значение:

```
=> ALTER TABLE vac SET (autovacuum_enabled = off);
=> UPDATE vac SET s = 'B' WHERE id % 4 = 0 OR id = 1;
UPDATE 251
=> SELECT * FROM need_vacuum WHERE tablename = 'public.vac' \gx
-[ RECORD 1 ]---+-----------
tablename       | public.vac
dead_tup        | 251
max_dead_tup    | 250
ins_tup         | 1000
max_ins_tup     | 1200
last_autovacuum |
is_it_time      | Пора!
```

Теперь условие срабатывания выполняется. Включим автоочистку и после непродолжительной паузы обнаружим, что таблица обработана, а статистика использования сброшена:

```
=> ALTER TABLE vac SET (autovacuum_enabled = on);
=> SELECT * FROM need_vacuum WHERE tablename = 'public.vac' \gx
-[ RECORD 1 ]---+------------------------------
tablename       | public.vac
dead_tup        | 0
max_dead_tup    | 250
ins_tup         | 0
max_ins_tup     | 1200
last_autovacuum | 2026-04-05 20:52:11.763126+03
is_it_time      |
```

7.2. Заморозка

Для экспериментов с заморозкой создадим новую таблицу. Установим для нее минимальное значение параметра *fillfactor* так, чтобы на каждой странице помещалось всего две строки — так будет удобнее наблюдать за происходящим. И как и раньше, отключим автоматику, чтобы управлять временем очистки самостоятельно.

```
=> CREATE TABLE tfreeze(
  id integer,
  s char(300)
) WITH (fillfactor = 10, autovacuum_enabled = off);
```

Очередной вариант функции для просмотра страниц с помощью расширения pageinspect будет отображать диапазон страниц, расшифровывать признак заморозки (f) и показывать возраст транзакции xmin (для этого используется системная функция age; сам возраст, конечно, не хранится в страницах данных):

```
=> CREATE FUNCTION heap_page(
  relname text, pageno_from integer, pageno_to integer
)
RETURNS TABLE(
  ctid tid, state text,
  xmin text, xmin_age integer, xmax text
) AS $$
SELECT (pageno,lp)::text::tid AS ctid,
       CASE lp_flags
         WHEN 0 THEN 'unused'
         WHEN 1 THEN 'normal'
         WHEN 2 THEN 'redirect to '||lp_off
         WHEN 3 THEN 'dead'
       END AS state,
       t_xmin || CASE
         WHEN (t_infomask & 256+512) = 256+512 THEN ' f'
         WHEN (t_infomask & 256) > 0 THEN ' c'
         WHEN (t_infomask & 512) > 0 THEN ' a'
         ELSE ''
       END AS xmin,
       age(t_xmin) AS xmin_age,
       t_xmax || CASE
         WHEN (t_infomask & 1024) > 0 THEN ' c'
         WHEN (t_infomask & 2048) > 0 THEN ' a'
         ELSE ''
       END AS xmax
FROM generate_series(pageno_from, pageno_to) p(pageno),
     heap_page_items(get_raw_page(relname, pageno))
ORDER BY pageno, lp;
$$ LANGUAGE sql;
```

Вставляем в таблицу некоторое количество строк и сразу выполняем очистку, чтобы создалась карта видимости:

```
=> CREATE EXTENSION IF NOT EXISTS pg_visibility;
=> INSERT INTO tfreeze(id, s)
  SELECT id, 'FOO'||id FROM generate_series(1,100) id;
INSERT 0 100
=> VACUUM tfreeze;
```

Будем наблюдать за первыми двумя страницами с помощью расширения pg_visibility. После очистки обе страницы оказываются отмеченными в карте видимости (all_visible), но не в карте заморозки (all_frozen) — они еще содержат незамороженные версии строк:

```
=> SELECT *
FROM generate_series(0,1) g(blkno),
     pg_visibility_map('tfreeze',g.blkno)
ORDER BY g.blkno;
 blkno | all_visible | all_frozen
-------+-------------+------------
     0 | t           | f
     1 | t           | f
(2 rows)
```

У транзакции, создавшей строки, возраст xmin_age равен 1, поскольку это последняя транзакция, которая выполнялась в системе:

```
=> SELECT * FROM heap_page('tfreeze',0,1);
 ctid  | state  | xmin  | xmin_age | xmax
-------+--------+-------+----------+------
 (0,1) | normal | 877 c |        1 | 0 a
 (0,2) | normal | 877 c |        1 | 0 a
 (1,1) | normal | 877 c |        1 | 0 a
 (1,2) | normal | 877 c |        1 | 0 a
(4 rows)
```

7.3. Управление заморозкой

Заморозкой управляют четыре основных конфигурационных параметра. Значения всех четырех представляют возраст транзакций и определяют моменты, начиная с которых:

- версии начинают замораживаться — *vacuum_freeze_min_age*;
- выполняется агрессивная заморозка — *vacuum_freeze_table_age*;
- заморозка срабатывает аварийно — *autovacuum_freeze_max_age*;
- заморозка работает в приоритетном режиме — *vacuum_failsafe_age*. v. 14

Минимальный возраст для заморозки

Минимальный возраст транзакции xmin, при котором версию строки можно замораживать, определяется параметром *vacuum_freeze_min_age*. Чем меньше это значение, тем сильнее могут вырасти накладные расходы: если строка окажется «горячей» и будет активно изменяться, то заморозка все новых и новых версий будет пропадать без пользы. Установка относительно большого значения параметра позволяет выждать. 50 млн

> Если вызвать заморозку вручную командой VACUUM FREEZE, будут заморожены все версии строк без оглядки на возраст транзакций, как будто *vacuum_freeze_min_age* = 0. Именно поэтому в рассмотренном ранее примере была заморожена буквально только что появившаяся версия. *с. 136*

Чтобы посмотреть, как происходит заморозка, уменьшим значение до единицы:

```
=> ALTER SYSTEM SET vacuum_freeze_min_age = 1;
=> SELECT pg_reload_conf();
```

Обновим теперь одну строку на нулевой странице. Новая версия попадет на ту же страницу благодаря небольшому значению *fillfactor*:

```
=> UPDATE tfreeze SET s = 'BAR' WHERE id = 1;
```

Возраст транзакций увеличился на единицу. Табличные страницы выглядят сейчас следующим образом:

```
=> SELECT * FROM heap_page('tfreeze',0,1);
 ctid  | state  | xmin  | xmin_age | xmax
-------+--------+-------+----------+------
 (0,1) | normal | 877 c |        2 | 878
 (0,2) | normal | 877 c |        2 | 0 a
 (0,3) | normal | 878   |        1 | 0 a
 (1,1) | normal | 877 c |        2 | 0 a
 (1,2) | normal | 877 c |        2 | 0 a
(5 rows)
```

Теперь строки старше *vacuum_freeze_min_age* = 1 подлежат заморозке. Но очистка рассматривает только те страницы, которые не отмечены в карте видимости: *с. 131*

```
=> SELECT *
FROM generate_series(0,1) g(blkno),
     pg_visibility_map('tfreeze',g.blkno)
ORDER BY g.blkno;
 blkno | all_visible | all_frozen
-------+-------------+------------
     0 | f           | f
     1 | t           | f
(2 rows)
```

> Очистка пропускает страницы, отмеченные в карте видимости, только если подряд идут минимум 32 такие страницы[1]. Если же признаки чередуются чаще, карта видимости игнорируется: пропускать отдельные страницы не имеет большого смысла, поскольку они все равно окажутся в оперативной памяти благодаря предвыборке. Именно поэтому таблица `tfreeze` создана достаточно большой, хотя нас интересуют только две страницы.

Предыдущая команда `UPDATE` сбросила бит видимости нулевой страницы, а вот первую страницу очистка пропустит:

```
=> VACUUM tfreeze;
VACUUM
=> SELECT * FROM heap_page('tfreeze',0,1);
 ctid  |     state     | xmin  | xmin_age | xmax
-------+---------------+-------+----------+------
 (0,1) | redirect to 3 |       |          |
 (0,2) | normal        | 877 f |        2 | 0 a
 (0,3) | normal        | 878 f |        1 | 0 a
 (1,1) | normal        | 877 c |        2 | 0 a
 (1,2) | normal        | 877 c |        2 | 0 a
(5 rows)
```

v. 16 Обратите внимание, что на нулевой странице были заморожены обе версии строк, хотя возраст транзакции `xmin` во второй версии не превышает значение параметра *vacuum_freeze_min_age*. Поскольку страницу все равно приходится менять, на ней замораживаются все версии строк за горизонтом базы данных — это практически не увеличивает стоимость операции, но может сэкономить усилия в будущем.

[1] backend/access/heap/vacuumlazy.c, функция heap_vac_scan_next_block.

Теперь обе страницы снова отмечены в карте видимости. Несмотря на то что первая страница содержит незамороженные строки, очистка будет пропускать ее, если в ней ничего не изменится:

```
=> SELECT * FROM generate_series(0,1) g(blkno),
     pg_visibility_map('tfreeze',g.blkno)
ORDER BY g.blkno;
 blkno | all_visible | all_frozen
-------+-------------+------------
     0 | t           | t
     1 | t           | f
(2 rows)
```

Возраст для агрессивной заморозки

Итак, если на странице остались только актуальные версии, видимые во всех снимках, очистка не заморозит их. Справиться с проблемой позволяет параметр *vacuum_freeze_table_age*. Его значение определяет возраст транзакции, при котором очистка игнорирует карту видимости и замораживает все страницы, предотвращая последствия переполнения счетчика транзакций. Такой режим очистки называется агрессивным; поскольку его основная задача состоит в заморозке, я буду употреблять и термин «агрессивная заморозка».

150 млн

с. 134

Для каждой таблицы в системном каталоге хранится номер транзакции, про которую известно, что все более старые транзакции гарантированно заморожены:

```
=> SELECT relfrozenxid, age(relfrozenxid)
FROM pg_class WHERE relname = 'tfreeze';
 relfrozenxid | age
--------------+-----
          877 |   2
(1 row)
```

С возрастом этой запомненной транзакции и сравнивается значение параметра *vacuum_freeze_table_age* для принятия решения об агрессивной заморозке.

Благодаря карте заморозки нет необходимости полностью сканировать таблицу при очистке; достаточно просмотреть только те страницы, которые

еще не отмечены в карте. Карта заморозки не только существенно уменьшает объем работы, но и дает устойчивость к прерываниям: если процесс очистки остановить и начать заново, ему не придется возвращаться к страницам, которые уже были обработаны и отмечены в карте в прошлый раз.

Для заморозки всей таблицы уменьшим значение *vacuum_freeze_table_age* до двух, чтобы выполнилось условие агрессивной заморозки:

```
=> ALTER SYSTEM SET vacuum_freeze_table_age = 2;
=> SELECT pg_reload_conf();
```

Выполним очистку:

```
=> VACUUM VERBOSE tfreeze;
INFO:  aggressively vacuuming "internals.public.tfreeze"
INFO:  finished vacuuming "internals.public.tfreeze": index scans: 0
pages: 0 removed, 50 remain, 50 scanned (100.00% of total), 0 eagerly
scanned
tuples: 0 removed, 100 remain, 0 are dead but not yet removable
removable cutoff: 879, which was 0 XIDs old when operation ended
new relfrozenxid: 879, which is 2 XIDs ahead of previous value
frozen: 48 pages from table (96.00% of total) had 96 tuples frozen
visibility map: 0 pages set all-visible, 48 pages set all-frozen (48
were all-visible)
index scan not needed: 0 pages from table (0.00% of total) had 0 dead
item identifiers removed
avg read rate: 0.000 MB/s, avg write rate: 0.000 MB/s
buffer usage: 116 hits, 0 reads, 0 dirtied
WAL usage: 97 records, 0 full page images, 6615 bytes, 0 buffers full
system usage: CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s
VACUUM
```

Теперь, поскольку агрессивная очистка проверяет всю таблицу, номер замороженной транзакции можно увеличить — в страницах гарантированно не осталось более старой незамороженной транзакции:

```
=> SELECT relfrozenxid, age(relfrozenxid)
FROM pg_class WHERE relname = 'tfreeze';
 relfrozenxid | age
--------------+-----
          879 |   0
(1 row)
```

Разумеется, на первой странице теперь заморожены все версии строк:

```
=> SELECT * FROM heap_page('tfreeze',0,1);
 ctid  |     state      | xmin  | xmin_age | xmax
-------+----------------+-------+----------+------
 (0,1) | redirect to 3 |       |          |
 (0,2) | normal         | 877 f |        2 | 0 a
 (0,3) | normal         | 878 f |        1 | 0 a
 (1,1) | normal         | 877 f |        2 | 0 a
 (1,2) | normal         | 877 f |        2 | 0 a
(5 rows)
```

Кроме того, первая страница отмечена в карте заморозки:

```
=> SELECT * FROM generate_series(0,1) g(blkno),
  pg_visibility_map('tfreeze',g.blkno)
ORDER BY g.blkno;
 blkno | all_visible | all_frozen
-------+-------------+------------
     0 | t           | t
     1 | t           | t
(2 rows)
```

Агрессивная заморозка всех страниц в таблице выполняется не чаще, чем раз в *vacuum_freeze_table_age – vacuum_freeze_min_age* транзакций (100 млн при значениях параметров по умолчанию). Она будет запускаться реже, если очистка в неагрессивном режиме тоже сможет продвинуть значение `relfrozenxid`[1]. В любом случае надо учитывать, что слишком большое значение параметра *vacuum_freeze_min_age* вместо экономии может привести к избыточным срабатываниям агрессивной заморозки и увеличению накладных расходов.

v. 15

Возраст для аварийного срабатывания автоочистки

Бывают ситуации, когда предыдущих двух настроек оказывается недостаточно для своевременной заморозки версий строк. Автоочистку можно отключить, а обычную очистку — не запускать (так делать не надо, но технически это возможно). Также могут присутствовать таблицы, не попадающие в очередь автоочистки, или неактивные базы данных (например,

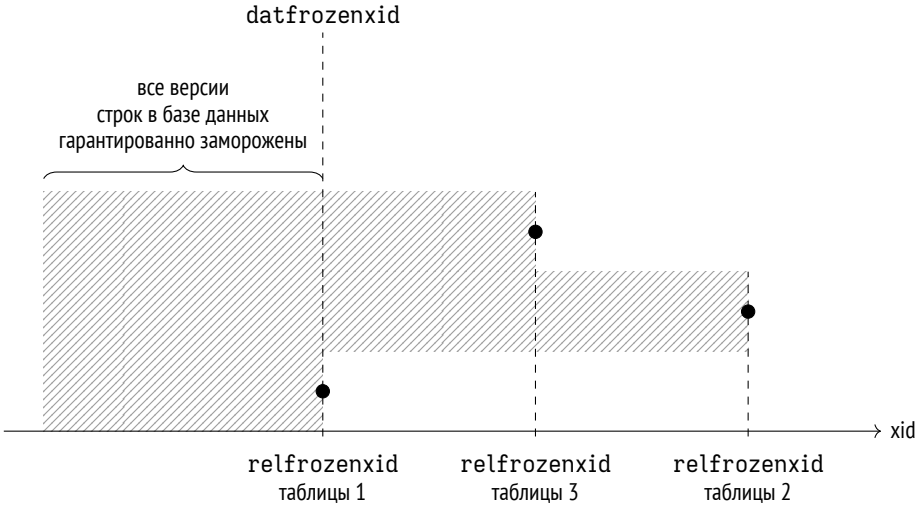
[1] commitfest.postgresql.org/37/3433.

с. 139 template0), в которые автоочистка вообще не приходит. Для подобных случаев предусмотрено *аварийное* срабатывание автоочистки в агрессивном режиме, что позволяет предотвратить переполнение счетчика транзакций.

Агрессивная очистка базы данных запускается аварийно, когда возраст какой-нибудь незамороженной транзакции в этой базе может превысить
200 млн значение *autovacuum_freeze_max_age*[1]. Даже если отключить автоочистку, она все равно запустится — правда, не сразу, а при первом ручном выполнении очистки или перезапуске сервера.

Определяющим для аварийного срабатывания является возраст самой старой из транзакций pg_class.relfrozenxid по всем таблицам базы, поскольку все еще более старые транзакции гарантированно заморожены. Номер этой транзакции запоминается в системном каталоге:

```
=> SELECT datname, datfrozenxid, age(datfrozenxid) FROM pg_database;
  datname  | datfrozenxid | age
-----------+--------------+-----
 postgres  |          744 | 135
 template1 |          744 | 135
 template0 |          744 | 135
 internals |          744 | 135
(4 rows)
```



[1] backend/access/transam/varsup.c, функция SetTransactionIdLimit.

Предел для значения *autovacuum_freeze_max_age* составляет 2 млрд транзакций (чуть менее половины круга), а значение по умолчанию в 10 раз меньше. В этом есть смысл: большое значение увеличивает риск того, что за оставшееся до переполнения счетчика время автоочистка просто не успеет заморозить все необходимые версии строк. В этом случае PostgreSQL аварийно остановится, чтобы предотвратить возможные проблемы, и запуск потребует вмешательства администратора.

Кроме того, значение этого параметра ограничивает размер структуры clog. *с. 85*
В кластере не может остаться более старых незамороженных транзакций, чем старейшая из `datfrozenxid` по всем базам данных, а для замороженных транзакций статус хранить не нужно. Лишние файлы-сегменты clog удаляются автоочисткой[1].

Изменение параметра *autovacuum_freeze_max_age* требует перезапуска сервера. Но при необходимости все рассмотренные выше параметры управления заморозкой можно переопределить с помощью параметров хранения на уровне отдельных таблиц. Обратите внимание на то, что названия всех параметров начинаются на «auto»: *autovacuum_freeze_min_age*, *autovacuum_freeze_table_age* и *autovacuum_freeze_max_age*.

Возраст для приоритетного режима заморозки

v. 14

Если автоочистка уже работает над предотвращением переполнения счетчика транзакций, но явно не успевает, срабатывает «предохранитель»: автоочистка прекращает учитывать задержку *autovacuum_vacuum_cost_delay* (*vacuum_cost_delay*) и перестает очищать индексы, чтобы как можно быстрее заморозить версии строк.

Приоритетный режим заморозки включается[2], если возраст какой-нибудь незамороженной транзакции в базе данных может превысить значение параметра *vacuum_failsafe_age*. Предполагается, что значение этого параметра должно быть выше, чем значение *autovacuum_freeze_max_age*.

1,6 млрд

[1] backend/commands/vacuum.c, функция vac_truncate_clog.
[2] backend/access/heap/vacuumlazy.c, функция lazy_check_wraparound_failsafe.

v. 12 При ручном вызове, когда цель состоит не в освобождении места на страницах данных, а в максимально быстрой заморозке, очистку индексов тоже имеет смысл отключить. Это можно сделать либо явно, используя команду `VACUUM (freeze, index_cleanup false)`, либо с помощью параметра хранения *vacuum_index_cleanup*.

7.4. Нетерпеливое сканирование

v. 18

Когда очистка не занимается агрессивной заморозкой, она пропускает страницы, отмеченные в карте видимости, поскольку вычищать из них нечего. Но из этого правила есть исключения.

Про одно я уже вскользь упоминал: очистка *вынуждена* сканировать страницы, отмеченные в карте видимости, если не набирается хотя бы 32 страницы, идущие подряд. Пропускать такие небольшие группы не только бесполезно, но и вредно: может сбиться алгоритм операционной системы, отвечающий за предвыборку страниц.

Другое исключение более интересно. Очистка может *намеренно* просканировать некоторое количество страниц, отмеченных в карте видимости (но не в карте заморозки). Такое сканирование называется «нетерпеливым» (eager).

И в том, и в другом случае к прочитанным страницам применяются обычные правила заморозки, что дает шанс заморозить все версии строк на странице, отметить ее в карте заморозки и таким образом уменьшить объем работы в агрессивном режиме. С другой стороны, чтение лишних страниц — это дополнительная нагрузка, и она не должна мешать основной деятельности.

Если не удается заморозить несколько страниц подряд, то, скорее всего, все версии строк в этом фрагменте таблицы еще недостаточно старые, так что продолжать попытки бесполезно. Размер фрагмента фиксирован (4096 страниц), а доля неудачных попыток ограничена значением парамет-
0.03 ра *vacuum_max_eager_freeze_failure_rate* (или одноименного параметра хранения). Настройки по умолчанию дают право на 122 промаха, после чего нетерпеливое сканирование текущего фрагмента прерывается.

Но в другом месте таблицы ситуация может отличаться, поэтому при входе в следующий фрагмент счетчик неудачных попыток сбрасывается и нетерпеливое сканирование продолжается.

Таким образом, в худшем случае нетерпеливое сканирование зря прочитает *vacuum_max_eager_freeze_failure_rate* от количества страниц, отмеченных в карте видимости, но не отмеченных в карте заморозки. Установка этого параметра в ноль фактически запрещает нетерпеливое сканирование.

Но и при удачном стечении обстоятельств замораживать сразу много страниц не нужно, чтобы не сильно увеличивать время работы и не превращать обычную очистку в агрессивную. Поэтому число успешных попыток тоже ограничено и не может превышать 20 % от количества страниц, отмеченных в карте видимости, но не в карте заморозки.

Чтобы посмотреть на заморозку при нетерпеливом сканировании, вернем исходное значение возраста агрессивной заморозки, но сохраним установленный ранее минимальный возраст срабатывания обычной очистки.

```
=> ALTER SYSTEM RESET vacuum_freeze_table_age;
=> SELECT pg_reload_conf();
=> SHOW vacuum_freeze_min_age;
 vacuum_freeze_min_age
-----------------------
 1
(1 row)
```

Нетерпеливое сканирование происходит не всегда; одно из условий — не слишком маленькая таблица, минимум 8192 страницы. Поэтому наполним `tfreeze` заново достаточным количеством строк (напомню, что на одной странице помещается две версии):

```
=> TRUNCATE tfreeze;
=> INSERT INTO tfreeze(id, s)
SELECT id, 'FOO'||id FROM generate_series(1,8192*2) id;
INSERT 0 16384
```

Затем очистим таблицу, чтобы все страницы отметились в карте видимости. И финальный подготовительный штрих: выполним транзакцию, чтобы увеличить возраст всех версий на единицу — теперь они попадают под условие заморозки.

```
=> VACUUM tfreeze;
=> SELECT pg_current_xact_id();
 pg_current_xact_id
--------------------
                881
(1 row)

=> SELECT * FROM heap_page('tfreeze',0,8191);
   ctid   | state  | xmin  | xmin_age | xmax
----------+--------+-------+----------+------
 (0,1)    | normal | 880 c |        2 | 0 a
 (0,2)    | normal | 880 c |        2 | 0 a
 (1,1)    | normal | 880 c |        2 | 0 a
 (1,2)    | normal | 880 c |        2 | 0 a
 ...
 (8191,1) | normal | 880 c |        2 | 0 a
 (8191,2) | normal | 880 c |        2 | 0 a
(16384 rows)

=> SELECT * FROM generate_series(0,8191) g(blkno),
    pg_visibility_map('tfreeze',g.blkno)
ORDER BY g.blkno;
 blkno | all_visible | all_frozen
-------+-------------+------------
     0 | t           | f
     1 | t           | f
 ...
  8191 | t           | f
(8192 rows)
```

Наконец, все готово, запускаем очистку с параметром verbose:

```
=> VACUUM VERBOSE tfreeze;
INFO:  vacuuming "internals.public.tfreeze"
INFO:  disabling eager scanning after freezing 1638 eagerly scanned
blocks of relation "internals.public.tfreeze"
INFO:  finished vacuuming "internals.public.tfreeze": index scans: 0
pages: 0 removed, 8192 remain, 1640 scanned (20.02% of total), 1639
eagerly scanned
tuples: 0 removed, 16384 remain, 0 are dead but not yet removable
removable cutoff: 882, which was 0 XIDs old when operation ended
frozen: 1640 pages from table (20.02% of total) had 3280 tuples frozen
visibility map: 0 pages set all-visible, 1640 pages set all-frozen
(1640 were all-visible)
index scan not needed: 0 pages from table (0.00% of total) had 0 dead
item identifiers removed
```

```
avg read rate: 0.000 MB/s, avg write rate: 0.000 MB/s
buffer usage: 3299 hits, 0 reads, 0 dirtied
WAL usage: 3281 records, 0 full page images, 215167 bytes, 0 buffers
full
system usage: CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s
VACUUM
```

Мы видим, что нетерпеливое сканирование было прервано после успешной заморозки 1638 страниц: это как раз 20 % от 8192. Заметьте, что несколькими строками ниже сказано, что всего было заморожено 1640 страниц: как я говорил, нетерпеливое сканирование — не единственный случай, когда обычная очистка читает страницы, отмеченные в карте видимости. Например, последняя страница сканируется всегда: если в ней есть версии строк, очистка не будет пытаться усекать таблицу, чтобы не захватывать впустую исключительную блокировку. *с. 133*

```
SELECT * FROM generate_series(0,8191) g(blkno),
    pg_visibility_map('tfreeze',g.blkno)
ORDER BY g.blkno;
 blkno | all_visible | all_frozen
-------+-------------+------------
     0 | t           | t
     1 | t           | t
 ...
  1637 | t           | t
  1638 | t           | t
  1639 | t           | f
  1640 | t           | f
 ...
  8189 | t           | f
  8190 | t           | f
  8191 | t           | t
(8192 rows)
```

7.5. Заморозка при загрузке

Данные, которые не должны меняться, можно заморозить сразу при начальной загрузке. Для этого используется команда `COPY` с параметром `FREEZE`.

Загружать данные с заморозкой можно только в таблицу, созданную или опустошенную (командой `TRUNCATE`) в этой же транзакции. Обе эти операции монопольно блокируют таблицу. Такое ограничение необходимо, *с. 261*

поскольку замороженные версии считаются видимыми во всех снимках данных независимо от уровня изоляции; иначе транзакции неожиданно обнаруживали бы свежезамороженные строки прямо по мере их загрузки. Но пока таблица заблокирована, другие транзакции не смогут получить к ней доступ.

Тем не менее формально нарушение изоляции все же возможно. В отдельном сеансе начнем транзакцию с уровнем изоляции Repeatable Read:

```
=> BEGIN ISOLATION LEVEL REPEATABLE READ;
=> SELECT 1; -- построен снимок
```

Опустошим таблицу `tfreeze` и в той же транзакции загрузим в нее новые строки. (Если бы читающая транзакция успела обратиться к `tfreeze`, команда `TRUNCATE` была бы заблокирована.)

```
=> BEGIN;
=> TRUNCATE tfreeze;
=> COPY tfreeze FROM stdin WITH FREEZE;
1 FOO
2 BAR
3 BAZ
\.
=> COMMIT;
```

Теперь читающая транзакция видит новые данные в нарушение изоляции:

```
=> SELECT count(*) FROM tfreeze;
 count
-------
     3
(1 row)
=> COMMIT;
```

Но поскольку загрузка данных вряд ли происходит регулярно, в большинстве случаев это не представляет проблемы.

v. 14 *с. 127* При загрузке с заморозкой сразу же создается карта видимости, а в заголовках страниц проставляется признак видимости:

```
=> SELECT * FROM pg_visibility_map('tfreeze',0);
 all_visible | all_frozen
-------------+------------
 t           | t
(1 row)
=> SELECT flags & 4 > 0 AS all_visible
FROM page_header(get_raw_page('tfreeze',0));
 all_visible
-------------
 t
(1 row)
```

Таким образом, после загрузки с заморозкой очистка уже не будет обрабатывать таблицу (если, конечно, данные в ней останутся неизменными). К сожалению, это пока не работает для toast-таблиц: при загрузке длинных значений очистке придется лишний раз переписать всю toast-таблицу, чтобы проставить признак видимости в заголовки всех страниц. v. 14

7.6. Регулирование нагрузки и мониторинг

Очистка не блокирует другие процессы, поскольку работает постранично, но тем не менее создает нагрузку на систему и может оказывать заметное влияние на производительность.

Отслеживание хода выполнения очистки позволяет обнаружить ситуации, когда процесс проявляет избыточную активность или работает неэффективно, не справляясь с невычищенными и незамороженными версиями, нарастающими как снежный ком.

Ручная очистка

Очистка выполняет примерно *vacuum_cost_limit* условных единиц работы, а затем засыпает на *vacuum_cost_delay* единиц времени. Такое чередование работы и ожидания позволяет более или менее равномерно распределить влияние очистки на систему, не допуская пиковых нагрузок. 200 0

Нулевое значение по умолчанию для *vacuum_cost_delay* фактически означает, что (ручная) очистка вовсе не засыпает, так что конкретное значение *vacuum_cost_limit* не играет никакой роли. Это сделано из соображения, что если уж администратору пришлось запускать очистку вручную, то он, вероятно, хочет выполнить ее как можно быстрее.

Тем не менее если все-таки установить время сна, то указанный в параметре *vacuum_cost_limit* объем работы будет складываться из стоимостей работы со страницами в буферном кеше. Каждое чтение страницы оценивается в *vacuum_cost_page_hit* единиц, если страница нашлась в буферном кеше, и в *vacuum_cost_page_miss* единиц, если не нашлась[1]. Если же страница не была грязной, а в результате работы очистки на ней что-то поменялось, это стоит дополнительных *vacuum_cost_page_dirty* единиц[2].

с. 183
1
2
20

Если оставить значение *vacuum_cost_limit* по умолчанию, за один цикл очистка может обработать от 200 страниц в лучшем случае (если все они закешированы и в результате работы не появилось ни одной новой грязной) до всего 9 в худшем (если все страницы читаются с диска и становятся грязными).

Команда `VACUUM` с указанием `VERBOSE` выполняет очистку и выводит отчет о проделанной работе, а представление `pg_stat_progress_vacuum` показывает состояние уже запущенного процесса.

v. 13

Похожее представление есть и для анализа (`pg_stat_progress_analyze`), хотя анализ выполняется быстро и обычно не доставляет проблем.

Вставим в таблицу больше строк и все обновим, чтобы очистка выполнялась ощутимое время:

```
=> TRUNCATE vac;
=> INSERT INTO vac(id,s)
  SELECT id, 'A' FROM generate_series(1,5_000_000) id;
=> UPDATE vac SET s  = 'B';
UPDATE 5000000
```

Для целей демонстрации уменьшим размер памяти, выделенной под хранение идентификаторов, до абсурдно скромного значения:

[1] backend/storage/buffer/bufmgr.c, функция ReadBuffer_common.
[2] backend/storage/buffer/bufmgr.c, функция MarkBufferDirty.

```
=> ALTER SYSTEM SET maintenance_work_mem = '1MB';
=> SELECT pg_reload_conf();
```

Запустим очистку и, пока она работает, обратимся несколько раз к представлению pg_stat_progress_vacuum:

```
=> VACUUM VERBOSE vac;
```

```
SELECT * FROM pg_stat_progress_vacuum \gx
-[ RECORD 1 ]--------+--------------
pid                  | 20941
datid                | 16391
datname              | internals
relid                | 16486
phase                | scanning heap
heap_blks_total      | 98483
heap_blks_scanned    | 21329
heap_blks_vacuumed   | 0
index_vacuum_count   | 0
max_dead_tuple_bytes | 67108864
dead_tuple_bytes     | 0
num_dead_item_ids    | 0
indexes_total        | 0
indexes_processed    | 0
delay_time           | 0
```

```
SELECT * FROM pg_stat_progress_vacuum \gx
-[ RECORD 1 ]--------+------------------
pid                  | 17602
datid                | 16391
datname              | internals
relid                | 16486
phase                | vacuuming indexes
heap_blks_total      | 172414
heap_blks_scanned    | 36833
heap_blks_vacuumed   | 0
index_vacuum_count   | 0
max_dead_tuple_bytes | 1048576
dead_tuple_bytes     | 1114112
num_dead_item_ids    | 2136372
indexes_total        | 1
indexes_processed    | 0
delay_time           | 0
```

v. 17 Представление, в частности, показывает:

- `phase` — текущий этап очистки (выше я описал основные, но вообще их больше[1]);
- `heap_blks_total` — общее число страниц таблицы;
- `heap_blks_scanned` — число просканированных страниц;
- `heap_blks_vacuumed` — число уже очищенных страниц;
- `index_vacuum_count` — количество проходов по индексам;
- `dead_tuple_bytes` и `num_dead_item_ids` — объем и количество мертвых версий, собранных за данный проход.

Общий прогресс очистки определяется отношением `heap_blks_vacuumed` к `heap_blks_total`, но нужно учитывать, что из-за сканирования индексов это значение изменяется «рывками».

Основное внимание стоит обратить на количество циклов очистки (поле `index_vacuum_count`): значение больше единицы сигнализирует о том, что выделенной памяти не хватило для завершения очистки за один проход по индексам. Впрочем, вероятность этого при разумном значении *maintenance_work_mem* мала. Эффективность хранения можно оценить по отношению `dead_tuple_bytes` к `num_dead_item_ids`: в данном примере на один `tid` расходуется примерно полбайта.

off v. 18 Если включить параметр *track_cost_delay_timing*, поле `delay_time` покажет фактическое время сна. Надо иметь в виду, что на некоторых платформах отслеживание времени может существенно замедлить выполнение, так что этой возможностью следует пользоваться с осторожностью.

Вывод завершившейся к этому времени команды `VACUUM VERBOSE` показывает общую картину:

[1] postgrespro.ru/docs/postgresql/18/progress-reporting#VACUUM-PHASES.

```
INFO:  vacuuming "internals.public.vac"
INFO:  finished vacuuming "internals.public.vac": index scans: 3
pages: 0 removed, 172414 remain, 172414 scanned (100.00% of total), 0
eagerly scanned
tuples: 5000000 removed, 5000000 remain, 0 are dead but not yet
removable
removable cutoff: 886, which was 0 XIDs old when operation ended
new relfrozenxid: 886, which is 3 XIDs ahead of previous value
frozen: 86208 pages from table (50.00% of total) had 5000000 tuples
frozen
visibility map: 172414 pages set all-visible, 172414 pages set
all-frozen (0 were all-visible)
index scan needed: 86207 pages from table (50.00% of total) had 5000000
dead item identifiers removed
index "vac_s": pages: 8652 in total, 4325 newly deleted, 4325 currently
deleted, 0 reusable
avg read rate: 342.318 MB/s, avg write rate: 341.067 MB/s
buffer usage: 223204 hits, 263773 reads, 262809 dirtied
WAL usage: 591745 records, 177971 full page images, 1300060561 bytes,
14908 buffers full
system usage: CPU: user: 1.03 s, system: 0.76 s, elapsed: 6.01 s
VACUUM
```

В данном случае очистке потребовалось выполнить целых три прохода по индексам.

Автоочистка

Регулирование нагрузки для автоматической очистки[1] работает примерно так же, как и для обычной. Но чтобы они могли работать с разной интенсивностью, для автоочистки предусмотрены собственные параметры:

- *autovacuum_vacuum_cost_limit*; -1
- *autovacuum_vacuum_cost_delay*. 2ms

Если какой-либо из этих параметров равен –1, для него используется значение соответствующего параметра обычной очистки. То есть по умолчанию для *autovacuum_vacuum_cost_limit* применяется значение *vacuum_cost_limit*.

[1] backend/postmaster/autovacuum.c, функция autovac_balance_cost.

Единицы работы автоочистки, ограниченные в одном цикле пределом *autovacuum_vacuum_cost_limit*, расходуются всеми рабочими процессами, так что общая нагрузка на систему остается примерно одинаковой при любом их количестве. Поэтому если стоит задача ускорить автоочистку, при увеличении параметра *autovacuum_max_workers* стоит пропорционально увеличить и значение *autovacuum_vacuum_cost_limit*.

При необходимости для разных таблиц можно устанавливать собственные значения с помощью параметров хранения (*autovacuum_vacuum_cost_limit* и *autovacuum_vacuum_cost_delay*) и их toast-версий.

Представления `need_vacuum` и `need_analyze`, которые мы уже рассмотрели, позволяют отслеживать список таблиц, требующих очистки и анализа. Увеличение длины списка говорит о том, что автоочистка не справляется и ее надо ускорить, уменьшив паузу (*autovacuum_vacuum_cost_delay*) или увеличив объем работы между паузами (*autovacuum_vacuum_cost_limit*). Не исключено, что понадобится также увеличить степень параллелизма (*autovacuum_max_workers*).

Помимо запросов есть и штатный способ мониторинга автоочистки — вывод информации (аналогичной той, которую показывает команда `VACUUM VERBOSE`) в журнал сообщений сервера для дальнейшего анализа.

-1 Установим нулевое значение параметра *log_autovacuum_min_duration*, при котором в журнал попадают все запуски автоочистки:

```
=> ALTER SYSTEM SET log_autovacuum_min_duration = 0;
```

v. 18 Включим также отслеживание времени сна очистки. Для ручной очистки, которая по умолчанию не спит, это не было бы информативно, а вот для автоочистки может дать информацию к размышлению.

```
=> ALTER SYSTEM SET track_cost_delay_timing = on;
=> SELECT pg_reload_conf();
```

Снова обновим все строки таблицы, чтобы автоочистке было чем заняться:

```
=> UPDATE vac SET s = 'C';
UPDATE 5000000
```

```
postgres$ tail -n 20 /home/postgres/logfile
2026-04-05 20:54:21.347 MSK [51046] LOG:  automatic vacuum of table
"internals.public.vac": index scans: 3
    pages: 0 removed, 172414 remain, 172414 scanned (100.00% of total),
0 eagerly scanned
    tuples: 5000000 removed, 5000000 remain, 0 are dead but not yet
removable
    removable cutoff: 888, which was 0 XIDs old when operation ended
    new relfrozenxid: 888, which is 2 XIDs ahead of previous value
    frozen: 86208 pages from table (50.00% of total) had 5000000 tuples
frozen
    visibility map: 172414 pages set all-visible, 172414 pages set
all-frozen (0 were all-visible)
    index scan needed: 86208 pages from table (50.00% of total) had
5000000 dead item identifiers removed
    index "vac_s": pages: 16923 in total, 4325 newly deleted, 8650
currently deleted, 4325 reusable
    delay time: 68789.809 ms
    avg read rate: 26.782 MB/s, avg write rate: 24.155 MB/s
    buffer usage: 236780 hits, 288039 reads, 259792 dirtied
    WAL usage: 616268 records, 187331 full page images, 1502256915
bytes, 9413 buffers full
    system usage: CPU: user: 5.13 s, system: 4.50 s, elapsed: 84.02 s
2026-04-05 20:54:22.349 MSK [51046] LOG:  automatic analyze of table
"internals.public.vac"
    delay time: 663.495 ms
    avg read rate: 214.227 MB/s, avg write rate: 0.008 MB/s
    buffer usage: 2742 hits, 27421 reads, 1 dirtied
    WAL usage: 5 records, 1 full page images, 5509 bytes, 0 buffers full
    system usage: CPU: user: 0.20 s, system: 0.06 s, elapsed: 0.99 s
```

Из журнала следует, что бóльшую часть времени автоочистка простаивала, так что для увеличения скорости имеется достаточный резерв.

Общее время, затраченное на очистку и анализ таблицы (в миллисекундах), можно узнать из системной статистики: v. 18

```
=> SELECT total_vacuum_time, total_autovacuum_time,
  total_analyze_time, total_autoanalyze_time
FROM pg_stat_all_tables WHERE relname = 'vac' \gx
-[ RECORD 1 ]----------+------
total_vacuum_time      | 6025
total_autovacuum_time  | 84026
total_analyze_time     | 0
total_autoanalyze_time | 1825
```

8

Перестроение таблиц и индексов

8.1. Полная очистка

Необходимость

Обычная очистка освобождает больше места, чем самоочистка, но и она не всегда решает задачу полностью.

Если файлы таблицы или индекса по каким-то причинам сильно выросли в размерах, то обычная очистка освобождает место внутри существующих страниц, но число страниц в большинстве случаев не уменьшается. Единственная ситуация, при которой очистка возвращает место операционной
с. 133 системе, — образование полностью пустых страниц в самом конце файла, что происходит нечасто.

Излишний размер приводит к неприятным последствиям:

- замедляется полное сканирование таблицы (или индекса);
- может потребоваться больший буферный кеш (ведь кешируются страницы, а плотность полезной информации в них падает);
- в B-дереве может появиться лишний уровень, который будет замедлять индексный доступ;
- нерационально расходуется место на диске и в резервных копиях.

Если доля полезной информации в файлах опустилась ниже некоторого разумного предела, администратор может выполнить *полную очистку*[1] командой VACUUM FULL. Она последовательно перестраивает сначала таблицу, а затем и все ее индексы с нуля, максимально компактно упаковывая данные (с учетом параметра *fillfactor*). В процессе перестроения объекта на диске хранятся и старые, и новые файлы[2], поэтому для работы полной очистки может потребоваться довольно много свободного места.

с. 111

Следует учитывать, что на все время перестроения таблица полностью блокируется и для записи, и для чтения.

Оценка плотности информации

Для иллюстрации вставим в таблицу некоторое количество строк, отключив автоматическую очистку:

```
=> TRUNCATE vac;
=> ALTER TABLE vac SET (autovacuum_enabled = off);
=> INSERT INTO vac(id,s)
  SELECT id, id::text FROM generate_series(1,500000) id;
```

Плотность информации удобно оценивать с помощью специального расширения pgstattuple:

```
=> CREATE EXTENSION pgstattuple;
=> SELECT * FROM pgstattuple('vac') \gx
-[ RECORD 1 ]------+---------
table_len          | 70623232
tuple_count        | 500000
tuple_len          | 64500000
tuple_percent      | 91.33
dead_tuple_count   | 0
dead_tuple_len     | 0
dead_tuple_percent | 0
free_space         | 416328
free_percent       | 0.59
```

[1] postgrespro.ru/docs/postgresql/18/routine-vacuuming#VACUUM-FOR-SPACE-RECOVERY.
[2] backend/commands/cluster.c.

Функция читает всю таблицу и выводит статистику по распределению места в файлах. Поле `tuple_percent` показывает процент места, занятого полезными данными (версиями строк). Его значение неизбежно меньше 100 % из-за накладных расходов на служебную информацию внутри страницы, но тем не менее в этом примере оно довольно велико.

Для индекса выводится другая информация, но поле `avg_leaf_density` имеет тот же смысл: процент полезной информации (в листовых страницах).

```
=> SELECT * FROM pgstatindex('vac_s') \gx
-[ RECORD 1 ]------+----------
version            | 4
tree_level         | 3
index_size         | 114302976
root_block_no      | 2825
internal_pages     | 376
leaf_pages         | 13576
empty_pages        | 0
deleted_pages      | 0
avg_leaf_density   | 53.83
leaf_fragmentation | 10.59
```

Использованные функции расширения `pgstattuple` читают таблицу или индекс целиком, чтобы получить точную информацию. В случае большого объекта это может оказаться слишком затратно, поэтому в расширении есть функция `pgstattuple_approx`, которая пропускает страницы, отмеченные в карте видимости, и показывает примерные цифры.

Еще более быстрый, но и еще менее точный способ — прикинуть отношение объема данных к размеру файла по системному каталогу[1].

Вот какой размер занимают таблица и индекс:

```
=> SELECT pg_size_pretty(pg_table_size('vac')) AS table_size,
        pg_size_pretty(pg_indexes_size('vac')) AS index_size;
 table_size | index_size
------------+------------
 67 MB      | 109 MB
(1 row)
```

Теперь удалим 90 % всех строк:

[1] wiki.postgresql.org/wiki/Show_database_bloat.

```
=> DELETE FROM vac WHERE id % 10 != 0;
DELETE 450000
```

После обычной очистки размер файлов не меняется, поскольку нет пустых «хвостовых» страниц:

```
=> VACUUM vac;
=> SELECT pg_size_pretty(pg_table_size('vac')) AS table_size,
       pg_size_pretty(pg_indexes_size('vac')) AS index_size;
 table_size | index_size
------------+------------
 67 MB      | 109 MB
(1 row)
```

Но плотность информации уменьшилась примерно в десять раз:

```
=> SELECT vac.tuple_percent,
         vac_s.avg_leaf_density
FROM pgstattuple('vac') vac,
     pgstatindex('vac_s') vac_s;
 tuple_percent | avg_leaf_density
---------------+------------------
          9.13 |             6.66
(1 row)
```

Сейчас таблица и индекс размещаются в файлах со следующими именами:

```
=> SELECT pg_relation_filepath('vac') AS vac_filepath,
       pg_relation_filepath('vac_s') AS vac_s_filepath \gx
-[ RECORD 1 ]--+-----------------
vac_filepath   | base/16391/16526
vac_s_filepath | base/16391/16527
```

Теперь проверим, что получится после полной очистки.

```
=> VACUUM FULL vac;
```

Пока команда работает, ход ее выполнения можно отслеживать в представлении `pg_stat_progress_cluster` (похожем на представление для обычной очистки `pg_stat_progress_vacuum`): v. 12

```
=> SELECT * FROM pg_stat_progress_cluster \gx
-[ RECORD 1 ]-------+------------------
pid                 | 53523
datid               | 16391
datname             | internals
relid               | 16486
command             | VACUUM FULL
phase               | seq scanning heap
cluster_index_relid | 0
heap_tuples_scanned | 43088
heap_tuples_written | 43088
heap_blks_total     | 8621
heap_blks_scanned   | 7429
index_rebuild_count | 0
```

```
VACUUM
```

Этапы выполнения полной очистки[1], конечно, отличаются от этапов обычной очистки.

После полной очистки файлы поменялись на новые:

```
=> SELECT pg_relation_filepath('vac') AS vac_filepath,
       pg_relation_filepath('vac_s') AS vac_s_filepath \gx
-[ RECORD 1 ]--+-----------------
vac_filepath   | base/16391/16538
vac_s_filepath | base/16391/16541
```

Размер таблицы и индекса существенно уменьшился:

```
=> SELECT pg_size_pretty(pg_table_size('vac')) AS table_size,
       pg_size_pretty(pg_indexes_size('vac')) AS index_size;
 table_size | index_size
------------+------------
 6904 kB    | 6504 kB
(1 row)
```

Плотность информации, соответственно, увеличилась, причем в индексе она увеличилась даже по сравнению с первоначальной. Заново создать индекс (B-дерево) по имеющимся данным обычно выгоднее, чем вставлять данные строка за строкой в уже имеющийся индекс:

[1] postgrespro.ru/docs/postgresql/18/progress-reporting#CLUSTER-PHASES.

```
=> SELECT vac.tuple_percent, vac_s.avg_leaf_density
FROM pgstattuple('vac') vac,
     pgstatindex('vac_s') vac_s;
 tuple_percent | avg_leaf_density
---------------+------------------
         91.23 |            91.03
(1 row)
```

Заморозка

При перестроении таблицы версии строк замораживаются, поскольку это ничего не стоит по сравнению с остальным объемом работы:

```
=> SELECT * FROM heap_page('vac',0,0) LIMIT 5;
 ctid  | state  | xmin  | xmin_age | xmax
-------+--------+-------+----------+------
 (0,1) | normal | 892 f |        4 | 0 a
 (0,2) | normal | 892 f |        4 | 0 a
 (0,3) | normal | 892 f |        4 | 0 a
 (0,4) | normal | 892 f |        4 | 0 a
 (0,5) | normal | 892 f |        4 | 0 a
(5 rows)
```

Однако страницы не отмечаются в карте видимости и заморозки, и в заголовке страницы не проставляется признак видимости (как это происходит при выполнении команды COPY с параметром FREEZE): *с. 161*

```
=> SELECT * FROM pg_visibility_map('vac',0);
 all_visible | all_frozen
-------------+------------
 f           | f
(1 row)
=> SELECT flags & 4 > 0 all_visible
FROM page_header(get_raw_page('vac',0));
 all_visible
-------------
 f
(1 row)
```

И только после выполнения команды VACUUM (или срабатывания автоочистки) ситуация исправляется:

```
=> VACUUM vac;
=> SELECT * FROM pg_visibility_map('vac',0);
 all_visible | all_frozen
-------------+------------
 t           | t
(1 row)
=> SELECT flags & 4 > 0 AS all_visible
FROM page_header(get_raw_page('vac',0));
 all_visible
-------------
 t
(1 row)
```

Фактически это означает, что такая страница будет перезаписана еще раз — даже если при перестроении все версии строк табличной страницы находились за горизонтом базы данных.

8.2. Другие способы перестроения

Аналоги полной очистки

Помимо полной очистки есть еще несколько команд, которые перестраивают таблицы и индексы полностью. Все они монопольно блокируют работу с таблицей, все они удаляют старые файлы данных и создают новые.

с. 409 Команда CLUSTER во всем аналогична VACUUM FULL, но дополнительно упорядочивает версии строк в файлах в соответствии с одним из имеющихся индексов. Это дает планировщику возможность в некоторых случаях более
с. 415 эффективно использовать индексный доступ. Однако надо понимать, что кластеризация не поддерживается: при последующих изменениях таблицы физический порядок версий строк будет нарушаться.

С точки зрения кода VACUUM FULL — просто частный случай CLUSTER, не требующий переупорядочивания строк[1].

[1] backend/commands/cluster.c.

Команда REINDEX перестраивает индекс или несколько индексов[1]. VACUUM FULL и CLUSTER фактически используют механизм этой команды, когда завершают работу с таблицей и доходят до индексов.

Команда TRUNCATE[2] удаляет все табличные строки и логически соответствует команде DELETE без условия WHERE. Но DELETE только помечает версии строк как удаленные, что требует дальнейшей очистки. TRUNCATE же просто создает новый чистый файл — как правило, это намного быстрее.

с. 87

Перестроение без долгих блокировок

Полная очистка не предполагает регулярного использования, так как полностью блокирует всякий доступ к таблице (включая и выполнение запросов) на все время своей работы. Для высокодоступных систем это обычно неприемлемо.

с. 261

Существует несколько расширений (например, pg_repack[3]), позволяющих перестроить таблицу и индексы практически без прерывания обслуживания. Исключительная блокировка таблицы все равно требуется, но лишь в начале и в конце работы, и на короткое время. Достигается это более сложной реализацией: пока идет перестроение, триггер сохраняет изменения данных исходной таблицы, а затем утилита применяет их к новой таблице и подменяет исходную в системном каталоге.

Интересное решение предлагает утилита pgcompacttable[4]. Идея состоит в многократном фиктивном (не меняющем данные) обновлении строк таблицы, в результате которого актуальные версии строк постепенно перемещаются внутри файла ближе к его началу. Между сериями обновлений запускается очистка, которая удаляет неактуальные версии и постепенно усекает файл. Такой подход отнимает существенно больше времени и ресурсов, но не требует дополнительного места для перестроения таблицы и не создает

с. 133

[1] backend/commands/indexcmds.c.
[2] backend/commands/tablecmds.c, функция ExecuteTruncate.
[3] github.com/reorg/pg_repack.
[4] github.com/dataegret/pgcompacttable.

пиковых нагрузок. Кратковременные исключительные блокировки при усечении таблицы возникают и в этом случае, но они обрабатываются очисткой достаточно мягко.

8.3. Профилактика

Читающие запросы

Одной из причин раздувания файлов является совмещение долго выполняющихся транзакций, удерживающих горизонт базы, с активным обновлением данных.

с. 106

Сами по себе долгие (читающие) транзакции не вызывают никаких проблем. Поэтому обычное решение — разнести нагрузку на разные машины: OLAP-транзакции выполнять на реплике, а быстрые OLTP-транзакции оставить на основном сервере. Конечно, это усложняет и удорожает систему, но может оказаться необходимым.

Бывают ситуации, когда долгие транзакции вызваны не необходимостью, а ошибками в приложении или драйвере. Если проблему не удается решить цивилизованным путем, у администратора в распоряжении есть пара параметров: *idle_in_transaction_session_timeout*, который определяет максимальное время жизни бездействующей транзакции, и *transaction_timeout*, который ограничивает общую продолжительность транзакции. По прошествии указанного времени (если оно не равно нулю) транзакция прерывается.

0ms
0ms
v. 17

Обновление данных

Вторая причина раздувания состоит в одномоментном изменении большого числа строк. Обновление всех строк таблицы может привести к увеличению количества версий в два раза, а очистка не успеет вмешаться. Самоочистка может сгладить проблему, но и только.

Добавим к таблице столбец `processed`, в котором будем отмечать обработанные строки:

```
=> ALTER TABLE vac ADD processed boolean DEFAULT false;
=> SELECT pg_size_pretty(pg_table_size('vac'));
 pg_size_pretty
----------------
 6936 kB
(1 row)
```

После обновления всех строк размер таблицы увеличивается примерно вдвое:

```
=> UPDATE vac SET processed = true;
UPDATE 50000
=> SELECT pg_size_pretty(pg_table_size('vac'));
 pg_size_pretty
----------------
 14 MB
(1 row)
```

Решение состоит в уменьшении объема изменений, выполняемых в одной транзакции, и разнесении изменений во времени, чтобы очистка успела удалить ненужные версии строк и освободить место под новые версии внутри уже существующих страниц. Если считать, что обновление каждой строки в принципе может быть зафиксировано независимо, то в качестве шаблона можно использовать запрос, выделяющий в таблице пакет строк определенного размера:

```
SELECT идентификатор
FROM таблица
WHERE фильтрация уже обработанных строк
LIMIT размер пакета
FOR UPDATE SKIP LOCKED
```

В этом фрагменте кода выбирается и сразу блокируется набор строк не более заданного размера. При этом строки, уже заблокированные другими транзакциями, пропускаются — они попадут в другой пакет в следующий раз. *с. 285* Это довольно универсальный и удобный способ, позволяющий легко менять размер пакета и выполнять повторную обработку в случае каких-либо сбоев. Сбросим признак обработки и выполним полную очистку, чтобы вернуть размер к исходному:

```
=> UPDATE vac SET processed = false;
=> VACUUM FULL vac;
```

После обновления первого пакета размер таблицы немного увеличивается:

```
=> WITH batch AS (
  SELECT id FROM vac WHERE NOT processed LIMIT 1000
  FOR UPDATE SKIP LOCKED
)
UPDATE vac SET processed = true
WHERE id IN (SELECT id FROM batch);
UPDATE 1000
=> SELECT pg_size_pretty(pg_table_size('vac'));
 pg_size_pretty
----------------
 7064 kB
(1 row)
```

Но больше размер практически не растет, поскольку новые версии строк добавляются на место вычищенных версий:

```
=> VACUUM vac;
=> WITH batch AS (
  SELECT id FROM vac WHERE NOT processed LIMIT 1000
  FOR UPDATE SKIP LOCKED
)
UPDATE vac SET processed = true
WHERE id IN (SELECT id FROM batch);
UPDATE 1000
=> SELECT pg_size_pretty(pg_table_size('vac'));
 pg_size_pretty
----------------
 7072 kB
(1 row)
```

Часть II

Буферный кеш и журнал

9

Буферный кеш

9.1. Ввод-вывод и кеширование

Общие соображения

Операции ввода-вывода можно разделить на прямые и буферизованные.

Прямой ввод-вывод (Direct I/O) перемещает данные непосредственно между памятью пользовательского процесса и диском. Если при этом поддерживается прямой доступ к памяти (DMA), то такая операция даже не нагружает процессор.

Буферизованный ввод-вывод задействует операционную систему и ее кеш. Пользовательский процесс запрашивает данные с помощью системного вызова. Если нужные данные есть в кеше, операционная система возвращает результат немедленно; в противном случае она предварительно читает данные с диска. Так или иначе, данные сначала (возможно, заранее) помещаются в кеш в памяти ядра операционной системы, а затем копируются в память пользовательского процесса: в нашем случае — в буферный кеш PostgreSQL.

> Хотя здесь мы говорим про оперативную память и диски, та же идея кеширования применяется для компенсации разницы в производительности любых двух типов памяти, одна из которых быстрее, но дороже и меньше по объему, а другая — медленнее, но дешевле и больше. В быстрой памяти невозможно разместить *все* данные из медленной памяти. Но в большинстве случаев активная работа ведется одновременно только с небольшой выборкой, поэтому, выделив часть быстрой памяти под хранение «горячих» данных, удается существенно экономить на обращениях к медленному

> устройству. В современных вычислительных системах кеширование применяется повсеместно, как на программном, так и на аппаратном уровне. У одного только процессора можно насчитать три-четыре уровня кеша; свой кеш бывает также и у контроллеров дисковых массивов, и у самих дисков.

Операции ввода-вывода можно классифицировать и иначе.

При *синхронном* вводе-выводе процесс отправляет команду и блокируется. Пока операция выполняется, процесс вынужден простаивать, дожидаясь ее завершения.

При *асинхронном* вводе-выводе (Async I/O) процесс формирует и отправляет задание и продолжает работу. За результатом он возвращается позже, в удобный для себя момент. Если операция уже выполнилась, процесс получает результат сразу; если нет — блокируется до ее завершения, как при синхронном вводе-выводе.

Довольно очевидно, что прямой ввод-вывод может обеспечить более высокую пропускную способность и меньшую задержку получения данных, а асинхронный ввод-вывод позволяет меньше простаивать и выполнять больше полезной работы. Но с точки зрения программирования обычный синхронный буферизованный ввод-вывод гораздо проще, а проигрывает совсем не так сильно, как можно было бы подумать. PostgreSQL тридцать лет использовал именно синхронный буферизованный ввод-вывод. Дело в том, что хотя код СУБД при этом работает последовательно, ввод-вывод фактически выполняется асинхронно: операционная система во многих случаях предугадывает последовательность чтений и заранее, в фоновом для СУБД режиме, считывает нужные данные в свой кеш. На операционную систему перекладываются все заботы о предвыборке данных и об отложенной записи грязных страниц.

Мотивацией для собственной реализации асинхронного ввода-вывода является уменьшение зависимости от операционной системы: не все системы выполняют предвыборку одинаково хорошо, а СУБД в любом случае лучше знает, какие страницы ей понадобятся и когда. Кроме того, привлекает возможность асинхронно выполнять синхронизацию данных с диском (`fsync`).

Переход на прямой ввод-вывод тоже сулит преимущества: избавление от двойной буферизации (в кеше операционной системы и буферном кеше

PostgreSQL) и потенциально существенное повышение производительности. Однако при этом теряется вся поддержка операционной системы. Придется реализовать собственную предвыборку данных, учитывающую особенности методов доступа, и самостоятельно сбрасывать грязные страницы на диск, равномерно распределяя пишущую нагрузку. Понадобится и алгоритм объединения мелких запросов на ввод-вывод в более крупные. Без всего этого прямой ввод-вывод только замедлит систему.

Текущее положение дел

v. 18

Работа над тем, чтобы отвязать PostgreSQL от файловой системы, началась уже несколько лет назад.

Прямой ввод-вывод включается параметром *debug_io_direct*, но пока не реализовано все необходимое, эта настройка имеет смысл только для разработчиков ядра.

off

v. 16

Асинхронный ввод-вывод [1] доступен пока только для чтения; запись и синхронизация по-прежнему остаются синхронными. Имеются несколько реализаций ввода-вывода, одна из которых указывается в параметре *io_method*: можно выбрать тот способ, который поддерживается на конкретной платформе и показывает лучшие результаты в конкретной системе.

worker

- При значении `worker` ввод-вывод выполняется отдельными рабочими процессами[2]. Этот способ работает на любых платформах и выбирается по умолчанию.

  Чтобы прочитать страницу или несколько подряд идущих страниц, процесс (обслуживающий или системный) помещает свой запрос в очередь, общую для всех рабочих процессов. В какой-то момент один из них выполнит запрошенную операцию, например прочитает требуемые страницы в буферный кеш.

  Количество рабочих процессов постоянно и определяется параметром *io_workers*. На серверах с большим количеством ядер значение параметра имеет смысл увеличить (максимальное значение — 32).

  3

[1] backend/storage/aio/README.md.
[2] backend/storage/aio/method_worker.c.

```
=> SELECT pid, backend_type
FROM pg_stat_activity
WHERE backend_type = 'io worker';
  pid  | backend_type
-------+--------------
 55418 | io worker
 55419 | io worker
 55420 | io worker
(3 rows)
```

- При значении `io_uring` ввод-вывод выполняется самим процессом с помощью одноименного интерфейса ядра Linux и работает только в этой операционной системе[1].

  Интерфейс io_uring использует два кольцевых буфера в памяти пользовательского процесса (отсюда и название — user-space ring): один для запросов на ввод-вывод, другой — для получения результата этих запросов. Поскольку буферы находятся в памяти процесса, взаимодействие с интерфейсом происходит без системных вызовов и не приводит к переключению контекста.

- При значении `sync` используется обычный синхронный ввод-вывод.

Интерфейс io_uring потенциально эффективнее рабочих процессов, но есть нюансы. Помимо собственно ввода-вывода, у прочитанных страниц требуется проверять контрольные суммы, а пока ввод-вывод остается буферизованным, надо еще и копировать прочитанные данные из кеша операционной системы в буферный кеш PostgreSQL. При выборе `io_uring` эта работа выполняется самим процессом, при `worker` — распределяется между рабочими процессами. Поэтому единственный интенсивно читающий процесс, скорее всего, выиграет от способа `worker`, особенно при достаточном количестве рабочих процессов, а множество читающих процессов, порождающих массу мелких одностраничных запросов, наоборот, могут упереться в возможности ограниченного пула рабочих процессов.

Асинхронный ввод-вывод настраивается, по сути, парой значений: количеством асинхронных операций ввода-вывода, которые могут выполняться одновременно, и количеством идущих подряд страниц, которые будут читаться одной операцией. Однако параметров несколько больше.

[1] backend/storage/aio/method_io_uring.c.

Параметром *io_max_concurrency* мы задаем максимальное количество асинхронных операций в расчете на один процесс. Размер структур данных, которые создаются для асинхронного ввода-вывода в разделяемой памяти при запуске экземпляра, зависит от произведения этого значения на максимальное число процессов в системе (включая служебные). При значении по умолчанию количество выбирается автоматически исходя из размеров буферного кеша, но не превышает 64[1]. -1

Само по себе значение *io_max_concurrency* не ограничивает количество операций одного процесса. Такое ограничение устанавливают два других параметра: *effective_io_concurrency* для обычного чтения данных из таблиц и *maintenance_io_concurrency* для таких служебных задач, как, например, очистка. Установка этих параметров в ноль фактически отключает асинхронный ввод-вывод. Поскольку степень параллелизма зависит от возможностей устройства ввода-вывода, эти параметры можно устанавливать и на уровне табличных пространств. 16 16

Максимальный объем данных, который может быть прочитан одной операцией ввода-вывода, устанавливается параметром *io_combine_limit*; значение по умолчанию соответствует 16 стандартным 8-килобайтным страницам. Если операционная система поддерживает более крупные операции, увеличение значения может пойти на пользу OLAP-системам, в которых большие таблицы часто сканируются полностью. 128 kB

Параметр *io_combine_limit* можно менять на лету, но в пределах, установленных другим параметром, *io_max_combine_limit*, изменение которого требует перезапуска сервера. 128 kB

Пока асинхронно выполняются только следующие операции:

- последовательное чтение (Seq Scan); *с. 374*
- сканирование по битовой карте (Bitmap Heap Scan); *с. 427*
- очистка и анализ таблиц, а также очистка B-деревьев и индексов GiST и SP-GiST; *с. 125*
- прогрев буферного кеша (расширение `pg_prewarm`);

[1] backend/storage/aio/aio_init.c, функция AioChooseMaxConcurrency.

- проверка целостности отношений (функция `verify_heapam` расширения `amcheck`).

Если асинхронный ввод-вывод отключен, при этих операциях PostgreSQL старается подсказать операционной системе глубину желаемой предвыборки, ориентируясь на значение параметра *effective_io_concurrency* или *maintenance_io_concurrency*. Также есть пара операций, которые пока не охвачены асинхронным вводом-выводом, но тоже инициируют предвыборку:

- чтение индексных страниц при удалении строк из таблицы [1];
- чтение страниц журнала при восстановлении[2].

Представление `pg_aios` показывает дескрипторы асинхронного ввода-вывода, которые определяют операции и хранят состояния этих операций, пока они выполняются. Впрочем, представление предназначено не для обычного мониторинга, а скорее для разработчиков ядра.

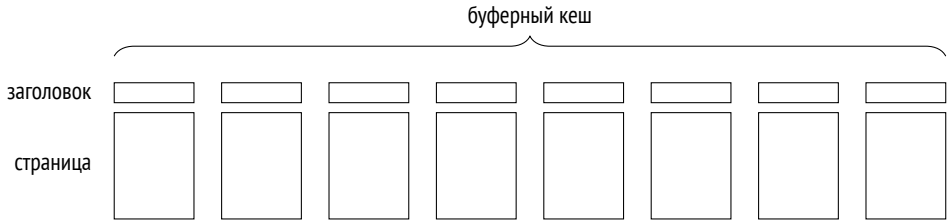
9.2. Устройство буферного кеша

Буферный кеш PostgreSQL[3] хранит страницы отношений, сглаживая разницу между временем доступа к оперативной памяти (наносекунды) и к дискам (миллисекунды).

Буферный кеш располагается в общей памяти сервера и доступен всем процессам. Он занимает бóльшую часть общей памяти и является одной из самых важных и сложных структур данных. Понимание принципа работы кеша важно само по себе; к тому же многие другие структуры (такие как вложенные транзакции, статусы транзакций clog, журнальные записи) используют похожее, хотя и более простое кеширование.

Кеш называется буферным, потому что представляет собой массив *буферов*. Каждый буфер резервирует фрагмент памяти, достаточный для одной страницы данных и ее заголовка[4].

[1] backend/access/heap/heapam.c, функция index_delete_prefetch_buffer.
[2] backend/access/transam/xlogprefetcher.c.
[3] backend/storage/buffer/README.
[4] include/storage/buf_internals.h.



Заголовок содержит информацию о буфере и загруженной в него странице, такую как:

- физическое расположение страницы (идентификатор файла отношения, слой и номер блока в слое);
- признак того, что данные на странице изменились и рано или поздно должны быть записаны на диск (такую страницу называют *грязной*);
- число обращений к буферу (usage count);
- счетчик закреплений буфера (pin count или reference count).

Чтобы получить доступ к странице данных отношения, процесс запрашивает ее у менеджера буферов[1] и получает номер буфера, содержащего эту страницу. Процесс читает данные из страницы в кеше и там же может менять их. На время работы со страницей буфер *закрепляется*, чтобы менеджер не заменил ее на другую. При каждом закреплении увеличивается счетчик обращений к странице. Помимо закреплений, применяются и другие блокировки. *с. 306*

Пока страница находится в буферном кеше, работа с ней не приводит к файловым операциям.

Внутрь буферного кеша позволяет заглянуть расширение `pg_buffercache`:

```
=> CREATE EXTENSION pg_buffercache;
```

Создадим таблицу и вставим в нее одну строку:

```
=> CREATE TABLE cacheme(
  id integer
) WITH (autovacuum_enabled = off);
```

[1] backend/storage/buffer/bufmgr.c.

```
=> INSERT INTO cacheme VALUES (1);
```

Теперь в буферном кеше содержится табличная страница с только что вставленной строкой. В этом можно убедиться, отобрав буферы, относящиеся к конкретной таблице. Такой запрос нам еще пригодится, поэтому обернем его в функцию:

```
=> CREATE FUNCTION buffercache(rel regclass)
RETURNS TABLE(
  bufferid integer, relfork text, relblk bigint,
  isdirty boolean, usagecount smallint, pins integer
) AS $$
SELECT bufferid,
  CASE relforknumber
    WHEN 0 THEN 'main'
    WHEN 1 THEN 'fsm'
    WHEN 2 THEN 'vm'
  END,
  relblocknumber,
  isdirty,
  usagecount,
  pinning_backends
FROM pg_buffercache
WHERE relfilenode = pg_relation_filenode(rel)
AND relforknumber = 0 -- только main
ORDER BY relforknumber, relblocknumber;
$$ LANGUAGE sql;
=> SELECT * FROM buffercache('cacheme');
 bufferid | relfork | relblk | isdirty | usagecount | pins
----------+---------+--------+---------+------------+------
      355 | main    |      0 | t       |          1 |    0
(1 row)
```

Страница грязная (isdirty), поскольку изменена и еще не записана на диск. Счетчик обращений (usagecount) равен единице.

9.3. Попадание в кеш

Когда менеджеру буферов требуется прочитать страницу[1], он сначала пытается найти ее в буферном кеше. Для быстрого поиска нужного буфера используется хеш-таблица[2], хранящая номера буферов.

[1] backend/storage/buffer/bufmgr.c, функция ReadBuffer_common.
[2] backend/storage/buffer/buf_table.c.

> Многие современные языки программирования включают хеш-таблицы как один из базовых типов данных. Часто они называются ассоциативным массивом. И действительно, с точки зрения пользователя они выглядят как массив, но в качестве индекса (который называется *ключом хеширования*) можно использовать любой тип данных, например текстовую строку, а не целое число.
>
> Диапазон возможных значений ключа может быть очень велик, но одновременно в хеш-таблице хранится не так много значений. Идея хеширования состоит в том, что значение ключа с помощью *хеш-функции* отображается в некоторое целое число. Полученное число или часть его битов используется как индекс обычного массива. Элементы такого массива называют *корзинами хеш-таблицы*.
>
> Хорошая хеш-функция распределяет ключи по корзинам равномерно, но тем не менее она может выдать одинаковые числа для разных ключей и таким образом направить ключи в одну и ту же корзину; это называется *коллизией*. Поэтому вместе со значением в корзине сохраняется и ключ хеширования, а при чтении значения по ключу перепроверяются все ключи, которые попали в одну корзину.

Известно много вариантов реализации хеш-таблиц. Для буферного кеша используется динамически расширяемая таблица с разрешением хеш-коллизий с помощью цепочек[1].

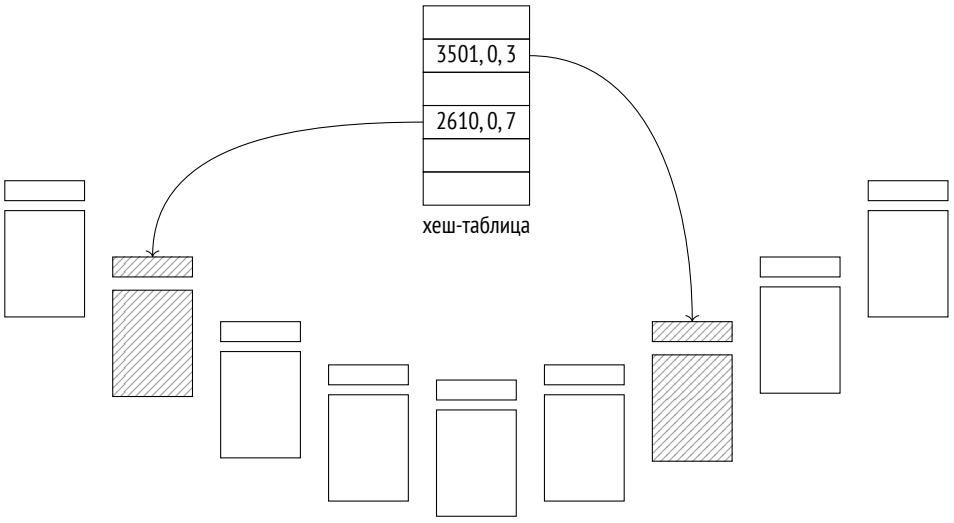
Ключом хеширования служат идентификатор файла отношения, тип слоя и номер страницы внутри файла этого слоя. Таким образом, зная страницу, можно быстро найти содержащий ее буфер или удостовериться, что страницы нет в кеше.

> Использование хеш-таблицы давно вызывает нарекания. Такая структура никак не помогает найти все буферы, занятые страницами определенного отношения. А это требуется при удалении (`DROP`), опустошении (`TRUNCATE`) или усечении хвостовой части таблицы во время очистки, чтобы убрать из кеша соответствующие страницы[2]. *с. 133*
> Но хорошую замену пока никто не предложил.

Если номер буфера найден в хеш-таблице, менеджер закрепляет этот буфер и возвращает его номер процессу. После этого процесс может работать со страницей в буферном кеше, и это не приводит к вводу-выводу.

Чтобы закрепить буфер, нужно увеличить счетчик `pin count` в его заголовке; один и тот же буфер могут одновременно закрепить несколько процессов.

[1] backend/utils/hash/dynahash.c.
[2] backend/storage/buffer/bufmgr.c, функция DropRelFileNodeBuffers.



Пока буфер закреплен (т. е. значение счетчика больше нуля), считается, что буфер используется и его содержимое не должно «радикально» измениться. Например, в странице может появиться новая версия строки (она не будет видна благодаря правилам видимости), но страница не может быть заменена на другую.

Команда EXPLAIN с параметрами analyze и buffers не просто показывает план запроса, но и выполняет его, а также выводит количество использованных буферов:

```
=> EXPLAIN (analyze, costs off, timing off, summary off)
  SELECT * FROM cacheme;
                    QUERY PLAN
-------------------------------------------------
 Seq Scan on cacheme (actual rows=1.00 loops=1)
   Buffers: shared hit=1
 Planning:
   Buffers: shared hit=2 read=6
(4 rows)
```

Здесь hit=1 говорит о том, что потребовалось прочитать одну страницу и она была найдена в кеше.

Закрепление буфера увеличило счетчик обращений на единицу:

```
=> SELECT * FROM buffercache('cacheme');
 bufferid | relfork | relblk | isdirty | usagecount | pins
----------+---------+--------+---------+------------+------
      355 | main    |      0 | t       |          2 |    0
(1 row)
```

Теперь обновим строку, чтобы в странице были две версии: одна актуальная, а вторая — мертвая. Это понадобится нам чуть позже.

```
=> UPDATE cacheme SET id = 2;
UPDATE 1
```

Воспользуемся тем, что открытый курсор удерживает закрепление, чтобы иметь возможность быстро прочитать следующую строку выборки. Это позволит нам увидеть закрепление в процессе работы запроса:

```
=> BEGIN;
=> DECLARE c CURSOR FOR SELECT * FROM cacheme;
=> FETCH c;
 id
----
  2
(1 row)
=> SELECT * FROM buffercache('cacheme');
 bufferid | relfork | relblk | isdirty | usagecount | pins
----------+---------+--------+---------+------------+------
      355 | main    |      0 | t       |          4 |    1
(1 row)
```

В некоторых случаях, когда требуется именно закрепленный буфер, процесс встает в очередь и засыпает до того момента, когда он сможет получить монопольный доступ к буферу. Пример такой операции — заморозка[1]. *с. 134*

Но, как правило, если процессу мешает закрепление, он просто пропускает такой буфер и выбирает другой. Примером может служить очистка таблицы. В данном случае страница будет пропущена, поскольку закрепление, установленное другим процессом, препятствует физическому удалению мертвой версии:

[1] backend/storage/buffer/bufmgr.c, функция LockBufferForCleanup.

```
=> VACUUM VERBOSE cacheme;
INFO:  vacuuming "internals.public.cacheme"
INFO:  finished vacuuming "internals.public.cacheme": index
scans: 0
pages: 0 removed, 1 remain, 1 scanned (100.00% of total), 0
eagerly scanned
tuples: 0 removed, 2 remain, 0 are dead but not yet removable
tuples missed: 1 dead from 1 pages not removed due to cleanup
lock contention
removable cutoff: 908, which was 0 XIDs old when operation ended
new relfrozenxid: 905, which is 1 XIDs ahead of previous value
frozen: 0 pages from table (0.00% of total) had 0 tuples frozen
visibility map: 0 pages set all-visible, 0 pages set all-frozen
(0 were all-visible)
index scan not needed: 0 pages from table (0.00% of total) had 0
dead item identifiers removed
avg read rate: 0.000 MB/s, avg write rate: 137.061 MB/s
buffer usage: 33 hits, 0 reads, 3 dirtied
WAL usage: 4 records, 3 full page images, 25030 bytes, 0 buffers
full
system usage: CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s
VACUUM
```

При переходе курсора на другую страницу или после его закрытия закрепление снимается. В нашем примере это происходит одновременно с окончанием транзакции:

```
=> COMMIT;
=> SELECT * FROM buffercache('cacheme') WHERE relfork = 'main';
 bufferid | relfork | relblk | isdirty | usagecount | pins
----------+---------+--------+---------+------------+------
      355 | main    |      0 | t       |          4 |    0
(1 row)
```

Так же закрепление работает и при изменении страницы. Например, добавим в таблицу еще одну строку (она попадет в ту же самую страницу):

```
=> INSERT INTO cacheme VALUES (2);
=> SELECT * FROM buffercache('cacheme') WHERE relfork = 'main';
 bufferid | relfork | relblk | isdirty | usagecount | pins
----------+---------+--------+---------+------------+------
      355 | main    |      0 | t       |          5 |    0
(1 row)
```

Страница не записывается на диск немедленно, а остается грязной в буферном кеше: происходит экономия и на чтении с диска, и на записи.

9.4. Промах кеша

Если в хеш-таблице не нашлось записи, соответствующей странице, значит, эта страница отсутствует в буферном кеше. В таком случае для страницы выбирается (и тут же закрепляется) новый буфер, страница читается в этот буфер, а ссылки в хеш-таблице изменяются соответствующим образом.

Уберем страницу из буфера (вместо этого можно было бы очистить весь буферный кеш, перезапустив экземпляр): v. 17

```
=> SELECT pg_buffercache_evict(355);
 pg_buffercache_evict
----------------------
 (t,t)
(1 row)
```

Попытка прочитать страницу приведет к промаху кеша, и страница будет загружена в заново отведенный для нее буфер:

```
=> EXPLAIN (analyze, costs off, timing off, summary off)
  SELECT * FROM cacheme;
                   QUERY PLAN
-------------------------------------------------
 Seq Scan on cacheme (actual rows=2.00 loops=1)
   Buffers: shared read=1 dirtied=1
 Planning:
   Buffers: shared hit=2
(4 rows)
```

Теперь вместо `hit` команда показывает `read`, что означает промах кеша. При этом в результате запроса страница стала грязной (`dirtied`), так как запрос изменил биты-подсказки на странице. *с. 86*

Запрос к буферному кешу покажет, что счетчик использований только что добавленной страницы равен единице:

```
=> SELECT * FROM buffercache('cacheme');
 bufferid | relfork | relblk | isdirty | usagecount | pins
----------+---------+--------+---------+------------+------
      378 | main    |      0 | t       |          1 |    0
(1 row)
```

Общую статистику использования буферного кеша для таблицы можно получить из представления `pg_statio_all_tables`:

```
=> SELECT heap_blks_read, heap_blks_hit
FROM pg_statio_all_tables
WHERE relname = 'cacheme';
 heap_blks_read | heap_blks_hit
----------------+---------------
              1 |             9
(1 row)
```

Аналогичные представления есть для индексов и для последовательностей.

Поиск буфера и вытеснение

Поиск подходящего буфера[1] — непростая задача. Есть два варианта развития событий.

1. Сразу после запуска сервера кеш содержит только пустые, свободные буферы, и все они связаны в общий список.

   Пока остаются свободные буферы, первый из них считается подходящим и убирается из списка.

   Буфер может вернуться[2] в список свободных буферов только в том случае, когда страница исчезает, а не заменяется другой страницей. Это происходит при удалении (`DROP`), опустошении (`TRUNCATE`) или усечении хвостовой части таблиц во время очистки.

с. 133

2. Рано или поздно свободные буферы заканчиваются (поскольку обычно размер базы данных превышает объем памяти, выделенной под кеш), и тогда менеджер должен выбрать один из занятых буферов и вытеснить

[1] backend/storage/buffer/freelist.c, функция StrategyGetBuffer.
[2] backend/storage/buffer/freelist.c, функция StrategyFreeBuffer.

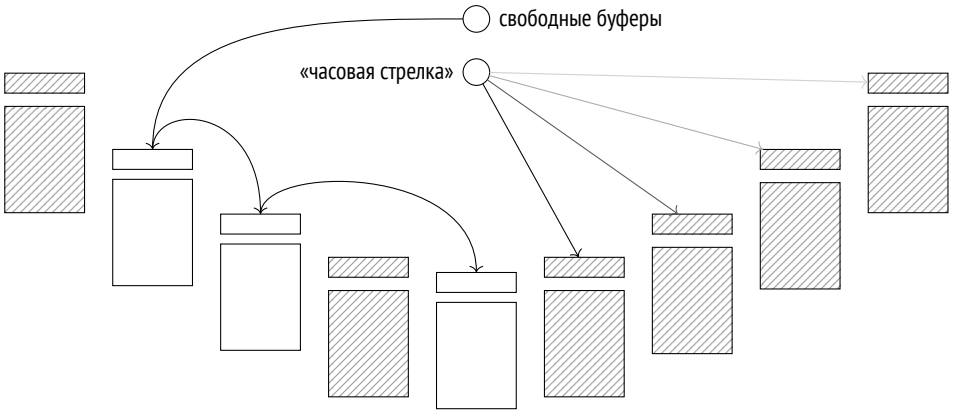
находящуюся там страницу. Для этого используется «часовой» алгоритм (clock sweep). Имеется «часовая стрелка», указывающая на один из буферов. Стрелка пробегает буферный кеш по кругу, уменьшая счетчики обращений к страницам на единицу. Поиск завершается, когда стрелка доходит до незакрепленного буфера с нулевым счетчиком.

С одной стороны, счетчик обращений увеличивается каждый раз, когда буфер используется (закрепляется). С другой — уменьшается при поиске буфера для вытеснения. Таким образом, в первую очередь вытесняются те страницы, к которым реже обращались в последнее время, а активно используемые страницы остаются в кеше.

Можно заметить, что если все буферы имеют ненулевой счетчик обращений, то «часовой стрелке» придется сделать больше одного круга, уменьшая счетчики, пока какой-нибудь из них не обратится наконец в ноль. Чтобы избежать долгого «наматывания кругов», максимальное значение счетчика обращений ограничено числом 5.

После того как буфер найден, надо удалить из хеш-таблицы ссылку на старую страницу, еще находящуюся в буфере.

Но если буфер оказался грязным, то есть содержит измененные данные, старую страницу нельзя просто выбросить — сначала менеджер буферов сохраняет ее на диск. *с. 226*



В найденный буфер — среди свободных или среди занятых — менеджер буферов читает новую страницу. Для этого используется буферизованный

ввод-вывод, поэтому страница будет прочитана с диска только в том случае, если операционная система не найдет ее в своем собственном кеше.

> СУБД, использующие прямой ввод-вывод и не зависящие от кеша операционной системы, различают логические чтения (из оперативной памяти, то есть из буферного кеша) и физические чтения (с диска). С точки зрения PostgreSQL страница либо прочитана из буферного кеша, либо запрошена у операционной системы, но у СУБД нет возможности узнать, была ли она в последнем случае найдена в оперативной памяти или прочитана с диска.

В хеш-таблицу помещается ссылка на новую страницу, а буфер закрепляется. Это приводит к увеличению счетчика обращений, то есть к установке его в единицу. В итоге у буфера появляется время нарастить счетчик обращений, пока «часовая стрелка» обходит по кругу буферный кеш.

9.5. Массовое вытеснение

При операциях, выполняющих массовое чтение или запись данных, есть опасность быстрого вытеснения полезных страниц из буферного кеша «одноразовыми» данными.

Чтобы этого избежать, для массовых операций в буферном кеше используются относительно небольшие *буферные кольца* (buffer ring), и вытеснение происходит в их пределах, не затрагивая остальные буферы.

> Наряду с термином «buffer ring» в коде используется и «ring buffer», то есть «кольцевой буфер». Но такой перевод вызвал бы путаницу, поскольку сам (кольцевой) буфер состоит из буферов (буферного кеша). Термин «кольцо» лишен такого недостатка.

Буферное кольцо заданного размера представляет собой массив буферов, которые используются «по кругу», один за другим. Сначала буферное кольцо пусто, и буферы набираются в него из буферного кеша обычным образом. А затем начинает действовать вытеснение, но только внутри кольца[1].

При этом буферы, входящие в кольцо, не исключаются из буферного кеша и в принципе могут быть использованы и другими операциями. Поэтому если очередной буфер оказывается закрепленным или имеет более одного

[1] backend/storage/buffer/freelist.c, функция GetBufferFromRing.

обращения, он убирается из буферного кольца и заменяется каким-нибудь другим буфером.

Существует три стратегии вытеснения[1].

Стратегия массового чтения (bulkread) применяется при последовательном сканировании больших таблиц, размер которых превышает четверть буферного кеша. *с. 374*

При использовании этой стратегии грязные страницы не записываются на диск, чтобы освободить буфер; вместо этого буфер отключается от кольца и заменяется другим. Так чтению не надо дожидаться записи, и оно выполняется быстрее.

Если в момент начала сканирования таблицы процесс обнаруживает, что эта таблица уже сканируется, он присоединяется к существующему буферному кольцу и получает часть таблицы без избыточного ввода-вывода[2]. Когда первый процесс заканчивает сканирование, второй отдельно дочитывает пропущенное начало таблицы.

Кольцо должно быть достаточно большим, чтобы буферы успевали освобождаться до того, как потребуются повторно, и задерживались в кольце достаточно долго, чтобы несколько читающих процессов могли синхронизироваться. При асинхронном вводе-выводе одновременно могут выполняться *effective_io_concurrency* операций, каждая из которых может читать *io_combine_limit* байт, что и определяет необходимый размер. Если асинхронный ввод-вывод отключен, произведение обратится в ноль; на этот случай добавляется еще 256 Кбайт (32 стандартные страницы). Но чтобы кольцо не получилось уж слишком большим, его размер ограничен отношением объема буферного кеша к максимальному количеству процессов в системе: кеша должно хватить на всех. v. 18 *с. 183* 16 128K

Таким образом, с настройками по умолчанию — при достаточном большом буферном кеше — размер кольца будет составлять $32 + 16 \cdot 16 = 288$ буферов.

Стратегия массовой записи (bulkwrite) применяется для операций `COPY FROM`, `CREATE TABLE AS SELECT`, `CREATE MATERIALIZED VIEW` и тех вариантов

[1] backend/storage/buffer/freelist.c, функция GetAccessStrategy.
[2] backend/access/common/syncscan.c.

`ALTER TABLE`, которые вызывают перезапись таблицы. Выделяется довольно большое кольцо размером 16 Мбайт (2048 стандартных страниц), но не больше $\frac{1}{8}$ от размера всего буферного кеша.

Стратегия очистки (vacuum) применяется процессами очистки и анализа при полном сканировании таблиц без учета карты видимости. Размер кольца определяется параметром *vacuum_buffer_usage_limit*, но может быть переопределен в команде `VACUUM` или `ANALYZE`.

2MB v. 16

Буферные кольца не всегда защищают от ненужного вытеснения. При изменении большого числа строк командами `UPDATE` или `DELETE` таблица сканируется, используя стратегию массового чтения, но из-за того, что страницы изменяются, буферное кольцо вырождается и фактически не работает.

Другой важный случай — хранение в базе данных больших значений, для которых используется TOAST. Доступ к toast-таблице всегда происходит по индексу, несмотря на потенциально большой объем читаемых данных, и, следовательно, не использует буферное кольцо.

с. 33

Посмотрим теперь на стратегию массового чтения. Для упрощения расчетов создадим таблицу так, чтобы при вставке одна строка занимала целую страницу. Размер буферного кеша по умолчанию — 16 384 страницы по 8 Кбайт. Значит, чтобы сканирование таблицы использовало буферное кольцо, таблица должна занимать больше 4096 страниц.

```
=> CREATE TABLE big(
  id integer PRIMARY KEY GENERATED ALWAYS AS IDENTITY,
  s char(1000)
) WITH (fillfactor = 10);
=> INSERT INTO big(s) SELECT 'FOO' FROM generate_series(1,4096+1);
```

Проанализируем таблицу:

```
=> ANALYZE big;
=> SELECT relname, relfilenode, relpages
FROM pg_class
WHERE relname IN ('big', 'big_pkey');
```

```
 relname  | relfilenode | relpages
----------+-------------+----------
 big      |       16569 |     4097
 big_pkey |       16575 |       14
(2 rows)
```

Очистим кеш от табличных страниц, прочитанных во время анализа, после чего прочитаем всю таблицу: v. 18

```
=> SELECT pg_buffercache_evict_relation('big'::regclass);
 pg_buffercache_evict_relation
-------------------------------
 (4101,4099,0)
(1 row)
=> EXPLAIN (analyze, costs off, timing off, summary off, buffers off)
SELECT * FROM big;
                  QUERY PLAN
-----------------------------------------------
 Seq Scan on big (actual rows=4097.00 loops=1)
(1 row)
```

Размер буферного кольца ограничен из-за небольшого объема буферного кеша — табличными страницами в нем занято только 94 буфера вместо ожидаемых 288:

```
=> SELECT count(*)
FROM pg_buffercache
WHERE relfilenode = pg_relation_filenode('big'::regclass);
 count
-------
    94
(1 row)
```

Если же таблица читается с помощью индекса, то буферное кольцо не используется:

```
=> EXPLAIN (analyze, costs off, timing off, summary off, buffers off)
SELECT * FROM big
ORDER BY id;
                           QUERY PLAN
-----------------------------------------------------------------
 Index Scan using big_pkey on big (actual rows=4097.00 loops=1)
   Index Searches: 1
(2 rows)
```

В результате в буферном кеше оказывается вся таблица и весь индекс:

```
=> SELECT relfilenode, count(*)
FROM pg_buffercache
WHERE relfilenode IN (
  pg_relation_filenode('big'), pg_relation_filenode('big_pkey')
)
GROUP BY relfilenode;
 relfilenode | count
-------------+-------
       16569 |  4097
       16575 |    14
(2 rows)
```

9.6. Настройка размера

128MB Размер буферного кеша определяется параметром *shared_buffers*. Значение по умолчанию заведомо занижено; имеет смысл увеличить его сразу же после установки PostgreSQL. Изменение параметра требует перезапуска, поскольку общая память выделяется под кеш при старте сервера.

Из каких соображений выбирать подходящее значение?

Даже самая большая база имеет ограниченный набор «горячих» данных, с которыми ведется активная работа в каждый момент времени. В идеале именно этот набор и должен помещаться в буферный кеш (конечно, с запасом под неизбежные «одноразовые» данные). Если размер кеша будет меньше, то активно используемые страницы будут постоянно вытеснять друг друга, создавая избыточный ввод-вывод. Но и бездумно увеличивать кеш тоже неправильно, поскольку это повышает накладные расходы на его поддержание, а оперативная память требуется и для других нужд.

Оптимальный размер буферного кеша будет разным в разных системах: он зависит и от общего доступного объема памяти, и от данных, и от профиля нагрузки. К сожалению, нет такого волшебного значения или даже формулы, одинаково хорошо подходящей всем.

Надо также принимать во внимание, что — пока не включен прямой ввод-вывод — промах кеша PostgreSQL не обязательно приводит к физическому

вводу-выводу. При небольшом буферном кеше свободное место будет задействовано под кеш операционной системы, что может в определенной степени сглаживать проблему. Но, в отличие от СУБД, операционная система ничего не знает о смысле прочитанных данных, поэтому использует другую стратегию вытеснения.

Стандартная рекомендация — взять в качестве первого приближения $\frac{1}{4}$ оперативной памяти, а дальше смотреть по ситуации.

Лучше всего провести эксперименты: увеличить или уменьшить размер кеша и сравнить характеристики системы. Конечно, для этого надо иметь тестовый стенд, аналогичный производственной системе, и уметь воспроизводить типовую нагрузку.

Но некоторый анализ все же возможен.

Во-первых, с помощью уже знакомого расширения `pg_buffercache` можно смотреть на текущее состояние кеша под разными углами. Например, изучить распределение буферов по степени их использования:

```
=> SELECT usagecount, count(*)
FROM pg_buffercache
GROUP BY usagecount
ORDER BY usagecount;
 usagecount | count
------------+-------
          1 |  4148
          2 |   146
          3 |    21
          4 |    15
          5 |   209
            | 11845
(6 rows)
```

Пустые значения счетчика соответствуют свободным буферам. В данном случае это неудивительно, поскольку сервер практически бездействует. Большинство занятых буферов содержат страницы системных таблиц, которые были прочитаны обслуживающим процессом для заполнения своего кеша системного каталога и для выполнения запросов.

Можно посмотреть, какая часть каждого из отношений закеширована и насколько активно используются эти данные (в этом запросе активными считаются страницы со счетчиком использования больше единицы):

```
=> SELECT c.relname,
  count(*) blocks,
  round( 100.0 * 8192 * count(*) /
    pg_table_size(c.oid) ) AS "% of rel",
  round( 100.0 * 8192 * count(*) FILTER (WHERE b.usagecount > 1) /
    pg_table_size(c.oid) ) AS "% hot"
FROM pg_buffercache b
  JOIN pg_class c ON pg_relation_filenode(c.oid) = b.relfilenode
WHERE b.reldatabase IN (
  0, -- общие объекты кластера
  (SELECT oid FROM pg_database
   WHERE datname = current_database())
)
AND b.usagecount IS NOT NULL
GROUP BY c.relname, c.oid
ORDER BY 2 DESC
LIMIT 5;
```

```
   relname    | blocks | % of rel | % hot
--------------+--------+----------+-------
 big          |   4097 |      100 |     2
 pg_attribute |     58 |       94 |    50
 pg_proc      |     24 |       20 |    15
 pg_class     |     17 |       94 |    67
 big_pkey     |     14 |      100 |   100
(5 rows)
```

Здесь видно, что таблица big и ее индекс полностью закешированы, но их страницы не используются активно.

Стоит учитывать, что запросы к pg_buffercache:

- надо повторять несколько раз, так как цифры будут меняться в определенных пределах;
- не надо выполнять непрерывно, потому что представление блокирует, хоть и кратковременно, просматриваемые буферы.

v. 16 Поэтому для мониторинга лучше пользоваться другими функциями расширения — pg_buffercache_summary и pg_buffercache_usage_counts, — которые требуют меньше блокировок и за счет этого позволяют быстро оценить общую (хотя, возможно, немного рассогласованную) картину происходящего.

v. 16 Во-вторых, подробная информация о вводе-выводе, проходящем через буферный кеш, накапливается в представлении pg_stat_io. Например, можно

изучить количество операций ввода-вывода в разбивке по стратегиям использования буферных колец (контекст `normal` соответствует обычным операциям без кольца):

```
=> SELECT context,
  sum(reads) AS reads, sum(writes) AS writes,
  sum(hits) AS hits, sum(evictions) AS evictions, sum(reuses) AS reuses
FROM pg_stat_io
WHERE object = 'relation'
  AND context in ('normal','bulkread','bulkwrite','vacuum')
GROUP BY context;
```

```
  context  | reads  | writes |   hits   | evictions | reuses
-----------+--------+--------+----------+-----------+--------
 vacuum    | 138091 | 539140 |    89134 |      1846 | 656633
 normal    |  95057 | 342503 | 71883823 |    271326 |
 bulkwrite |      0 |      0 |        8 |         0 |      0
 bulkread  |  30506 | 155238 |    77778 |    175469 |  99285
(4 rows)
```

Запрос также показывает число попаданий в кеш (`hits`) и вытеснений из кеша: «обычные» вытеснения учитываются в поле `evictions`, а вытеснения страниц из буферного кольца — в поле `reuses`. Из этих цифр можно сделать определенные выводы о характере использования кеша.

Кроме количества операций чтения и записи (`reads` и `writes`), из представления можно узнать и объем ввода-вывода (`read_bytes` и `write_bytes`) в байтах: v. 18

```
=> SELECT context,
  pg_size_pretty(read_bytes) AS read_bytes,
  round(read_bytes/reads/8192, 1) AS pages_per_read,
  pg_size_pretty(write_bytes) AS write_bytes,
  round(write_bytes/writes/8192, 1) AS pages_per_write
FROM (
  SELECT context,
    sum(reads) AS reads,
    sum(read_bytes) AS read_bytes,
    sum(writes) AS writes,
    sum(write_bytes) AS write_bytes
  FROM pg_stat_io
  WHERE object = 'relation'
  GROUP BY context
  HAVING sum(reads) + sum(writes) > 0
);
```

```
 context  | read_bytes | pages_per_read | write_bytes | pages_per_write
----------+------------+----------------+-------------+-----------------
 vacuum   | 5147 MB    |            4.8 | 4212 MB     |             1.0
 normal   | 760 MB     |            1.0 | 2676 MB     |             1.0
 bulkread | 2155 MB    |            9.0 | 1213 MB     |             1.0
(3 rows)
```

Здесь мы видим, что очистка и последовательное сканирование читают сразу несколько страниц за одну операцию ввода-вывода, а вот запись пока всегда одностраничная.

При включенном параметре *track_io_timing* в этом же представлении отслеживается и время, затраченное на ввод-вывод. Однако, как я уже говорил, такое отслеживание может приводить к существенным накладным расходам на некоторых платформах.

с. 166

Представление `pg_stat_io` показывает данные, сгруппированные по типам процессов. Детализацию до конкретного обслуживающего процесса (пока он работает) можно получить функцией `pg_stat_get_backend_io`, которая возвращает таблицу с тем же набором полей.

v. 18

Изучение данных `pg_buffercache` и `pg_stat_io` в различных разрезах может дать полезную информацию для размышлений.

9.7. Прогрев кеша

После перезапуска сервера должно пройти некоторое время, чтобы кеш «прогрелся» — набрал актуальные активно использующиеся данные. Иногда может оказаться полезным сразу прочитать в кеш данные определенных таблиц, и для этого предназначено расширение `pg_prewarm`:

```
=> CREATE EXTENSION pg_prewarm;
```

Кроме обычного чтения таблиц в буферный кеш (или только в кеш операционной системы), расширение позволяет сохранять актуальное состояние кеша на диск и восстанавливать его же после перезагрузки сервера. Для этого надо добавить библиотеку расширения в параметр *shared_preload_libraries* и перезагрузить сервер:

```
=> ALTER SYSTEM SET shared_preload_libraries = 'pg_prewarm';
```

```
postgres$ pg_ctl restart -l /home/postgres/logfile
```

После перезапуска, если значение параметра *pg_prewarm.autoprewarm* не менялось, будет автоматически запущен фоновый процесс autoprewarm leader, который раз в *pg_prewarm.autoprewarm_interval* единиц времени будет сбрасывать на диск список страниц, находящихся в кеше (этот процесс использует один из слотов *max_parallel_processes*).

on

300s

> Изначально процесс назывался autoprewarm master, но в версии 13 и он попал под раздачу.

```
postgres$ ps -o pid,command \
--ppid `head -n 1 /usr/local/pgsql/data/postmaster.pid` | \
grep prewarm
  57518 postgres: autoprewarm leader
```

Сейчас, после перезагрузки, таблица big в кеше отсутствует:

```
=> SELECT count(*)
FROM pg_buffercache
WHERE relfilenode = pg_relation_filenode('big'::regclass);
 count
-------
     0
(1 row)
```

Если есть обоснованные предположения, что все содержимое таблицы будет активно использоваться, а доступ к диску недопустимо снизит время отклика запросов, таблицу можно заранее прочитать в буферный кеш:

```
=> SELECT pg_prewarm('big');
 pg_prewarm
------------
       4097
(1 row)
=> SELECT count(*)
FROM pg_buffercache
WHERE relfilenode = pg_relation_filenode('big'::regclass);
 count
-------
  4097
(1 row)
```

Список страниц сбрасывается в файл PGDATA/autoprewarm.blocks. Можно подождать, пока процесс autoprewarm leader отработает в первый раз, но мы инициируем запись вручную:

```
=> SELECT autoprewarm_dump_now();
 autoprewarm_dump_now
----------------------
                 4262
(1 row)
```

Число сброшенных страниц больше 4097, поскольку включает все занятые буферы. Файл записывается в текстовом формате и содержит идентификаторы базы данных, табличного пространства и файла, номер слоя и номер блока:

```
postgres$ head -n 10 /usr/local/pgsql/data/autoprewarm.blocks
<<4262>>
0,1664,1262,0,0
0,1664,1260,0,0
16391,1663,1259,0,0
16391,1663,1259,0,1
16391,1663,1259,0,2
16391,1663,1259,0,3
16391,1663,1259,0,4
16391,1663,1259,0,5
16391,1663,1259,0,6
```

Теперь снова перезапустим сервер.

```
postgres$ pg_ctl restart -l /home/postgres/logfile
```

Таблица сразу же оказывается в кеше:

```
=> SELECT count(*)
FROM pg_buffercache
WHERE relfilenode = pg_relation_filenode('big'::regclass);
 count
-------
  4097
(1 row)
```

Это обеспечивается тем же самым процессом autoprewarm leader: он читает файл, разделяет страницы по базам данных, сортирует их (чтобы чтение

с диска было по возможности последовательным) и передает отдельному рабочему процессу autoprewarm worker для обработки.

9.8. Локальный кеш

Исключением из общего правила являются временные таблицы. Поскольку временные данные видны только одному процессу, им нечего делать в общем буферном кеше. Поэтому для временных данных используется кеш в локальной памяти процесса, владеющего таблицей[1].

В целом локальный буферный кеш устроен так же, как и общий:

- для поиска страниц используется собственная хеш-таблица;
- страницы вытесняются по обычному алгоритму (но без использования буферных колец);
- страницы могут закрепляться, чтобы избежать вытеснения.

Однако его реализация серьезно упрощается, поскольку отпадает необходимость в блокировках в оперативной памяти (так как доступ к буферам имеет только один процесс) и в защите данных от сбоя (так как данные в любом случае существуют максимум до конца сеанса). *с. 306*, *с. 211*

Память под локальный кеш выделяется постепенно, по мере необходимости, поскольку временные таблицы используются далеко не во всех сеансах. Максимальный объем кеша одного сеанса ограничен параметром *temp_buffers*. `8MB`

> Параметр с созвучным именем *temp_file_limit* относится не к временным таблицам, а к файлам, которые могут создаваться при выполнении запросов для временного хранения промежуточных данных.

В выводе команды `EXPLAIN` обращения к локальному буферному кешу отмечаются словом `local` вместо `shared`:

[1] backend/storage/buffer/localbuf.c.

```
=> CREATE TEMPORARY TABLE tmp AS SELECT 1;
=> EXPLAIN (analyze, costs off, timing off, summary off)
  SELECT * FROM tmp;
                QUERY PLAN
-------------------------------------------
 Seq Scan on tmp (actual rows=1.00 loops=1)
   Buffers: local hit=1
 Planning:
   Buffers: shared hit=12 read=7
(4 rows)
```

Данные о вводе-выводе страниц локальных кешей процессов позволяет по-
v. 16 лучить уже знакомое нам представление pg_stat_io:

```
=> SELECT context, sum(reads) reads, sum(writes) writes,
  sum(hits) hits, sum(evictions) evictions, sum(reuses) reuses
FROM pg_stat_io
WHERE object = 'temp relation'
GROUP BY context;
 context | reads | writes | hits | evictions | reuses
---------+-------+--------+------+-----------+--------
 normal  |     0 |      0 |    1 |         0 |
(1 row)
```

10

Журнал предзаписи

10.1. Журналирование

В случае сбоя, например при выключении электропитания либо при отказе СУБД или операционной системы, теряется все содержимое оперативной памяти; остается только информация, записанная на диск. Чтобы сервер мог стартовать после сбоя, необходимо восстановить согласованность данных. Если при сбое пострадал сам диск, та же задача возникает и при восстановлении из резервной копии.

Один из теоретически возможных способов — все время поддерживать данные на диске в согласованном виде. Фактически это означает необходимость постоянной записи отдельных страниц на диск (а случайная запись обходится дороже последовательной), причем в таком порядке, что ни в какой момент времени согласованность не нарушается (а это трудно, особенно для сложно организованных индексов).

PostgreSQL, как и большинство СУБД, использует другой подход.

В процессе работы часть актуальных данных хранится только в оперативной памяти и записывается на диск (или на другой энергонезависимый носитель) отложенно. Поэтому данные на диске рассогласованы в течение всего времени работы сервера — разные страницы оказываются записанными по состоянию на разные моменты времени. При этом каждое действие в оперативной памяти (например, изменение страницы в буферном кеше) *журналируется*: создается и сохраняется на диске журнальная запись,

содержащая минимальную информацию, достаточную для повторения того же действия в случае необходимости[1].

Журнальная запись об изменении страницы в обязательном порядке попадает на диск *перед тем*, как туда попадет измененная страница. Отсюда и название: *журнал предзаписи* (write-ahead log). Это требование дает возможность в случае сбоя прочитать с диска журнал и повторить те операции, которые были выполнены до сбоя, но результат которых не успел дойти до диска из оперативной памяти и пропал.

Как правило, вести журнал эффективнее, чем записывать отдельные страницы. Журнал представляет собой последовательный поток, с записью которого неплохо справляются даже HDD-диски. К тому же журнальная запись может быть меньше страницы по размеру.

Журналировать нужно все операции, при выполнении которых есть риск получить несогласованность на диске в случае сбоя. В частности, в журнал записываются следующие действия:

- изменение страниц отношений в буферном кеше — сброс измененной страницы на диск откладывается;
- фиксация и отмена транзакций — изменение статуса происходит в буферах clog и тоже попадает на диск не сразу;
- файловые операции (создание и удаление файлов и каталогов, например, при создании таблицы) — такие операции должны происходить синхронно с изменением данных.

В журнал не записываются:

- операции с нежурналируемыми (`UNLOGGED`) таблицами;

v. 15

- операции с нежурналируемыми последовательностями (как правило, относящимися к нежурналируемым таблицам);
- операции с временными таблицами и последовательностями — время жизни таких объектов в любом случае не превышает время жизни создавшего их сеанса.

[1] postgrespro.ru/docs/postgresql/18/wal-intro.

Журнал используется не только для восстановления после сбоя, но и для восстановления на произвольный момент времени из резервной копии, а также для репликации.

10.2. Устройство журнала

Логическая структура

Логически журнал[1] можно представить себе как последовательность записей различной длины. Каждая запись содержит *данные* о некоторой операции, предваренные стандартным *заголовком*[2]. В заголовке, в числе прочего, указаны:

- номер транзакции, к которой относится запись;
- менеджер ресурсов, ответственный за интерпретацию записи[3];
- контрольная сумма для обнаружения повреждений данных;
- длина записи;
- ссылка на предыдущую запись журнала.

  Обычно журнал читается в прямом направлении, но, например, утилита pg_rewind просматривает его в обратном.

Сами данные имеют разный формат и смысл. Например, они могут представлять собой некоторый фрагмент страницы, который надо записать поверх ее содержимого с определенным смещением. Менеджер ресурсов «понимает», как расшифровать и воспроизвести запись. Есть отдельные менеджеры для таблиц, для каждого типа индекса, для статуса транзакций и других сущностей. Полный их список можно получить командой `pg_waldump -r list`.

[1] postgrespro.ru/docs/postgresql/18/wal-internals; backend/access/transam/README.
[2] include/access/xlogrecord.h.
[3] include/access/rmgrlist.h.

В разделяемой памяти сервера для журнала выделены специальные бу-
-1 феры. Размер журнального кеша задается параметром *wal_buffers*. Значе-
ние по умолчанию подразумевает автоматическую настройку: выделяется
с. 216 16 Мбайт (размер журнального сегмента) или меньше при совсем небольших размерах буферного кеша.

Журнальный кеш похож по своему устройству на буферный, но работает преимущественно в режиме кольцевого буфера: записи добавляются в «голову», а сохраняются на диск с «хвоста». Слишком маленький размер журнального кеша приведет к тому, что синхронизация с диском будет выполняться чаще, чем необходимо.

Позиция уже сохраненных записей («хвост» буфера) и позиция вставки («голова» буфера) в ненагруженной системе почти всегда будут совпадать:

```
=> SELECT pg_current_wal_lsn(), pg_current_wal_insert_lsn();
 pg_current_wal_lsn | pg_current_wal_insert_lsn
--------------------+---------------------------
 2/8121CB28         | 2/8121CB28
(1 row)
```

Чтобы сослаться на определенную запись, используется тип данных pg_lsn (log sequence number, LSN), представляющий 64-битное смещение в байтах до журнальной записи относительно начала журнала. LSN выводится как два 32-битных числа в шестнадцатеричной системе через косую черту.

Создадим таблицу:

```
=> CREATE TABLE wal(id integer);
=> INSERT INTO wal VALUES (1);
```

Начнем транзакцию и запомним позицию вставки в журнал:

```
=> BEGIN;
=> SELECT pg_current_wal_insert_lsn();
 pg_current_wal_insert_lsn
---------------------------
 2/81235AE8
(1 row)
```

Теперь выполним какую-нибудь операцию, например обновим строку:

```
=> UPDATE wal SET id = id + 1;
```

Изменение выполняется в странице, находящейся в буферном кеше в оперативной памяти. Информация об изменении сохраняется и в журнальную страницу, также в оперативной памяти. Поэтому позиция вставки увеличивается:

```
=> SELECT pg_current_wal_insert_lsn();
 pg_current_wal_insert_lsn
---------------------------
 2/81235B30
(1 row)
```

Чтобы гарантировать, что измененная страница данных будет вытеснена на диск строго позже, чем журнальная запись, в заголовке страницы сохраняется номер LSN последней записи, относящейся к этой странице. Позицию можно проверить с помощью расширения `pageinspect`:

```
=> SELECT lsn FROM page_header(get_raw_page('wal',0));
    lsn
------------
 2/81235B30
(1 row)
```

Для всего кластера баз данных используется один общий журнал, и в него все время попадают новые записи. Поэтому номер LSN на странице может оказаться меньше, чем значение, которое вернула чуть раньше функция `pg_current_wal_insert_lsn`. Но в системе, где ничего не происходит, цифры совпадут.

Теперь завершим транзакцию:

```
=> COMMIT;
```

Запись о фиксации также попадает в журнал, и позиция снова меняется:

```
=> SELECT pg_current_wal_insert_lsn();
 pg_current_wal_insert_lsn
---------------------------
 2/81235B58
(1 row)
```

с. 85 Фиксация меняет статус транзакции в страницах clog, располагающихся
с. 309 в SLRU-кеше. Чтобы страницу clog нельзя было вытеснить на диск раньше, чем соответствующую журнальную запись, для страниц clog тоже отслеживается номер LSN последней журнальной записи. Но эта информация хранится не в самой странице, а в оперативной памяти.

с. 235 В какой-то момент созданные журнальные записи окажутся на диске. После этого страницы данных и clog могут быть вытеснены из кеша. Если бы потребовалось вытеснить их раньше, это было бы обнаружено, и журнальные записи были бы принудительно сброшены на диск первыми[1].

Зная две позиции LSN, можно получить размер журнальных записей между ними (в байтах) простым вычитанием одной позиции из другой. Надо только привести позиции к типу pg_lsn:

```
=> SELECT '2/81235B58'::pg_lsn - '2/81235AE8'::pg_lsn;
 ?column?
----------
      112
(1 row)
```

В данном случае обновление строки и фиксация потребовали около сотни байтов в журнале.

Таким же способом можно оценить, какой объем журнальных записей генерируется сервером за единицу времени при определенной нагрузке. Эта информация потребуется при настройке контрольной точки.

Физическая структура

На диске журнал хранится в каталоге PGDATA/pg_wal в виде файлов-сегмен-
16MB тов. Параметр *wal_segment_size*, доступный только для чтения, показывает размер сегмента.

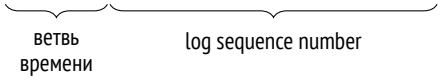
[1] backend/storage/buffer/bufmgr.c, функция FlushBuffer.

Для нагруженных систем увеличение размера сегментов может снизить накладные расходы, но эта настройка выполняется только при инициализации кластера (`initdb --wal-segsize`).

Журнальные записи попадают в текущий файл; когда он заканчивается — начинает использоваться следующий.

Можно узнать, в каком файле мы найдем нужную позицию и с каким смещением от начала файла:

```
=> SELECT file_name, upper(to_hex(file_offset)) file_offset
FROM pg_walfile_name_offset('2/81235AE8');
         file_name          | file_offset
----------------------------+-------------
 000000010000000200000081 | 235AE8
```



Имя файла состоит из двух частей. Старшие восемь шестнадцатеричных разрядов показывают номер *ветви времени*, которая используется при восстановлении из резервной копии, а остаток соответствует старшим разрядам LSN (младшие разряды LSN показывает поле `file_offset`).

Журнальные файлы можно посмотреть специальной функцией:

```
=> SELECT *
FROM pg_ls_waldir()
WHERE name = '000000010000000200000081';
           name            |   size   |      modification
---------------------------+----------+------------------------
 000000010000000200000081 | 16777216 | 2026-04-05 20:54:38+03
(1 row)
```

Заглянуть в заголовки созданных журнальных записей можно либо утилитой pg_waldump, либо с помощью расширения `pg_walinspect`: v. 15

```
=> CREATE EXTENSION pg_walinspect;
=> SELECT start_lsn, resource_manager, xid, record_type,
  left(description,44) description, block_ref
FROM pg_get_wal_records_info('2/81235AE8', '2/81235B58') \gx
```

```
-[ RECORD 1 ]----+-----------------------------------------------
start_lsn        | 2/81235AE8
resource_manager | Heap
xid              | 917
record_type      | HOT_UPDATE
description      | old_xmax: 917, old_off: 1, old_infobits: [],
block_ref        | blkref #0: rel 1663/16391/16586 fork main blk 0
-[ RECORD 2 ]----+-----------------------------------------------
start_lsn        | 2/81235B30
resource_manager | Transaction
xid              | 917
record_type      | COMMIT
description      | 2026-04-05 20:54:38.825403+03
block_ref        |
```

Здесь мы видим заголовки двух записей.

с. 115 Первая — операция `HOT_UPDATE`, относящаяся к менеджеру ресурсов `Heap`. В столбце `block_ref` указаны имя файла и номер обновленной табличной страницы:

```
=> SELECT pg_relation_filepath('wal');
 pg_relation_filepath
----------------------
 base/16391/16586
(1 row)
```

Вторая запись — `COMMIT`, относящаяся к менеджеру ресурсов `Transaction`.

10.3. Контрольная точка

Для восстановления согласованности после сбоя необходимо последовательно читать журнал и применять к страницам те записи, которые относятся к пропавшим изменениям. Чтобы их обнаружить, надо сравнить LSN страницы на диске с LSN журнальной записи. Однако остается непонятным, с какой именно записи следует начинать восстановление. Если начать слишком поздно, то к страницам, записанным на диск раньше этого момента, будут применены не все изменения, а это приведет к необратимому повреждению данных. Начать же с самого начала нереально: невозможно хранить такой потенциально огромный объем данных, и невозможно смириться со столь же огромным временем восстановления. Нужна такая постепенно

продвигающаяся вперед *контрольная точка*, с которой можно безопасно начинать восстановление; предшествующие ей журнальные записи, соответственно, можно удалять.

Самый простой вариант получить контрольную точку — периодически приостанавливать работу системы и сбрасывать на диск все грязные страницы буферного и других кешей. Такой способ, конечно, никуда не годится, поскольку в эти моменты система будет замирать на неопределенное, но весьма существенное время.

Поэтому на практике контрольная точка растягивается во времени и фактически превращается в отрезок. Выполнением контрольной точки занимается специальный фоновый процесс checkpointer[1].

Начало контрольной точки. Сначала процесс контрольной точки сбрасывает на диск структуры, которые можно записать одномоментно благодаря их небольшому объему. К ним относятся объекты из кеша SLRU (такие как статусы транзакций clog или вложенные транзакции) и некоторые другие. *с. 309*

Выполнение контрольной точки. Основное время выполнения контрольной точки занимает запись грязных страниц буферного кеша[2].

Сначала в заголовках всех грязных на текущий момент (момент начала контрольной точки) буферов проставляется специальный флаг. Это происходит быстро, поскольку не связано с вводом-выводом.

Затем процесс контрольной точки постепенно проходит по всем буферам и сбрасывает помеченные на диск. Страницы при этом не вытесняются из кеша, а только записываются, поэтому не нужно обращать внимания ни на число обращений к буферу, ни на его закрепление.

Страницы сбрасываются по порядку номеров, чтобы запись на диск была по возможности последовательной, а не случайной. При этом запись в разные табличные пространства (которые могут соответствовать разным физическим устройствам) чередуется для равномерного распределения нагрузки. *с. 26*

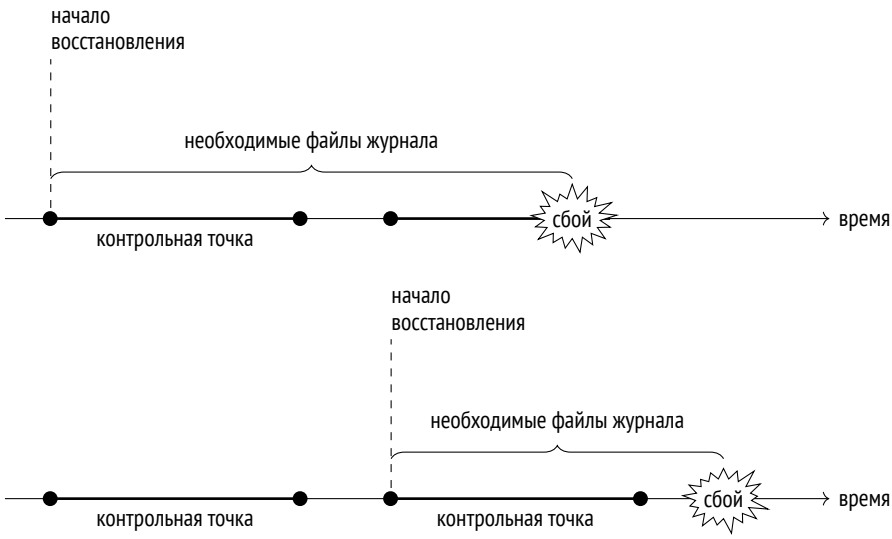
[1] backend/postmaster/checkpointer.c;
backend/access/transam/xlog.c, функция CreateCheckPoint.

[2] backend/storage/buffer/bufmgr.c, функция BufferSync.

Помеченные буферы могут также быть записаны и обслуживающими процессами — смотря кто доберется до буфера первым. В любом случае при записи снимается установленный ранее флаг, так что (для целей контрольной точки) буфер будет записан только один раз.

Естественно, в ходе выполнения контрольной точки страницы продолжают изменяться в буферном кеше. Но новые грязные буферы не помечены флагом, и процесс контрольной точки не будет их записывать.

Завершение контрольной точки. Когда все буферы, которые были грязными *на момент начала* контрольной точки, оказываются записанными, контрольная точка считается *завершенной*. Теперь (но не раньше) момент *начала* используется в качестве той точки, с которой надо начинать восстановление. Все предшествующие журнальные записи больше не нужны.



В конце своей работы процесс создает журнальную запись об окончании контрольной точки, указывая LSN момента ее начала.

Кроме того, в файле PGDATA/global/pg_control обновляется указание на последнюю пройденную контрольную точку. (До завершения процесса pg_control указывает на предыдущую.)

Чтобы окончательно разобраться, что на что указывает, рассмотрим небольшой пример.

Сделаем несколько страниц в буферном кеше грязными:

```
=> UPDATE big SET s = 'FOO';
=> SELECT count(*) FROM pg_buffercache WHERE isdirty;
 count
-------
  4131
(1 row)
```

Запомним текущую позицию в журнале:

```
=> SELECT pg_current_wal_insert_lsn();
 pg_current_wal_insert_lsn
---------------------------
 2/81AC6580
(1 row)
```

Теперь выполним контрольную точку вручную. Все грязные страницы будут сброшены на диск; поскольку в системе ничего не происходит, новых грязных страниц не появится:

```
=> CHECKPOINT;
=> SELECT count(*) FROM pg_buffercache WHERE isdirty;
 count
-------
     0
(1 row)
```

Посмотрим, как контрольная точка отразилась в журнале. Запись CHECKPOINT_REDO отмечает начало контрольной точки, а CHECKPOINT_ONLINE — ее завершение: v. 17

```
=> SELECT start_lsn,
  resource_manager,
  record_type,
  left(description,46)||'...' description
FROM pg_get_wal_records_info('2/81AC6580','FFFFFFFF/FFFFFFFF')
  -- pg_get_wal_records_info_till_end_of_wal до v.16
WHERE record_type LIKE 'CHECKPOINT%'
\gx
```

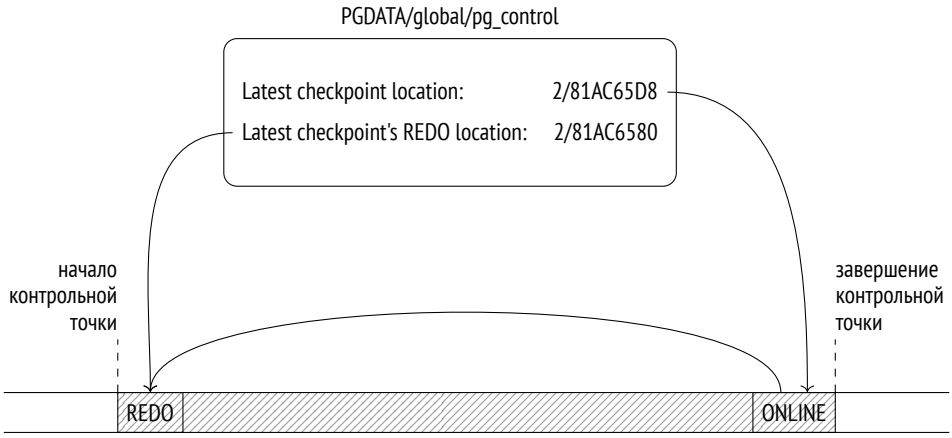
```
-[ RECORD 1 ]----+-------------------------------------------------
start_lsn        | 2/81AC6580
resource_manager | XLOG
record_type      | CHECKPOINT_REDO
description      | wal_level replica...
-[ RECORD 2 ]----+-------------------------------------------------
start_lsn        | 2/81AC65D8
resource_manager | XLOG
record_type      | CHECKPOINT_ONLINE
description      | redo 2/81AC6580; tli 1; prev tli 1; fpw true; ...
```

Между двумя этими позициями в журнале будут располагаться другие записи, порожденные продолжающим свою работу сервером.

> До версии PostgreSQL 10 для журнала предзаписи часто использовался акроним XLOG (transaction log). Следы этого именования остались не только в названии менеджера ресурсов, но и в исходном коде.

Ту же информацию найдем и в управляющем файле:

```
postgres$ /usr/local/pgsql/bin/pg_controldata \
-D /usr/local/pgsql/data | egrep 'Latest.*location'
Latest checkpoint location:           2/81AC65D8
Latest checkpoint's REDO location:    2/81AC6580
```



10.4. Восстановление

При старте сервера первым делом запускается процесс postmaster. В свою очередь он запускает процесс startup[1], задача которого — обеспечить восстановление, если произошел сбой.

Чтобы определить, требуется ли восстановление, startup читает управляющий файл pg_control и проверяет статус кластера. Утилита pg_controldata позволяет заглянуть в содержимое файла:

```
postgres$ /usr/local/pgsql/bin/pg_controldata \
-D /usr/local/pgsql/data | grep state
Database cluster state:               in production
```

Статус аккуратно остановленного сервера — «shut down». Статус «in production» у неработающего сервера свидетельствует о сбое. В этом случае процесс startup автоматически выполняет восстановление с позиции начала последней пройденной контрольной точки из того же файла pg_control.

> Если в каталоге PGDATA присутствует файл backup_label из резервной копии, позиция начала восстановления читается из него.

Процесс последовательно читает журнал, начиная с найденной позиции, и применяет журнальные записи к страницам данных, если LSN страницы меньше LSN записи. Если же LSN страницы оказался больше, то запись применять не нужно, а точнее говоря — нельзя, поскольку записи рассчитаны на строго последовательное применение.

Однако некоторые записи формируются как *полный образ страницы* (full page image, FPI). Полный образ можно применить к любому состоянию страницы, поскольку оно все равно будет стерто. Такие изменения называют *идемпотентными*. Другой пример идемпотентного действия — запись об изменении статуса транзакции: каждой транзакции соответствуют отдельные фиксированные биты в clog, установка которых не зависит от их предыдущего значения. Поэтому внутри страниц clog нет необходимости хранить LSN последнего изменения.

[1] backend/postmaster/startup.c;
backend/access/transam/xlog.c, функция StartupXLOG.

Журнальные записи применяются к страницам в буферном кеше, как это происходит и при обычной работе.

Аналогично журнальные записи применяются и к файлам: например, если запись говорит о том, что файл должен существовать, а его нет, — файл создается.

После окончания восстановления все нежурналируемые отношения перезаписываются с помощью образов в init-файлах.

с. 30

В заключение выполняется контрольная точка, чтобы зафиксировать на диске восстановленное состояние.

На этом работа процесса startup заканчивается.

> В классическом виде процесс восстановления состоит из двух этапов. На первом (roll forward) накатываются журнальные записи, при этом сервер повторяет потерянную при сбое работу. На втором (roll back) откатываются транзакции, которые не были зафиксированы на момент сбоя.
>
> PostgreSQL не нуждается во втором этапе. После восстановления в clog не будет установлен ни бит фиксации, ни бит обрыва (что формально соответствует выполняющейся транзакции), но поскольку точно известно, что транзакция уже не выполняется, она будет считаться оборванной[1].

Можно сымитировать сбой, принудительно остановив сервер в режиме immediate:

```
postgres$ pg_ctl stop -m immediate
```

Проверим состояние кластера:

```
postgres$ /usr/local/pgsql/bin/pg_controldata \
-D /usr/local/pgsql/data | grep 'state'
Database cluster state:               in production
```

При запуске сервера процесс startup понимает, что произошел сбой, и запускает восстановление:

[1] backend/access/heap/heapam_visibility.c, функция HeapTupleSatisfiesMVCC.

```
postgres$ pg_ctl start -l /home/postgres/logfile
postgres$ tail -n 8 /home/postgres/logfile
LOG:  database system was interrupted; last known up at 2026-04-05
20:54:39 MSK
LOG:  database system was not properly shut down; automatic recovery in
progress
LOG:  redo starts at 2/81AC6580
LOG:  invalid record length at 2/81AC6650: expected at least 24, got 0
LOG:  redo done at 2/81AC65D8 system usage: CPU: user: 0.00 s, system:
0.00 s, elapsed: 0.00 s
LOG:  checkpoint starting: end-of-recovery immediate wait
LOG:  checkpoint complete: wrote 0 buffers (0.0%), wrote 3 SLRU
buffers; 0 WAL file(s) added, 0 removed, 0 recycled; write=0.001 s,
sync=0.001 s, total=0.007 s; sync files=2, longest=0.001 s,
average=0.001 s; distance=0 kB, estimate=0 kB; lsn=2/81AC6650, redo
lsn=2/81AC6650
LOG:  database system is ready to accept connections
```

При нормальном останове сервера процесс postmaster отключает всех клиентов и затем выполняет финальную контрольную точку, чтобы сбросить грязные страницы на диск.

Запомним текущую позицию в журнале:

```
=> SELECT pg_current_wal_insert_lsn();
 pg_current_wal_insert_lsn
---------------------------
 2/81AC66C8
(1 row)
```

Теперь аккуратно остановим сервер:

```
postgres$ pg_ctl stop
```

Проверим состояние кластера:

```
postgres$ /usr/local/pgsql/bin/pg_controldata \
-D /usr/local/pgsql/data | grep state
Database cluster state:               shut down
```

Посмотрев журнал, обнаружим в конце запись о финальной контрольной точке (CHECKPOINT_SHUTDOWN):

```
postgres$ /usr/local/pgsql/bin/pg_waldump \
-p /usr/local/pgsql/data/pg_wal -s 2/81AC66C8
rmgr: XLOG        len (rec/tot):    114/   114, tx:          0, lsn:
2/81AC66C8, prev 2/81AC6650, desc: CHECKPOINT_SHUTDOWN redo 2/81AC66C8;
tli 1; prev tli 1; fpw true; wal_level replica; xid 0:920; oid 24778;
multi 1; offset 0; oldest xid 744 in DB 1; oldest multi 1 in DB 1;
oldest/newest commit timestamp xid: 0/0; oldest running xid 0; shutdown
pg_waldump: error: error in WAL record at 2/81AC66C8: invalid record
length at 2/81AC6740: expected at least 24, got 0
```

Последнее сообщение от pg_waldump говорит о том, что утилита дочитала журнал до конца.

Снова запустим экземпляр:

```
postgres$ pg_ctl start -l /home/postgres/logfile
```

10.5. Фоновая запись

Если обслуживающему процессу потребуется вытеснить из буфера грязную страницу, ему придется самостоятельно записать ее на диск. Это нехорошая ситуация, приводящая к ожиданиям, — гораздо лучше, когда запись происходит асинхронно, в фоновом режиме.

Частично работу по сбросу грязных страниц берет на себя процесс контрольной точки, но этого недостаточно.

Поэтому существует также процесс *фоновой записи* bgwriter[1], который применяет тот же алгоритм поиска буферов, что и механизм вытеснения, но с двумя отличиями:

- используется собственная «часовая стрелка», которая может опережать «часовую стрелку» вытеснения, но никогда не отстает от нее;
- при обходе буферов счетчик обращений не уменьшается.

[1] backend/postmaster/bgwriter.c.

Грязная страница сбрасывается на диск, если буфер не закреплен и имеет нулевое число обращений. Таким образом, фоновый процесс записи идет впереди вытеснения и заранее сбрасывает те буферы, которые с большой вероятностью вскоре будут вытеснены.

Это увеличивает шансы на то, что буферы, выбранные при вытеснении, не будут грязными.

10.6. Настройка

Настройка контрольной точки

Продолжительность контрольной точки (точнее говоря, продолжительность этапа записи грязных буферов) определяется значением параметра *checkpoint_completion_target*. Оно показывает, какую часть времени между началами двух соседних контрольных точек будет происходить запись. Увеличивать значение до единицы не рекомендуется, поскольку может получиться так, что следующая контрольная точка должна будет начаться до того, как полностью завершилась предыдущая. Катастрофы не случится, поскольку в любом случае в один момент времени выполняется только одна контрольная точка, но нормальный ритм работы может быть нарушен.

0.9
v. 14

К настройке остальных параметров можно подойти следующим образом. Сначала определимся, какой объем журнальных файлов допустимо сохранять между двумя последовательными контрольными точками. Чем больше объем, тем меньше накладных расходов, но в любом случае это значение будет ограничено доступным свободным местом и допустимым временем восстановления.

Можно посчитать, за какое время при *обычной* нагрузке будет генерироваться этот объем. Для этого надо запомнить начальную позицию вставки в журнал и периодически проверять разность текущей и запомненной позиций.

Полученное время будем считать обычным интервалом между контрольными точками и запишем его в параметр *checkpoint_timeout*. Значение по умолчанию, скорее всего, слишком мало; обычно время увеличивают, например, до получаса.

5min
с. 245

Однако возможно (и даже вероятно), что *иногда* нагрузка будет выше, и за указанное в параметре время будет сгенерирован слишком большой объем журнальных записей. В этом случае контрольная точка должна выполнять-
1GB ся чаще. Для этого параметром *max_wal_size* ограничим объем журнальных файлов, необходимых для восстановления. При превышении этого предела [1] сервер инициирует внеплановую контрольную точку.

Журнальные файлы, необходимые для восстановления, содержат записи за прошедшую завершенную контрольную точку и за текущую, еще не завершенную. Поэтому для оценки общего объема надо умножить известный объем между контрольными точками на $(1 + checkpoint_completion_target)$.

При такой настройке бóльшая часть контрольных точек происходит по расписанию раз в *checkpoint_timeout* единиц времени. Но при повышенной нагрузке контрольная точка вызывается чаще, по достижении журналом объема *max_wal_size*.

Процесс контрольной точки постоянно сопоставляет свой фактический прогресс с ожидаемым [2]:

фактический прогресс определяется долей уже просмотренных страниц буферного кеша;

ожидаемый прогресс по времени определяется долей уже потраченного времени, исходя из того, что контрольная точка должна завершиться за интервал *checkpoint_timeout* × *checkpoint_completion_target*;

ожидаемый прогресс по объему определяется долей уже заполненных журнальных файлов, планируемое количество которых вычисляется исходя из значения *max_wal_size* × *checkpoint_completion_target*.

Если запись грязных страниц опережает график, она приостанавливается; если отстает хотя бы по одному из двух параметров — догоняет без задержек[3]. Учет и времени, и объема позволяет однотипно управлять скоростью выполнения контрольных точек как по расписанию, так и по требованию.

[1] backend/access/transam/xlog.c, функции XLogCheckpointNeeded и CalculateCheckpointSegments.

[2] backend/postmaster/checkpointer.c, функция IsCheckpointOnSchedule.

[3] backend/postmaster/checkpointer.c, функция CheckpointWriteDelay.

После прохождения контрольной точки сервер избавляется от ненужных для восстановления (и других задач) журнальных записей[1]. При этом часть файлов удаляется, а остальные переименовываются и используются заново. Их количество зависит от интенсивности генерации журнальных записей, но в любом случае в PGDATA/pg_wal поддерживается минимальный объем *min_wal_size*[2].

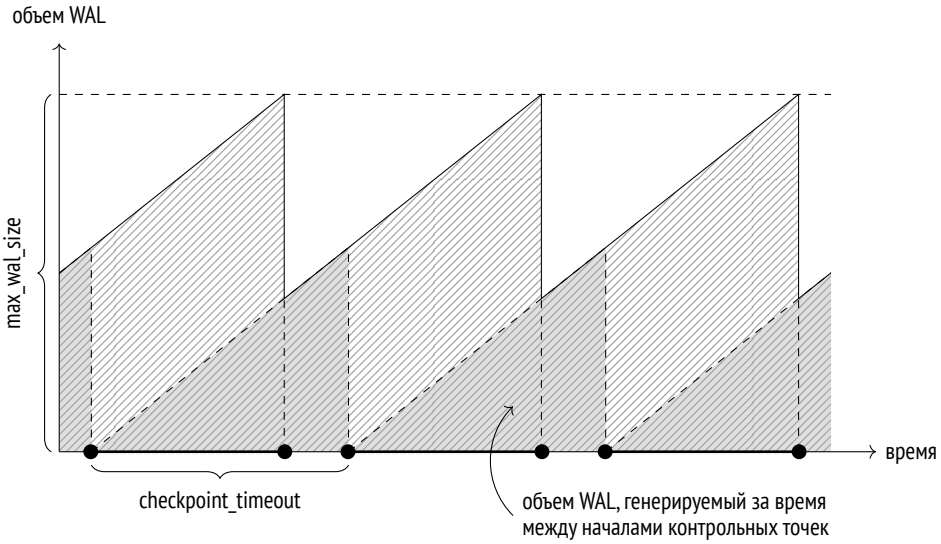
80MB

Обычно переименование позволяет сэкономить на постоянном создании и удалении файлов, но при необходимости такое поведение можно отключить с помощью параметра *wal_recycle*.

v. 12

on

Рисунок показывает, как меняется объем необходимых серверу журнальных файлов в нормальных условиях.



Важно понимать, что реальный объем журнальных файлов на сервере может быть больше, а в некоторых случаях может и превысить значение *max_wal_size*:

- Параметр *max_wal_size* задает ориентир, но не служит жестким ограничением. Если нагрузка возрастет, запись может отстать от графика.

[1] backend/access/transam/xlog.c, функция RemoveOldXlogFiles.
[2] backend/access/transam/xlog.c, функция XLOGfileslop.

- Сервер не имеет права стирать журнальные файлы, еще не переданные через слоты репликации и еще не записанные в архив при непрерывном архивировании. Этот функционал требует постоянного мониторинга, потому что можно легко переполнить дисковую память сервера.

v. 12 0MB
- Некоторое количество файлов может быть зарезервировано администратором с помощью параметра *wal_keep_size*.

Настройка фоновой записи

Процесс фоновой записи имеет смысл настраивать после контрольной точки. Совместно эти процессы должны успевать записывать грязные буферы до того, как они потребуются обслуживающим процессам.

Процесс фоновой записи выполняет работу циклами, засыпая в промежутках на *bgwriter_delay* единиц времени. 200ms

Количество страниц, которые будут записаны за один цикл работы, определяется по среднему количеству буферов, запрошенных обслуживающими процессами с прошлого запуска (используется скользящее среднее, чтобы сгладить неравномерность между запусками, но при этом не зависеть от давней истории). Вычисленное количество дополнительно умножается на коэффициент *bgwriter_lru_multiplier*. Но в любом случае за один цикл работы не будет записано более *bgwriter_lru_maxpages* страниц. 2 100

Если процесс не обнаружил ни одного грязного буфера (то есть в системе ничего не происходит), он «впадает в спячку». После первого же обращения серверного процесса за буфером процесс просыпается и продолжает работать обычным образом.

Мониторинг

Настройку контрольной точки и фоновой записи можно и нужно корректировать, получая обратную связь от мониторинга.

Если контрольные точки, вызванные превышением объема журнальных файлов, выполняются чаще, чем указано в параметре *checkpoint_warning*, в журнал сообщений выводится предупреждение. Значение параметра стоит привести в соответствие с предполагаемой пиковой нагрузкой. 30s

Детальная информация о ходе выполнения контрольных точек выводится в журнал сообщений сервера при включенном параметре *log_checkpoints*. on

Поменяем что-нибудь в данных и инициируем контрольную точку:

```
=> UPDATE big SET s = 'BAR';
=> CHECKPOINT;
```

В журнале сообщений мы увидим, сколько буферов было записано, как изменился состав журнальных файлов после контрольной точки, сколько времени заняла контрольная точка и расстояние (в байтах) между началами соседних контрольных точек:

```
postgres$ tail -n 2 /home/postgres/logfile
LOG:  checkpoint starting: immediate force wait
LOG:  checkpoint complete: wrote 4097 buffers (25.0%), wrote 3 SLRU
buffers; 0 WAL file(s) added, 1 removed, 0 recycled; write=0.012 s,
sync=0.007 s, total=0.025 s; sync files=3, longest=0.006 s,
average=0.003 s; distance=9213 kB, estimate=9213 kB; lsn=2/823C5CB0,
redo lsn=2/823C5C58
```

Но наиболее полезная информация для целей настройки — статистика работы процесса контрольной точки в представлении `pg_stat_checkpointer`. v. 17

> До версии 17 контрольная точка и фоновая запись разделяли одно представление `pg_stat_bgwriter`, поскольку раньше обе задачи выполнялись процессом фоновой записи; в версии 9.2 контрольную точку выделили в отдельный процесс, но представление долгое время оставалось общим.

```
=> SELECT * FROM pg_stat_checkpointer \gx
-[ RECORD 1 ]-------+------------------------------
num_timed           | 0
num_requested       | 26
num_done            | 21
restartpoints_timed | 0
restartpoints_req   | 0
restartpoints_done  | 0
write_time          | 106014
sync_time           | 2660
buffers_written     | 78463
slru_written        | 26
stats_reset         | 2026-04-05 20:51:34.261075+03
```

В числе прочего представление показывает количество выполненных контрольных точек:

- num_timed — по расписанию (по истечении интервала времени *checkpoint_timeout*);
- num_requested — по требованию (в том числе по достижении объема журнала *max_wal_size*).

Большое значение в столбце num_requested (по сравнению с num_timed) говорит о том, что в реальности контрольные точки происходят чаще, чем предполагалось.

Поля, начинающиеся на restartpoints, актуальны только для реплики.

Представление pg_stat_bgwriter показывает статистику фоновой записи:

```
=> SELECT * FROM pg_stat_bgwriter \gx
-[ RECORD 1 ]----+------------------------------
buffers_clean    | 20899
maxwritten_clean | 192
buffers_alloc    | 513174
stats_reset      | 2026-04-05 20:51:34.261075+03
```

Для настройки полезно значение maxwritten_clean: это число показывает, сколько раз процесс фоновой записи прекращал цикл работы из-за превышения *bgwriter_lru_maxpages*.

В хорошо настроенной системе обслуживающие процессы практически не выполняют запись; основной ее объем приходится на процессы контрольной точки (buffers_written) и фоновой записи (buffers_clean).

Кроме подробных сведений о контрольной точке и фоновой записи, важно иметь информацию и о журнале предзаписи в целом. Самую общую статистику — количество журнальных записей и полных образов страниц, а также
v. 14 занимаемый ими объем — можно получить из представления pg_stat_wal:

```
=> SELECT wal_records, wal_fpi, pg_size_pretty(wal_bytes) wal_bytes
FROM pg_stat_wal;
 wal_records | wal_fpi | wal_bytes
-------------+---------+-----------
    44246365 |  608009 | 10126 MB
(1 row)
```

Наиболее детальная информация о вводе-выводе, связанном с журналом (в том числе и разбивка по типам процессов, хотя здесь я ее не показываю), содержится в уже знакомом представлении pg_stat_io: v. 18 *с. 204*

```
=> SELECT context, writes, pg_size_pretty(write_bytes) write_bytes,
  pg_size_pretty(write_bytes/writes) bytes_per_write, fsyncs
FROM (
  SELECT context, sum(writes) writes, sum(write_bytes) write_bytes,
    sum(fsyncs) fsyncs
  FROM pg_stat_io
  WHERE object = 'wal'
  GROUP BY context
);
 context | writes | write_bytes | bytes_per_write | fsyncs
---------+--------+-------------+-----------------+--------
 normal  | 887267 | 10 GB       | 12 kB           |   3880
 init    |     85 | 1360 MB     | 16 MB           |     85
(2 rows)
```

Для журнала поле context может принимать следующие значения:

- init — инициализация сегмента WAL (а также заполнение его нулями при установленном параметре *wal_init_zero* — отсюда и 16 Мбайт данных, записанных каждой операцией); on
- normal — операции с уже существующими сегментами.

Статистику отдельного обслуживающего процесса во время его работы можно получить функцией pg_stat_get_backend_wal. v. 18

11

Режимы журнала

11.1. Производительность

При обычной работе сервера происходит постоянная, но последовательная запись журнальных файлов. Поскольку случайный доступ практически отсутствует, с этой задачей вполне могут справляться и HDD-диски. Но такой характер нагрузки существенно отличается от того, как происходит доступ к файлам данных. Поэтому может быть выгодно разместить журнал на отдельном физическом носителе, создав вместо каталога PGDATA/pg_wal символьную ссылку на каталог примонтированной файловой системы.

v. 17

Есть пара ситуаций, при которых журнальные файлы необходимо не только писать, но и читать. Первая — понятный случай восстановления после сбоя. Вторая возникает при использовании потоковой репликации. Процесс walsender[1] старается читать записи WAL из журнального кеша[2], но если реплика не успевает получать журнальные записи, пока нужные страницы еще находятся в памяти основного сервера, данные будут прочитаны с диска. Однако и в этом случае доступ будет последовательным, а не случайным.

Запись журнала происходит в одном из двух режимов:

- синхронном — при фиксации транзакции продолжение работы невозможно, пока все журнальные записи об этой транзакции не окажутся на диске;
- асинхронном — транзакция завершается немедленно, а журнал записывается в фоновом режиме.

[1] backend/replication/walsender.c.

[2] backend/access/transam/xlog.c, функция WALReadFromBuffers.

Режим устанавливается параметром *synchronous_commit*. on

Синхронный режим. Для надежного сохранения факта фиксации недостаточно просто передать журнальные записи операционной системе — необходимо синхронизировать их с диском. Поскольку синхронизация связана с реальным (то есть медленным) вводом-выводом, выгодно выполнять ее как можно реже.

Для этого обслуживающий процесс, завершающий транзакцию и записывающий журнал, может делать небольшую паузу, определяемую параметром *commit_delay*. Но происходит это только в том случае, если 0s
в системе имеется не менее *commit_siblings* активных транзакций[1]: за 5
время ожидания некоторые из них могут завершиться, и тогда удастся синхронизировать все журнальные записи за один прием. Это похоже на то, как вы придерживаете двери лифта, чтобы кто-то успел заскочить в кабину.

Со значениями по умолчанию пауза не выдерживается. Изменять значение параметра *commit_delay* имеет смысл только в системах, выполняющих большое количество коротких OLTP-транзакций.

После возможной паузы процесс, завершающий транзакцию, сбрасывает на диск все накопившиеся журнальные записи и синхронизирует их с диском (важно, что на диске оказывается запись о фиксации и все предыдущие записи, относящиеся к этой транзакции; все остальное записывается просто потому, что не увеличивает стоимость).

С этого момента транзакция считается надежно завершенной[2] — гарантируется долговечность (Durability из набора требований ACID). Поэтому синхронный режим используется по умолчанию.

Обратная сторона состоит в том, что синхронная запись увеличивает время отклика (команда `COMMIT` не возвращает управление до окончания синхронизации) и уменьшает производительность системы, особенно OLTP.

[1] backend/access/transam/xlog.c, функция XLogFlush.
[2] backend/access/transam/xlog.c, функция RecordTransactionCommit.

Асинхронный режим. Чтобы перейти на асинхронную запись[1], нужно установить для параметра *synchronous_commit* значение «off».

В асинхронном режиме сброс журнальных записей выполняет процесс walwriter[2], чередуя циклы работы с ожиданием. Продолжительность па-
200ms уз устанавливается параметром *wal_writer_delay*.

Проснувшись после очередной паузы, процесс проверяет, не появились ли в кеше новые *полностью* заполненные страницы WAL. Если появились, то процесс записывает их на диск, игнорируя текущую страницу. Если же с прошлого раза ни одна страница журнала не заполнилась до конца, процесс записывает текущую недозаполненную страницу, раз уж все равно проснулся[3].

Этот алгоритм нацелен на то, чтобы по возможности не сбрасывать одну и ту же страницу несколько раз, что важно при большом потоке изменений.

Хотя журнальный кеш и используется как кольцевой буфер, запись останавливается, когда достигает последней страницы кеша; следующий после паузы цикл записи начнется с первой страницы. Поэтому в самом худшем случае walwriter добирается до журнальной записи с третьей попытки: сначала будут сброшены заполненные страницы в конце кеша, затем в начале, и, наконец, дело дойдет до недозаполненной страницы, содержащей искомую запись. В большинстве же случаев хватает одного-двух циклов.

1MB Синхронизация данных происходит через каждые *wal_writer_flush_after* мегабайт и один раз в конце цикла записи.

Асинхронная запись эффективнее синхронной: фиксация изменений не ждет физической записи на диск. Однако надежность уменьшается: в случае сбоя зафиксированные данные могут пропасть, если после фиксации прошло менее 3 × *wal_writer_delay* единиц времени (что при настройке по умолчанию составляет 0,6 секунды).

[1] postgrespro.ru/docs/postgresql/18/wal-async-commit.
[2] backend/postmaster/walwriter.c.
[3] backend/access/transam/xlog.c, функция XLogBackgroundFlush.

В реальности эти режимы дополняют друг друга. Даже при синхронной фиксации журнальные записи долгой транзакции могут записываться асинхронно, чтобы освободить буферы WAL. А если при сбросе страницы из буферного кеша окажется, что соответствующая журнальная запись еще не на диске, она будет сброшена немедленно даже в асинхронном режиме, поскольку иначе невозможно продолжать работу.

В основном же непростой выбор — эффективность или долговечность — остается за проектировщиком системы.

Параметр *synchronous_commit* можно устанавливать в том числе и для отдельных транзакций. Если на уровне приложения получается разделить транзакции на абсолютно критичные (например, работающие с финансовыми данными) и менее важные, то можно увеличить производительность, жертвуя надежностью только части транзакций.

Чтобы получить некоторое представление о том, какой выигрыш дает асинхронная фиксация, сравним время отклика и пропускную способность при двух режимах на эталонном тесте pgbench[1].

Инициализируем необходимые таблицы и запустим 30-секундный тест в синхронном режиме:

```
postgres$ /usr/local/pgsql/bin/pgbench -i internals
postgres$ /usr/local/pgsql/bin/pgbench -T 30 internals
pgbench (18.3)
starting vacuum...end.
transaction type: <builtin: TPC-B (sort of)>
scaling factor: 1
query mode: simple
number of clients: 1
number of threads: 1
maximum number of tries: 1
duration: 30 s
number of transactions actually processed: 42850
number of failed transactions: 0 (0.000%)
latency average = 0.700 ms
initial connection time = 0.515 ms
tps = 1428.341142 (without initial connection time)
```

И затем проведем такой же тест в асинхронном режиме:

[1] postgrespro.ru/docs/postgresql/18/pgbench.

```
=> ALTER SYSTEM SET synchronous_commit = off;
=> SELECT pg_reload_conf();
postgres$ /usr/local/pgsql/bin/pgbench -T 30 internals
pgbench (18.3)
starting vacuum...end.
transaction type: <builtin: TPC-B (sort of)>
scaling factor: 1
query mode: simple
number of clients: 1
number of threads: 1
maximum number of tries: 1
duration: 30 s
number of transactions actually processed: 68607
number of failed transactions: 0 (0.000%)
latency average = 0.437 ms
initial connection time = 0.710 ms
tps = 2286.926300 (without initial connection time)
```

При асинхронной фиксации в этом простом тесте время отклика (latency) существенно уменьшилось, а пропускная способность (tps) увеличилась. Разумеется, в каждой конкретной системе при конкретной нагрузке соотношение будет своим, но видно, что при коротких OLTP-транзакциях эффект может быть весьма значительным.

Восстановим значения параметров по умолчанию:

```
=> ALTER SYSTEM RESET synchronous_commit;
=> SELECT pg_reload_conf();
```

11.2. Надежность

Очевидно, что механизм журналирования должен гарантировать возможность восстановления после сбоя в любых ситуациях (не связанных, конечно, с разрушением носителя данных). На согласованность данных влияет много факторов, из которых наиболее важны кеширование, повреждение данных и неатомарность записи[1].

[1] postgrespro.ru/docs/postgresql/18/wal-reliability.

Кеширование

На пути данных к энергонезависимому хранилищу (такому как пластина жесткого диска) стоят многочисленные кеши.

Системный вызов записи на диск приводит лишь к тому, что операционная система переносит данные в свой кеш (также находящийся в оперативной памяти и подверженный сбоям). Фактическая запись происходит асинхронно, в зависимости от настроек планировщика ввода-вывода операционной системы.

Когда планировщик решает записать данные, они попадают в кеш накопителя (жесткого диска). Электроника накопителя тоже может отложить запись, например собирая данные в группы, которые выгоднее записать одновременно. RAID-контроллер добавляет еще один уровень кеширования между операционной системой и диском.

Если не предпринимать специальных мер, момент надежного сохранения данных остается неизвестным. Обычно это и не важно, поскольку есть журнал, но сами журнальные записи обязаны немедленно сохраняться надежным образом[1]. Это верно и для асинхронного режима, поскольку иначе нельзя гарантировать, что журнальные записи попадут на диск раньше измененных данных.

Контрольная точка также должна сохранять данные надежно, чтобы грязные страницы действительно были сброшены на диск, а не остались в кеше операционной системы. Кроме того, контрольная точка синхронизирует с диском и все файловые операции, выполненные ранее другими процессами (такие как запись страницы или удаление файла): к завершению контрольной точки результаты всех этих действий тоже должны гарантированно оказаться на диске[2].

Есть и некоторые другие ситуации, требующие надежной записи, например выполнение нежурналируемых операций на уровне журнала minimal.

Операционные системы предоставляют разные средства, которые должны гарантировать немедленную запись данных в энергонезависимую память.

[1] backend/access/transam/xlog.c, функция issue_xlog_fsync.
[2] backend/storage/sync/sync.c.

Они сводятся к двум основным: либо после записи выполняется отдельная команда синхронизации (fsync, fdatasync), либо необходимость синхронизации (или даже прямой записи, минуя кеш операционной системы) указывается при открытии файла или записи в него.

Утилита pg_test_fsync помогает определить наиболее подходящий способ синхронизации журнала для конкретной операционной системы и конкретной файловой системы; выбранный способ указывается в параметре *wal_sync_method*. Для других ситуаций способ определяется автоматически[1].

Тонкий момент состоит в том, что при выборе метода надо учитывать характеристики аппаратуры. Например, если используется контроллер с батареей резервного питания, нет резона не использовать его кеш, поскольку батарея позволит сохранить данные в случае сбоя электропитания.

Асинхронный режим фиксации и отсутствие синхронизации — принципи-
on ально разные режимы. Отключение синхронизации (параметр *fsync*) еще больше ускоряет работу системы, но при любом сбое все данные кластера будут безвозвратно потеряны. Асинхронный режим дает гарантию восстановления согласованного состояния данных после сбоя, но, возможно, часть последних транзакций будет в этом состоянии отсутствовать.

Повреждение данных

Оборудование несовершенно, и данные могут повредиться в памяти или на носителе, измениться при передаче по интерфейсным кабелям. Часть таких ошибок обрабатывается на аппаратном уровне, но часть — нет.

Чтобы вовремя обнаружить возникшую проблему, журнальные записи всегда снабжаются контрольными суммами.

Страницы данных также можно защитить контрольными суммами[2]. Это
v. 12 можно сделать либо при инициализации кластера, либо с помощью утилиты pg_checksums[3] при остановленном сервере[4].

[1] backend/storage/file/fd.c, функция pg_fsync.
[2] backend/storage/page/README.
[3] postgrespro.ru/docs/postgresql/18/app-pgchecksums.
[4] commitfest.postgresql.org/51/5323.

В производственной среде контрольные суммы обязательно должны быть включены, несмотря на (незначительные) накладные расходы на их вычисление и проверку. Это увеличивает шансы обнаружить сбой вовремя. Но не дает полной гарантии:

- контрольные суммы проверяются только при обращении к странице, поэтому повреждение может долгое время оставаться незамеченным и попасть во все резервные копии, так что не сохранится ни одного источника верной информации;
- страница, заполненная нулями, считается корректной, поэтому если файловая система по ошибке «обнулит» файл, проверка не обнаружит проблему;
- контрольные суммы защищают только основной слой отношений, а остальные слои и все прочие файлы (например, статусы транзакций clog) ничем не защищены.

Убедимся, что контрольные суммы включены, проверив значение параметра *data_checksums*, доступного только для чтения:

```
=> SHOW data_checksums;
 data_checksums
----------------
 on
(1 row)
```

Остановим сервер и обнулим несколько байтов в нулевой странице основного слоя таблицы:

```
=> SELECT pg_relation_filepath('wal');
 pg_relation_filepath
----------------------
 base/16391/16586
(1 row)
postgres$ pg_ctl stop
postgres$ dd if=/dev/zero of=/usr/local/pgsql/data/base/16391/16586 \
oflag=dsync conv=notrunc bs=1 count=8
8+0 records in
8+0 records out
8 bytes copied, 0.0036238 s, 2.2 kB/s
```

Снова запустим сервер:

```
postgres$ pg_ctl start -l /home/postgres/logfile
```

В принципе, сервер можно было бы и не останавливать. Достаточно, чтобы страница записалась на диск и была вытеснена из кеша (иначе сервер будет продолжать работать с закешированной страницей). Но такой сценарий сложнее воспроизвести.

Теперь пробуем прочитать таблицу:

```
=> SELECT * FROM wal LIMIT 1;
ERROR:  invalid page in block 0 of relation "base/16391/16586"
```

Если данные невозможно восстановить из резервной копии, стоит хотя бы попытаться прочитать поврежденную страницу (естественно, с риском получить искаженную информацию). Для этого надо установить параметр *ignore_checksum_failure*:

off

```
=> SET ignore_checksum_failure = on;
=> SELECT * FROM wal LIMIT 1;
WARNING:  ignoring checksum failure in block 0 of relation
"base/16391/16586"
 id
----
  2
(1 row)
```

В данном случае все прошло успешно, потому что мы испортили некритичную часть заголовка страницы (номер LSN последней журнальной записи), а не сами данные.

Неатомарность записи

Страница базы данных обычно занимает 8 Кбайт, а на низком уровне запись происходит блоками, которые в большинстве случаев имеют меньший размер (как правило, 512 байт или 4 Кбайта). Поэтому при сбое страница данных может записаться частично. При восстановлении бессмысленно применять к такой поврежденной странице обычные журнальные записи.

Для защиты от частичной записи PostgreSQL сохраняет в журнале полный образ страницы (full page image, FPI) при первом ее изменении после начала контрольной точки. *с. 223*

Этим поведением управляет параметр *full_page_writes*. Отказ от записи полных образов может привести к неустранимому повреждению данных, но полные образы могут появляться в журнале даже при отключенном параметре из-за битов-подсказок. Их изменения считаются некритичными и не журналируются сами по себе — первый же запрос, обратившийся к странице, заново установит необходимые биты. Однако изменение любого, даже несущественного бита приводит к изменению контрольной суммы. Поэтому при включенных контрольных суммах (или при установленном параметре *wal_log_hints*) первое изменение битов-подсказок после контрольной точки вызывает запись полного образа страницы[1], невзирая на значение *full_page_writes*.

on

с. 86

off

Если при восстановлении в журнале встречается полный образ страницы, он безусловно (без проверки LSN) записывается на диск, поскольку, как и всякая журнальная запись, он защищен контрольной суммой и не может быть незаметно испорчен. И уже к этому гарантированно корректному образу применяются обычные журнальные записи.

Наличие полных образов существенно увеличивает объем генерируемых журнальных записей, несмотря на то что механизм журналирования исключает из полного образа незанятое место внутри страницы[2]. Ситуацию можно значительно улучшить за счет сжатия полных образов, которое включается параметром *wal_compression*.

off

Проведем простой эксперимент с помощью утилиты pgbench. Выполним контрольную точку и сразу же запустим тест с фиксированным количеством транзакций:

```
=> CHECKPOINT;
=> SELECT pg_current_wal_insert_lsn();
```

[1] backend/storage/buffer/bufmgr.c, функция MarkBufferDirtyHint.
[2] backend/access/transam/xloginsert.c, функция XLogRecordAssemble.

```
 pg_current_wal_insert_lsn
---------------------------
 2/87C95BC8
(1 row)
postgres$ /usr/local/pgsql/bin/pgbench -t 20000 internals
=> SELECT pg_current_wal_insert_lsn();
 pg_current_wal_insert_lsn
---------------------------
 2/8928F088
(1 row)
```

Размер журнальных записей:

```
=> SELECT pg_size_pretty('2/8928F088'::pg_lsn - '2/87C95BC8'::pg_lsn);
 pg_size_pretty
----------------
 22 MB
(1 row)
```

Полные образы в нашем случае составляют значительную часть сгенерированного объема, в чем можно убедиться, посмотрев статистику утилитой pg_waldump с ключом `--stats` или функцией `pg_get_wal_stats` расширения
v. 15 `pg_walinspect`.

Таблица показывает количество журнальных записей (`N`), объем обычных записей (`Record size`) и объем полных образов (`FPI size`) для каждого типа ресурсов (`Type`):

```
postgres$ /usr/local/pgsql/bin/pg_waldump --stats \
-p /usr/local/pgsql/data/pg_wal -s 2/87C95BC8 -e 2/8928F088
WAL statistics between 2/87C95BC8 and 2/8928F088:
Type            N      (%)   Record size      (%)   FPI size      (%)
----            -      ---   -----------      ---   --------      ---
XLOG         1810 (  1.48)         88690 (  1.11)   14572848 ( 98.95)
Transaction 20001 ( 16.38)        680114 (  8.50)          0 (  0.00)
Storage         1 (  0.00)            42 (  0.00)          0 (  0.00)
Standby         3 (  0.00)           222 (  0.00)          0 (  0.00)
Heap2       20233 ( 16.57)       1317150 ( 16.47)      16384 (  0.11)
Heap        80015 ( 65.54)       5911923 ( 73.91)      77844 (  0.53)
Btree          15 (  0.01)           888 (  0.01)      59920 (  0.41)
           ------               --------             --------
Total      122078                7999029 [35.20%]   14726996 [64.80%]
```

Соотношение будет меньше, если страницы успеют измениться между контрольными точками несколько раз. Это еще один повод не выполнять контрольные точки слишком часто.

Теперь посмотрим, как помогает сжатие, на том же примере:

```
=> ALTER SYSTEM SET wal_compression = on; -- или pglz
=> SELECT pg_reload_conf();
=> CHECKPOINT;
=> SELECT pg_current_wal_insert_lsn();
 pg_current_wal_insert_lsn
---------------------------
 2/8928F158
(1 row)
postgres$ /usr/local/pgsql/bin/pgbench -t 20000 internals
=> SELECT pg_current_wal_insert_lsn();
 pg_current_wal_insert_lsn
---------------------------
 2/89D256F8
(1 row)
```

Размер журнальных записей со сжатием:

```
=> SELECT pg_size_pretty('2/89D256F8'::pg_lsn - '2/8928F158'::pg_lsn);
 pg_size_pretty
----------------
 11 MB
(1 row)
postgres$ /usr/local/pgsql/bin/pg_waldump --stats \
-p /usr/local/pgsql/data/pg_wal -s 2/8928F158 -e 2/89D256F8
WAL statistics between 2/8928F158 and 2/89D256F8:
Type            N      (%)   Record size      (%)   FPI size      (%)
----            -      ---   -----------      ---   --------      ---
XLOG         1793 (  1.47)         91399 (  1.14)    2792815 ( 98.79)
Transaction 20001 ( 16.39)        680114 (  8.50)          0 (  0.00)
Storage         1 (  0.00)            42 (  0.00)          0 (  0.00)
CLOG            1 (  0.00)            34 (  0.00)          0 (  0.00)
Standby         4 (  0.00)           276 (  0.00)          0 (  0.00)
Heap2       20227 ( 16.57)       1320696 ( 16.50)        255 (  0.01)
Heap        80005 ( 65.56)       5909689 ( 73.85)      17798 (  0.63)
Btree           5 (  0.00)           275 (  0.00)      16283 (  0.58)
          ------               --------            --------
Total      122037               8002525 [73.89%]    2827151 [26.11%]
```

v. 15 Помимо традиционного алгоритма сжатия PGLZ, доступны два других: LZ4 и Zstandard. Алгоритм LZ4 обычно требует меньше ресурсов процессора, показывая примерно ту же степень сжатия. Алгоритм Zstandard сочетает словарное сжатие с энтропийным кодированием; он сильнее нагружает процессор, но сжимает данные лучше.

```
=> ALTER SYSTEM SET wal_compression = zstd;
=> SELECT pg_reload_conf();
=> CHECKPOINT;
=> SELECT pg_current_wal_insert_lsn();
 pg_current_wal_insert_lsn
---------------------------
 2/89D257C8
(1 row)
postgres$ /usr/local/pgsql/bin/pgbench -t 20000 internals
=> SELECT pg_current_wal_insert_lsn();
 pg_current_wal_insert_lsn
---------------------------
 2/8A6C6378
(1 row)
```

Размер журнальных записей со сжатием Zstandard:

```
=> SELECT pg_size_pretty('2/8A6C6378'::pg_lsn - '2/89D257C8'::pg_lsn);
 pg_size_pretty
----------------
 9859 kB
(1 row)
```

Вывод: при наличии большого числа полных образов страниц (из-за контрольных сумм или *full_page_writes*, то есть почти всегда) имеет смысл воспользоваться сжатием. Несколько алгоритмов позволяют найти компромисс между дополнительной нагрузкой на процессор и сокращением объема журнальных записей.

11.3. Уровни журнала

Основная задача журнала предзаписи — обеспечить возможность восстановления после сбоя. Но журнал можно использовать и для решения других

задач, расширив состав входящих в него записей. Есть несколько уровней журналирования: minimal, replica и logical. Каждый следующий уровень включает в себя все, что попадает в журнал предыдущего уровня, и добавляет некоторую новую информацию.

Используемый уровень задается параметром *wal_level*; его изменение требует перезапуска сервера. replica

Minimal

Начальный уровень minimal гарантирует только восстановление после сбоя. Для экономии места операции с отношениями, созданными или опустошенными в текущей транзакции, не записываются в журнал, если они связаны со вставкой большого объема данных (в частности, это касается таких команд, как `CREATE TABLE AS SELECT` или `CREATE INDEX`)[1]. Вместо журналирования необходимые данные сразу же сбрасываются на диск, а изменения в системном каталоге становятся видимыми при фиксации транзакции.

Если сбой произойдет в процессе выполнения операции, то уже записанные на диск данные останутся невидимыми и не нарушат согласованности. Если же сбой произойдет после того, как операция завершится, то все необходимые данные для применения последующих журнальных записей уже будут находиться на диске.

Объем данных, который требуется записать в только что созданное отношение, чтобы такая оптимизация начала работать, определяется значением параметра *wal_skip_threshold*. v. 13 2MB

Посмотрим, что попадает в журнал на уровне minimal.

По умолчанию используется более высокий уровень replica, рассчитанный на поддержку репликации. При установке минимального уровня потребуется также обнулить значение параметра *max_wal_senders*, который определяет допустимое количество процессов walsender: 10

```
=> ALTER SYSTEM SET wal_level = minimal;
=> ALTER SYSTEM SET max_wal_senders = 0;
```

[1] include/utils/rel.h, макрос RelationNeedsWAL.

Изменение параметров требует перезапуска сервера:

```
postgres$ pg_ctl restart -l /home/postgres/logfile
```

Запомним текущую позицию в журнале:

```
=> SELECT pg_current_wal_insert_lsn();
 pg_current_wal_insert_lsn
---------------------------
 2/8A6C82B0
(1 row)
```

Опустошим таблицу и в той же транзакции вставим новые строки, превысив предел *wal_skip_threshold*:

```
=> BEGIN;
=> TRUNCATE TABLE wal;
=> INSERT INTO wal
     SELECT id FROM generate_series(1,100_000) id;
=> COMMIT;
```

> Я использую команду TRUNCATE, а не создаю отдельную таблицу, поскольку при этом генерируется меньше журнальных записей.

v. 15 С помощью уже знакомого расширения pg_walinspect посмотрим содержимое журнала.

```
=> SELECT start_lsn,
  resource_manager AS rmgr,
  record_type,
  (regexp_match(block_ref, '[0-9]+\/[0-9]+\/[0-9]+'))[1] AS rel
FROM -- pg_get_wal_records_info_till_end_of_wal до v.16
       pg_get_wal_records_info('2/8A6C82B0','FFFFFFFF/FFFFFFFF');
 start_lsn  |    rmgr     | record_type |       rel
------------+-------------+-------------+-----------------
 2/8A6C82B0 | Storage     | CREATE      |
 2/8A6C82E0 | Heap        | UPDATE      | 1663/16391/1259
 2/8A6C8360 | Btree       | INSERT_LEAF | 1663/16391/2662
 2/8A6C83A0 | Btree       | INSERT_LEAF | 1663/16391/2663
 2/8A6C83E0 | Btree       | INSERT_LEAF | 1663/16391/3455
 2/8A6C8420 | Transaction | COMMIT      |
(6 rows)
```

Первая запись (`CREATE`) говорит о создании нового файла для отношения (поскольку при опустошении таблица фактически перезаписывается). *с. 177*

Далее следуют несколько записей, относящихся к системному каталогу. Они отражают изменение таблицы `pg_class` (`UPDATE`) и добавление записей к трем индексам, построенным на этой таблице (`INSERT_LEAF`). В этом можно убедиться, сопоставив путь к файлу, показанный в столбце `rel`, с именем отношения. *с. 30*

Последней идет запись о фиксации транзакции. Вставка данных в таблицу не журналируется.

Replica

Когда система восстанавливается после сбоя, данные, сохраненные на диске, обновляются до согласованного состояния с помощью журнальных записей. То же самое происходит и при восстановлении из резервной копии. Но в этом случае могут использоваться заархивированные журнальные записи, чтобы не просто восстановить согласованность, а довести состояние данных до целевой точки восстановления. Количество таких записей может быть весьма велико (например, они могут охватывать несколько дней), то есть период восстановления будет содержать не одну контрольную точку, а множество. Поэтому минимального уровня журнала недостаточно — нельзя повторить операцию, если она не журналируется. Для возможности восстановления из резервной копии в журнал должны попадать *все* операции.

То же самое верно и для репликации: все, что не журналируется, не будет передано на реплику и не будет на ней воспроизведено.

Но если реплика используется для выполнения запросов, все еще больше усложняется. Во-первых, нужна информация об исключительных блокировках, возникающих на основном сервере, поскольку они могут конфликтовать с запросами на реплике. *с. 261* Во-вторых, нужно уметь строить снимки данных, а для этого необходима информация о выполняющихся транзакциях. *с. 97* В случае реплики нужно учитывать не только локальные транзакции, но и транзакции на основном сервере.

Единственный способ передать эту информацию реплике — периодически записывать ее в журнал[1]. Этим занимается процесс фоновой записи bgwriter[2]: при наличии активности в системе он раз в 15 секунд (интервал не настраивается) создает запись типа RUNNING_XACTS.

Возможность восстановить данные из резервной копии и использовать физическую репликацию гарантируется на уровне журнала replica.

Именно этот уровень используется по умолчанию, поэтому просто сбросим установленные выше параметры и перезагрузим сервер:

```
=> ALTER SYSTEM RESET wal_level;
=> ALTER SYSTEM RESET max_wal_senders;

postgres$ pg_ctl restart -l /home/postgres/logfile
```

Повторим ту же последовательность действий, что и в прошлый раз (но ограничимся вставкой одной строки, чтобы не загромождать вывод):

```
=> SELECT pg_current_wal_insert_lsn();
 pg_current_wal_insert_lsn
---------------------------
 2/8ACE9650
(1 row)

=> BEGIN;
=> TRUNCATE TABLE wal;
=> INSERT INTO wal VALUES (42);
=> COMMIT;
```

Теперь проверим журнальные записи.

```
=> SELECT start_lsn,
  resource_manager AS rmgr,
  record_type,
  (regexp_match(block_ref, '[0-9]+\/[0-9]+\/[0-9]+'))[1] AS rel
FROM pg_get_wal_records_info('2/8ACE9650','FFFFFFFF/FFFFFFFF');
```

[1] backend/storage/ipc/standby, функция LogStandbySnapshot.
[2] backend/postmaster/bgwriter.c.

```
 start_lsn  |    rmgr     |   record_type   |       rel
------------+-------------+-----------------+------------------
 2/8ACE9650 | Heap2       | PRUNE_ON_ACCESS | 1663/16391/1259
 2/8ACE9688 | Standby     | LOCK            |
 2/8ACE96B8 | Storage     | CREATE          |
 2/8ACE96E8 | Heap        | UPDATE          | 1663/16391/1259
 2/8ACE9768 | Btree       | INSERT_LEAF     | 1663/16391/2662
 2/8ACE97A8 | Btree       | INSERT_LEAF     | 1663/16391/2663
 2/8ACE97E8 | Btree       | INSERT_LEAF     | 1663/16391/3455
 2/8ACE9828 | Heap        | INSERT+INIT     | 1663/16391/24814
 2/8ACE9868 | Standby     | LOCK            |
 2/8ACE9898 | Standby     | RUNNING_XACTS   |
 2/8ACE98D0 | Transaction | COMMIT          |
(11 rows)
```

К записям, которые появляются на уровне minimal, добавились:

- записи менеджера ресурсов Standby, связанные с репликацией: LOCK (блокировки) и RUNNING_XACTS (активные транзакции);
- запись об инициализации новой страницы и одновременной вставке в нее строки INSERT+INIT.

Logical

Наконец, максимальный уровень logical обеспечивает возможность логического декодирования и логической репликации. Он должен быть включен на публикующем сервере.

С точки зрения журнальных записей этот уровень практически не отличается от уровня replica — в имеющиеся записи добавляются идентификаторы строк[1]; появляются записи, связанные с источниками репликации и аннулированием кеша, а также произвольные логические записи, которые могут добавлять приложения. В основном же логическое декодирование зависит от информации о выполняющихся транзакциях (RUNNING_XACTS): она позволяет строить снимки данных, чтобы отслеживать изменения системного каталога.

[1] postgrespro.ru/docs/postgresql/18/sql-altertable#SQL-ALTERTABLE-REPLICA-IDENTITY.

Часть III

Блокировки

12

Блокировки отношений

12.1. Общие сведения о блокировках

Блокировки (locks), называемые также *замка́ми*, упорядочивают конкурентный доступ к разделяемым ресурсам.

Под конкурентным доступом понимается одновременное обращение к ресурсу нескольких процессов. Сами процессы могут при этом выполняться как параллельно (если позволяет аппаратура), так и последовательно в режиме разделения времени — это не важно. Если нет конкуренции, то нет и нужды в блокировках (например, общий буферный кеш требует блокировок, а локальный — нет).

Перед тем как обратиться к ресурсу, процесс обязан *захватить* (acquire) блокировку, ассоциированную с этим ресурсом, а по окончании работы — *освободить* (release) ее, чтобы ресурсом могли воспользоваться другие. Когда блокировками управляет СУБД, установленный порядок поддерживается автоматически; если блокировки устанавливает приложение, то обязанность соблюдать правила ложится на него.

На низком уровне блокировка представляется участком разделяемой памяти, в котором некоторым образом отмечается статус блокировки (свободна или захвачена) и, возможно, дополнительная информация (такая как номер процесса и время захвата).

> Можно заметить, что такой участок разделяемой памяти сам по себе является ресурсом. Конкурентный доступ к нему упорядочивается с помощью специальных примитивов синхронизации (таких как семафоры или мьютексы), предоставляемых операционной системой. Они обеспечивают строго последовательное выполнение кода, обращающегося к разделяемому ресурсу. На самом низком уровне эти примитивы

реализуются на основе атомарных инструкций процессора (таких как test-and-set или compare-and-swap).

Защищаемым ресурсом в принципе может быть все что угодно, лишь бы этот ресурс можно было однозначно идентифицировать и сопоставить его с адресом блокировки.

Например, ресурсом может быть объект, с которым работает СУБД, такой как таблица (идентифицируется с помощью `oid` в системном каталоге), страница данных (идентифицируется именем файла и позицией внутри файла), версия строки (идентифицируется страницей и смещением внутри страницы). Ресурсом может быть структура в памяти, такая как хеш-таблица или буфер (идентифицируется заранее присвоенным номером). Иногда даже бывает удобно использовать абстрактные ресурсы, не имеющие никакого физического смысла.

Захват блокировки возможен не всегда: ресурс может оказаться уже занятым кем-то другим. Тогда процесс либо встает в очередь (если механизм блокировки дает такую возможность), либо повторяет попытку захвата блокировки через определенное время. Так или иначе, это приводит к тому, что процесс вынужден простаивать в ожидании освобождения ресурса.

Выделю два фактора, сильно влияющих на эффективность блокировок.

Гранулярность — степень детализации блокировки. Гранулярность важна, если ресурсы образуют иерархию.

Например, таблица состоит из страниц, которые содержат версии строк. Все эти объекты могут выступать в качестве ресурсов. Если установить блокировку на уровне таблицы (крупная гранулярность), процессы не смогут работать одновременно, даже если они обращаются к разным страницам или строкам.

Блокировка отдельных строк (мелкая гранулярность) лишена этого недостатка, но количество блокировок сильно увеличивается. Чтобы информация о них не заняла слишком много памяти, могут применяться разные методы, например *повышение уровня* (эскалация): когда количество низкоуровневых, мелкогранулярных блокировок превышает определенный предел, они заменяются на одну блокировку более высокого уровня.

Набор режимов, в которых могут захватываться блокировки.

Часто применяют всего два режима. *Исключительный* (exclusive), или *монопольный*, режим несовместим ни с каким другим режимом, в том числе с самим собой. В *разделяемом* (shared) режиме блокировка может захватываться несколькими процессами одновременно. Разделяемый режим может использоваться для чтения ресурса, а исключительный — для изменения.

В общем случае режимов может быть больше. Имена режимов не имеют значения, важна лишь матрица их совместимости друг с другом.

Чем мельче гранулярность блокировки и чем больше можно выделить совместимых режимов, тем больше возможностей для распараллеливания.

Блокировки можно классифицировать по времени использования.

Длительные блокировки захватываются на потенциально продолжительное время (обычно до конца транзакции) и чаще всего относятся к таким ресурсам, как отношения и строки. Как правило, системы баз данных управляют такими блокировками автоматически, но пользователь тем не менее может некоторым образом контролировать этот процесс.

Длительные блокировки характеризуются большим количеством режимов, позволяющих выполнять различные одновременные действия над данными. Обычно такие блокировки снабжаются развитой инфраструктурой (очередью ожидания, обнаружением взаимоблокировок, мониторингом), поскольку затраты на поддержание этих средств оказываются несравнимо меньше стоимости операций над защищаемыми данными.

Короткие блокировки захватываются на доли секунды (а зачастую — на время выполнения нескольких инструкций процессора) и обычно относятся к структурам данных в общей памяти. Такими блокировками СУБД управляют полностью автоматически.

Для коротких блокировок характерны минимум режимов и простая инфраструктура, могут отсутствовать даже средства мониторинга.

В PostgreSQL используются самые разные виды блокировок[1]. К длительным относятся *«тяжелые» блокировки* (на уровне отношений и других объектов);

[1] backend/storage/lmgr/README.

с. 268 отдельно выделяются *блокировки на уровне строк*. К коротким блокировкам
с. 305 относятся различные *блокировки в оперативной памяти*. Особняком стоят
с. 298 *предикатные блокировки*, которые, несмотря на название, блокировками не являются.

12.2. Тяжелые блокировки

Тяжелые (heavyweight) блокировки относятся к длительным. Они устанавливаются на уровне *объектов*: в первую очередь отношений, но также и некоторых других. Эти блокировки обычно защищают объекты от одновременного изменения или от использования во время реорганизации, но могут применяться и для других нужд. Такая нечеткая формулировка связана с тем, что блокировки из этой группы используются для самых разных целей, а объединяет их лишь то, как они устроены.

Обычно под словом «блокировка», если нет явных уточнений, понимаются именно такие, тяжелые блокировки.

Тяжелые блокировки располагаются в общей памяти сервера[1] и доступны для изучения в представлении `pg_locks`. Их количество ограничено про-
64 изведением значений двух параметров сервера: *max_locks_per_transaction* и
100 *max_connections*.

Пул блокировок — общий для всех транзакций, то есть одна транзакция может захватить больше блокировок, чем *max_locks_per_transaction*. Важно лишь, чтобы общее число блокировок в системе не превысило установленный предел. Пул создается при запуске, так что изменение любого из двух указанных параметров требует перезагрузки сервера.

Если ресурс уже заблокирован в несовместимом режиме, процесс, пытающийся захватить блокировку, ставится в очередь. Ожидающие процессы не тратят процессорное время: они засыпают и затем пробуждаются операционной системой при освобождении ресурса.

с. 286 Возможна ситуация *взаимоблокировки* (deadlock), или *тупика* двух транзакций, при которой первой из них для продолжения работы требуется ресурс,

[1] backend/storage/lmgr/lock.c.

занятый второй транзакцией, а второй в это время необходим ресурс, занятый первой. Это простой случай; попасть в тупиковую ситуацию могут и более двух транзакций. При взаимоблокировке ожидание длилось бы бесконечно, поэтому PostgreSQL автоматически определяет тупиковые ситуации и аварийно прерывает одну из транзакций, чтобы остальные могли продолжить работу.

Поскольку разные типы тяжелых блокировок служат разным целям, защищают разные ресурсы и имеют разные режимы, каждый тип придется рассмотреть по отдельности.

В приведенном списке названия соответствуют значениям столбца `locktype` представления `pg_locks`:

transactionid и **virtualxid** — блокировка номера транзакции; *с. 259*

relation — блокировка отношения; *с. 261*

tuple — блокировка версии строки; *с. 275*

object — блокировка объекта, который не является отношением; *с. 293*

extend — блокировка файлов отношений при добавлении новых страниц; *с. 295*

page — блокировка страницы (используется некоторыми типами индексов); *с. 296*

advisory — рекомендательная блокировка. *с. 296*

Практически все тяжелые блокировки устанавливаются автоматически, когда в этом возникает необходимость, и автоматически же освобождаются при завершении транзакции. Есть и исключения: например, блокировки отношений можно запросить явно, а управление рекомендательными блокировками полностью находится в руках пользователя.

12.3. Блокировки номеров транзакций

Каждая транзакция всегда удерживает исключительную блокировку своего собственного номера (и виртуального, и — если есть — реального). *с. 91*

Используются два режима блокировки, исключительный и разделяемый. Матрица конфликтов очень проста: разделяемый режим совместим сам с собой; исключительный режим несовместим ни с каким режимом.

	Shared	Exclusive
Shared		×
Exclusive	×	×

Чтобы дождаться окончания какой-либо транзакции, надо запросить блокировку ее номера (в любом режиме). Поскольку сама транзакция уже удерживает исключительную блокировку своего номера, ее не удастся получить. Запрашивающий процесс встанет в очередь и уснет. При завершении транзакции блокировка снимется, и тогда ожидающий процесс будет разбужен. Конечно, получить запрошенную блокировку он не сможет — ресурс уже исчез. Но это и не требуется.

Начнем транзакцию в отдельном сеансе и получим номер обслуживающего процесса:

```
=> BEGIN;
=> SELECT pg_backend_pid();
 pg_backend_pid
----------------
          64412
(1 row)
```

Только что начатая транзакция удерживает исключительную блокировку собственного виртуального номера:

```
=> SELECT locktype, virtualxid, mode, granted
FROM pg_locks WHERE pid = 64412;
  locktype  | virtualxid |     mode      | granted
------------+------------+---------------+---------
 virtualxid | 4/2        | ExclusiveLock | t
(1 row)
```

Здесь `locktype` — это тип блокировки, `virtualxid` — виртуальный номер транзакции (идентифицирующий ресурс), `mode` — режим (в данном случае исключительный). Флаг `granted` показывает, удалось ли получить запрашиваемую блокировку.

Когда транзакция получает реальный номер, его блокировка также добавляется к списку:

```
=> SELECT pg_current_xact_id();
 pg_current_xact_id
--------------------
             191442
(1 row)
```

```
=> SELECT locktype, virtualxid, transactionid AS xid, mode, granted
FROM pg_locks WHERE pid = 64412;
   locktype    | virtualxid |  xid   |     mode      | granted
---------------+------------+--------+---------------+---------
 virtualxid    | 4/2        |        | ExclusiveLock | t
 transactionid |            | 191442 | ExclusiveLock | t
(2 rows)
```

Теперь транзакция удерживает исключительную блокировку обоих своих номеров.

12.4. Блокировки отношений

Для блокировки отношений определено целых восемь различных режимов[1]. Такое разнообразие необходимо для того, чтобы как можно большее количество команд, относящихся к одному отношению (таблице, индексу или другому объекту), могло выполняться одновременно.

Ниже показана матрица конфликтов, дополненная примерами команд, которые требуют соответствующих режимов блокировок. Нет смысла пытаться запоминать эти режимы или искать логику в их названиях. Но внимательно рассмотреть таблицу, сделать общие выводы и обращаться к ней по мере необходимости — безусловно, стоит.

[1] postgrespro.ru/docs/postgresql/18/explicit-locking#LOCKING-TABLES.

	AS	RS	RE	SUE	S	SRE	E	AE	
Access Share								×	SELECT
Row Share							×	×	SELECT FOR UPDATE/SHARE
Row Exclusive					×	×	×	×	INSERT, UPDATE, DELETE
Share Update Exclusive				×	×	×	×	×	VACUUM, CREATE INDEX CONCURRENTLY
Share			×	×		×	×	×	CREATE INDEX
Share Row Exclusive			×	×	×	×	×	×	CREATE TRIGGER
Exclusive		×	×	×	×	×	×	×	REFRESH MAT. VIEW CONCURRENTLY
Access Exclusive	×	×	×	×	×	×	×	×	DROP, TRUNCATE, VACUUM FULL, LOCK TABLE, REFRESH MAT. VIEW

Первый режим, Access Share, самый слабый. Он совместим с любым другим, кроме последнего, Access Exclusive. Поэтому команда `SELECT` не мешает выполнению почти никаких команд, но не даст, например, удалить таблицы, к которым обращается запрос. Режим Access Exclusive, наоборот, самый строгий: он несовместим ни с одним режимом, то есть является исключительным.

Первые четыре режима допускают одновременное изменение данных в таблице, а следующие четыре — нет. Например, команда `CREATE INDEX` использует режим Share. Он совместим сам с собой (то есть можно одновременно создавать несколько индексов для одной таблицы) и с режимами читающих команд. Таким образом, команды `SELECT` будут выполняться, а вот команды `INSERT`, `UPDATE` и `DELETE` будут заблокированы.

И наоборот — незавершенные транзакции, изменяющие данные в таблице, будут блокировать работу команды `CREATE INDEX`. Поэтому и существует вариант `CREATE INDEX CONCURRENTLY`, использующий более слабый режим Share Update Exclusive: такая команда создает индекс дольше обычной (и даже может завершиться ошибкой), зато допускает одновременное изменение данных.

Команда `ALTER TABLE` имеет много вариантов, которые требуют разных режимов блокировки (Share Update Exclusive, Share Row Exclusive, Access Exclusive). Все режимы, разумеется, описаны в документации[1].

[1] postgrespro.ru/docs/postgresql/18/sql-altertable.

Примеры в этой части книги будут использовать уже знакомую нам таблицу счетов:

```
=> TRUNCATE accounts;
=> INSERT INTO accounts(id, client, amount)
VALUES (1, 'alice',   100.00),
       (2, 'bob',     200.00),
       (3, 'charlie', 300.00);
```

Запрос к таблице блокировок понадобится еще не раз, поэтому создадим для него представление, в котором для краткости вывода свернем все идентификаторы в один общий столбец:

```
=> CREATE VIEW locks AS
SELECT pid,
  locktype,
  CASE locktype
    WHEN 'relation' THEN relation::regclass::text
    WHEN 'transactionid' THEN transactionid::text
    WHEN 'virtualxid' THEN virtualxid
  END AS lockid,
  mode,
  granted
FROM pg_locks
ORDER BY 1, 2, 3;
```

Транзакция в первом сеансе (она не завершалась) обновляет строку. Вместе с таблицей блокируются и все индексы этой таблицы, поэтому появляются две новые блокировки с типом `relation` и режимом Row Exclusive:

```
    => UPDATE accounts SET amount = amount + 100.00 WHERE id = 1;
```

```
=> SELECT locktype, lockid, mode, granted
FROM locks WHERE pid = 64412;
   locktype    |    lockid     |       mode       | granted
---------------+---------------+------------------+---------
 relation      | accounts      | RowExclusiveLock | t
 relation      | accounts_pkey | RowExclusiveLock | t
 transactionid | 191442        | ExclusiveLock    | t
 virtualxid    | 4/2           | ExclusiveLock    | t
(4 rows)
```

12.5. Очередь ожидания

Тяжелые блокировки предоставляют «честную» очередь ожидания[1]. Процесс встает в очередь, если пытается захватить блокировку, несовместимую с режимом уже захваченной блокировки или с режимом любого из процессов, уже ожидающих в очереди.

Пока в первом сеансе транзакция выполняет обновление, попробуем в другом сеансе создать индекс по таблице:

```
=> SELECT pg_backend_pid();
 pg_backend_pid
----------------
          64897
(1 row)
=> CREATE INDEX ON accounts(client);
```

Команда «подвисает» в ожидании освобождения ресурса. Транзакция пытается получить блокировку таблицы в режиме Share, но не может:

```
=> SELECT locktype, lockid, mode, granted
FROM locks WHERE pid = 64897;
  locktype  |  lockid  |     mode      | granted
------------+----------+---------------+---------
 relation   | accounts | ShareLock     | f
 virtualxid | 6/3      | ExclusiveLock | t
(2 rows)
```

Пусть теперь в третьем сеансе будет запущена команда `VACUUM FULL`. Она тоже встанет в очередь, поскольку необходимый ей режим Access Exclusive несовместим ни с одним режимом:

```
=> SELECT pg_backend_pid();
 pg_backend_pid
----------------
          65106
(1 row)
=> VACUUM FULL accounts;
```

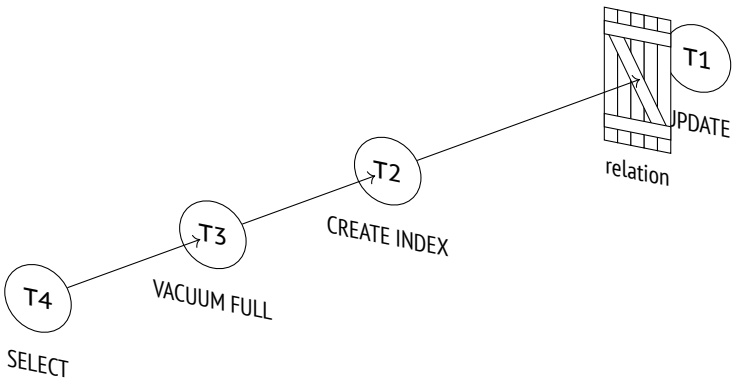
[1] backend/storage/lmgr/lock.c, функция LockAcquire.

```
=> SELECT locktype, lockid, mode, granted
FROM locks WHERE pid = 65106;
   locktype    |  lockid  |        mode         | granted
---------------+----------+---------------------+---------
 relation      | accounts | AccessExclusiveLock | f
 transactionid | 191446   | ExclusiveLock       | t
 virtualxid    | 8/4      | ExclusiveLock       | t
(3 rows)
```

Теперь и все последующие претенденты будут попадать в очередь, уже независимо от режима блокировки. Даже обычные запросы SELECT будут честно вставать за VACUUM FULL, хоть и совместимы с блокировкой Row Exclusive, которую удерживает первый сеанс, выполняющий команду UPDATE.

```
=> SELECT pg_backend_pid();
 pg_backend_pid
----------------
          65322
(1 row)
=> SELECT * FROM accounts;
```

```
=> SELECT locktype, lockid, mode, granted
FROM locks WHERE pid = 65322;
  locktype  |  lockid  |      mode       | granted
------------+----------+-----------------+---------
 relation   | accounts | AccessShareLock | f
 virtualxid | 10/3     | ExclusiveLock   | t
(2 rows)
```



Общую картину ожиданий дает функция pg_blocking_pids. Она показывает номера процессов, которые стоят в очереди перед указанным и либо удерживают, либо запрашивают несовместимую блокировку:

```
=> SELECT pid,
  pg_blocking_pids(pid),
  wait_event_type,
  state,
  left(query,50) AS query
FROM pg_stat_activity
WHERE pid IN (64412,64897,65106,65322) \gx
-[ RECORD 1 ]----+--------------------------------------------------
pid              | 64412
pg_blocking_pids | {}
wait_event_type  | Client
state            | idle in transaction
query            | UPDATE accounts SET amount = amount + 100.00 WHERE
-[ RECORD 2 ]----+--------------------------------------------------
pid              | 64897
pg_blocking_pids | {64412}
wait_event_type  | Lock
state            | active
query            | CREATE INDEX ON accounts(client);
-[ RECORD 3 ]----+--------------------------------------------------
pid              | 65106
pg_blocking_pids | {64412,64897}
wait_event_type  | Lock
state            | active
query            | VACUUM FULL accounts;
-[ RECORD 4 ]----+--------------------------------------------------
pid              | 65322
pg_blocking_pids | {65106}
wait_event_type  | Lock
state            | active
query            | SELECT * FROM accounts;
```

Более детальное представление можно получить, аккуратно анализируя содержимое таблицы pg_locks[1].

Когда транзакция завершается (неважно, фиксацией или обрывом), все ее блокировки снимаются[2]. Процесс, стоящий первым в очереди, получает запрашиваемую блокировку и пробуждается.

[1] wiki.postgresql.org/wiki/Lock_dependency_information.

[2] backend/storage/lmgr/lock.c, функции LockReleaseAll и LockRelease.

Неочевидный момент состоит в том, что при откате к точке сохранения (командой ROLLBACK TO SAVEPOINT или при попадании в секцию EXCEPTION PL/pgSQL-блока) снимаются все блокировки, захваченные транзакцией с момента установки этой точки. Это связано с откатом вложенной транзакции и освобождением ее блокировок. *с. 92*

В нашем примере завершение транзакции в первом сеансе приведет к последовательному выполнению всех запросов, стоящих в очереди:

```
=> ROLLBACK;
ROLLBACK
```

```
 CREATE INDEX
```

```
  VACUUM
```

```
   id | client  | amount
  ----+---------+--------
    1 | alice   | 100.00
    2 | bob     | 200.00
    3 | charlie | 300.00
  (3 rows)
```

13

Блокировки строк

13.1. Устройство

Благодаря изоляции на основе снимков табличные строки не требуется блокировать при чтении. Но нельзя допустить, чтобы две транзакции изменяли одну и ту же строку в один момент времени. Строки нужно блокировать, вот только тяжелые блокировки плохо подходят для этого: каждая из них занимает место в разделяемой памяти сервера (сотни байт, не считая вспомогательной инфраструктуры), а внутренние механизмы не рассчитаны на работу с огромным количеством одновременно существующих блокировок.

В некоторых СУБД эта проблема решается повышением уровня блокировки: если блокировок уровня строк становится слишком много, они заменяются одной более общей блокировкой (например, уровня страницы или всей таблицы). Это упрощает реализацию, но может приводить к сильному снижению эффективности.

В PostgreSQL информация о том, что строка заблокирована, хранится только в заголовке версии строки. Фактически это просто признаки в страницах данных, а не настоящие блокировки — в оперативной памяти они никак не отражаются.

с. 87 Обычно строка блокируется при изменении или удалении. В обоих случаях актуальная версия строки помечается как удаленная. Признаком служит номер транзакции в поле `xmax`, и этот же номер (в сочетании с дополнительными информационными битами) указывает на то, что строка заблокирована. Когда какая-либо транзакция собирается изменить строку, но видит в поле `xmax` актуальной версии номер незавершенной транзакции, она обязана

дождаться ее завершения. После этого все блокировки будут сняты, и ожидающая транзакция сможет продолжить свою операцию над строкой.

Такое решение позволяет блокировать сколько угодно строк, не потребляя никаких ресурсов.

Обратная сторона медали состоит в том, что без информации о блокировке в оперативной памяти другие процессы не могут встать в очередь. Поэтому приходится все-таки использовать и обычные тяжелые блокировки. Дождаться освобождения строки означает дождаться окончания блокирующей транзакции, а для этого нужно запросить блокировку ее номера. Таким образом, число используемых тяжелых блокировок пропорционально числу одновременно работающих процессов, а не количеству изменяемых строк.

13.2. Режимы блокировки строки

Существует четыре режима, в которых можно заблокировать строку[1]. Два режима представляют исключительные блокировки, которые одновременно может удерживать только одна транзакция, и еще два — разделяемые блокировки, которые могут удерживаться несколькими транзакциями.

Матрица конфликтов выглядит следующим образом:

	Key Share	Share	No Key Update	Update
Key Share				×
Share			×	×
No Key Update		×	×	×
Update	×	×	×	×

Исключительные режимы

Режим Update предполагает изменение любых полей строки или ее удаление, а режим No Key Update — изменение только тех полей, которые не входят

[1] postgrespro.ru/docs/postgresql/18/explicit-locking#LOCKING-ROWS.

в уникальные индексы (иными словами, такое изменение, которое не может затронуть внешние ключи).

Команда UPDATE сама выбирает минимальный подходящий режим блокировки; поскольку ключи обычно не меняются, чаще всего строки блокируются в режиме No Key Update.

Создадим функцию, которая с помощью расширения pageinspect покажет интересующую нас информацию о версиях строк, а именно поле xmax и некоторые информационные биты:

```
=> CREATE FUNCTION row_locks(relname text, pageno integer)
RETURNS TABLE(
  ctid tid, xmax text,
  lock_only text, is_multi text,
  keys_upd text, keyshr text, shr text
) AS $$
SELECT (pageno,lp)::text::tid,
  t_xmax,
  CASE WHEN t_infomask  & 128   = 128   THEN 't' END,
  CASE WHEN t_infomask  & 4096  = 4096  THEN 't' END,
  CASE WHEN t_infomask2 & 8192  = 8192  THEN 't' END,
  CASE WHEN t_infomask  & 16    = 16    THEN 't' END,
  CASE WHEN t_infomask  & 16+64 = 16+64 THEN 't' END
FROM heap_page_items(get_raw_page(relname,pageno))
ORDER BY lp;
$$ LANGUAGE sql;
```

Начнем транзакцию и обновим сумму первого счета в таблице accounts (ключ не меняется) и номер второго счета (ключ меняется):

```
=> BEGIN;
=> UPDATE accounts SET amount = amount + 100.00 WHERE id = 1;
=> UPDATE accounts SET id = 20 WHERE id = 2;
```

Заглянем в страницу:

```
=> SELECT * FROM row_locks('accounts',0) LIMIT 2;
 ctid  |  xmax  | lock_only | is_multi | keys_upd | keyshr | shr
-------+--------+-----------+----------+----------+--------+-----
 (0,1) | 191451 |           |          |          |        |
 (0,2) | 191451 |           |          | t        |        |
(2 rows)
```

Режим блокировки определяется информационным битом keys_updated.

```
=> ROLLBACK;
```

То же самое поле xmax используется как признак блокирования командой SELECT FOR, но в этом случае проставляется дополнительный информационный бит xmax_lock_only. Он говорит о том, что версия строки только заблокирована, но не удалена и по-прежнему является актуальной:

```
=> BEGIN;
=> SELECT * FROM accounts WHERE id = 1 FOR NO KEY UPDATE;
=> SELECT * FROM accounts WHERE id = 2 FOR UPDATE;
=> SELECT * FROM row_locks('accounts',0) LIMIT 2;
 ctid  |  xmax  | lock_only | is_multi | keys_upd | keyshr | shr
-------+--------+-----------+----------+----------+--------+-----
 (0,1) | 191452 | t         |          |          |        |
 (0,2) | 191452 | t         |          | t        |        |
(2 rows)
=> ROLLBACK;
```

Разделяемые режимы

Режим Share гарантирует, что прочитанная строка не будет изменена другой транзакцией. Режим Key Share допускает изменение любых полей строки, кроме ключевых.

Из разделяемых режимов само ядро PostgreSQL использует только Key Share при проверке внешних ключей. Он совместим с исключительным режимом No Key Update, то есть проверка внешних ключей не мешает одновременному обновлению любых неключевых полей. Конечно, приложения могут использовать любой подходящий режим.

Подчеркну еще раз, что при чтении не используются никакие блокировки уровня строки.

```
=> BEGIN;
=> SELECT * FROM accounts WHERE id = 1 FOR KEY SHARE;
=> SELECT * FROM accounts WHERE id = 2 FOR SHARE;
```

В версиях строк видим:

```
=> SELECT * FROM row_locks('accounts',0) LIMIT 2;
 ctid  |  xmax  | lock_only | is_multi | keys_upd | keyshr | shr
-------+--------+-----------+----------+----------+--------+-----
 (0,1) | 191453 | t         |          |          | t      |
 (0,2) | 191453 | t         |          |          | t      | t
(2 rows)
```

В обоих случаях установлен бит xmax_keyshr_lock, а режим Share можно распознать, посмотрев на другие информационные биты[1].

13.3. Мультитранзакции

Признак блокировки представлен номером блокирующей транзакции в поле xmax, но разделяемые блокировки могут удерживаться несколькими транзакциями одновременно. Как в одно поле записать сразу несколько номеров?

Для разделяемых блокировок применяются так называемые *мультитранзакции*[2] (MultiXact). Мультитранзакция — это группа транзакций, которой присвоен отдельный номер. Детальная информация об участниках такой группы и режимах их блокировок хранится в файлах в каталоге PGDATA/pg_multixact. Поскольку количество участников мультитранзакций может быть произвольным, работа со страницами в памяти организована
с. 309 с помощью двух SLRU-кешей: один для состава мультитранзакций, а второй для смещений, по которым следует искать информацию в первом кеше. Размер кешей настраивается параметрами *multixact_member_buffers* (32) и *multixact_offset_buffers* (16).

Номер мультитранзакции имеет ту же размерность, что и обычный номер транзакции (32 бита), но номера выделяются независимо, то есть номера транзакций и мультитранзакций могут пересекаться. Чтобы отличить один вид транзакций от другого, используется дополнительный информационный бит xmax_is_multi.

[1] include/access/htup_details.h.
[2] backend/access/transam/multixact.c.

Добавим к имеющимся блокировкам еще одну исключительную, выполненную другой транзакцией (это можно сделать, поскольку режимы Key Share и No Key Update совместимы между собой):

```
=> BEGIN;
=> UPDATE accounts SET amount = amount + 100.00 WHERE id = 1;
```

```
=> SELECT * FROM row_locks('accounts',0) LIMIT 2;
 ctid  |  xmax  | lock_only | is_multi | keys_upd | keyshr | shr
-------+--------+-----------+----------+----------+--------+-----
 (0,1) | 1      |           | t        |          |        |
 (0,2) | 191453 | t         |          |          | t      | t
(2 rows)
```

В первой строке обычный номер заменен на номер мультитранзакции — об этом говорит бит `xmax_is_multi`.

Чтобы не вникать в детали реализации мультитранзакций, можно воспользоваться расширением `pgrowlocks`, которое позволяет увидеть всю информацию о всех типах блокировок строк в удобном виде:

```
=> CREATE EXTENSION pgrowlocks;
=> SELECT * FROM pgrowlocks('accounts') \gx
-[ RECORD 1 ]---------------------------------
locked_row | (0,1)
locker     | 1
multi      | t
xids       | {191453,191454}
modes      | {"For Key Share","No Key Update"}
pids       | {65884,66190}
-[ RECORD 2 ]---------------------------------
locked_row | (0,2)
locker     | 191453
multi      | f
xids       | {191453}
modes      | {"For Share"}
pids       | {65884}
```

Такой запрос внешне похож на обращение к представлению `pg_locks`, но функция `pgrowlocks` читает табличные страницы, так как в оперативной памяти нет информации о блокировках уровня строки.

```
=> COMMIT;
```

```
    => ROLLBACK;
```

Поскольку для мультитранзакций выделяются 32-битные номера, из-за ограничения разрядности счетчика с ними возникает та же проблема пе-
с. 134 реполнения (xid wraparound), что и с обычным номером. Поэтому для номеров мультитранзакций необходимо выполнять аналог заморозки — заменять старые номера на новые (или на обычный номер транзакции, если в момент заморозки блокировка удерживается уже только одной транзакцией)[1].

Но если заморозка обычных номеров транзакций выполняется только для поля `xmin` (так как версии с непустым полем `xmax` неактуальны и будут очищены), то для мультитранзакций, наоборот, замораживается поле `xmax`: актуальная версия строки может постоянно блокироваться все новыми транзакциями в разделяемом режиме.

За заморозку мультитранзакций отвечают параметры сервера, аналогичные тем, что управляют обычной заморозкой: *vacuum_multixact_freeze_min_age*, *vacuum_multixact_freeze_table_age*, *autovacuum_multixact_freeze_max_age*, а так-
v. 14 же *vacuum_multixact_failsafe_age*.

13.4. Очередь ожидания

Исключительные режимы

Из-за того, что блокировка строки — просто признак, очередь организована весьма нетривиально. Когда транзакция собирается изменить строку, она выполняет следующую последовательность действий[2]:

1) если поле `xmax` и информационные биты версии строки указывают на то, что строка заблокирована в несовместимом режиме, захватывает исключительную тяжелую блокировку изменяемой версии строки;

2) при необходимости дожидается освобождения несовместимых блокировок, запрашивая блокировку номера транзакции `xmax` (или нескольких транзакций, если `xmax` — мультитранзакция);

[1] backend/access/heap/heapam.c, функция FreezeMultiXactId.
[2] backend/access/heap/README.tuplock.

3) прописывает в версию строки свой номер (в поле xmax) и устанавливает необходимые информационные биты;

4) освобождает блокировку версии строки, если она захватывалась в п. 1.

Блокировка *версии* строки (tuple lock) является еще одним видом тяжелых блокировок; она имеет тип tuple. Не путайте ее с блокировкой собственно строки (row lock).

Может показаться, что шаги 1 и 4 избыточны и достаточно ограничиться только ожиданием блокирующих транзакций. Но если строку одновременно пытаются обновить несколько транзакций, все они будут ждать завершения транзакции, работающей над строкой в данный момент. При завершении этой транзакции между ожидающими возникнет состояние гонки за право обладания строкой, а это может привести к неопределенно долгому ожиданию для отдельных «невезучих» транзакций. Такая ситуация называется *ресурсным голоданием* (starvation).

Блокировка версии строки выделяет первую в очереди транзакцию и гарантирует, что именно она получит блокировку следующей.

Однако лучше один раз увидеть. Поскольку в процессе работы возникает довольно много разных блокировок, и каждой соответствует отдельная строка в таблице pg_locks, я создам еще одно представление над pg_locks. Оно показывает информацию компактно и оставляет только интересные нам сейчас блокировки (относящиеся к таблице accounts и к самой транзакции, но без блокировки виртуальных номеров):

```
=> CREATE VIEW locks_accounts AS
SELECT pid,
  locktype,
  CASE locktype
    WHEN 'relation' THEN relation::regclass::text
    WHEN 'transactionid' THEN transactionid::text
    WHEN 'tuple' THEN relation::regclass||'('||page||','||tuple||')'
  END AS lockid,
  mode,
  granted
FROM pg_locks
WHERE locktype in ('relation','transactionid','tuple')
  AND (locktype != 'relation' OR relation = 'accounts'::regclass)
ORDER BY 1, 2, 3;
```

Начнем первую транзакцию и обновим строку:

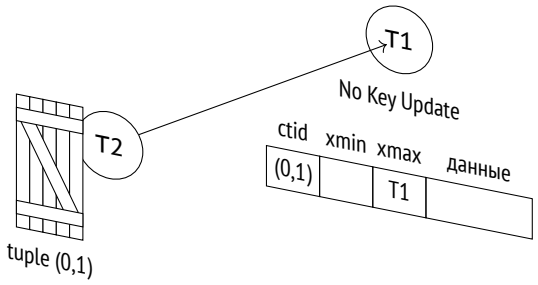
```
=> BEGIN;
=> SELECT txid_current(), pg_backend_pid();
 txid_current | pg_backend_pid
--------------+----------------
       191456 |          66190
(1 row)
=> UPDATE accounts SET amount = amount + 100.00 WHERE id = 1;
```

Транзакция успешно выполняет четыре шага последовательности и теперь удерживает блокировку таблицы:

```
=> SELECT * FROM locks_accounts WHERE pid = 66190;
  pid  |   locktype    |  lockid  |       mode       | granted
-------+---------------+----------+------------------+---------
 66190 | relation      | accounts | RowExclusiveLock | t
 66190 | transactionid | 191456   | ExclusiveLock    | t
(2 rows)
```

Начинаем вторую транзакцию и пытаемся обновить ту же строку. Транзакция подвисает в ожидании блокировки:

```
=> BEGIN;
=> SELECT txid_current(), pg_backend_pid();
 txid_current | pg_backend_pid
--------------+----------------
       191457 |          66267
(1 row)
=> UPDATE accounts SET amount = amount + 100.00 WHERE id = 1;
```



Вторая транзакция дошла только до второго шага. Поэтому, помимо блокировки таблицы и собственного номера, она добавляет в pg_locks еще две

строки: захваченную на первом шаге блокировку типа tuple и запрошенную на втором шаге блокировку номера первой транзакции:

```
=> SELECT * FROM locks_accounts WHERE pid = 66267;
  pid  |   locktype    |    lockid     |       mode       | granted
-------+---------------+---------------+------------------+---------
 66267 | relation      | accounts      | RowExclusiveLock | t
 66267 | transactionid | 191456        | ShareLock        | f
 66267 | transactionid | 191457        | ExclusiveLock    | t
 66267 | tuple         | accounts(0,1) | ExclusiveLock    | t
(4 rows)
```

Третья транзакция дойдет только до первого шага. Она попытается захватить блокировку версии строки и остановится уже на этом:

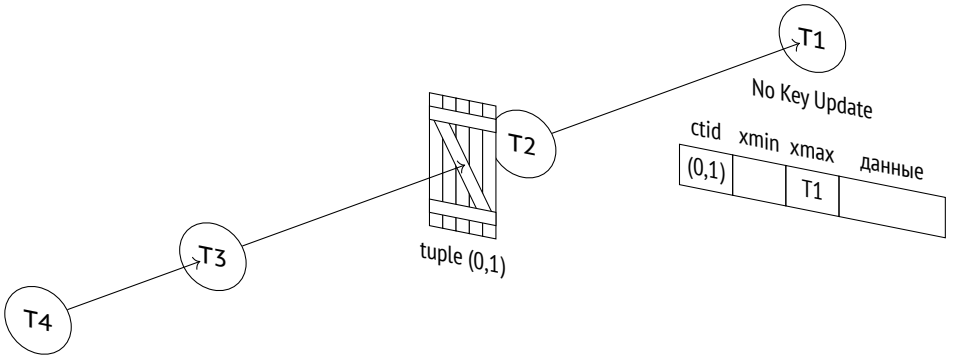
```
=> BEGIN;
=> SELECT txid_current(), pg_backend_pid();
 txid_current | pg_backend_pid
--------------+----------------
       191458 |          66344
(1 row)
=> UPDATE accounts SET amount = amount + 100.00 WHERE id = 1;
```

```
=> SELECT * FROM locks_accounts WHERE pid = 66344;
  pid  |   locktype    |    lockid     |       mode       | granted
-------+---------------+---------------+------------------+---------
 66344 | relation      | accounts      | RowExclusiveLock | t
 66344 | transactionid | 191458        | ExclusiveLock    | t
 66344 | tuple         | accounts(0,1) | ExclusiveLock    | f
(3 rows)
```

Четвертая и последующие транзакции, желающие обновить ту же самую строку, ничем не будут отличаться от третьей — все они будут ожидать одну и ту же блокировку версии строки.

```
=> BEGIN;
=> SELECT txid_current(), pg_backend_pid();
 txid_current | pg_backend_pid
--------------+----------------
       191459 |          66421
(1 row)
=> UPDATE accounts SET amount = amount + 100.00 WHERE id = 1;
```

```
=> SELECT * FROM locks_accounts WHERE pid = 66344;
  pid  |   locktype    |    lockid     |       mode       | granted
-------+---------------+---------------+------------------+---------
 66344 | relation      | accounts      | RowExclusiveLock | t
 66344 | transactionid | 191458        | ExclusiveLock    | t
 66344 | tuple         | accounts(0,1) | ExclusiveLock    | f
(3 rows)
```



Общую картину текущих ожиданий можно посмотреть в представлении pg_stat_activity, добавив информацию о блокирующих процессах:

```
=> SELECT pid,
  wait_event_type,
  wait_event,
  pg_blocking_pids(pid)
FROM pg_stat_activity
WHERE pid IN (66190,66267,66344,66421);
  pid  | wait_event_type |  wait_event   | pg_blocking_pids
-------+-----------------+---------------+------------------
 66190 | Client          | ClientRead    | {}
 66267 | Lock            | transactionid | {66190}
 66344 | Lock            | tuple         | {66267}
 66421 | Lock            | tuple         | {66267,66344}
(4 rows)
```

Если первая транзакция завершится откатом, вся эта конструкция будет работать так, как и следовало бы ожидать: оставшиеся транзакции продвинутся в очереди в том же порядке на один шаг.

Но все-таки более вероятно, что первая транзакция завершится фиксацией. На уровне изоляции Repeatable Read или Serializable это приведет к обрыву

второй транзакции с ошибкой сериализации [1] (а затем и остальных, стоящих в очереди). Но на уровне Read Committed измененная строка перечитывается, и возобновляется попытка ее обновления.

Итак, первая транзакция завершается:

```
=> COMMIT;
```

Вторая транзакция пробуждается и успешно выполняет третий и четвертый шаги последовательности:

```
UPDATE 1
```

```
=> SELECT * FROM locks_accounts WHERE pid = 66267;
  pid  |   locktype    |  lockid  |       mode       | granted
-------+---------------+----------+------------------+---------
 66267 | relation      | accounts | RowExclusiveLock | t
 66267 | transactionid | 191457   | ExclusiveLock    | t
(2 rows)
```

Как только вторая транзакция освобождает блокировку версии строки, просыпается и третья транзакция, но она обнаруживает в поле xmax новой версии строки уже другой номер. На этом последовательность блокирования, приведенная выше, завершается неуспехом. На уровне изоляции Read Committed выполняется повторная попытка заблокировать строку[2], но эта попытка уже не следует приведенной последовательности. Третья транзакция будет ожидать завершения второй без попыток захватить блокировку версии строки:

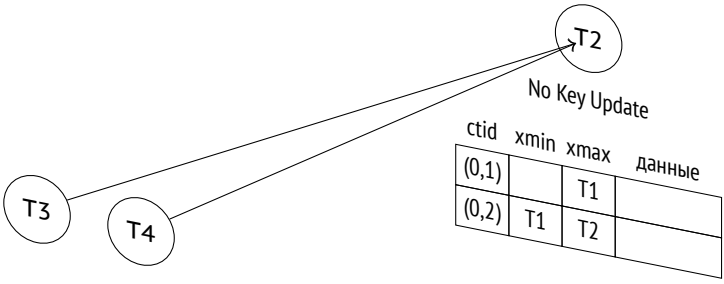
```
=> SELECT * FROM locks_accounts WHERE pid = 66344;
  pid  |   locktype    |  lockid  |       mode       | granted
-------+---------------+----------+------------------+---------
 66344 | relation      | accounts | RowExclusiveLock | t
 66344 | transactionid | 191457   | ShareLock        | f
 66344 | transactionid | 191458   | ExclusiveLock    | t
(3 rows)
```

То же самое происходит и с четвертой транзакцией:

[1] backend/executor/nodeModifyTable.c, функция ExecUpdate.
[2] backend/access/heap/heapam_handler.c, функция heapam_tuple_lock.

```
=> SELECT * FROM locks_accounts WHERE pid = 66421;
  pid  |   locktype    |  lockid  |       mode       | granted
-------+---------------+----------+------------------+---------
 66421 | relation      | accounts | RowExclusiveLock | t
 66421 | transactionid | 191457   | ShareLock        | f
 66421 | transactionid | 191459   | ExclusiveLock    | t
(3 rows)
```

Теперь и третья, и четвертая транзакции ожидают завершения второй с перспективой гонки за захват блокировки. Очередь как таковая распалась.



Если бы, пока очередь еще существовала, в нее успели встать несколько других транзакций, все они оказались бы невольными участниками этой гонки.

Вывод: одновременно обновлять одну и ту же строку таблицы во многих параллельных процессах — не самая удачная идея. Это создает горячую точку, которая при достаточно высокой нагрузке становится узким местом и вызывает проблемы с производительностью.

Завершим все начатые транзакции.

```
=> COMMIT;
```

```
UPDATE 1
=> COMMIT;
```

```
UPDATE 1
=> COMMIT;
```

Разделяемые режимы

Разделяемые блокировки строк применяются PostgreSQL только для проверки ссылочной целостности. Использование их в нагруженном приложении может приводить к ресурсному голоданию, и двухуровневая схема блокирования этому никак не препятствует.

Напомню еще раз последовательность действий, которую выполняет транзакция при блокировке строки:

1) если поле xmax и информационные биты версии строки указывают на то, что строка заблокирована в *несовместимом* режиме, захватывает исключительную тяжелую блокировку изменяемой версии строки;
2) при необходимости дожидается освобождения *несовместимых* блокировок, запрашивая блокировку номера транзакции xmax (или нескольких транзакций, если xmax — мультитранзакция);
3) прописывает в версию строки свой номер (в поле xmax) и устанавливает необходимые информационные биты;
4) освобождает блокировку версии строки, если она захватывалась в п. 1.

Первые два шага говорят о том, что если режимы блокировок *совместимы*, транзакция пройдет *без очереди*.

Повторим наш эксперимент с чистого листа.

```
=> TRUNCATE accounts;
=> INSERT INTO accounts(id, client, amount)
VALUES (1,'alice',100.00),
       (2,'bob',200.00),
       (3,'charlie',300.00);
```

Начинаем первую транзакцию:

```
=> BEGIN;
=> SELECT txid_current(), pg_backend_pid();
 txid_current | pg_backend_pid
--------------+----------------
       191462 |          66190
(1 row)
```

Первая транзакция блокирует строку в разделяемом режиме:

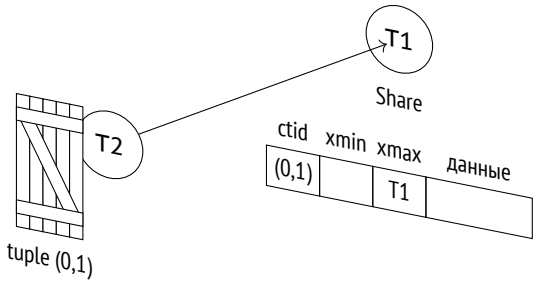
```
=> SELECT * FROM accounts WHERE id = 1 FOR SHARE;
```

Вторая транзакция пытается обновить ту же строку, но не может — режимы Share и No Key Update несовместимы:

```
=> BEGIN;
=> SELECT txid_current(), pg_backend_pid();
 txid_current | pg_backend_pid
--------------+----------------
       191463 |          66267
(1 row)
=> UPDATE accounts SET amount = amount + 100.00 WHERE id = 1;
```

Вторая транзакция ждет завершения первой и удерживает блокировку версии строки — как и в прошлом примере:

```
=> SELECT * FROM locks_accounts WHERE pid = 66267;
  pid  |   locktype    |    lockid     |       mode       | granted
-------+---------------+---------------+------------------+---------
 66267 | relation      | accounts      | RowExclusiveLock | t
 66267 | transactionid | 191462        | ShareLock        | f
 66267 | transactionid | 191463        | ExclusiveLock    | t
 66267 | tuple         | accounts(0,1) | ExclusiveLock    | t
(4 rows)
```

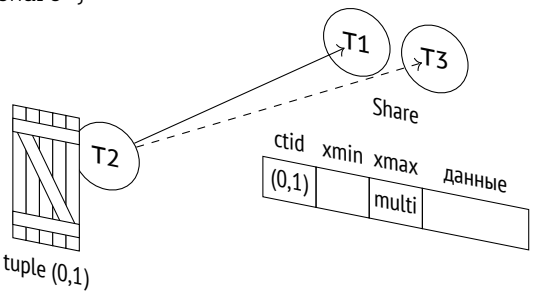


Пусть теперь третья транзакция блокирует строку в разделяемом режиме. Ее блокировка совместима с уже имеющейся блокировкой, и поэтому третья транзакция проходит без очереди:

```
=> BEGIN;
=> SELECT txid_current(), pg_backend_pid();
 txid_current | pg_backend_pid
--------------+----------------
       191464 |          66344
(1 row)
=> SELECT * FROM accounts WHERE id = 1 FOR SHARE;
```

И вот уже две транзакции блокируют строку:

```
=> SELECT * FROM pgrowlocks('accounts') \gx
-[ RECORD 1 ]-------------------------
locked_row | (0,1)
locker     | 2
multi      | t
xids       | {191462,191464}
modes      | {"For Share","For Share"}
pids       | {66190,66344}
```



Теперь если первая транзакция завершится, вторая будет разбужена, но увидит, что блокировка строки никуда не исчезла, и снова встанет в очередь — на этот раз за третьей транзакцией:

```
=> COMMIT;
```

```
=> SELECT * FROM locks_accounts WHERE pid = 66267;
  pid  |   locktype    |    lockid     |       mode       | granted
-------+---------------+---------------+------------------+---------
 66267 | relation      | accounts      | RowExclusiveLock | t
 66267 | transactionid | 191463        | ExclusiveLock    | t
 66267 | transactionid | 191464        | ShareLock        | f
 66267 | tuple         | accounts(0,1) | ExclusiveLock    | t
(4 rows)
```

И только когда третья транзакция завершится (и если за это время не появятся другие разделяемые блокировки), вторая сможет выполнить обновление.

```
|||     => COMMIT;
```

```
||      UPDATE 1
||      => COMMIT;
```

Проверка внешних ключей вряд ли создаст проблемы, поскольку ключевые поля обычно не меняются, а режимы Key Share и No Key Update совместимы. Но от идеи использования разделяемых режимов блокировок строк в приложении в большинстве случаев лучше отказаться.

13.5. Блокировка без ожидания

Обычно команды SQL ожидают освобождения необходимых им ресурсов. Но иногда хочется отказаться от выполнения команды, если блокировку не удалось получить сразу же. Для этого такие команды, как `SELECT`, `LOCK` и `ALTER`, позволяют использовать предложение `NOWAIT`.

Заблокируем строку:

```
=> BEGIN;
=> UPDATE accounts SET amount = amount + 100.00 WHERE id = 1;
```

Команда с предложением `NOWAIT` немедленно завершается ошибкой, если ресурс оказался занят:

```
|       => SELECT * FROM accounts
|       FOR UPDATE NOWAIT;
|       ERROR:  could not obtain lock on row in relation "accounts"
```

В прикладном коде такую ошибку можно перехватить и обработать.

У команд `UPDATE` и `DELETE` предложение `NOWAIT` не предусмотрено. Стандартный прием состоит в том, чтобы сначала попробовать заблокировать строку командой `SELECT FOR UPDATE NOWAIT`, а затем — если получилось — обновить или удалить ее.

В редких случаях бывает удобно пропускать уже заблокированные строки и сразу же получать свободные для обработки. Так работает команда SELECT FOR с предложением SKIP LOCKED:

```
=> SELECT * FROM accounts
ORDER BY id
FOR UPDATE SKIP LOCKED
LIMIT 1;
 id | client | amount
----+--------+--------
  2 | bob    | 200.00
(1 row)
```

В этом примере первая — заблокированная — строка была пропущена, и запрос заблокировал и вернул вторую.

Такой подход позволяет организовать многопоточную обработку очередей или пакетную обработку групп строк. Но не стоит искать для этой команды другие применения — большинство задач решаются более простыми средствами. *с. 179*

Наконец, есть способ не допустить длительного ожидания с помощью установки тайм-аута:

```
=> SET lock_timeout = '1s';
=> ALTER TABLE accounts DROP COLUMN amount;
ERROR:  canceling statement due to lock timeout
```

Команда завершается ошибкой, поскольку не сумела получить блокировку в течение одной секунды. Конечно, тайм-аут можно устанавливать не только на уровне сеанса, но и, например, для отдельной транзакции.

Этот прием позволяет исключить длительную приостановку работы с таблицей, когда команда, требующая исключительную блокировку, выполняется под нагрузкой. При получении ошибки команду можно повторить через некоторое время.

> Параметр *lock_timeout* отличается от параметра *statement_timeout* тем, что ограничивает время ожидания блокировки, а не общее время выполнения оператора.

```
=> ROLLBACK;
```

13.6. Взаимоблокировки

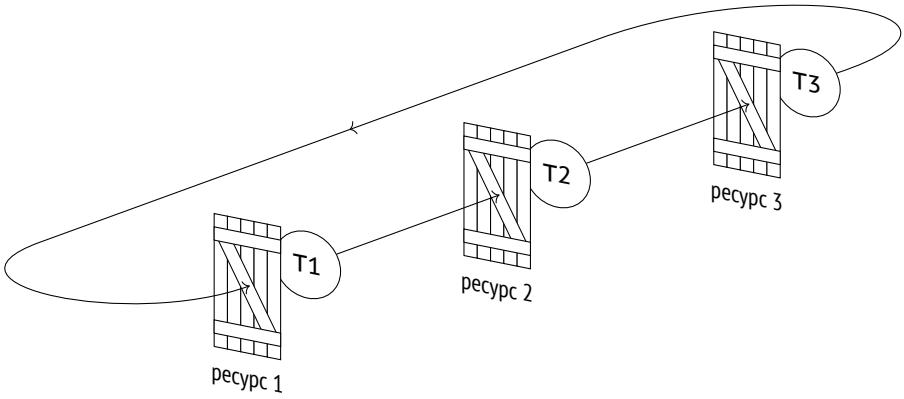
Одной транзакции для продолжения работы может требоваться ресурс, захваченный другой транзакцией; та может нуждаться в ресурсе, захваченном третьей транзакцией, и так далее. Транзакции выстраиваются в очередь, используя механизм тяжелых блокировок.

Но возможна ситуация, когда транзакции, за которой стоит очередь, понадобится еще один ресурс, и ей самой потребуется встать за ним в ту же самую очередь. Возникнет *взаимоблокировка*[1]: очередь замкнется в круг и не сможет уже рассосаться сама собой.

Чтобы наглядно представить себе картину блокировок, построим граф ожиданий. Вершины этого графа — выполняющиеся процессы. Его ребра — стрелки, проведенные от процессов, ожидающих получения блокировок, к процессам, которые удерживают эти блокировки. Если в графе есть *контур*, то есть из какой-либо вершины можно по стрелкам добраться до нее же самой, — значит, произошла взаимоблокировка.

На рисунках я показываю не процессы, а транзакции. Обычно такая замена допустима, поскольку один процесс выполняет одну транзакцию, а блокировки не захватываются вне транзакций. Но в общем случае нужно говорить о процессах, поскольку не все блокировки снимаются сразу после завершения транзакции.

с. 301



[1] postgrespro.ru/docs/postgresql/18/explicit-locking#LOCKING-DEADLOCKS.

Если взаимоблокировка возникла и ни один участник не установил тайм-аут ожидания, транзакции будут ждать друг друга бесконечно. Поэтому менеджер блокировок[1] автоматически отслеживает такие ситуации.

Однако проверка требует определенных усилий, которые не хочется прилагать всякий раз, когда запрашивается новая блокировка (все-таки взаимоблокировки достаточно редки). Поэтому когда процесс пытается захватить блокировку и не может, он просто встает в очередь и засыпает, но при этом автоматически устанавливается таймер на время, указанное в параметре *deadlock_timeout*[2]. Если ресурс освободится раньше — отлично, на проверке удалось сэкономить. Но если по истечении *deadlock_timeout* единиц времени ожидание продолжается, ожидающий процесс будет разбужен и инициирует проверку[3]. 1s

По сути, проверка и состоит в построении графа ожиданий и поиска в нем контуров[4]. На все время проверки останавливается любая работа с тяжелыми блокировками, чтобы «заморозить» текущее состояние графа.

Если проверка не выявила взаимоблокировок, процесс снова засыпает; рано или поздно до него дойдет очередь.

Если же взаимоблокировка обнаружена, одна из транзакций принудительно обрывается, чтобы освободить захваченные ей блокировки и дать возможность остальным транзакциям продолжить работу. В большинстве случаев прерывается та транзакция, которая инициировала проверку, но если в контур входит рабочий процесс автоочистки, не занятый заморозкой в связи с переполнением счетчика транзакций, то прерывается автоочистка как менее приоритетная.

Взаимоблокировки обычно означают, что приложение спроектировано неправильно. Обнаружить такие ситуации можно двумя способами: во-первых, будут появляться сообщения в журнале сервера, а во-вторых, будет увеличиваться значение `deadlocks` в таблице `pg_stat_database`.

[1] backend/storage/lmgr/README.
[2] backend/storage/lmgr/proc.c, функция ProcSleep.
[3] backend/storage/lmgr/proc.c, функция CheckDeadLock.
[4] backend/storage/lmgr/deadlock.c.

Взаимоблокировка при обновлении строк

Хотя в конечном счете к взаимоблокировкам приводят тяжелые блокировки, наиболее частая причина их возникновения — разный порядок блокирования табличных строк.

Пусть первая транзакция собирается перевести 100 ₽ с первого счета на второй. Для этого она сначала уменьшает первый счет:

```
=> BEGIN;
=> UPDATE accounts SET amount = amount - 100.00 WHERE id = 1;
UPDATE 1
```

В это же время вторая транзакция собирается перевести 10 ₽ со второго счета на первый. Она начинает с того, что уменьшает второй счет:

```
=> BEGIN;
=> UPDATE accounts SET amount = amount - 10.00 WHERE id = 2;
UPDATE 1
```

Теперь первая транзакция пытается увеличить второй счет, но обнаруживает, что строка заблокирована:

```
=> UPDATE accounts SET amount = amount + 100.00 WHERE id = 2;
```

Затем вторая транзакция пытается увеличить первый счет, но тоже блокируется:

```
=> UPDATE accounts SET amount = amount + 10.00 WHERE id = 1;
```

Возникает циклическое ожидание, которое никогда не завершится само по себе. Через секунду первая транзакция, не получив доступа к ресурсу, инициирует проверку взаимоблокировки и обрывается сервером:

```
ERROR:  deadlock detected
DETAIL:  Process 65884 waits for ShareLock on transaction 191470;
blocked by process 66190.
Process 66190 waits for ShareLock on transaction 191469; blocked by
process 65884.
HINT:  See server log for query details.
CONTEXT:  while updating tuple (0,2) in relation "accounts"
```

Теперь вторая транзакция может продолжить работу. Она пробуждается и выполняет обновление:

```
    UPDATE 1
```

Завершим транзакции.

```
    => ROLLBACK;
```

```
=> ROLLBACK;
```

Правильный способ выполнения таких операций — блокирование ресурсов в одном и том же порядке. Например, в данном случае можно блокировать счета в порядке возрастания их номеров.

Взаимоблокировка двух команд UPDATE

Бывают случаи, когда взаимоблокировка кажется невозможной, но все-таки возникает.

Удобно и привычно воспринимать команды SQL как атомарные. Но возьмем `UPDATE`: команда блокирует строки по мере их обновления (а не все сразу), и это происходит не одномоментно. Поэтому если одна команда `UPDATE` обновляет несколько строк в одном порядке, а другая — в другом, они могут взаимозаблокироваться.

Для воспроизведения создадим индекс по столбцу `amount`, построенный по убыванию суммы:

```
=> CREATE INDEX ON accounts(amount DESC);
```

Чтобы успеть увидеть происходящее, напишем замедляющую функцию:

```
=> CREATE FUNCTION inc_slow(n numeric)
RETURNS numeric
AS $$
  SELECT pg_sleep(1);
  SELECT n + 100.00;
$$ LANGUAGE sql;
```

Первая команда UPDATE будет обновлять всю таблицу. План выполнения — последовательный просмотр всей таблицы:

```
=> EXPLAIN (costs off)
  UPDATE accounts SET amount = inc_slow(amount);
        QUERY PLAN
--------------------------
 Update on accounts
   -> Seq Scan on accounts
(2 rows)
```

Расположим версии строк на табличной странице в порядке возрастания суммы. Для этого вставим их заново в опустошенную таблицу:

```
=> TRUNCATE accounts;
=> INSERT INTO accounts(id, client, amount)
VALUES (1,'alice',100.00),
       (2,'bob',200.00),
       (3,'charlie',300.00);
=> ANALYZE accounts;
=> SELECT ctid, * FROM accounts;
 ctid  | id | client  | amount
-------+----+---------+--------
 (0,1) |  1 | alice   | 100.00
 (0,2) |  2 | bob     | 200.00
 (0,3) |  3 | charlie | 300.00
(3 rows)
```

При последовательном сканировании строки будут обновляться в том же по-
с. 198 рядке (для больших таблиц это не всегда верно).

Запускаем обновление:

```
=> UPDATE accounts SET amount = inc_slow(amount);
```

А пока оно работает, в другом сеансе запретим использование последовательного сканирования:

```
=> SET enable_seqscan = off;
```

В этом случае для следующего оператора UPDATE планировщик решает использовать сканирование индекса:

```
=> EXPLAIN (costs off)
  UPDATE accounts SET amount = inc_slow(amount)
  WHERE amount > 100.00;
                      QUERY PLAN
------------------------------------------------------
 Update on accounts
   -> Index Scan using accounts_amount_idx on accounts
        Index Cond: (amount > 100.00)
(3 rows)
```

Под условие попадают вторая и третья строки, причем, поскольку индекс построен по убыванию суммы, строки будут обновляться в обратном порядке.

Запускаем следующее обновление:

```
=> UPDATE accounts SET amount = inc_slow(amount)
WHERE amount > 100.00;
```

Заглянув в табличную страницу с помощью расширения pgrowlocks, мы увидим, что первый оператор успел обновить первую строку (0,1), а второй — последнюю (0,3):

```
=> SELECT locked_row, locker, modes FROM pgrowlocks('accounts');
 locked_row | locker |       modes
------------+--------+-------------------
 (0,1)      | 191476 | {"No Key Update"}    ⟵ первый
 (0,3)      | 191477 | {"No Key Update"}    ⟵ второй
(2 rows)
```

Проходит еще секунда. Первый оператор обновил вторую строку, а второй хотел бы это сделать, но не может.

```
=> SELECT locked_row, locker, modes FROM pgrowlocks('accounts');
 locked_row | locker |       modes
------------+--------+-------------------
 (0,1)      | 191476 | {"No Key Update"}
 (0,2)      | 191476 | {"No Key Update"}    ⟵ первый успел раньше
 (0,3)      | 191477 | {"No Key Update"}
(3 rows)
```

Теперь первый оператор хотел бы обновить последнюю строку таблицы, но она уже заблокирована вторым. Произошла взаимоблокировка.

Одна из транзакций прерывается:

```
ERROR:  deadlock detected
DETAIL:  Process 66267 waits for ShareLock on transaction 191476;
blocked by process 66190.
Process 66190 waits for ShareLock on transaction 191477; blocked by
process 66267.
HINT:  See server log for query details.
CONTEXT:  while updating tuple (0,2) in relation "accounts"
```

А другая завершает выполнение:

```
UPDATE 3
```

Несмотря на кажущуюся невероятность, такие ситуации реально возникают в нагруженных системах, обновляющих пакеты строк.

14

Блокировки разных объектов

14.1. Блокировки неотношений

Когда требуется заблокировать ресурс, не являющийся *отношением* в понимании PostgreSQL, используются тяжелые блокировки типа `object`[1]. Таким ресурсом может быть почти все, что можно найти в системном каталоге, например табличные пространства, подписки, схемы, роли, политики, перечислимые типы данных.

Начнем транзакцию и создадим в ней таблицу:

```
=> BEGIN;
=> CREATE TABLE example(n integer);
```

Теперь посмотрим на блокировки неотношений в `pg_locks`:

```
=> SELECT database,
  (
    SELECT datname FROM pg_database WHERE oid = database
  ) AS dbname,
  classid,
  (
    SELECT relname FROM pg_class WHERE oid = classid
  ) AS classname,
  objid,
  mode,
  granted
FROM pg_locks
WHERE locktype = 'object'
  AND pid = pg_backend_pid() \gx
```

[1] backend/storage/lmgr/lmgr.c, функции LockDatabaseObject и LockSharedObject.

```
-[ RECORD 1 ]--------------
database  | 16391
dbname    | internals
classid   | 2615
classname | pg_namespace
objid     | 2200
mode      | AccessShareLock
granted   | t
```

Блокируемый ресурс определяется здесь тремя полями:

database — oid базы данных, к которой относится блокируемый объект (или ноль, если это общий объект кластера);

classid — oid из pg_class, который соответствует имени таблицы системного каталога, определяющей тип ресурса;

objid — oid из таблицы системного каталога, на которую указал classid.

В столбце database указана база данных internals, к которой подключен сеанс. Столбец classid указывает на таблицу pg_namespace, содержащую схемы.

Теперь мы можем расшифровать objid:

```
=> SELECT nspname FROM pg_namespace WHERE oid = 2200;
 nspname
---------
 public
(1 row)
```

Таким образом, заблокирована схема public, чтобы никто не мог удалить ее, пока транзакция еще не завершена.

Соответственно, при удалении объектов захватываются исключительные блокировки как самого объекта, так и всех, от которых он зависит[1].

```
=> ROLLBACK;
```

[1] backend/catalog/dependency.c, функция performDeletion.

14.2. Блокировки расширения отношения

Когда число строк в отношении растет, PostgreSQL по возможности использует для вставки свободное место в имеющихся страницах. Но очевидно, что в какой-то момент приходится добавлять и новые, то есть *расширять отношение*. Физически страницы добавляются в конец соответствующего файла-сегмента (и это может привести к созданию нового сегмента).

Чтобы два процесса не начали добавлять страницы одновременно, эта операция защищена специальной тяжелой блокировкой с типом `extend`[1]. Такая же блокировка используется и при очистке индексов, чтобы другие процессы не могли добавить новые страницы во время сканирования.

Блокировка расширения отношения ведет себя немного иначе, чем рассмотренные ранее:

- она снимается сразу по завершении расширения, не дожидаясь конца транзакции;
- она не может приводить к взаимоблокировкам, поэтому в процедуре обнаружения для нее сделано исключение: она не попадает в граф ожиданий.

  Тем не менее процедура проверки все равно вызывается, если ожидание расширения происходит дольше, чем установлено параметром *deadlock_timeout*. Это ненормальная ситуация, но она может возникнуть при большом потоке вставок из большого числа параллельно работающих процессов. В этом случае проверка может запускаться многократно, фактически парализуя нормальную работу системы.

При интенсивной вставке блокировка может стать узким местом, поэтому файлы таблиц расширяются не на одну страницу, а сразу на несколько (пропорционально числу ожидающих блокировку процессов, но не более чем на 64 страницы за один раз)[2]. Однако к файлам индексов на основе B-дерева по-прежнему добавляется только по одной странице[3]. v. 16

[1] backend/storage/lmgr/lmgr.c, функция LockRelationForExtension.
[2] backend/access/heap/hio.c, функция RelationAddBlocks.
[3] backend/access/nbtree/nbtpage.c, функция _bt_allocbuf.

14.3. Блокировки страниц

Тяжелая блокировка на уровне страницы с типом page [1] применяется в един-
с. 612 ственном случае, связанном с особенностями GIN-индексов.

GIN-индексы позволяют ускорять поиск элементов в составных значениях, например слов в текстовых документах. Такие индексы в первом приближении можно представить как обычное B-дерево, в котором хранятся не сами документы, а отдельные слова этих документов. Поэтому при добавлении нового документа индекс приходится перестраивать довольно сильно, внося в него каждое слово, входящее в документ.

Чтобы улучшить производительность, GIN-индексы предоставляют воз-
можность отложенной вставки, которая включается параметром хранения
on *fastupdate*. Новые слова сначала быстро добавляются в неупорядоченный *список ожидания* (pending list), а спустя какое-то время все накопившееся перемещается в основную индексную структуру. Экономия происходит за счет того, что разные документы с большой вероятностью содержат повторяющиеся слова.

Чтобы исключить перемещение слов из списка ожидания в основной индекс одновременно несколькими процессами, на время переноса метастраница индекса блокируется в исключительном режиме. Это не мешает обычному использованию индекса.

Как и блокировка расширения отношения, блокировка страницы снимается сразу по завершении работы (а не в конце транзакции) и не может приводить к взаимоблокировкам.

14.4. Рекомендательные блокировки

В отличие от других тяжелых блокировок (таких как блокировки отношений), *рекомендательные блокировки* [2] (advisory locks) никогда не устанавливаются автоматически, ими управляет разработчик приложения. Их удобно

[1] backend/storage/lmgr/lmgr.c, функция LockPage.
[2] postgrespro.ru/docs/postgresql/18/explicit-locking#ADVISORY-LOCKS.

использовать, если приложению для каких-то целей требуется нестандартная логика блокирования.

Допустим, имеется ресурс, не соответствующий никакому объекту базы данных (который мы могли бы заблокировать командами типа `SELECT FOR` или `LOCK TABLE`). Для блокировки надо сопоставить ресурс с числовым идентификатором. Если у ресурса есть уникальное имя, то простой вариант — взять хеш-код от имени:

```
=> SELECT hashtext('ресурс1');
 hashtext
-----------
 243773337
(1 row)
```

Имеется целое семейство функций[1] для управления рекомендательными блокировками. Их названия начинаются на `pg_advisory` и содержат дополнительные слова, уточняющие назначение функции:

lock — захватить блокировку;

try — захватить блокировку, если это можно сделать без ожидания;

unlock — освободить блокировку;

share — использовать разделяемый режим (по умолчанию используется исключительный режим);

xact — блокировать до конца транзакции (по умолчанию — до конца сеанса).

Для примера захватим исключительную блокировку до конца сеанса:

```
=> BEGIN;
=> SELECT pg_advisory_lock(hashtext('ресурс1'));
=> SELECT locktype, objid, mode, granted
FROM pg_locks WHERE locktype = 'advisory' AND pid = pg_backend_pid();
 locktype |   objid   |     mode      | granted
----------+-----------+---------------+---------
 advisory | 243773337 | ExclusiveLock | t
(1 row)
```

[1] postgrespro.ru/docs/postgresql/18/functions-admin#FUNCTIONS-ADVISORY-LOCKS.

Чтобы блокирование действительно работало, другие процессы также должны придерживаться установленного порядка получения доступа к ресурсу. За соблюдением этого правила должно следить приложение.

Установленная блокировка не снимается и после завершения транзакции:

```
=> COMMIT;
=> SELECT locktype, objid, mode, granted
FROM pg_locks WHERE locktype = 'advisory' AND pid = pg_backend_pid();
 locktype |   objid   |     mode      | granted
----------+-----------+---------------+---------
 advisory | 243773337 | ExclusiveLock | t
(1 row)
```

После окончания работы с ресурсом освобождаем блокировку явно:

```
=> SELECT pg_advisory_unlock(hashtext('ресурс1'));
```

14.5. Предикатные блокировки

Термин *предикатная блокировка* появился еще при первых попытках реализовать полную изоляцию на основе блокировок[1]. Проблема, с которой тогда столкнулись, заключалась в том, что даже блокировка всех прочитанных и измененных строк не дает полной изоляции: в таблице могут появиться *новые* строки, попадающие под прежние условия отбора, что приводит к возникновению *фантомов*.

с. 51

Поэтому было предложено блокировать не строки, а условия (предикаты). При выполнении запроса с предикатом $a > 10$ блокировка самого предиката не даст добавить в таблицу новые строки, попадающие под это условие, и позволит избежать фантомов. Сложность в том, что при появлении запроса с другим условием, например $a < 20$, требуется определить, пересекаются ли эти условия. В общем случае это алгоритмически неразрешимая задача; на практике ее можно решить только для предикатов, имеющих очень простой вид (как в приведенном примере).

[1] *K. P. Eswaran, J. N. Gray, R. A. Lorie, I. L. Traiger*. The notions of consistency and predicate locks in a database system.

В PostgreSQL уровень изоляции Serializable достигается иначе. Для него используется протокол *сериализуемой* изоляции на основе снимков (Serializable Snapshot Isolation, SSI)[1]. Термин *предикатная блокировка* остался, но смысл его в корне изменился. Фактически такие «блокировки» ничего не блокируют, а используются для отслеживания зависимостей по данным между транзакциями.

Доказано, что изоляция на основе снимков, реализованная на уровне Repeatable Read, допускает *аномалию несогласованной записи* и *аномалию только читающей транзакции*, но никакие другие аномалии невозможны. Двум названным аномалиям соответствуют определенные закономерности в графе зависимостей между транзакциями, которые можно обнаруживать, не затрачивая на это слишком много ресурсов. *с. 67*

Сложность в том, что необходимо различать зависимости двух видов:

- одна транзакция читает строку, которая затем изменяется другой транзакцией (RW-зависимость);
- одна транзакция изменяет строку, которую затем читает другая транзакция (WR-зависимость).

WR-зависимости можно обнаружить, используя уже имеющиеся обычные блокировки, а вот RW-зависимости приходится отслеживать дополнительно с помощью предикатных блокировок. Такое отслеживание включается при выборе уровня изоляции Serializable, и именно поэтому важно, чтобы *все* (или по крайней мере все взаимосвязанные) транзакции использовали этот уровень. Если какая-либо транзакция будет работать на другом уровне, она не будет устанавливать (и проверять) предикатные блокировки, и уровень Serializable выродится до Repeatable Read.

Еще раз повторюсь: несмотря на название, предикатные блокировки ничего не блокируют. Вместо этого при фиксации транзакции выполняется проверка, и если обнаруживается «нехорошая» структура зависимостей, которая может свидетельствовать об аномалии, транзакция обрывается.

[1] backend/storage/lmgr/README-SSI;
backend/storage/lmgr/predicate.c.

Создадим таблицу с данными и индекс на ней, занимающий некоторое количество страниц (для этого указано низкое значение *fillfactor*):

```
=> CREATE TABLE pred(n numeric, s text);
=> INSERT INTO pred(n) SELECT n FROM generate_series(1,10000) n;
=> CREATE INDEX ON pred(n) WITH (fillfactor = 10);
=> ANALYZE pred;
```

Если запрос выполняется с помощью последовательного сканирования таблицы, предикатная блокировка устанавливается на всю таблицу целиком (даже если под условия фильтрации попадают не все строки).

```
=> SELECT pg_backend_pid();
 pg_backend_pid
----------------
          70273
(1 row)
=> BEGIN ISOLATION LEVEL SERIALIZABLE;
=> EXPLAIN (analyze, costs off, timing off, summary off, buffers off)
  SELECT * FROM pred WHERE n > 100;
                   QUERY PLAN
-------------------------------------------------
 Seq Scan on pred (actual rows=9900.00 loops=1)
   Filter: (n > '100'::numeric)
   Rows Removed by Filter: 100
(3 rows)
```

Хотя предикатные блокировки используют собственную инфраструктуру, они отображаются в представлении pg_locks вместе с тяжелыми блокировками. Любые предикатные блокировки всегда захватываются в одном специальном режиме SIRead (Serializable Isolation Read):

```
=> SELECT relation::regclass, locktype, page, tuple
FROM pg_locks WHERE mode = 'SIReadLock' AND pid = 70273
ORDER BY 1, 2, 3, 4;
 relation | locktype | page | tuple
----------+----------+------+-------
 pred     | relation |      |
(1 row)
```

```
    => ROLLBACK;
```

Стоит иметь в виду, что предикатные блокировки не всегда снимаются сразу по завершении транзакции, ведь они нужны, чтобы отслеживать зависимости *между* транзакциями. Но в любом случае управление ими происходит автоматически.

Если же запрос выполняется с помощью индексного сканирования, ситуация меняется в лучшую сторону. В случае B-дерева достаточно установить предикатную блокировку на прочитанные табличные строки и на просмотренные листовые страницы индекса. Тем самым «блокируются» не только конкретные значения, но и весь прочитанный диапазон.

```
    => BEGIN ISOLATION LEVEL SERIALIZABLE;
    => EXPLAIN (analyze, costs off, timing off, summary off, buffers off)
      SELECT * FROM pred WHERE n BETWEEN 1000 AND 1001;
                              QUERY PLAN
    ------------------------------------------------------------------
     Index Scan using pred_n_idx on pred (actual rows=2.00 loops=1)
       Index Cond: ((n >= '1000'::numeric) AND (n <= '1001'::numeric))
       Index Searches: 1
    (3 rows)
```

```
=> SELECT relation::regclass, locktype, page, tuple
FROM pg_locks WHERE mode = 'SIReadLock' AND pid = 70273
ORDER BY 1, 2, 3, 4;
  relation  | locktype | page | tuple
------------+----------+------+-------
 pred       | tuple    |    4 |    96
 pred       | tuple    |    4 |    97
 pred_n_idx | page     |   28 |
(3 rows)
```

Число листовых страниц, покрывающих прочитанный диапазон, может измениться, например, из-за расщепления индексных страниц при вставке новых строк. Но реализация это учитывает и блокирует новые страницы:

```
=> INSERT INTO pred
  SELECT 1000+(n/1000.0) FROM generate_series(1,999) n;
```

```
=> SELECT relation::regclass, locktype, page, tuple
FROM pg_locks WHERE mode = 'SIReadLock' AND pid = 70273
ORDER BY 1, 2, 3, 4;
  relation  | locktype | page | tuple
------------+----------+------+-------
 pred       | tuple    |    4 |    96
 pred       | tuple    |    4 |    97
 pred_n_idx | page     |   28 |
 pred_n_idx | page     |  266 |
 pred_n_idx | page     |  267 |
 pred_n_idx | page     |  268 |
 pred_n_idx | page     |  269 |
(7 rows)
```

На каждую прочитанную версию строки создается отдельная блокировка, но потенциально таких версий может быть очень много. Для предикатных блокировок используется отдельный пул, память под который выделяется при старте сервера. Общее количество предикатных блокировок ограничено произведением значений параметров *max_pred_locks_per_transaction* и *max_connections* (несмотря на названия параметров, учет по отдельным транзакциям не ведется).

64
100

По сути, возникает та же проблема, что и с блокировкой строк, но решается она по-другому: *повышением уровня* блокировок[1].

Значения, при которых повышается уровень, можно задать конфигурационными параметрами. Если количество блокировок версий строк, относящихся к одной странице, превышает *max_pred_locks_per_page*, такие блокировки заменяются на одну блокировку уровня страницы.

2

```
=> EXPLAIN (analyze, costs off, timing off, summary off, buffers off)
  SELECT * FROM pred WHERE n BETWEEN 1000 AND 1002;
                            QUERY PLAN
-------------------------------------------------------------------
 Index Scan using pred_n_idx on pred (actual rows=3.00 loops=1)
   Index Cond: ((n >= '1000'::numeric) AND (n <= '1002'::numeric))
   Index Searches: 1
(3 rows)
```

[1] backend/storage/lmgr/predicate.c, функция PredicateLockAcquire.

Вместо трех блокировок типа tuple появилась одна типа page:

```
=> SELECT relation::regclass, locktype, page, tuple
FROM pg_locks WHERE mode = 'SIReadLock' AND pid = 70273
ORDER BY 1, 2, 3, 4;
  relation  | locktype | page | tuple
------------+----------+------+-------
 pred       | page     |    4 |
 pred_n_idx | page     |   28 |
 pred_n_idx | page     |  266 |
 pred_n_idx | page     |  267 |
 pred_n_idx | page     |  268 |
 pred_n_idx | page     |  269 |
(6 rows)


=> ROLLBACK;
```

По такому же принципу повышается уровень страничных блокировок. Если число блокировок страниц, относящихся к одному отношению, превышает *max_pred_locks_per_relation*, такие блокировки заменяются на одну блокировку уровня отношения. (При отрицательном значении в качестве предела берется значение *max_pred_locks_per_transaction*, деленное на модуль значения этого параметра; таким образом, по умолчанию предел составит 32).

-2

64

Конечно, повышение уровня блокировок неизбежно приводит к тому, что большее число транзакций напрасно завершается ошибкой сериализации. Пропускная способность системы в итоге снижается. Нужно искать баланс между расходом оперативной памяти на хранение блокировок и производительностью.

Предикатные блокировки поддерживают индексы следующих типов:

- B-деревья;
- хеш-индексы, GiST и GIN.

Если используется индексный доступ, а индекс не работает с предикатными блокировками, то блокировка накладывается на весь индекс целиком. Конечно, это тоже увеличивает число лишних обрывов транзакций.

Для более эффективной работы на уровне Serializable следует явно помечать только читающие транзакции как `READ ONLY`. Если менеджер блокировок убедится в невозможности конфликтов читающей транзакции с другими транзакциями[1], он сможет освободить уже установленные для нее предикатные блокировки и не задействовать новые. А если дополнительно объявить такую транзакцию откладываемой (`DEFERRABLE`), это позволит избе-
с. 71 жать аномалии только читающей транзакции.

[1] backend/storage/lmgr/predicate.c, макрос SxactIsROSafe.

15

Блокировки в памяти

15.1. Спин-блокировки

Для защиты структур в разделяемой оперативной памяти используются не обычные тяжелые блокировки, а несколько видов более легких и менее затратных блокировок.

Самые простые из них — *спин-блокировки*, или *спинлоки* (spinlock). Они предназначены для захвата на очень короткое время (несколько инструкций процессора) и защищают отдельные участки памяти от одновременного изменения.

Спин-блокировки реализуются на основе атомарных инструкций процессора, таких как compare-and-swap[1]. Они поддерживают единственный исключительный режим. Если блокировка занята, ожидающий процесс выполняет активное ожидание — команда повторяется («крутится» в цикле, отсюда и название). Если блокировку не удается получить за определенное время, процесс ненадолго приостанавливается и затем снова начинает цикл активного ожидания.

Такая стратегия имеет смысл, если вероятность конфликта оценивается как очень низкая, и поэтому, получив отказ, можно ожидать освобождения блокировки уже через несколько инструкций.

Спин-блокировки не позволяют обнаруживать взаимоблокировки. С прикладной точки зрения остается только верить в то, что разработчики PostgreSQL используют эти блокировки правильно.

[1] backend/storage/lmgr/s_lock.c.

15.2. Легкие блокировки

Следом идут так называемые *легкие блокировки*[1] (lightweight locks, lwlocks). Их захватывают на время, необходимое для работы со структурой данных (например, с хеш-таблицей или списком указателей). Как правило, легкие блокировки удерживаются недолго, но в некоторых случаях они защищают операции ввода-вывода, так что, в принципе, время может оказаться и значительным.

Имеются два режима: исключительный (для изменения данных) и разделяемый (только для чтения). Поддерживается очередь ожидания, но пока блокировка удерживается в разделяемом режиме, другие читающие процессы проходят вне очереди. В системах с высокой степенью параллельности и большой нагрузкой это может приводить к неприятным эффектам[2].

Механизм проверки взаимоблокировок не предусмотрен; корректное использование легких блокировок, как и спин-блокировок, остается на совести разработчиков PostgreSQL.

15.3. Примеры

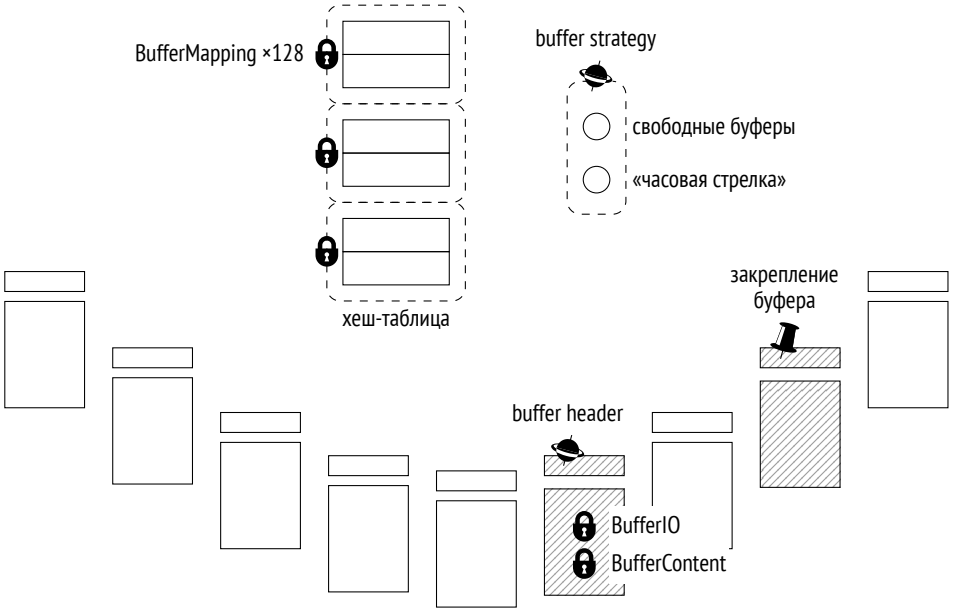
Чтобы получить некоторое представление о том, как и где используются спинлоки и легкие блокировки, рассмотрим три примера структур в разделяемой памяти: буферный кеш, буферы журнала предзаписи и кеш SLRU. Я назову далеко не все блокировки; полная картина слишком сложна и представляет интерес только для разработчиков ядра.

Буферный кеш

с. 183 Чтобы обратиться к хеш-таблице, позволяющей найти буфер в кеше, процесс должен захватить легкую блокировку BufferMapping: в разделяемом режиме — для чтения, в исключительном — для изменений.

[1] backend/storage/lmgr/lwlock.c.

[2] www.postgresql.org/message-id/flat/CAPpHfdvJhO1qutziOp%3Ddy8TO8Xb4L38BxgKG4RPa1up1Lzh_UQ%40mail.gmail.com.



Обращения к хеш-таблице происходят очень активно, и зачастую эта блокировка становится узким местом. Для увеличения гранулярности она фактически устроена как *транш*, состоящий из 128 отдельных легких блокировок, каждая из которых защищает свою часть хеш-таблицы [1].

> Впервые единичную блокировку, защищающую хеш-таблицу, преобразовали в транш размером 16 в версии PostgreSQL 8.2 в 2006 году. Спустя 10 лет в версии 9.5 размер транша увеличили до 128. На современных многоядерных системах порой и этого значения оказывается мало.

Процесс получает доступ к заголовку буфера, захватывая спин-блокировку buffer header [2] (название условное, поскольку спин-блокировки не имеют внешних имен). Отдельные операции, такие как увеличение счетчика обращений, могут выполняться и без явных блокировок с помощью атомарных инструкций процессора.

Чтобы прочитать содержимое буфера, требуется блокировка BufferContent. Она располагается в заголовке данного буфера [3]. Обычно эта блокировка

[1] backend/storage/buffer/bufmgr.c;
include/storage/buf_internals.h, макрос BufMappingPartitionLock.

[2] backend/storage/buffer/bufmgr.c, функция LockBufHdr.

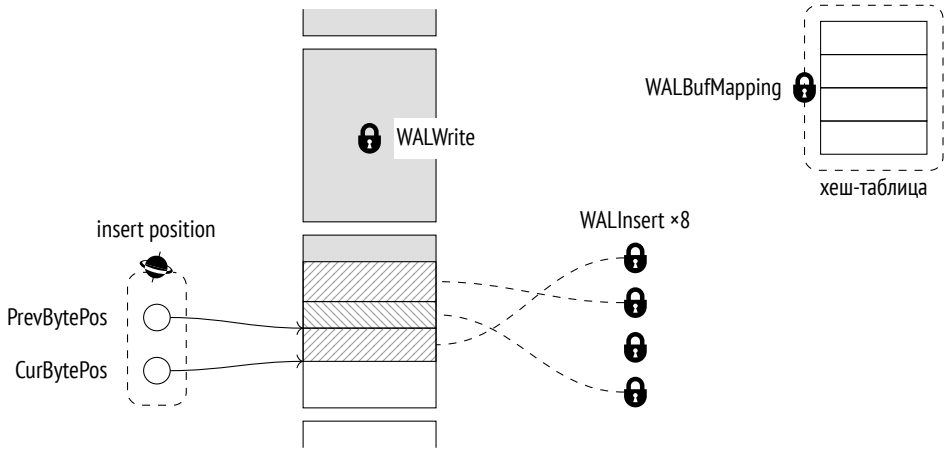
[3] include/storage/buf_internals.h.

захватывается только на время, необходимое для чтения указателей на вер-
с. 189 сии строк, а дальше достаточно защиты, предоставляемой закреплением буфера. Для изменения содержимого буфера блокировка BufferContent должна захватываться в исключительном режиме.

При чтении буфера с диска (или записи на диск) захватывается также блокировка BufferIO в заголовке буфера; фактически это не настоящая блокировка, а признак, играющий ее роль[1]. Блокировка сигнализирует другим процессам, заинтересованным в этой странице, что необходимо дождаться окончания ввода-вывода.

Указатель на свободные буферы и «часовая стрелка» механизма вытеснения также защищены одной общей спин-блокировкой buffer strategy[2].

Буферы журнала предзаписи



Для журнального кеша тоже используется хеш-таблица, содержащая отображение страниц в буферы. В отличие от хеш-таблицы буферного кеша, эта таблица защищена единственной легкой блокировкой WALBufMapping, поскольку размер журнального кеша гораздо меньше (обычно он равен одному сегменту) и обращение к буферам более упорядочено[3].

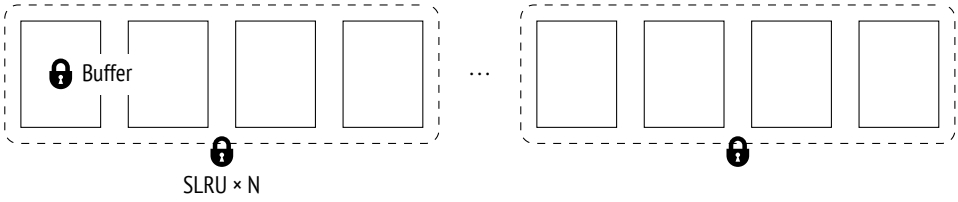
[1] backend/storage/buffer/bufmgr.c, функция StartBufferIO.
[2] backend/storage/buffer/freelist.c.
[3] backend/access/transam/xlog.c, функция AdvanceXLInsertBuffer.

Запись страниц журнала на диск защищена легкой блокировкой WALWrite, чтобы в каждый момент времени только один процесс мог выполнять эту операцию.

Чтобы создать журнальную запись, процесс сначала резервирует место внутри страницы WAL, а затем заполняет его необходимыми данными. Резервирование строго упорядочено; процесс должен захватить спин-блокировку insert position, защищающую указатель вставки[1]. Но заполнять уже зарезервированное место могут одновременно несколько процессов. Для этого процесс должен захватить *любую* из восьми легких блокировок, образующих транш WALInsert[2].

Кеш SLRU

Кеши SLRU (simple least-recently-used)[3] применяются для страниц таких структур данных, как мультитранзакции (multixact), статусы транзакций (clog) или вложенные транзакции (subtrans). В отличие от буферного кеша, кеш SLRU устроен довольно просто и рассчитан на то, что основной поток изменений и чтений приходится на последние страницы: например, завершение транзакции и проверка ее состояния обычно происходят примерно в одно время, после чего к статусу этой транзакции уже никто не обращается. Вытесняются страницы, к которым дольше всего не было обращений. *с. 86*



Кеш поделен на банки по 16 страниц; принадлежность страницы банку определяется четырьмя младшими битами ее номера. Такая схема позволяет обойтись без хеш-таблицы: внутри небольшого банка достаточно линейного

[1] backend/access/transam/xlog.c, функция ReserveXLogInsertLocation.
[2] backend/access/transam/xlog.c, функция WALInsertLockAcquire.
[3] backend/access/transam/slru.c.

поиска. Общий размер кеша, обслуживающего конкретную структуру данных, настраивается индивидуально.

Каждый банк защищен своей легкой блокировкой SLRU (имя транша зависит от структуры данных: например, транш для clog называется XactSLRU). Каждая страница также имеет собственную легкую блокировку Buffer (для clog — XactBuffer). Процессы удерживают блокировку банка для получения или изменения состояния входящих в него буферов. Получить блокировку банка требуется и для захвата блокировки страницы с целью записать ее на диск или прочитать с диска, но на время ввода-вывода блокировка банка освобождается, чтобы процессы могли работать с другими страницами этого банка.

15.4. Мониторинг ожиданий

Безусловно, блокировки необходимы для корректной работы, но они могут приводить и к нежелательным ожиданиям. Такие ожидания полезно отслеживать, чтобы разобраться в причине их возникновения.

Самый простой способ получить представление о длительных блокировках
off в системе — включить параметр *log_lock_waits*. В этом случае в журнал сообщений сервера будут попадать подробные сведения о блокировке каждый
1s раз, когда транзакция ждет дольше, чем *deadlock_timeout*. Эти данные выво-
с. 286 дятся при завершении проверки на взаимоблокировки, отсюда и имя параметра.

Но гораздо более полную и полезную информацию можно найти в представлении `pg_stat_activity`. Когда процесс — системный или обслуживающий — не может выполнять свою работу и ждет чего-либо, это ожидание отображается в столбцах `wait_event_type` (тип ожидания) и `wait_event` (имя конкретного ожидания).

Все ожидания можно разделить на несколько типов[1].

Большую категорию составляют ожидания различных блокировок:

Lock — тяжелых блокировок;

[1] postgrespro.ru/docs/postgresql/18/monitoring-stats#WAIT-EVENT-TABLE.

LWLock — легких блокировок;

BufferPin — закрепленного буфера.

Каждый тип конкретизируется именем ожидания. Например, для легких блокировок это будет имя блокировки или соответствующего транша[1].

Но процессы могут ожидать и других событий:

IO — ввода-вывода, когда требуется записать или прочитать данные;

Client — данных от клиента (в этом состоянии обычно проводит бóльшую часть времени psql);

IPC — данных от другого процесса;

Extension — специфического события, зарегистрированного расширением;

InjectionPoint — выполнения кода в точке внедрения с целью тестирования PostgreSQL (такое событие не появляется при эксплуатации). v. 17

Бывают ситуации, когда процесс просто не выполняет никакой полезной работы. К этой категории относятся ожидания:

Activity — фоновых процессов в своем основном цикле;

Timeout — таймера.

Как правило, такие ожидания «нормальны» и не говорят о каких-либо проблемах. Однако, например, событие SpinDelay типа Timeout свидетельствует об ожидании спин-блокировки. v. 17

Следует учитывать, что представление показывает только те ожидания, которые соответствующим образом обрабатываются в исходном коде[2]. Если имя типа ожидания не определено, процесс не находится ни в одном известном ожидании. Такое время следует считать *неучтенным*, так как это не означает со стопроцентной вероятностью, что процесс ничего не ждет; на самом деле неизвестно, что именно происходит в этот момент.

[1] postgrespro.ru/docs/postgresql/18/monitoring-stats#WAIT-EVENT-LWLOCK-TABLE.
[2] include/utils/wait_event.h.

```
=> SELECT backend_type, wait_event_type AS event_type, wait_event
FROM pg_stat_activity
ORDER BY backend_type;
         backend_type          | event_type |     wait_event
-------------------------------+------------+--------------------
 autovacuum launcher           | Activity   | AutovacuumMain
 background writer             | Activity   | BgwriterMain
 checkpointer                  | Activity   | CheckpointerMain
 client backend                |            |
 io worker                     | Activity   | IoWorkerMain
 io worker                     | Activity   | IoWorkerMain
 io worker                     | Activity   | IoWorkerMain
 logical replication launcher  | Activity   | LogicalLauncherMain
 walwriter                     | Activity   | WalWriterMain
(9 rows)
```

В данном случае во время обращения к представлению все фоновые служебные процессы «бездельничали», а обслуживающий процесс (client backend) был занят выполнением запроса и ничего не ждал.

15.5. Семплирование

К сожалению, представление pg_stat_activity показывает информацию об ожиданиях только *на текущий момент*; статистика не накапливается. Единственный способ получить картину ожиданий во времени — *семплирование* состояния представления с определенным интервалом.

Необходимо учитывать вероятностный характер семплирования. Чем меньше длится ожидание по отношению к периоду семплирования, тем меньше вероятность того, что это ожидание будет зафиксировано. Поэтому чем больше период, тем большее количество измерений требуется, чтобы получить достоверную картину. А повышение частоты семплирования приводит к увеличению накладных расходов. По этой же причине семплирование практически бесполезно для анализа короткоживущих сеансов.

Встроенных средств для семплирования не предусмотрено, но в качестве примера мы воспользуемся расширением pg_wait_sampling[1]. Необходимо

[1] github.com/postgrespro/pg_wait_sampling.

прописать библиотеку в параметр *shared_preload_libraries* и перезапустить сервер.

```
=> ALTER SYSTEM SET shared_preload_libraries = 'pg_wait_sampling';
postgres$ pg_ctl restart -l /home/postgres/logfile
```

Теперь установим расширение в базе данных:

```
=> CREATE EXTENSION pg_wait_sampling;
```

Расширение позволяет просмотреть последние ожидания, которые сохраняются в кольцевой буфер и постепенно перезаписываются новыми данными. Но гораздо интереснее получить профиль ожиданий — накопленную статистику за все время работы.

Например, посмотрим на ожидания, возникающие при работе эталонного тестирования. Запустим утилиту pgbench и, пока она работает, определим номер процесса:

```
postgres$ /usr/local/pgsql/bin/pgbench -T 60 internals
=> SELECT pid FROM pg_stat_activity
WHERE application_name = 'pgbench';
  pid
-------
 71963
(1 row)
```

Профиль ожиданий после окончания работы:

```
=> SELECT pid, event_type, event, count
FROM pg_wait_sampling_profile WHERE pid = 71963
ORDER BY count DESC LIMIT 5;
  pid  | event_type |   event    | count
-------+------------+------------+-------
 71963 | Client     | ClientRead |  2601
 71963 | IO         | WalSync    |  2092
 71963 |            |            |  1226
 71963 | IO         | WalWrite   |    22
 71963 | LWLock     | ProcArray  |     3
(5 rows)
```

10ms v. 1.1.6

С установками по умолчанию (параметр *pg_wait_sampling.profile_period*) значения сохраняются 100 раз в секунду. Поэтому, чтобы оценить длительность ожиданий в секундах, значение count надо делить на 100; общая длительность будет примерно соответствовать времени работы процесса.

В данном случае бóльшую часть времени утилита выполняла полезную работу. «Непроизводственные» ожидания в основном связаны с синхронизацией журнальных записей с диском. Этот пример хорошо иллюстрирует понятие *неучтенного* времени. Обработка события WALSync появилась в версии PostgreSQL 12; на более ранней версии профиль не содержал бы первой строки, хотя само ожидание, разумеется, имело бы место.

с. 99

Легкая блокировка 🔒 ProcArray защищает одноименную структуру данных в разделяемой памяти сервера.

А вот как будет выглядеть профиль, если искусственно замедлить работу файловой системы (например, с помощью slowfs[1]), чтобы любая операция ввода-вывода занимала 0,1 секунды:

```
postgres$ /usr/local/pgsql/bin/pgbench -T 60 internals
=> SELECT pid FROM pg_stat_activity WHERE application_name = 'pgbench';
  pid
-------
 72414
(1 row)
=> SELECT pid, event_type, event, count
FROM pg_wait_sampling_profile WHERE pid = 72414
ORDER BY count DESC LIMIT 5;
  pid  | event_type |     event      | count
-------+------------+----------------+-------
 72414 | IO         | WalWrite       |  5329
 72414 | LWLock     | WALWrite       |   167
 72414 |            |                |    50
 72414 | IO         | WalSync        |    43
 72414 | IO         | DataFileExtend |    19
(5 rows)
```

Теперь на первое место вышли операции ввода-вывода, главным образом связанные с записью журнала в синхронном режиме. Поскольку запись журнала на диск защищена легкой блокировкой 🔒 WALWrite, соответствующая строка также присутствует в профиле.

[1] github.com/nirs/slowfs.

Очевидно, что эта блокировка запрашивалась и в предыдущем примере, но, поскольку ее ожидание было короче, чем период семплирования, она либо попала в выборку очень небольшое количество раз, либо не попала вовсе. Это еще раз показывает, что анализ коротких ожиданий требует семплирования в течение достаточно продолжительного времени.

Часть IV

Выполнение запросов

16

Этапы выполнения запросов

16.1. Демонстрационная база данных

В предыдущих частях книги мы обходились простыми таблицами с несколькими строками. Эта и следующая части имеют дело с выполнением запросов, и нам понадобятся взаимосвязанные таблицы с большим количеством строк. Вместо того чтобы изобретать для каждого примера свой набор данных, я взял готовую демонстрационную базу данных авиаперевозок[1]. Демобаза существует в нескольких вариантах; я использую версию большого объема от 15.08.2017. Для ее установки надо разархивировать файл с копией базы данных[2] и выполнить его в psql.

При разработке демобазы мы старались сделать схему данных достаточно простой, чтобы ее было легко понять без особых пояснений, но в то же время достаточно сложной, чтобы она позволяла строить осмысленные запросы. База наполнена реалистичными данными, которые упрощают понимание примеров и с которыми интересно работать.

Ниже я кратко опишу основные объекты демобазы, а полное описание можно прочитать по ссылке в сноске.

Основной сущностью является **бронирование** (таблица `bookings`). В одно бронирование можно включить несколько пассажиров, каждому из которых выдается электронный **билет** (`tickets`). Пассажир не является отдельной сущностью; будем считать, что все пассажиры уникальны.

[1] postgrespro.ru/education/demodb.
[2] edu.postgrespro.ru/demo-big-20170815.zip.

Каждый билет включает один или несколько **перелетов** (`ticket_flights`). Несколько перелетов может появиться, когда выполняется полет с пересадками или билет взят «туда и обратно». В схеме данных нет жесткого ограничения, но предполагается, что все билеты в одном бронировании имеют одинаковый набор перелетов.

Каждый **рейс** (`flights`) следует из одного **аэропорта** (`airports`) в другой. Рейсы с одним номером имеют одинаковые пункты вылета и назначения, но будут отличаться датой отправления.

Представление `routes` выделяет из таблицы рейсов информацию о **маршрутах**, не зависящую от конкретных дат рейсов.

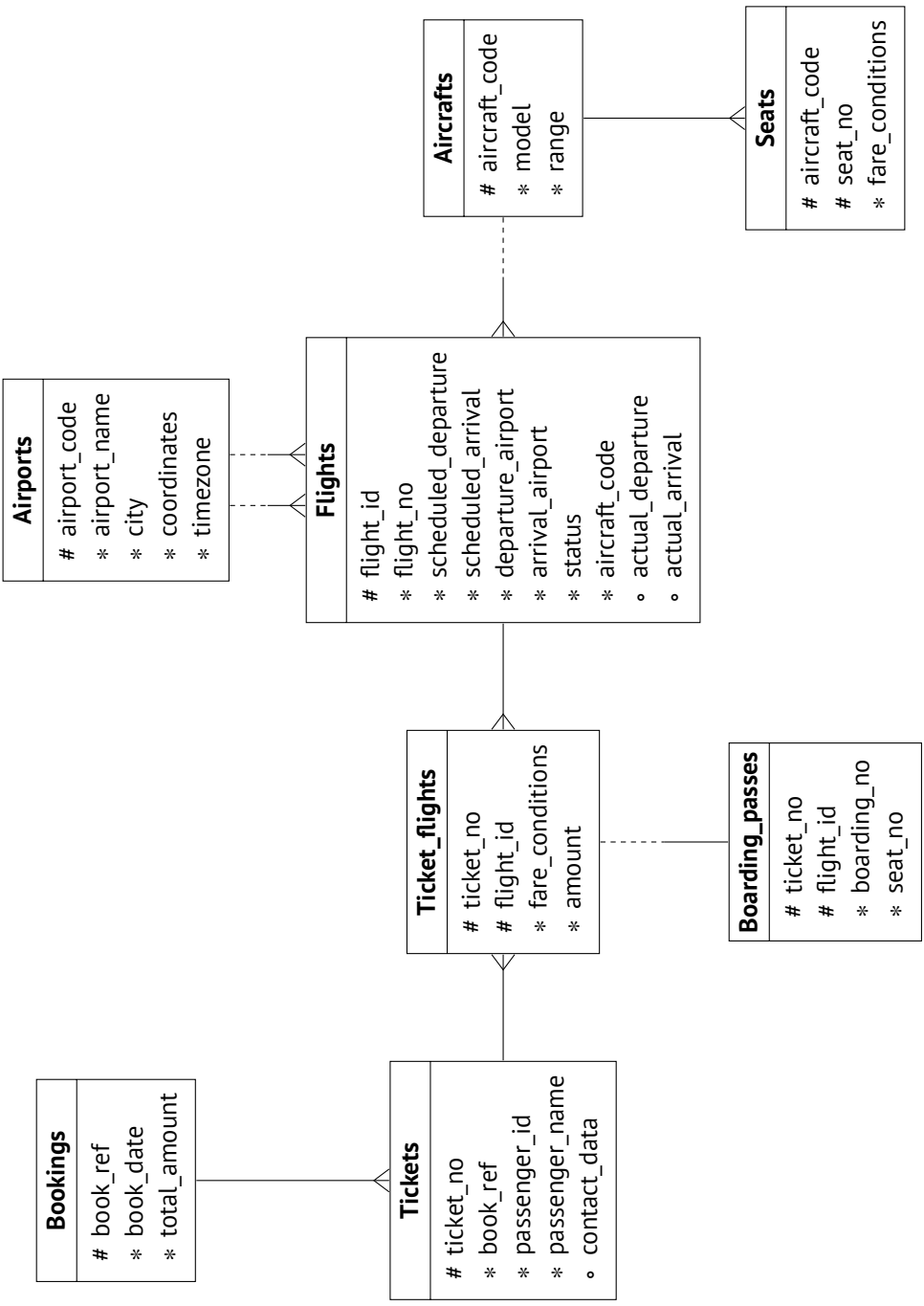
При регистрации на рейс пассажиру выдается **посадочный талон** (`boarding_passes`), в котором указано место в самолете. Пассажир может зарегистрироваться только на тот рейс, который есть у него в билете. Комбинация рейса и места в самолете должна быть уникальной, чтобы не допустить выдачу двух посадочных талонов на одно место.

Количество **мест** (`seats`) в самолете и их распределение по классам обслуживания зависят от конкретной модели **самолета** (`aircrafts`), выполняющего рейс. Предполагается, что у каждой модели есть только одна компоновка салона.

В одних таблицах используются суррогатные первичные ключи, в других — естественные (причем некоторые из них являются составными). Это сделано в демонстрационных целях и не является примером для подражания.

Демобазу можно рассматривать как резервную копию реальной системы: она содержит актуальные данные на определенный момент. Время этого момента сохранено в функции `bookings.now`. Ее можно использовать в запросах там, где в обычной жизни встретилась бы функция `now`.

Названия аэропортов, городов и моделей самолетов хранятся в таблицах `airports_data` и `aircrafts_data` на двух языках, русском и английском. В запросах обычно используются представления `airports` и `aircrafts`, показанные на ER-диаграмме. Они подставляют перевод в зависимости от
ru языка, выбранного в параметре *bookings.lang*. Но имена базовых таблиц могут появляться в планах выполнения запросов.


Aircrafts
aircraft_code
* model
* range
Seats
aircraft_code
seat_no
* fare_conditions
Airports
airport_code
* airport_name
* city
* coordinates
* timezone
Flights
flight_id
* flight_no
* scheduled_departure
* scheduled_arrival
* departure_airport
* arrival_airport
* status
* aircraft_code
o actual_departure
o actual_arrival
Ticket_flights
ticket_no
flight_id
* fare_conditions
* amount
Boarding_passes
ticket_no
flight_id
* boarding_no
* seat_no
Bookings
book_ref
* book_date
* total_amount
Tickets
ticket_no
* book_ref
* passenger_id
* passenger_name
o contact_data

16.2. Простой протокол запросов

Простой вариант клиент-серверного протокола PostgreSQL[1] позволяет отправить серверу текст SQL-запроса и получить в ответ все строки результата, сколько бы их ни было[2]. Запрос, поступающий серверу, проходит несколько этапов: он разбирается, трансформируется, планируется и, наконец, исполняется.

Разбор

Во-первых, текст запроса необходимо *разобрать*[3] (parse), чтобы понять, что именно требуется выполнить.

Лексический и синтаксический разборы. *Лексический анализатор* разбирает текст запроса на *лексемы*[4] (такие как ключевые слова, строковые и числовые литералы), а *синтаксический анализатор* убеждается, что полученный набор лексем соответствует грамматике языка[5]. PostgreSQL использует для разбора стандартные инструменты — утилиты Flex и Bison.

Разобранный запрос представляется в памяти обслуживающего процесса в виде абстрактного синтаксического дерева.

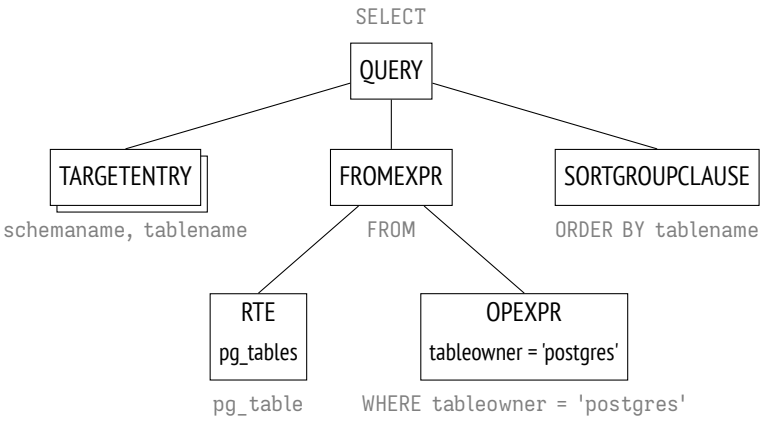
Возьмем для примера следующий запрос:

```
SELECT schemaname, tablename
FROM pg_tables
WHERE tableowner = 'postgres'
ORDER BY tablename;
```

Лексический анализатор выделит пять ключевых слов, пять идентификаторов, строковый литерал и три односимвольные лексемы (для запятой, знака равенства и точки с запятой). Синтаксический анализатор построит из этих

[1] postgrespro.ru/docs/postgresql/18/protocol.
[2] backend/tcop/postgres.c, функция exec_simple_query.
[3] postgrespro.ru/docs/postgresql/18/parser-stage; backend/parser/README.
[4] backend/parser/scan.l.
[5] backend/parser/gram.y.

лексем дерево, показанное на рисунке в сильно упрощенном виде. Рядом с узлами дерева подписаны части запроса, которые им соответствуют:



RTE — неочевидное сокращение от Range Table Entry. Именем range table в исходном коде PostgreSQL называются таблицы, подзапросы, результаты соединений — иными словами, *наборы строк*, над которыми работают операторы SQL[1].

Семантический разбор. Задача *семантического анализа*[2] — определить, есть ли в базе данных таблицы и другие объекты, на которые запрос ссылается по имени, и есть ли у пользователя право обращаться к этим объектам. Вся необходимая для семантического анализа информация хранится в системном каталоге. *с. 24*

Семантический анализатор получает от синтаксического анализатора дерево разбора и перестраивает его, дополняя ссылками на конкретные объекты базы данных, указанием типов данных и другой информацией.

Полное дерево, построенное в результате разбора, можно получить в журнале сообщений сервера, установив параметр *debug_print_parse*, хотя практического смысла в этом немного.

[1] include/nodes/parsenodes.h.
[2] backend/parser/analyze.c.

Трансформация

Далее запрос может *трансформироваться* (*переписываться*)[1].

Трансформации используются ядром для нескольких целей. Одна из них — заменять в дереве разбора имя представления на поддерево, соответствующее запросу этого представления.

Другой пример использования трансформаций ядром системы — реализация разграничения доступа на уровне строк[2] (row-level security).

v. 14 На этапе трансформации также происходит преобразование предложений SEARCH и CYCLE для рекурсивных запросов[3].

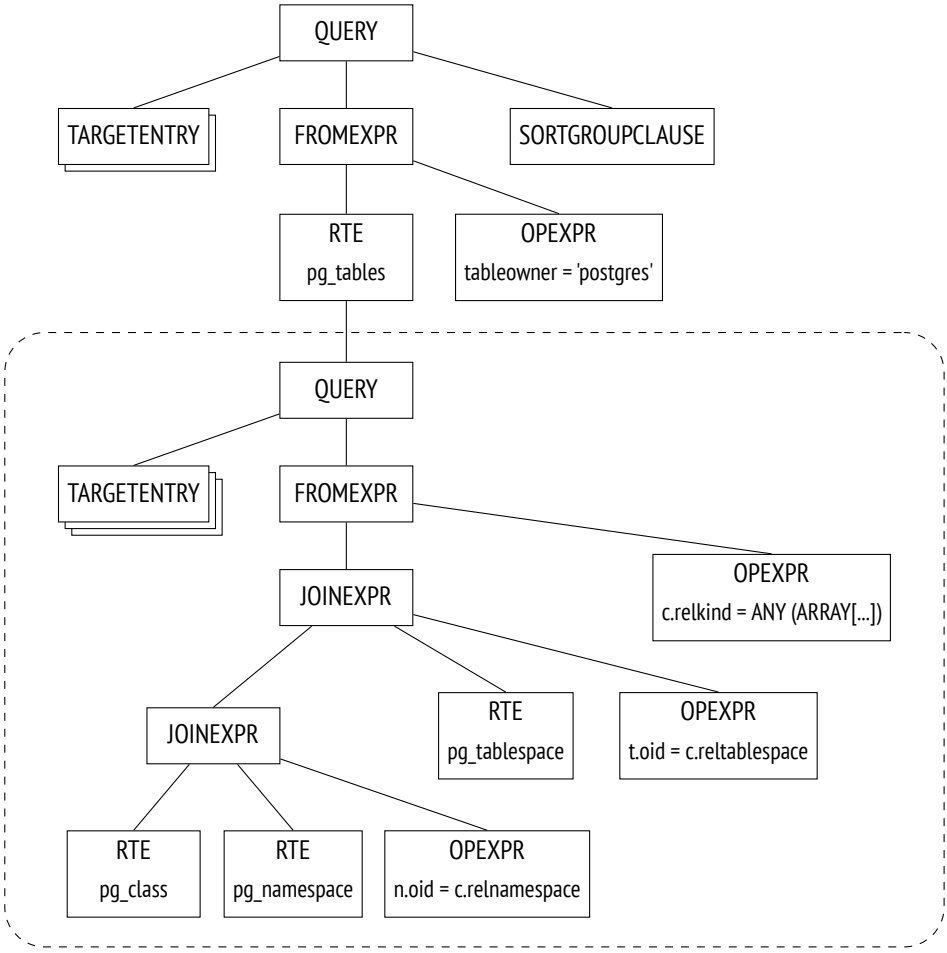
В примере выше pg_tables — представление; подставив его определение в текст запроса, мы получили бы:

```
SELECT schemaname, tablename
FROM (
    -- pg_tables
    SELECT n.nspname AS schemaname,
      c.relname AS tablename,
      pg_get_userbyid(c.relowner) AS tableowner,
      ...
    FROM pg_class c
      LEFT JOIN pg_namespace n ON n.oid = c.relnamespace
      LEFT JOIN pg_tablespace t ON t.oid = c.reltablespace
    WHERE c.relkind = ANY (ARRAY['r'::char, 'p'::char])
)
WHERE tableowner = 'postgres'
ORDER BY tablename;
```

На самом деле сервер не работает с текстовым представлением и выполняет все манипуляции только над деревом разбора. Его состояние после этапа трансформации показано на рисунке в упрощенном виде (а полный можно получить в журнале сообщений, установив параметр *debug_print_rewritten*).

Дерево разбора отражает синтаксическую структуру запроса, но ничего не говорит о том, в каком порядке будут выполнены операции.

[1] postgrespro.ru/docs/postgresql/18/rule-system.
[2] backend/rewrite/rowsecurity.c.
[3] backend/rewrite/rewriteSearchCycle.c.



PostgreSQL дает пользователю возможность написать свои собственные трансформации. Для этого используется система *правил перезаписи*[1] (rules).

> Поддержка правил была заявлена как одна из целей[2] разработки Postgres, так что правила существовали еще на этапе университетского проекта и затем неоднократно переосмысливались. Это мощный, но сложный в отладке и понимании механизм. Поступало даже предложение убрать правила из PostgreSQL, но оно не нашло общей поддержки. В большинстве случаев вместо правил удобнее и безопаснее использовать триггеры.

[1] postgrespro.ru/docs/postgresql/18/rules.
[2] *M. Stonebraker, L. A. Rowe*. The Design of Postgres.

Планирование

SQL — декларативный язык: запрос определяет, *какие* данные надо получить, но не говорит, *как именно* их получать.

Любой запрос можно выполнить разными способами. Для каждой операции, представленной в дереве разбора, могут существовать разные варианты ее реализации: например, данные из таблицы можно получить, прочитав всю таблицу (и отбросив ненужное), а можно найти подходящие строки с помощью индекса. Наборы строк всегда соединяются попарно, что приводит к огромному количеству вариантов, отличающихся порядком соединений. Кроме того, существуют разные алгоритмы соединений: например, можно перебирать строки одного набора и находить для них соответствие во втором наборе, а можно предварительно отсортировать оба набора и затем слить их вместе. Какие-то алгоритмы работают лучше в одних ситуациях, какие-то — в других.

Разница во времени выполнения между неоптимальными и оптимальными планами может составлять многие и многие порядки, поэтому *планировщик*[1], выполняющий *оптимизацию* разобранного запроса, является одним из самых сложных компонентов системы.

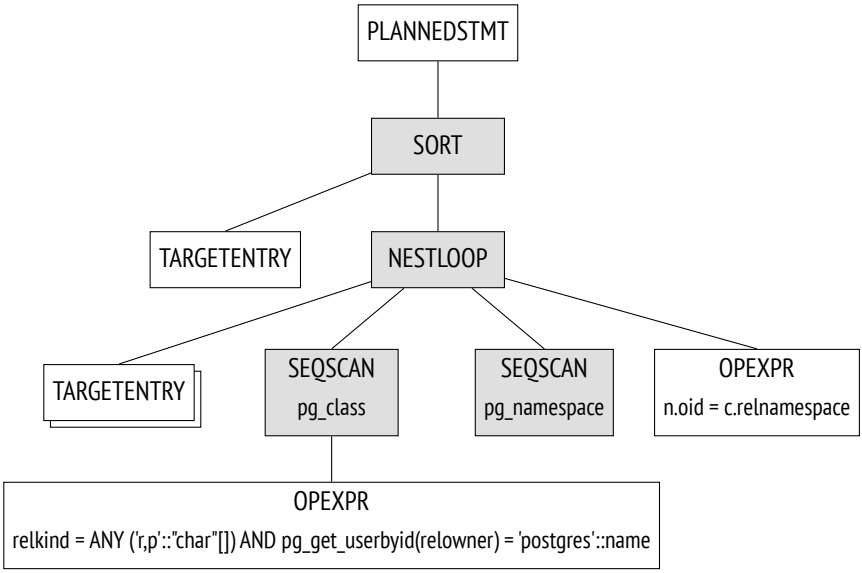
Дерево плана. План выполнения также представляется в виде дерева, но его узлы содержат не логические, а физические операции над данными.

Для исследовательских целей полное дерево плана можно получить в журнале сообщений сервера, установив параметр *debug_print_plan*. А на практике текстовое представление плана выводит команда `EXPLAIN`[2].

На рисунке выделены основные узлы дерева. Именно они и показаны ниже в выводе команды `EXPLAIN`.

с. 374 *с. 440* Узлы Seq Scan в плане запроса соответствуют чтению таблиц, а узел Nested Loop — соединению.

[1] postgrespro.ru/docs/postgresql/18/planner-optimizer.
[2] postgrespro.ru/docs/postgresql/18/using-explain.




```
=> EXPLAIN SELECT schemaname, tablename
FROM pg_tables
WHERE tableowner = 'postgres'
ORDER BY tablename;
                              QUERY PLAN
-----------------------------------------------------------------------
 Sort  (cost=22.91..22.92 rows=1 width=128)
   Sort Key: c.relname
   -> Nested Loop Left Join  (cost=0.00..22.90 rows=1 width=128)
       Join Filter: (n.oid = c.relnamespace)
       -> Seq Scan on pg_class c  (cost=0.00..21.79 rows=1 width=72)
           Filter: ((relkind = ANY ('{r,p}'::"char"[])) AND (pg_g...
       -> Seq Scan on pg_namespace n  (cost=0.00..1.05 rows=5 wid...
(7 rows)
```

Пока стоит обратить внимание на два момента:

- из трех таблиц запроса в дереве осталось только две: планировщик понял, что одна из таблиц не нужна для получения результата и ее можно удалить из дерева плана;

- каждый узел дерева снабжен информацией о предполагаемом числе продуцируемых строк (rows) и о стоимости (cost).

Перебор планов. PostgreSQL использует *стоимостной оптимизатор*[1]. Оптимизатор рассматривает потенциально возможные планы и оценивает предполагаемое количество ресурсов, необходимых для их выполнения (таких как операции ввода-вывода и такты процессора). Оценка оптимизатора, приведенная к числовому виду, называется *стоимостью* плана. Из всех просмотренных планов выбирается план с наименьшей стоимостью.

Проблема в том, что количество возможных планов экспоненциально зависит от количества соединяемых таблиц, и просто перебрать один за другим все варианты нереально даже для относительно простых запросов. Для сокращения пространства перебора традиционно используется алгоритм динамического программирования в сочетании с некоторыми эвристиками. Это позволяет увеличить количество таблиц в запросе, для которых может быть найдено математически точное решение за приемлемое время.

> Точное решение задачи оптимизации не гарантирует, что найденный план *действительно* будет лучшим, поскольку планировщик использует упрощенные математические модели и может действовать на основе не вполне точной входной информации.

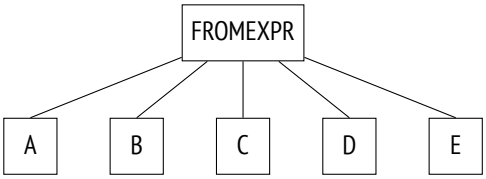
К сожалению, даже алгоритм динамического программирования занимает слишком много времени и требует слишком много памяти, когда количество соединений переваливает за десяток. Чтобы еще больше ограничить перебор, планировщик может зафиксировать порядок некоторых соединений в соответствии с синтаксической структурой запроса[2].

Рассмотрим запрос, в котором таблицы перечислены через запятую в предложении `FROM` без явного указания `JOIN`:

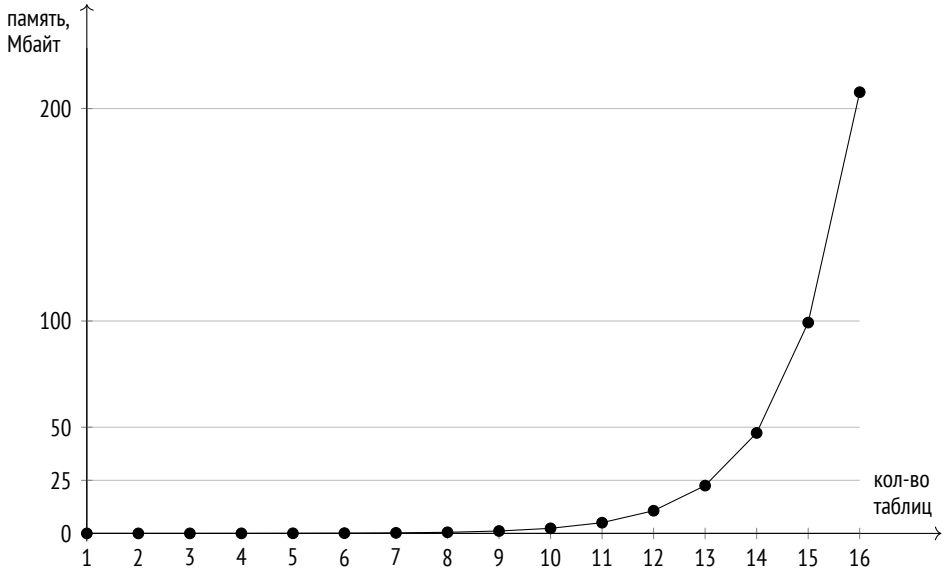
```
SELECT ...
FROM a, b, c, d, e
WHERE ...
```

Для такого запроса планировщик будет рассматривать все возможные комбинации попарных соединений. Запросу соответствует следующий (схематично показанный) фрагмент дерева разбора:

[1] backend/optimizer/README.
[2] postgrespro.ru/docs/postgresql/18/explicit-joins.



Ниже приведен график зависимости объема памяти, необходимого для планирования такого запроса, от количества таблиц в нем (данные получены командой EXPLAIN с параметром MEMORY): v. 17

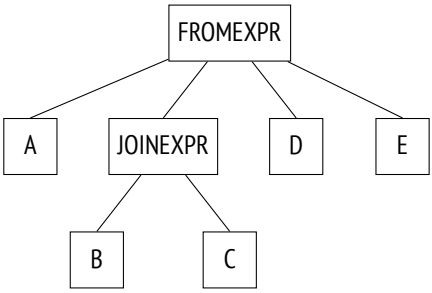


Время планирования также примерно удваивается с каждой добавляемой таблицей.

В другом примере соединения имеют структуру, определяемую предложением JOIN:

```
SELECT ...
FROM a,
  b JOIN c ON ...,
  d,
  e
WHERE ...
```

Дерево разбора отражает эту структуру:



Обычно планировщик уплощает дерево соединений, преобразуя его к тому же виду, что в первом примере. Алгоритм рекурсивно обходит дерево, заменяя узлы joinexpr плоским списком нижележащих элементов[1].

Но уплощение выполняется только при условии, что в плоском списке не
8 появится более *join_collapse_limit* элементов. В данном примере пять элементов, поэтому при любых значениях параметра, меньших 5, узел joinexpr уплощаться не будет.

Для планировщика это означает следующее:

- таблица B должна быть соединена с таблицей C (или наоборот, C с B — на порядок соединения в паре ограничение не накладывается);
- таблицы A, D, E и результат соединения B с C могут быть соединены в любом порядке.

При значении параметра *join_collapse_limit* = 1 порядок любых явных соединений JOIN будет сохранен.

Кроме того, операнды FULL OUTER JOIN *никогда* не уплощаются, какое бы значение ни принимал параметр *join_collapse_limit*.

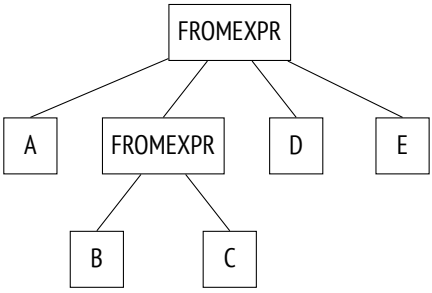
8 Параметр *from_collapse_limit* точно так же ограничивает уплощение подзапросов. Хотя внешне подзапросы не похожи на соединения JOIN, на уровне дерева разбора аналогия становится очевидной.

[1] backend/optimizer/plan/initsplan.c, функция deconstruct_jointree.

Вот пример запроса:

```
SELECT ...
FROM a,
  (
    SELECT ...
    FROM b, c
    WHERE ...
  ) bc,
  d, e
WHERE ...
```

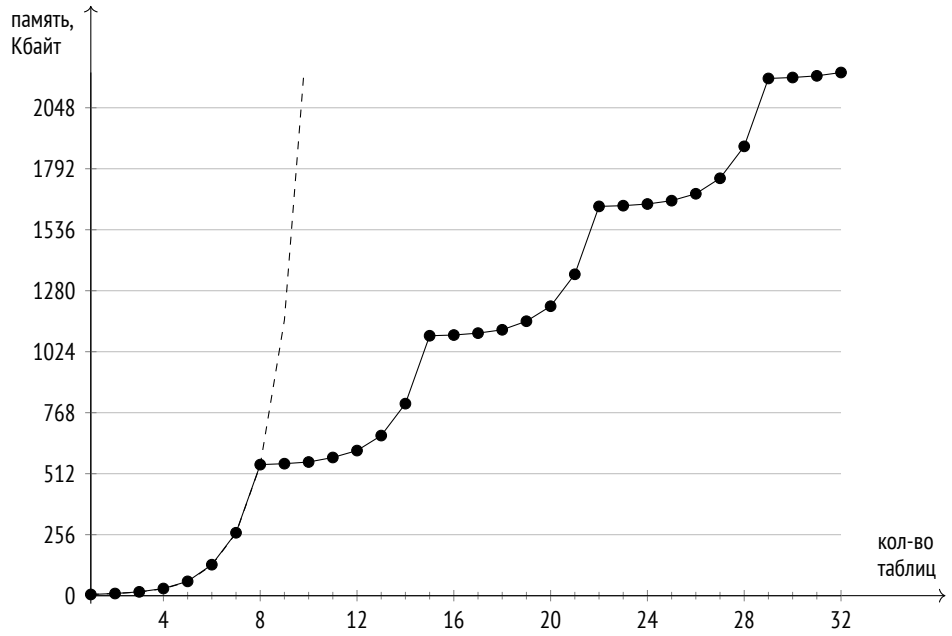
Соответствующее ему дерево соединений показано на рисунке. Вся разница в том, что вместо узла joinexpr стоит узел fromexpr (отсюда и не совсем очевидное название параметра).



Таким образом, количество элементов на одном уровне дерева после процедуры уплощения, как правило, ограничено значениями *join_collapse_limit* и *from_collapse_limit*.

Ниже показано, сколько памяти требуется планировщику для запроса, содержащего только соединения JOIN. На графике видно, что с увеличением количества таблиц расход ресурсов прибывает экспоненциальными «ступеньками», сохраняя в целом линейный рост. Начальная ступенька образована первыми восьмью (*join_collapse_limit*) элементами, следующая добавляет к этой группе еще семь элементов и т. д. Такой подход позволяет за разумное время планировать запросы даже с очень большим количеством JOIN-соединений.

Для наглядности пунктиром показан и предыдущий график для запросов с соединениями через запятую.



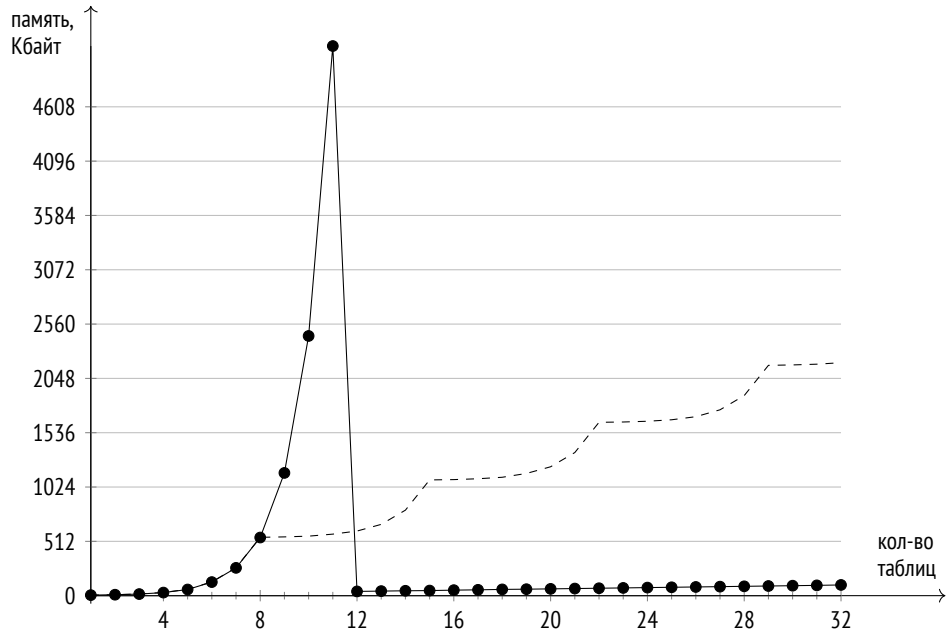
Остается, однако, проблема запросов с большим количеством соединений
on через запятую. Включенный параметр *geqo* переключает планировщик на использование *генетического алгоритма*[1], если число элементов достигает
12 значения параметра *geqo_threshold*. Генетический алгоритм работает существенно быстрее алгоритма динамического программирования (даже с учетом ограничений *join_collapse_limit* и *from_collapse_limit*), но не гарантирует нахождение оптимального плана. Для генетического алгоритма имеется целый ряд настроек[2], но я не буду их рассматривать.

Ниже показан график потребности в памяти для запроса, содержащего соединения через запятую, с включенным параметром *geqo*. Видно, что когда количество таблиц доходит до двенадцати (*geqo_threshold*), необходимость в ресурсах резко сокращается и далее растет линейно и очень медленно.

Для сравнения пунктиром показан график для запросов с `JOIN`-соединениями.

[1] postgrespro.ru/docs/postgresql/18/geqo;
backend/optimizer/geqo/geqo_main.c.

[2] postgrespro.ru/docs/postgresql/18/runtime-config-query#RUNTIME-CONFIG-QUERY-GEQO.



Неполное уплощение дерева или переключение на генетический алгоритм приводит к произвольному сокращению рассматриваемых вариантов плана и, как следствие, к риску упустить оптимальный план и нестабильным результатам планирования. В таких случаях автору запроса может потребоваться взять на себя разбиение запроса на независимо оптимизируемые части:

- Общие табличные выражения могут оптимизироваться отдельно от основного запроса; предложение `MATERIALIZED` гарантирует именно такое поведение[1]. v. 12
- Запросы внутри функций, написанных на любом языке, кроме SQL, оптимизируются отдельно от основного запроса (тело функции на SQL в некоторых случаях может подставляться в запрос[2]).
- Установка параметров *join_collapse_limit* и *from_collapse_limit* в единицу сохраняет порядок соединений, заданный синтаксической структурой запроса (что, скорее всего, слишком негибко).

[1] postgrespro.ru/docs/postgresql/18/queries-with.
[2] wiki.postgresql.org/wiki/Inlining_of_SQL_functions.

Выбор лучшего плана. Для клиента оптимальность плана зависит от того, как предполагается использовать результат. Если результат нужен целиком (например, для построения отчета), план должен оптимизировать получение всех строк выборки. Но чтобы как можно быстрее получить первые строки (например, для вывода на экран), может потребоваться совершенно другой план.

PostgreSQL решает эту задачу, вычисляя две компоненты стоимости:

```
=> EXPLAIN
SELECT schemaname, tablename
FROM pg_tables
WHERE tableowner = 'postgres'
ORDER BY tablename;
                              QUERY PLAN
-----------------------------------------------------------------------
 Sort  (cost=22.91..22.92 rows=1 width=128)
   Sort Key: c.relname
   -> Nested Loop Left Join  (cost=0.00..22.90 rows=1 width=128)
       Join Filter: (n.oid = c.relnamespace)
       -> Seq Scan on pg_class c  (cost=0.00..21.79 rows=1 width=72)
           Filter: ((relkind = ANY ('{r,p}'::"char"[])) AND (pg_g...
       -> Seq Scan on pg_namespace n  (cost=0.00..1.05 rows=5 wid...
(7 rows)
```

Первая компонента (начальная стоимость, startup cost) представляет затраты на подготовку к началу выполнения узла, а вторая (полная стоимость, total cost) — полные затраты на получение всех результирующих строк.

> Иногда можно встретить утверждение, что начальная стоимость — это стоимость получения первой строки выборки, но это не вполне точно.

Чтобы решить, каким планам отдавать предпочтение, планировщик смотрит, используется ли курсор (команда SQL `DECLARE` или *явное* объявление курсора в PL/pgSQL)[1]. Если нет, то предполагается немедленное получение всех результирующих строк клиентом, и из просмотренных планов оптимизатор выбирает план с наименьшей полной стоимостью.

Для запроса, который выполняется с помощью курсора, выбирается план, оптимизирующий получение не всех строк, а только доли, равной значению

[1] backend/optimizer/plan/planner.c, функция standard_planner.

параметра *cursor_tuple_fraction*. Говоря точнее[1], выбирается план с наименьшим значением выражения 0.1

$$\text{startup cost} + \textit{cursor_tuple_fraction}\,(\text{total cost} - \text{startup cost}).$$

Общая схема вычисления оценки. Чтобы получить общую оценку плана, необходимо оценить каждый из его узлов. Стоимость узла зависит от типа этого узла (очевидно, что стоимость чтения данных из таблицы и стоимость сортировки отличаются) и от объема обрабатываемых этим узлом данных (обычно чем меньше, тем дешевле). Тип узла известен, а для прогноза объема данных необходимо оценить *кардинальность* входных наборов (количество строк, принимаемых узлом на входе) и *селективность* узла (долю строк, которая останется у него на выходе). Для этого, в свою очередь, надо иметь *статистическую информацию* о данных: размер таблиц, распределение данных по столбцам. *с. 346*

Таким образом, оптимизация зависит от корректной статистики, собираемой и обновляемой процессом автоанализа.

Если в каждом узле плана кардинальность оценена правильно, то и стоимость обычно адекватно отражает реальные затраты. Основные ошибки планировщика связаны именно с неправильной оценкой кардинальности и селективности. Это может происходить из-за некорректной или устаревшей статистики, невозможности ее использования или — в меньшей степени — из-за несовершенства моделей, лежащих в основе планировщика.

Оценка кардинальности. Оценка кардинальности — рекурсивный процесс. Чтобы оценить кардинальность узла плана, надо:

1) оценить кардинальность каждого из дочерних узлов и получить количество строк, поступающих узлу на вход;

2) оценить селективность самого узла, то есть долю входящих строк, которая останется на выходе.

Произведение одного на другое и даст кардинальность узла.

[1] backend/optimizer/util/pathnode.c, функция compare_fractional_path_costs.

> Селективность представляется числом от 0 до 1. Селективность, близкая к нулю, обычно называется *высокой*, а близкая к единице — *низкой*. Это может показаться нелогичным, но селективность здесь понимается как *избирательность*: условие, выбирающее малую долю строк, обладает высокой селективностью (избирательностью), а условие, оставляющее почти все строки, — низкой.

Сначала рассчитываются кардинальности листовых узлов, в которых находятся методы доступа к данным. Для этого используется статистика, в частности общий размер таблицы.

Селективность условий, наложенных на таблицу, зависит от вида этих условий. В простейшем случае можно принять за селективность какую-то константу, хотя, конечно, планировщик старается использовать всю доступную информацию для уточнения оценки. Достаточно уметь оценивать селективность простых условий, а селективность условий, составленных с помощью логических операций, рассчитывается по формулам [1]:

$$\begin{aligned} sel_{\text{not } x} &= 1 - sel_x; \\ sel_{x \text{ and } y} &= sel_x \, sel_y; \\ sel_{x \text{ or } y} &= 1 - (1 - sel_x)(1 - sel_y) = sel_x + sel_y - sel_x \, sel_y . \end{aligned}$$

с. 365 К сожалению, эти формулы предполагают независимость предикатов x и y. В случае коррелированных предикатов такая оценка будет неточной.

Для оценки кардинальности соединений вычисляется кардинальность декартова произведения (равная произведению кардинальностей двух наборов данных) и оценивается селективность условий соединения, которая опять же зависит от типа условий.

Аналогично оценивается кардинальность и других узлов, таких как сортировка или агрегация.

Важно отметить, что ошибка расчета кардинальности, возникшая в нижних узлах плана, распространяется выше, приводя в итоге к неверной оценке и выбору неудачного плана. Этот неприятный эффект усугубляется тем, что планировщик располагает статистической информацией только о таблицах, но не о результатах соединений.

[1] backend/optimizer/path/clausesel.c, функции clauselist_selectivity_ext и clauselist_selectivity_or.

Оценка стоимости. Процесс оценки стоимости также рекурсивен. Чтобы вычислить стоимость поддерева плана, надо рассчитать стоимости дочерних узлов, сложить их и добавить стоимость самого узла.

Стоимость работы узла рассчитывается на основе математической модели операции, которую выполняет данный узел. Входными данными для оценки служит кардинальность, которая уже рассчитана. Отдельно оцениваются начальная и полная стоимости.

Некоторые операции не требуют никакой подготовки и начинают выполняться немедленно; у таких узлов начальная стоимость будет равна нулю. Другие операции, наоборот, требуют выполнения предварительных действий. Например, узел сортировки обычно должен получить от дочернего узла *все* данные, чтобы начать работу. У таких узлов начальная стоимость будет отлична от нуля — эту цену придется заплатить, даже если вышестоящему узлу (или клиенту) потребуется только одна строка из всей результирующей выборки.

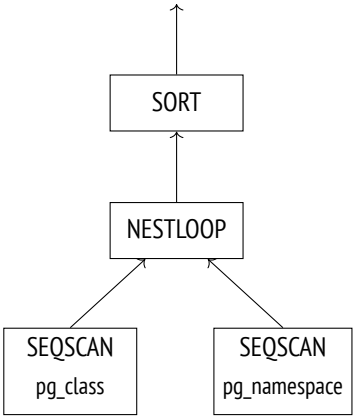
Стоимость отражает оценку планировщика и, если планировщик ошибается, может не коррелировать с реальным временем выполнения. Она нужна лишь для того, чтобы планировщик мог сравнивать разные планы *одного и того же* запроса в *одних и тех же* условиях. В остальных случаях сравнивать запросы (тем более разные) по стоимости — неправильно и бессмысленно. Например, стоимость могла быть недооценена из-за неправильной статистики; после актуализации статистики стоимость может вырасти, но стать более адекватной, и план на самом деле улучшится.

Исполнение

План, построенный в ходе оптимизации запроса, передается на *исполнение*[1]. В памяти обслуживающего процесса создается *портал*[2] — объект, хранящий состояние выполняющегося запроса. Состояние представляется в виде дерева, повторяющего структуру дерева плана. Фактически узлы дерева работают как конвейер, запрашивая и передавая друг другу строки.

[1] postgrespro.ru/docs/postgresql/18/executor; backend/executor/README.

[2] backend/utils/mmgr/portalmem.c.



Выполнение начинается с корня. Корневой узел (в примере это операция сортировки sort) обращается за данными к дочернему узлу. Получив все строки, узел выполняет сортировку и отдает данные выше, то есть клиенту.

Некоторые узлы (как nestloop на рисунке) соединяют данные, полученные из разных источников. Такой узел обращается за данными к двум дочерним узлам. Получив две строки, удовлетворяющие условию соединения, узел сразу же передает результирующую строку наверх (в отличие от сортировки, которая сначала вынуждена получить *все* строки). На этом выполнение узла прерывается до тех пор, пока вышестоящий узел не затребует следующую строку. Таким образом, если нужен только частичный результат (например, из-за ограничения LIMIT), операция не будет выполняться полностью.

Два листовых узла дерева seqscan представляют обращение к таблицам. Когда вышестоящий узел обращается за данными, листовой узел читает очередную строку из соответствующей таблицы и возвращает ее.

Таким образом, часть узлов не хранит строки, а немедленно передает их выше и тут же забывает, но некоторым узлам требуется сначала сохранить — *материализовать* — все строки. Один из таких узлов — сканирование общего табличного выражения CTE Scan. Материализацию включает либо разработ-
v. 12 чик (явным указанием MATERIALIZED), либо планировщик (чтобы не вычислять одно и то же табличное выражение несколько раз).

Объем данных, подлежащих материализации, заранее неизвестен. В надежде, что строк будет немного, сначала для хранилища выделяется фрагмент
4MB оперативной памяти обслуживающего процесса размером *work_mem*. Пока этой памяти хватает, данные сохраняются в ней[1].

[1] backend/utils/sort/tuplestore.c.

```
=> EXPLAIN (analyze, costs off, timing off, buffers off)
WITH t AS MATERIALIZED (
  SELECT * FROM seats
)
SELECT * FROM t;
                        QUERY PLAN
-----------------------------------------------------------
 CTE Scan on t (actual rows=1339.00 loops=1)
   Storage: Memory  Maximum Storage: 69kB
   CTE t
     -> Seq Scan on seats (actual rows=1339.00 loops=1)
(4 rows)
```

Если же память переполняется, *все* данные сбрасываются на диск во временный файл и при необходимости читаются из этого файла.

```
=> EXPLAIN (analyze, costs off, timing off, buffers off)
WITH t AS MATERIALIZED (
  SELECT * FROM tickets
)
SELECT * FROM t;
                          QUERY PLAN
-------------------------------------------------------------
 CTE Scan on t (actual rows=2949857.00 loops=1)
   Storage: Disk  Maximum Storage: 330144kB
   CTE t
     -> Seq Scan on tickets (actual rows=2949857.00 loops=1)
(4 rows)
```

Тот же механизм материализации используется рекурсивными запросами (для рабочей и промежуточной таблиц), оконными функциями (для текущей рамки), табличными функциями (для накопления результата), триггерами (для сторонних таблиц и переходных отношений) и т. п. Он же применяется и в узле Materialize, который планировщик может вставлять в план для материализации вычисленных строк.

Похожий, но более сложный механизм применяется для сортировки. *с. 489*

Таким образом, в одном плане может быть несколько узлов, которым необходимо хранилище данных, поэтому для запроса может быть выделено несколько фрагментов памяти, каждый размером *work_mem*. Общий объем оперативной памяти, доступный запросу, никак не ограничен, а предельный объем временных файлов можно установить параметром *temp_file_limit*. -1

16.3. Расширенный протокол запросов

При использовании простого протокола запросов каждая команда, даже если она повторяется из раза в раз, проходит все перечисленные выше этапы:

1) разбор;
2) трансформацию;
3) планирование;
4) исполнение.

Но нет никакого смысла разбирать один и тот же запрос заново. Нет смысла разбирать и запросы, отличающиеся только константами, — структура дерева разбора не изменится.

Еще одно неудобство простого способа выполнения запросов состоит в том, что клиент получает всю выборку сразу, сколько бы строк она ни содержала.

В принципе, оба ограничения можно преодолеть, используя команды SQL: первое — подготавливая запрос командой `PREPARE` и выполняя его с помощью `EXECUTE`, второе — создавая курсор командой `DECLARE` и извлекая строки с помощью `FETCH`. Но в этом случае на клиента ложится ответственность за именование создаваемых объектов, а сервер нагружается лишней работой по разбору дополнительных команд.

Поэтому расширенный клиент-серверный протокол позволяет детально управлять отдельными этапами выполнения операторов на уровне команд самого протокола.

Подготовка

На этапе *подготовки* запрос разбирается и трансформируется обычным образом, но полученное дерево разбора сохраняется в памяти обслуживающего процесса.

В PostgreSQL отсутствует глобальный кеш запросов. Минусы такого решения понятны — каждый обслуживающий процесс вынужден разбирать все запросы самостоятельно, даже если такой же запрос уже был разобран другим

процессом. Но есть и плюсы. Кеш в общей памяти легко может стать узким местом из-за необходимости блокировок. Один клиент, выполняющий множество мелких, но разных запросов (отличающихся, например, только константами), создает большую нагрузку на такой кеш, что может сказаться на производительности всего экземпляра. В PostgreSQL разбор запросов выполняется локально и поэтому не влияет на другие процессы.

с. 305

При подготовке запроса его можно параметризовать. Воспользуемся тем, что psql позволяет задействовать расширенный протокол:

v. 18

```
=> SELECT * FROM aircrafts WHERE aircraft_code = $1 \parse plane
```

Посмотреть именованные подготовленные операторы можно в представлении pg_prepared_statements:

```
=> SELECT name, statement, parameter_types
FROM pg_prepared_statements \gx
-[ RECORD 1 ]---+-----------------------------------------------
name            | plane
statement       | SELECT * FROM aircrafts WHERE aircraft_code = $1
parameter_types | {character}
```

Увидеть таким образом безымянные подготовленные операторы (которые, например, PL/pgSQL создает для каждого выражения) не получится. Конечно, сеансу доступны только собственные операторы; заглянуть в память другого сеанса невозможно.

Привязка параметров

Перед выполнением подготовленного запроса выполняется привязка фактических значений параметров:

```
=> \bind_named plane '733' \g
 aircraft_code |     model     | range
---------------+---------------+-------
 733           | Боинг 737-300 |  4200
(1 row)
```

Преимущество параметров подготовленных операторов перед конкатенацией литералов со строкой запроса — принципиальная невозможность

внедрения SQL-кода, поскольку никакое значение параметра не сможет изменить уже построенное дерево разбора. Чтобы достичь того же уровня безопасности без подготовленных операторов, требуется аккуратно экранировать каждое значение, полученное из ненадежного источника.

Планирование и исполнение

Когда дело доходит до выполнения подготовленного оператора, происходит планирование с учетом значений фактических параметров, после чего план передается на исполнение.

Учитывать значения параметров важно, поскольку оптимальные планы для разных значений могут не совпадать. Например, поиск очень дорогих бронирований использует индекс, так как планировщик предполагает, что подходящих строк не очень много:

```
=> CREATE INDEX ON bookings(total_amount);
=> EXPLAIN SELECT * FROM bookings
WHERE total_amount > 1000000;
                              QUERY PLAN
-----------------------------------------------------------------------
 Bitmap Heap Scan on bookings  (cost=81.86..9154.92 rows=4314 wid...
   Recheck Cond: (total_amount > '1000000'::numeric)
   -> Bitmap Index Scan on bookings_total_amount_idx  (cost=0.00....
        Index Cond: (total_amount > '1000000'::numeric)
(4 rows)
```

Однако под следующее условие попадают вообще все бронирования, поэтому индекс бесполезен, и таблица просматривается целиком:

```
=> EXPLAIN SELECT * FROM bookings WHERE total_amount > 100;
                             QUERY PLAN
--------------------------------------------------------------------
 Seq Scan on bookings  (cost=0.00..39876.88 rows=2111110 width=21)
   Filter: (total_amount > '100'::numeric)
(2 rows)
```

В некоторых случаях планировщик запоминает не только дерево разбора, но и план запроса, чтобы не выполнять планирование повторно. Такой план,

построенный без учета значений параметров, называется *общим планом* (в отличие от *частного плана*, учитывающего фактические значения)[1].

Очевидный случай, когда можно перейти на общий план без ущерба для эффективности, — запрос без параметров.

Подготовленные операторы с параметрами первые пять раз всегда оптимизируются с учетом фактических значений; при этом вычисляется средняя стоимость частных планов. Начиная с шестого раза, если общий план оказывается в среднем выгоднее, чем частные (с учетом того, что частные планы необходимо каждый раз строить заново[2]), планировщик запоминает общий план и дальше использует его, уже не повторяя оптимизацию.

Подготовленный оператор plane уже был один раз выполнен. Теперь перейдем с метакоманд psql на обычные команды SQL: это не совсем то же самое, что прямое использование команд протокола, но в конечном счете эффект тот же. Зато так будет проще увидеть, что после следующих трех выполнений по-прежнему используются частные планы — обратите внимание на значение параметра в плане запроса:

```
=> EXECUTE plane('763'); -- то же, что \bind_named, но в SQL
=> EXECUTE plane('773');
=> EXPLAIN EXECUTE plane('319');
                            QUERY PLAN
------------------------------------------------------------------
 Seq Scan on aircrafts_data ml  (cost=0.00..1.36 rows=1 width=40)
   Filter: (aircraft_code = '319'::bpchar)
(2 rows)
```

После еще одного, пятого выполнения планировщик переключится на использование общего плана — он совпадает с частными планами, имеет ту же стоимость, но его можно запомнить и больше не тратить ресурсы на оптимизацию. Команда EXPLAIN показывает теперь не значение параметра, а его номер:

```
=> EXECUTE plane('320');
```

[1] backend/utils/cache/plancache.c, функция choose_custom_plan.
[2] backend/utils/cache/plancache.c, функция cached_plan_cost.

```
=> EXPLAIN EXECUTE plane('321');
                              QUERY PLAN
-----------------------------------------------------------------------
 Seq Scan on aircrafts_data ml  (cost=0.00..1.36 rows=1 width=40)
   Filter: (aircraft_code = $1)
(2 rows)
```

Несложно представить ситуацию, в которой по неудачному стечению обстоятельств первые несколько частных планов будут более дорогими, чем общий план, а последующие оказались бы эффективнее общего — но планировщик уже не будет их рассматривать. Кроме того, планировщик сравнивает *оценки* стоимостей, а не фактические ресурсы, и это также может приводить к ошибкам.

v. 12 Поэтому при неправильном автоматическом решении можно принудительно выбрать общий либо частный план, установив соответствующее значение
auto параметра *plan_cache_mode*:

```
=> SET plan_cache_mode = 'force_custom_plan';
=> EXPLAIN EXECUTE plane('CN1');
                              QUERY PLAN
-----------------------------------------------------------------------
 Seq Scan on aircrafts_data ml  (cost=0.00..1.36 rows=1 width=40)
   Filter: (aircraft_code = 'CN1'::bpchar)
(2 rows)
```

v. 14 Представление `pg_prepared_statements` показывает в том числе и статистику выбора планов:

```
=> SELECT name, generic_plans, custom_plans
FROM pg_prepared_statements;
 name  | generic_plans | custom_plans
-------+---------------+--------------
 plane |             1 |            6
(1 row)
```

Получение результатов

Расширенный протокол запросов позволяет клиенту принимать результат построчно или частями по нескольку строк. Почти тот же эффект дает

использование SQL-курсоров (за исключением лишней работы для сервера и того факта, что планировщик оптимизирует получение не всей выборки, а первых *cursor_tuple_fraction* строк):

```
=> BEGIN;
=> DECLARE cur CURSOR FOR
  SELECT * FROM aircrafts ORDER BY aircraft_code;
=> FETCH 3 FROM cur;
 aircraft_code |      model       | range
---------------+------------------+-------
 319           | Аэробус A319-100 |  6700
 320           | Аэробус A320-200 |  5700
 321           | Аэробус A321-200 |  5600
(3 rows)
=> FETCH 2 FROM cur;
 aircraft_code |     model     | range
---------------+---------------+-------
 733           | Боинг 737-300 |  4200
 763           | Боинг 767-300 |  7900
(2 rows)
=> COMMIT;
```

Если запрос возвращает много строк и клиенту нужны они все, то огромное значение для скорости передачи данных играет размер выборки, получаемой за один раз. Чем больше выборка, тем меньше коммуникационных издержек на обращение к серверу и получение ответа. Но с ростом выборки эффект сокращения издержек уменьшается: разница между одной строкой и десятью может быть колоссальной, а между сотней и тысячей — уже почти незаметной.

17

Статистика

17.1. Базовая статистика

Базовая статистика уровня отношения[1] хранится в таблице pg_class системного каталога. К ней относятся:

- число строк в отношении (reltuples);
- размер отношения, в страницах (relpages);
- количество страниц, отмеченных в карте видимости (relallvisible).

с. 32

Вот эти значения на примере таблицы flights:

```
=> SELECT reltuples, relpages, relallvisible
FROM pg_class WHERE relname = 'flights';
 reltuples | relpages | relallvisible
-----------+----------+---------------
    214867 |     2624 |             0
(1 row)
```

Значение reltuples используется в качестве оценки кардинальности, когда запрос не накладывает никаких условий на строки таблицы:

```
=> EXPLAIN SELECT * FROM flights;
                          QUERY PLAN
---------------------------------------------------------------
 Seq Scan on flights  (cost=0.00..4772.67 rows=214867 width=63)
(1 row)
```

[1] postgrespro.ru/docs/postgresql/18/planner-stats.

Статистика собирается при анализе, ручном или автоматическом[1]. Однако ввиду особой важности базовая статистика рассчитывается также при выполнении некоторых операций (VACUUM FULL и CLUSTER[2], CREATE INDEX и REINDEX[3]) и уточняется при очистке[4]. с. 137

Для анализа случайно выбираются 300 × *default_statistics_target* строк. Поскольку размер выборки, достаточной для построения статистики заданной точности, слабо зависит от объема анализируемых данных, размер таблицы не учитывается[5]. 100

Строки выбираются из такого же количества (300 × *default_statistics_target*) случайных страниц[6]. Конечно, для небольшой таблицы количество прочитанных страниц и выбранных для анализа строк может оказаться меньше.

Поскольку в достаточно больших таблицах статистика собирается не по всем строкам, оценки могут немного расходиться с реальностью. Это нормально: статистика в любом случае не может все время быть точной, если данные изменяются. Для выбора адекватного плана обычно достаточно попадания в порядок.

Создадим копию таблицы flights с отключенной автоочисткой, чтобы управлять временем выполнения анализа:

```
=> CREATE TABLE flights_copy(LIKE flights)
WITH (autovacuum_enabled = false);
```

Для новой таблицы никакой статистики еще нет:

```
=> SELECT reltuples, relpages, relallvisible
FROM pg_class WHERE relname = 'flights_copy';
 reltuples | relpages | relallvisible
-----------+----------+---------------
        -1 |        0 |             0
(1 row)
```

[1] backend/commands/analyze.c, функция do_analyze_rel.
[2] backend/commands/cluster.c, функция copy_table_data.
[3] backend/catalog/heap.c, функция index_update_stats.
[4] backend/access/heap/vacuumlazy.c, функция heap_vacuum_rel.
[5] backend/commands/analyze.c, функция std_typanalyze.
[6] backend/commands/analyze.c, функция acquire_sample_rows; backend/utils/misc/sampling.c.

v. 14 Значение `reltuples` = −1 позволяет отличить таблицу, для которой статистика ни разу не собиралась, от действительно пустой таблицы без строк.

Но с большой вероятностью в таблицу будут добавлены какие-то строки сразу после создания. Поэтому, находясь в неведении, планировщик считает, что таблица занимает 10 страниц:

```
=> EXPLAIN SELECT * FROM flights_copy;
                           QUERY PLAN
-----------------------------------------------------------------
 Seq Scan on flights_copy  (cost=0.00..14.10 rows=410 width=170)
(1 row)
```

Количество строк рассчитывается исходя из размера одной строки; он отображается в плане запроса как width. Обычно для оценки используется среднее значение, вычисляемое при анализе, но в этом случае, поскольку статистика отсутствует, размер строки вычисляется приблизительно с учетом типов данных каждого из столбцов[1].

Теперь скопируем данные из таблицы `flights` и выполним анализ:

```
=> INSERT INTO flights_copy SELECT * FROM flights;
INSERT 0 214867
=> ANALYZE flights_copy;
```

Сейчас статистика совпадает с реальным количеством строк (размер таблицы таков, что статистика собирается по полным данным):

```
=> SELECT reltuples, relpages, relallvisible
FROM pg_class WHERE relname = 'flights_copy';
 reltuples | relpages | relallvisible
-----------+----------+--------------
    214867 |     2624 |             0
(1 row)
```

с. 422 Значение `relallvisible` используется при оценке стоимости сканирования только индекса. Оно обновляется при очистке:

```
=> VACUUM flights_copy;
```

[1] backend/access/table/tableam.c, функция table_block_relation_estimate_size.

```
=> SELECT relallvisible FROM pg_class WHERE relname = 'flights_copy';
 relallvisible
---------------
          2624
(1 row)
```

Теперь удвоим количество строк, не собирая статистику, и проверим оценку кардинальности в плане запроса:

```
=> INSERT INTO flights_copy SELECT * FROM flights;
=> SELECT count(*) FROM flights_copy;
 count
--------
 429734
(1 row)
=> EXPLAIN SELECT * FROM flights_copy;
                             QUERY PLAN
--------------------------------------------------------------------
 Seq Scan on flights_copy  (cost=0.00..9545.34 rows=429734 width=63)
(1 row)
```

Оценка оказалась точна, несмотря на устаревшие сведения в pg_class:

```
=> SELECT reltuples, relpages
FROM pg_class WHERE relname = 'flights_copy';
 reltuples | relpages
-----------+----------
    214867 |     2624
(1 row)
```

Дело в том, что планировщик повышает точность оценки, масштабируя значение reltuples в соответствии с отклонением реального размера файла данных от значения relpages[1]. Поскольку размер файла вырос в два раза по сравнению с relpages, количество строк скорректировалось исходя из предположения, что плотность данных не изменилась:

```
=> SELECT reltuples *
  (pg_relation_size('flights_copy') / 8192) / relpages AS tuples
FROM pg_class WHERE relname = 'flights_copy';
```

[1] backend/access/table/tableam.c, функция table_block_relation_estimate_size.

```
 tuples
--------
 429734
(1 row)
```

Конечно, такая корректировка работает не всегда (например, если удалить часть строк, оценка не изменится), но в ряде случаев позволяет «продержаться» до прихода анализа при крупных изменениях.

17.2. Неопределенные значения

Неопределенные значения, порицаемые теоретиками[1], играют тем не менее важную роль в реляционных базах данных как удобный способ представления того факта, что значение не существует или неизвестно.

Но особое значение требует и особого к себе отношения. Помимо теоретических неувязок, возникает множество сугубо практических сложностей, с которыми приходится считаться. Обычная булева логика превращается в трехзначную, а конструкция `NOT IN` ведет себя *неожиданно*. Непонятно, должны ли неопределенные значения считаться меньше обычных значений или больше (отсюда специальные предложения `NULLS FIRST` и `NULLS LAST` для сортировки). Не так уж очевидно, должны ли неопределенные значения учитываться в агрегатных функциях. Поскольку, строго говоря, неопределенные значения вовсе не являются значениями, то для их учета планировщику необходима дополнительная информация.

Помимо самой простой, базовой статистики на уровне отношений, при анализе собирается статистика для каждого столбца отношения. Она хранится в таблице системного каталога `pg_statistic`[2], но значительно проще пользоваться представлением `pg_stats`, которое показывает информацию в более удобном виде.

Доля неопределенных значений как раз относится к статистике уровня столбца; вычисленное при анализе значение показывает атрибут `null_frac`.

[1] citforum.ru/database/articles/evergreen_nulls.
[2] include/catalog/pg_statistic.h.

Например, чтобы найти еще не отправившиеся рейсы, можно воспользоваться тем, что время их вылета не определено:

```
=> EXPLAIN SELECT * FROM flights WHERE actual_departure IS NULL;
                          QUERY PLAN
---------------------------------------------------------------
 Seq Scan on flights  (cost=0.00..4772.67 rows=16473 width=63)
   Filter: (actual_departure IS NULL)
(2 rows)
```

Оценка вычисляется как общее число строк, умноженное на долю NULL:

```
=> SELECT round(reltuples * s.null_frac) AS rows
FROM pg_class
  JOIN pg_stats s ON s.tablename = relname
WHERE s.tablename = 'flights'
  AND s.attname = 'actual_departure';
 rows
-------
 16473
(1 row)
```

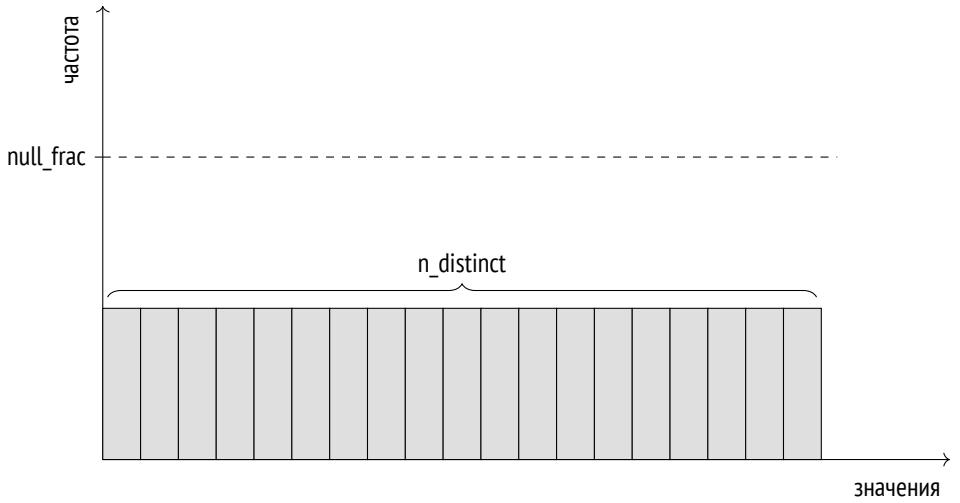
Точное значение:

```
=> SELECT count(*) FROM flights WHERE actual_departure IS NULL;
 count
-------
 16348
(1 row)
```

17.3. Уникальные значения

Поле n_distinct представления pg_stats показывает количество уникальных значений в столбце.

Если значение n_distinct отрицательно, то модуль этого числа равен не количеству, а доле уникальных значений. Например, –1 означает, что все значения в столбце уникальны, а –3 говорит о том, что каждое значение в среднем встречается в трех строках. Анализатор использует доли, когда вычисленное при анализе количество уникальных значений превышает 10 % от



общего количества строк; в этом случае пропорция, скорее всего, сохранится и при дальнейшем изменении данных[1].

Количество уникальных значений используется во всех случаях, которые предполагают равномерное распределение данных. Например, при оценке кардинальности условия «*столбец = выражение*», когда значение *выражения* неизвестно на этапе планирования, считается, что *выражение* может принимать любое из возможных значений столбца с равной вероятностью[2]:

```
=> EXPLAIN SELECT *
FROM flights
WHERE departure_airport = (
  SELECT airport_code FROM airports WHERE city = 'Санкт-Петербург'
);
                              QUERY PLAN
-----------------------------------------------------------------------
 Seq Scan on flights  (cost=30.56..5340.40 rows=2066 width=63)
   Filter: (departure_airport = (InitPlan 1).col1)
   InitPlan 1
     -> Seq Scan on airports_data ml  (cost=0.00..30.56 rows=1 wi...
         Filter: ((city ->> lang()) = 'Санкт-Петербург'::text)
(5 rows)
```

[1] backend/commands/analyze.c, функция compute_distinct_stats.
[2] backend/utils/adt/selfuncs.c, функция var_eq_non_const.

Здесь узел плана InitPlan выполняется один раз, и вычисленное значение используется в основном плане.

```
=> SELECT round(reltuples / s.n_distinct) AS rows
FROM pg_class
  JOIN pg_stats s ON s.tablename = relname
WHERE s.tablename = 'flights'
  AND s.attname = 'departure_airport';
 rows
------
 2066
(1 row)
```

Если количество уникальных значений вычисляется неверно (из-за ограниченности выборки, по которой проводится анализ), это количество можно указать для столбца явно:

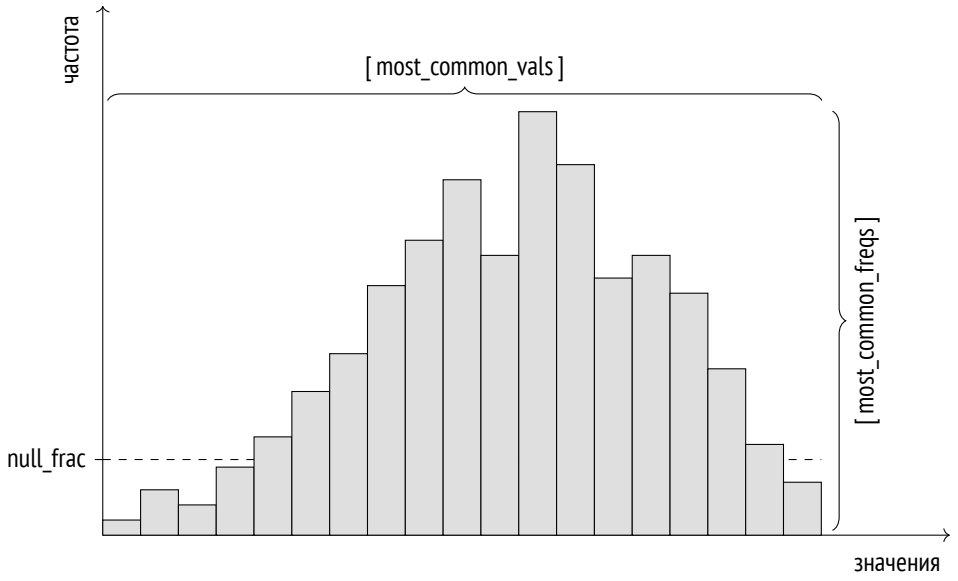
```
ALTER TABLE ... ALTER COLUMN ... SET (n_distinct = ...);
```

Если бы все данные были всегда распределены равномерно, этой информации (дополненной минимальным и максимальным значениями) было бы достаточно. Но при неравномерном распределении, которое на практике встречается гораздо чаще, такая оценка не будет точна:

```
=> SELECT min(cnt), round(avg(cnt)) avg, max(cnt) FROM (
  SELECT departure_airport, count(*) cnt
  FROM flights GROUP BY departure_airport
) t;
 min | avg  |  max
-----+------+-------
 113 | 2066 | 20875
(1 row)
```

17.4. Наиболее частые значения

Для уточнения оценки при неравномерном распределении собирается статистика по наиболее часто встречающимся значениям (most common values, MCV) и частоте их появления. Представление `pg_stats` показывает два этих массива в столбцах `most_common_vals` и `most_common_freqs`.



Вот пример такой статистики по частоте использования различных типов самолетов:

```
=> SELECT most_common_vals AS mcv,
  left(most_common_freqs::text,60) || '...' AS mcf
FROM pg_stats
WHERE tablename = 'flights' AND attname = 'aircraft_code' \gx
-[ RECORD 1 ]----------------------------------------------------------
mcv | {CN1,CR2,SU9,321,733,319,763,773}
mcf | {0.28123334,0.2653,0.264,0.0595,0.03816667,0.0374,0.0368,0.0...
```

Для оценки селективности условия «*столбец = значение*» достаточно найти значение в массиве `most_common_vals` и взять частоту из элемента массива `most_common_freqs` с тем же номером[1]:

```
=> EXPLAIN SELECT * FROM flights WHERE aircraft_code = '733';
                          QUERY PLAN
---------------------------------------------------------------
 Seq Scan on flights  (cost=0.00..5309.84 rows=8201 width=63)
   Filter: (aircraft_code = '733'::bpchar)
(2 rows)
```

[1] backend/utils/adt/selfuncs.c, функция var_eq_const.

```
=> SELECT round(reltuples * s.most_common_freqs[
  array_position((s.most_common_vals::text::text[]),'733')
])
FROM pg_class
  JOIN pg_stats s ON s.tablename = relname
WHERE s.tablename = 'flights'
  AND s.attname = 'aircraft_code';
 round
-------
  8201
(1 row)
```

Очевидно, что такая оценка будет близка к точному значению:

```
=> SELECT count(*) FROM flights WHERE aircraft_code = '733';
 count
-------
  8263
(1 row)
```

Список частых значений используется и для оценки селективности условий с неравенствами. Например, для условия вида «*столбец* < *значение*» надо найти в `most_common_vals` все значения, меньшие искомого, и просуммировать частоты из `most_common_freqs`[1].

Статистика частых значений отлично работает, когда количество различных значений не очень велико. Максимальный размер массивов определяется тем же параметром *default_statistics_target*, который ограничивает размер случайной выборки строк для анализа.

100

В некоторых случаях может иметь смысл увеличить значение параметра по умолчанию, чтобы расширить список частых значений и повысить точность оценок. Это можно сделать на уровне отдельного столбца:

```
ALTER TABLE ... ALTER COLUMN ... SET STATISTICS ...;
```

При этом увеличится и размер выборки, но только для указанной таблицы.

Поскольку в массиве частых значений сохраняются сами значения, массив может занимать довольно много места. Чтобы чрезмерно не увеличивать

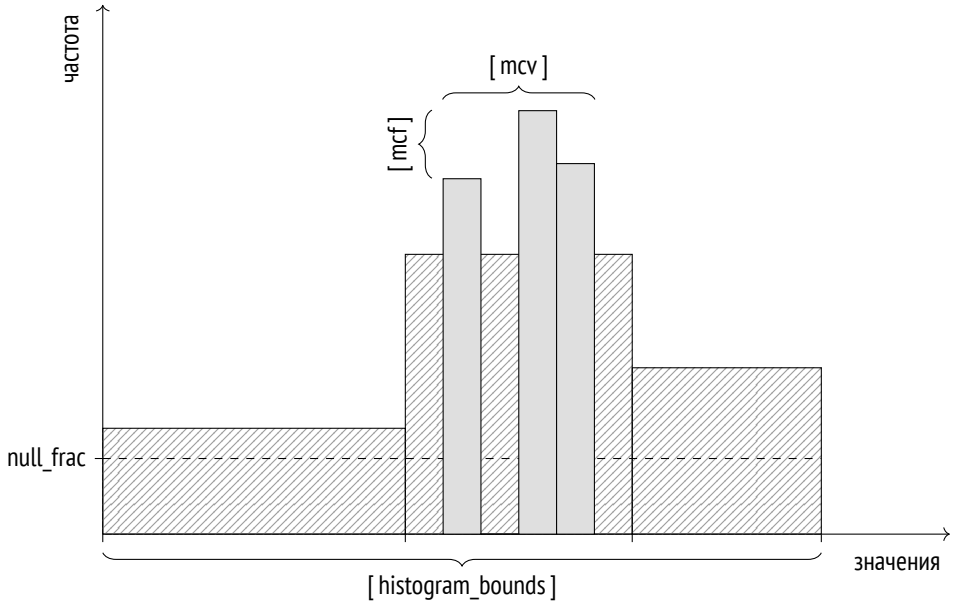
[1] backend/utils/adt/selfuncs.c, функция scalarineqsel.

pg_statistic и не нагружать планировщик бесполезной работой, из анализа и статистики исключаются значения, размер которых превышает 1 Кбайт. К тому же такие большие значения, скорее всего, уникальны и поэтому все равно не должны попасть в most_common_vals.

17.5. Гистограмма

Когда число различных значений слишком велико, чтобы записать их в массив, на помощь приходит гистограмма. Гистограмма состоит из нескольких *корзин*, в которые помещаются значения. Количество корзин ограничено все тем же параметром *default_statistics_target*.

Ширина корзин выбирается так, чтобы в каждую попало примерно одинаковое количество значений (на рисунке этому свойству соответствует одинаковая площадь больших заштрихованных прямоугольников). При этом учитываются только те значения, которые не попали в список наиболее частых. При таком построении суммарная частота значений одной (любой) корзины равна $\frac{1}{\textit{число корзин}}$.



Гистограмма хранится в поле histogram_bounds представления pg_stats как массив значений, ограничивающих корзины:

```
=> SELECT left(histogram_bounds::text,60) || '...' AS hist_bounds
FROM pg_stats s
WHERE s.tablename = 'boarding_passes' AND s.attname = 'seat_no';
                          hist_bounds
---------------------------------------------------------------
 {10A,10A,10F,10F,11G,12B,13B,13K,14C,14C,15B,15C,15C,16B,17B...
(1 row)
```

Гистограмма используется, в частности, для оценки селективности операторов «больше» или «меньше» вместе со списком наиболее частых значений[1]. Рассмотрим пример — количество посадочных талонов, выданных на дальние ряды:

```
=> EXPLAIN SELECT * FROM boarding_passes WHERE seat_no > '30B';
                             QUERY PLAN
--------------------------------------------------------------------
 Seq Scan on boarding_passes  (cost=0.00..157374.55 rows=2991709 ...
   Filter: ((seat_no)::text > '30B'::text)
(2 rows)
```

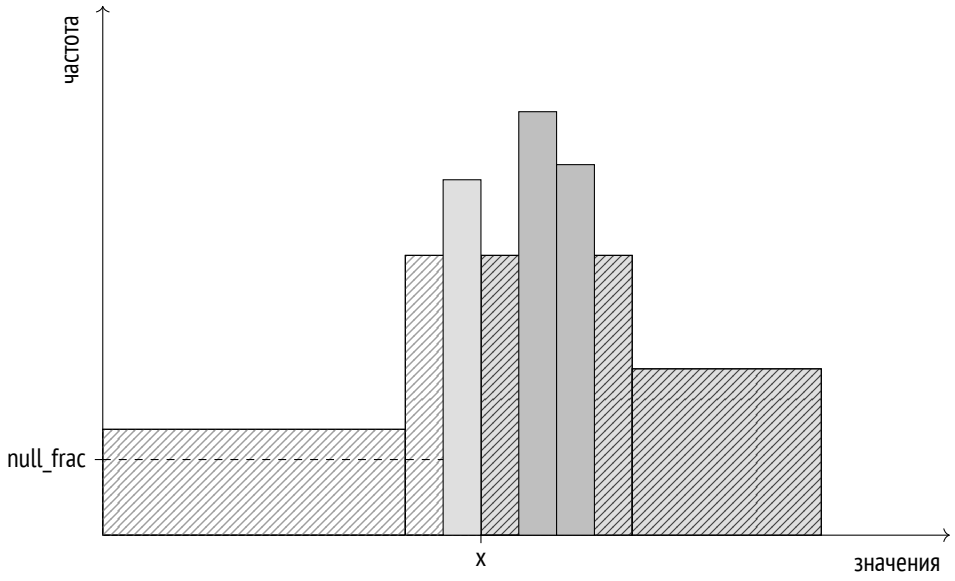
Номер места специально выбран так, что он лежит точно на границе корзин гистограммы.

Оценка селективности такого условия будет равна $\frac{N}{\text{число корзин}}$, где N — количество корзин, значения в которых удовлетворяют условию (то есть находятся справа от значения). При этом необходимо учесть, что наиболее частые значения не входят в гистограмму.

Вообще говоря, неопределенные значения тоже не входят в гистограмму, но в столбце seat_no их быть не может:

```
=> SELECT s.null_frac FROM pg_stats s
WHERE s.tablename = 'boarding_passes' AND s.attname = 'seat_no';
 null_frac
-----------
         0
(1 row)
```

[1] backend/utils/adt/selfuncs.c, функция ineq_histogram_selectivity.



Сначала найдем долю наиболее частых значений, которые удовлетворяют условию:

```
=> SELECT sum(s.most_common_freqs[
  array_position((s.most_common_vals::text::text[]),v)
])
FROM pg_stats s, unnest(s.most_common_vals::text::text[]) v
WHERE s.tablename = 'boarding_passes' AND s.attname = 'seat_no'
  AND v > '30B';
    sum
------------
 0.20833333
(1 row)
```

Общая доля наиболее частых значений (не учитываемая гистограммой) составляет:

```
=> SELECT sum(s.most_common_freqs[

  array_position((s.most_common_vals::text::text[]),v)
])
FROM pg_stats s, unnest(s.most_common_vals::text::text[]) v
WHERE s.tablename = 'boarding_passes' AND s.attname = 'seat_no';
```

```
    sum
-----------
 0.6747334
(1 row)
```

Поскольку интервал занимает ровное количество корзин гистограммы (из 100 возможных), получаем следующую оценку:

```
=> SELECT round( reltuples * (
    0.20833333 -- вклад частых значений
  + (1 - 0.6747334 - 0) * (52 / 100.0) -- вклад гистограммы
))
FROM pg_class WHERE relname = 'boarding_passes';
  round
---------
 2991708
(1 row)
```

В общем случае, когда значение лежит не на границе, с помощью линейной интерполяции учитывается доля корзины, в которой находится значение.

Точное значение составляет:

```
=> SELECT count(*) FROM boarding_passes WHERE seat_no > '30B';
  count
---------
 2993735
(1 row)
```

Увеличение параметра *default_statistics_target* может улучшить оценку, однако, как показывает пример, в сочетании со списком наиболее частых значений гистограмма обычно дает хороший результат даже при большом количестве уникальных значений в столбце:

```
=> SELECT n_distinct FROM pg_stats
WHERE tablename = 'boarding_passes' AND attname = 'seat_no';
 n_distinct
------------
        461
(1 row)
```

Увеличение точности оценки имеет смысл только в том случае, когда оно приводит к построению более качественного плана. Бездумное увеличение параметра может замедлить выполнение анализа и работу планировщика,

ничего не улучшив. С другой стороны, уменьшение параметра (вплоть до нуля) может ускорить анализ и планирование, но может и послужить причиной плохих планов. Такая «экономия» обычно неоправданна.

17.6. Статистика для нескалярных типов данных

Для нескалярных типов данных может собираться статистика по распределению не только самих значений, но и элементов, из которых эти значения состоят. Это позволяет более точно планировать запросы с участием столбцов не в первой нормальной форме.

- Массивы `most_common_elems` и `most_common_elem_freqs` показывают список *наиболее частых элементов* и их частоты.

  Статистика собирается и используется при оценке селективности для массивов [1] и типа данных `tsvector` [2].

- Массив `elem_count_histogram` показывает гистограмму *количества уникальных элементов* в значении.

  Статистика собирается и используется при оценке селективности только для массивов.

- Для непустых диапазонов собираются гистограммы распределения
v. 17 нижних и верхних границ (массив `range_bounds_histogram`) и длины (массив `range_length_histogram`). Отдельно отмечается доля пустых диапазонов (столбец `range_empty_frac`). Эти данные используются для оценки селективности различных операций с диапазонными типами [3].
v. 14 Такая же статистика используется и для мультидиапазонных типов [4].

[1] postgrespro.ru/docs/postgresql/18/arrays;
backend/utils/adt/array_typanalyze.c;
backend/utils/adt/array_selfuncs.c.

[2] postgrespro.ru/docs/postgresql/18/datatype-textsearch;
backend/tsearch/ts_typanalyze.c;
backend/tsearch/ts_selfuncs.c.

[3] postgrespro.ru/docs/postgresql/18/rangetypes;
backend/utils/adt/rangetypes_typanalyze.c;
backend/utils/adt/rangetypes_selfuncs.c.

[4] backend/utils/adt/multirangetypes_selfuncs.c.

17.7. Средний размер поля

Поле avg_width представления pg_stats показывает средний размер значений в столбце. Конечно, для таких типов, как integer или char(3), этот показатель всегда одинаков, но для типов данных с переменной длиной, таких как text, он может сильно отличаться от столбца к столбцу:

```
=> SELECT attname, avg_width FROM pg_stats
WHERE (tablename, attname) IN ( VALUES
  ('tickets', 'passenger_name'), ('ticket_flights','fare_conditions')
);
     attname     | avg_width
-----------------+-----------
 fare_conditions |         8
 passenger_name  |        16
(2 rows)
```

Эта статистика используется для оценки объема памяти, необходимой некоторым операциям, например сортировке или хешированию.

17.8. Корреляция

Поле correlation представления pg_stats показывает корреляцию между физическим расположением данных и логическим порядком в смысле операций сравнения. Если значения хранятся строго по возрастанию, корреляция будет близка к единице; если по убыванию — к минус единице. Чем более хаотично расположены данные на диске, тем ближе значение к нулю.

с. 546

```
=> SELECT attname, correlation
FROM pg_stats WHERE tablename = 'airports_data'
ORDER BY abs(correlation) DESC;
   attname    | correlation
--------------+-------------
 coordinates  |
 airport_code | -0.21120238
 city         |  -0.1970127
 airport_name | -0.18223621
 timezone     |  0.17961165
(5 rows)
```

с. 415 Корреляция используется для оценки стоимости индексного сканирования. Для столбца `coordinates` эта статистика не собирается, поскольку для типа данных `point` не определены операторы «больше» и «меньше».

17.9. Статистика по выражению

Обычно статистика по столбцу может использоваться, только если в операции сравнения слева или справа от оператора фигурирует сам столбец, а не выражение. Например, планировщик не знает, как изменится статистика после вычисления функции от столбца, и поэтому для условия «*вызов-функции = константа*» всегда использует фиксированную оценку в 0,5 %[1]:

```
=> EXPLAIN SELECT * FROM flights
WHERE extract(
  month FROM scheduled_departure AT TIME ZONE 'Europe/Moscow'
) = 1;
                              QUERY PLAN
-----------------------------------------------------------------------
 Seq Scan on flights  (cost=0.00..6384.17 rows=1074 width=63)
   Filter: (EXTRACT(month FROM (scheduled_departure AT TIME ZONE ...
(2 rows)

=> SELECT round(reltuples * 0.005)
FROM pg_class WHERE relname = 'flights';
 round
-------
  1074
(1 row)
```

Планировщику неизвестна семантика даже стандартных функций. Хотя нам из общих соображений понятно, что рейсов, совершенных в январе, будет примерно $\frac{1}{12}$ от общего количества, то есть на порядок больше спрогнозированного значения.

> PostgreSQL позволяет создать вспомогательную функцию для планировщика[2], которая подскажет ему ожидаемое количество строк на выходе некоторой функции при заданных аргументах. Поэтому про некоторые встроенные функции (например, `generate_series`) планировщик кое-что знает. Но далеко не все и не про все.

[1] backend/utils/adt/selfuncs.c, функция eqsel.

[2] postgrespro.ru/docs/postgresql/18/xfunc-optimization.

Чтобы исправить ситуацию, надо собрать статистику не по столбцу таблицы, а по выражению. Это можно сделать двумя способами.

Расширенная статистика по выражению

v. 14

Первый вариант — использовать *расширенную статистику*[1] по выражению. Такая статистика не собирается автоматически; необходимо вручную создать объект базы данных командой CREATE STATISTICS:

```
=> CREATE STATISTICS flights_expr_stat ON (extract(
    month FROM scheduled_departure AT TIME ZONE 'Europe/Moscow'
))
FROM flights;
```

После сбора статистики оценка исправляется:

```
=> ANALYZE flights;
=> EXPLAIN SELECT * FROM flights
WHERE extract(
  month FROM scheduled_departure AT TIME ZONE 'Europe/Moscow'
) = 1;
                              QUERY PLAN
-----------------------------------------------------------------------
 Seq Scan on flights  (cost=0.00..6384.17 rows=16502 width=63)
   Filter: (EXTRACT(month FROM (scheduled_departure AT TIME ZONE ...
(2 rows)
```

Чтобы собранная статистика использовалась, выражение в условии запроса должно быть записано в том же виде, что и в команде CREATE STATISTICS.

Размер собираемой расширенной статистики можно изменить отдельно командой ALTER STATISTICS, например: v. 13

```
=> ALTER STATISTICS flights_expr_stat SET STATISTICS 42;
```

Общая информация о расширенной статистике хранится в таблице системного каталога pg_statistic_ext, а собственно собранная статистика — в отдельной таблице pg_statistic_ext_data. Смысл такого разделения состоит v. 12

[1] postgrespro.ru/docs/postgresql/18/planner-stats#PLANNER-STATS-EXTENDED; backend/statistics/README.

в возможности ограничения доступа пользователей к чувствительной информации.

Доступную пользователю расширенную статистику по выражению можно посмотреть в более удобном виде с помощью специального представления:

```
=> SELECT left(expr,50) || '...' AS expr,
  null_frac, avg_width, n_distinct,
  most_common_vals AS mcv,
  left(most_common_freqs::text,50) || '...' AS mcf,
  correlation
FROM pg_stats_ext_exprs WHERE statistics_name = 'flights_expr_stat' \gx
-[ RECORD 1 ]----------------------------------------------------
expr        | EXTRACT(month FROM (scheduled_departure AT TIME ZO...
null_frac   | 0
avg_width   | 8
n_distinct  | 12
mcv         | {8,9,11,7,5,12,1,3,10,4,6,2}
mcf         | {0.121,0.1112,0.0796,0.0793,0.0791,0.07746667,0.07...
correlation | 0.08110773
```

Статистика для индекса по выражению

Второй способ исправить оценки кардинальности — воспользоваться тем,
с. 401 что при создании индекса по выражению для него собирается отдельная статистика, как для таблицы. Это удобно, если индекс действительно нужен.

```
=> DROP STATISTICS flights_expr_stat;
=> CREATE INDEX ON flights(extract(
  month FROM scheduled_departure AT TIME ZONE 'Europe/Moscow'
));
=> ANALYZE flights;
=> EXPLAIN SELECT * FROM flights
WHERE extract(
  month FROM scheduled_departure AT TIME ZONE 'Europe/Moscow'
) = 1;
                              QUERY PLAN
-----------------------------------------------------------------------
 Bitmap Heap Scan on flights  (cost=318.81..3237.23 rows=16824 wi...
   Recheck Cond: (EXTRACT(month FROM (scheduled_departure AT TIME...
   -> Bitmap Index Scan on flights_extract_idx  (cost=0.00..314.6...
       Index Cond: (EXTRACT(month FROM (scheduled_departure AT TI...
(4 rows)
```

Статистика для индексов по выражению хранится так же, как статистика по таблице. Например, из pg_stats можно получить количество уникальных значений, указав в качестве tablename имя индекса:

```
=> SELECT n_distinct FROM pg_stats
WHERE tablename = 'flights_extract_idx';
 n_distinct
------------
         12
(1 row)
```

Изменить точность статистики в случае индекса можно командой ALTER INDEX. Для этого сначала может потребоваться узнать, как называется столбец, соответствующий выражению. Например:

```
=> SELECT attname FROM pg_attribute
WHERE attrelid = 'flights_extract_idx'::regclass;
 attname
---------
 extract
(1 row)
=> ALTER INDEX flights_extract_idx
  ALTER COLUMN extract SET STATISTICS 42;
```

17.10. Многовариантная статистика

PostgreSQL дает возможность собирать *многовариантную статистику*, охватывающую не один, а несколько столбцов таблицы. Для этого необходимо вручную создать соответствующую расширенную статистику.

Реализовано три вида многовариантной статистики.

Функциональные зависимости между столбцами

Если значения в одном столбце определяются (полностью или частично) значениями другого столбца и в запросе указаны условия на оба таких столбца, оценка кардинальности будет занижена.

Рассмотрим запрос с двумя условиями:

```
=> SELECT count(*) FROM flights
WHERE flight_no = 'PG0007' AND departure_airport = 'VKO';
 count
-------
   396
(1 row)
```

Оценка оказывается сильно заниженной:

```
=> EXPLAIN SELECT * FROM flights
WHERE flight_no = 'PG0007' AND departure_airport = 'VKO';
                              QUERY PLAN
-----------------------------------------------------------------------
 Bitmap Heap Scan on flights  (cost=10.49..814.25 rows=14 width=63)
   Recheck Cond: (flight_no = 'PG0007'::bpchar)
   Filter: (departure_airport = 'VKO'::bpchar)
   -> Bitmap Index Scan on flights_flight_no_scheduled_departure_key
       (cost=0.00..10.48 rows=275 width=0)
       Index Cond: (flight_no = 'PG0007'::bpchar)
(6 rows)
```

Это известная *проблема коррелированных предикатов*. Планировщик полагается на то, что предикаты независимы, и вычисляет общую селективность
с. 336 как произведение селективностей условий, объединенных логическим «и». Это хорошо видно в приведенном плане: оценка в узле Bitmap Index Scan, полученная по условию на столбец `flight_no`, существенно уменьшается после фильтрации по условию на столбец `departure_airport` в узле Bitmap Heap Scan.

Однако мы понимаем, что номер рейса однозначно определяет аэропорты: фактически второе условие избыточно (конечно, если аэропорт указан правильно). В таких случаях расширенная статистика по функциональным зависимостям может улучшить оценку.

Создадим расширенную статистику по функциональной зависимости между двумя столбцами:

```
=> CREATE STATISTICS flights_dep_stat(dependencies)
ON flight_no, departure_airport FROM flights;
```

При очередном анализе таблицы желаемая статистика будет собрана, и оценка улучшится:

```
=> ANALYZE flights;
=> EXPLAIN SELECT * FROM flights
WHERE flight_no = 'PG0007' AND departure_airport = 'VKO';
                              QUERY PLAN
-----------------------------------------------------------------------
 Bitmap Heap Scan on flights  (cost=10.56..816.91 rows=276 width=63)
   Recheck Cond: (flight_no = 'PG0007'::bpchar)
   Filter: (departure_airport = 'VKO'::bpchar)
   -> Bitmap Index Scan on flights_flight_no_scheduled_departure_key
       (cost=0.00..10.49 rows=276 width=0)
       Index Cond: (flight_no = 'PG0007'::bpchar)
(6 rows)
```

В системном каталоге собранную статистику можно посмотреть следующим образом:

```
=> SELECT dependencies
FROM pg_stats_ext WHERE statistics_name = 'flights_dep_stat';
               dependencies
-----------------------------------------
 {"2 => 5": 1.000000, "5 => 2": 0.010300}
(1 row)
```

Числа 2 и 5 — номера столбцов таблицы из `pg_attribute`. Значения определяют степень функциональной зависимости: от 0 (зависимости нет) до 1 (значения в первом столбце полностью определяют значения во втором).

Многовариантное число различных значений

Информация о количестве уникальных комбинаций значений из нескольких столбцов позволяет точнее оценить кардинальность группировки по нескольким столбцам.

Например, количество возможных пар аэропортов отправления и прибытия оценивается планировщиком как квадрат количества аэропортов, но реальное значение сильно меньше, поскольку далеко не каждая пара аэропортов соединена прямым рейсом:

```
=> SELECT count(*)
FROM (
  SELECT DISTINCT departure_airport, arrival_airport FROM flights
) t;
 count
-------
   618
(1 row)
=> EXPLAIN SELECT DISTINCT departure_airport, arrival_airport
FROM flights;
                              QUERY PLAN
-----------------------------------------------------------------------
 HashAggregate  (cost=5847.01..5955.16 rows=10816 width=8)
   Group Key: departure_airport, arrival_airport
   -> Seq Scan on flights  (cost=0.00..4772.67 rows=214867 width=8)
(3 rows)
```

Создадим и соберем расширенную статистику по числу разных значений:

```
=> CREATE STATISTICS flights_nd_stat(ndistinct)
ON departure_airport, arrival_airport FROM flights;
=> ANALYZE flights;
```

Теперь оценка кардинальности исправилась:

```
=> EXPLAIN SELECT DISTINCT departure_airport, arrival_airport
FROM flights;
                              QUERY PLAN
-----------------------------------------------------------------------
 HashAggregate  (cost=5847.01..5853.19 rows=618 width=8)
   Group Key: departure_airport, arrival_airport
   -> Seq Scan on flights  (cost=0.00..4772.67 rows=214867 width=8)
(3 rows)
```

Собранную статистику можно увидеть в системном каталоге:

```
=> SELECT n_distinct
FROM pg_stats_ext WHERE statistics_name = 'flights_nd_stat';
  n_distinct
---------------
 {"5, 6": 618}
(1 row)
```

Многовариантные списки частых значений

v. 12

При неравномерном распределении значений одного только знания функциональной зависимости может быть недостаточно, поскольку оценка существенно зависит от конкретной пары значений. Например, планировщик ошибается, оценивая количество рейсов из Шереметьева, выполняемых Боингом 737:

```
=> SELECT count(*) FROM flights
WHERE departure_airport = 'SVO' AND aircraft_code = '733';
 count
-------
  2037
(1 row)
=> EXPLAIN SELECT * FROM flights
WHERE departure_airport = 'SVO' AND aircraft_code = '733';
                            QUERY PLAN
-------------------------------------------------------------------
 Seq Scan on flights  (cost=0.00..5847.00 rows=737 width=63)
   Filter: ((departure_airport = 'SVO'::bpchar) AND (aircraft_cod...
(2 rows)
```

В таком случае оценку можно уточнить, собирая статистику по многовариантным спискам частых значений[1]:

```
=> CREATE STATISTICS flights_mcv_stat(mcv)
ON departure_airport, aircraft_code FROM flights;
=> ANALYZE flights;
```

Новая оценка кардинальности гораздо точнее:

```
=> EXPLAIN SELECT * FROM flights
WHERE departure_airport = 'SVO' AND aircraft_code = '733';
                            QUERY PLAN
-------------------------------------------------------------------
 Seq Scan on flights  (cost=0.00..5847.00 rows=2127 width=63)
   Filter: ((departure_airport = 'SVO'::bpchar) AND (aircraft_cod...
(2 rows)
```

[1] backend/statistics/README.mcv;
backend/statistics/mcv.c.

Для ее получения планировщик воспользовался частотой, сохраненной в системном каталоге:

```
=> SELECT values, frequency
FROM pg_statistic_ext stx
  JOIN pg_statistic_ext_data stxd ON stx.oid = stxd.stxoid,
  pg_mcv_list_items(stxdmcv) m
WHERE stxname = 'flights_mcv_stat'
AND values = '{SVO,773}';
  values   |      frequency
-----------+---------------------
 {SVO,773} | 0.005066666666666666
(1 row)
```

В многовариантном списке, как и в обычном списке частых значений, со-
100 храняется *default_statistics_target* значений (если параметр задан на уровне столбцов, то используется наибольшее значение).

v. 13 Как и для расширенной статистики по выражению, при необходимости можно изменить размер списка:

```
ALTER STATISTICS ... SET STATISTICS ...;
```

Во всех примерах я использовал только два столбца, но многовариантную статистику можно создавать и по большему количеству.

В одном объекте можно комбинировать статистику разных типов, указывая их через запятую (а если не указать тип, для заданных столбцов будут собираться сразу все возможные варианты статистики).

v. 14 Более того, как и в статистике по выражению, в многовариантной статистике тоже можно использовать произвольные выражения, а не только имена столбцов.

18

Табличные методы доступа

18.1. Подключаемые движки хранения

Способ организации данных на диске, принятый в PostgreSQL, не является ни единственно возможным, ни наилучшим для всех типов нагрузки. Следуя идее расширяемости, PostgreSQL позволяет создавать и подключать различные *табличные методы доступа* (движки хранения данных), хотя в настоящее время «из коробки» доступен только один: v. 12

```
=> SELECT amname, amhandler FROM pg_am WHERE amtype = 't';
 amname |      amhandler
--------+----------------------
 heap   | heap_tableam_handler
(1 row)
```

Имя движка может указываться при создании таблицы (`CREATE TABLE ... USING`); по умолчанию используется движок, определяемый значением параметра *default_table_access_method*. Впоследствии движок можно сменить (`ALTER TABLE ... SET ACCESS METHOD`), но, конечно, ценой переписывания всех данных. heap v. 15

Чтобы ядро могло однотипно работать с разными движками, табличные методы доступа должны реализовывать специальный интерфейс[1]. Функция, указанная в столбце `amhandler`, возвращает интерфейсную структуру[2], содержащую всю необходимую для ядра информацию.

[1] postgrespro.ru/docs/postgresql/18/tableam.
[2] include/access/tableam.h.

Бóльшая часть компонентов ядра остается общей для любых табличных методов доступа:

- менеджер транзакций, включая поддержку ACID и изоляции на основе снимков;
- буферный менеджер;
- журнал, включая поддержку специфичных операций с помощью механизма унифицированных журнальных записей[1] или пользовательских
v. 15 менеджеров ресурсов[2];
- подсистема ввода-вывода;
- TOAST;
- оптимизатор и исполнитель запросов;
- индексная поддержка.

Не все эти компоненты могут быть нужны движку, но возможность их использования сохраняется.

В свою очередь, движок определяет:

- формат версии строки и структуру данных;
- реализацию сканирования таблицы и оценку его стоимости;
- реализацию вставок, удалений, обновлений и блокировок;
- правила видимости версий строк;
- процедуры очистки и анализа.

Исторически PostgreSQL использовал единственную систему хранения данных, встроенную в ядро без какого-либо определенного программного интерфейса. Из-за этого сейчас крайне сложно создать удачный интерфейс, который учитывал бы все сложившиеся особенности стандартного движка и при этом не мешал бы другим методам.

[1] postgrespro.ru/docs/postgresql/18/generic-wal.
[2] postgrespro.ru/docs/postgresql/18/custom-rmgr.

Поэтому и я не старался придерживаться строгого деления между ядром и табличными методами. Многое из сказанного в предыдущих частях книги формально относится не к функциям ядра, а к особенностям метода heap. Скорее всего, этот метод навсегда останется в PostgreSQL как универсальный стандартный движок, а другие методы будут занимать отдельные ниши, позволяя оптимальнее справляться с определенными типами нагрузки.

Освоение механизма табличных методов началось со следующих движков:

Zheap был призван справиться с проблемой раздувания таблиц[1]. Для этого он реализует обновление версий строк на месте и выносит исторические данные, необходимые для построения снимка, в отдельное undo-хранилище. Такой движок полезен при нагрузке, включающей активное обновление данных.

Устройство движка покажется знакомым пользователям Oracle, хотя есть и нюансы (например, интерфейс индексных методов не позволяет создавать индексы с собственной версионностью). *с. 394*

Zedstore реализует колоночное хранение[2] и должен быть эффективен для OLAP-запросов.

Данные организованы в основное B-дерево идентификаторов версий строк, а каждый столбец хранится в собственном B-дереве, связанном с основным (с перспективой сохранения в одном дереве сразу нескольких столбцов для получения гибридного хранилища).

К сожалению, эти движки не были доведены до промышленной готовности; работа над ними фактически прекратилась. В настоящее время активно развиваются другие проекты, например:

OrioleDB — движок, спроектированный с учетом возможностей современного оборудования и особенностей облачных сервисов[3]. Метод доступа хранит данные в таблицах, организованных как деревья, и устраняет раздувание таблиц за счет использования журнала отката.

[1] github.com/EnterpriseDB/zheap.
[2] github.com/greenplum-db/postgres/tree/zedstore.
[3] github.com/orioledb/orioledb.

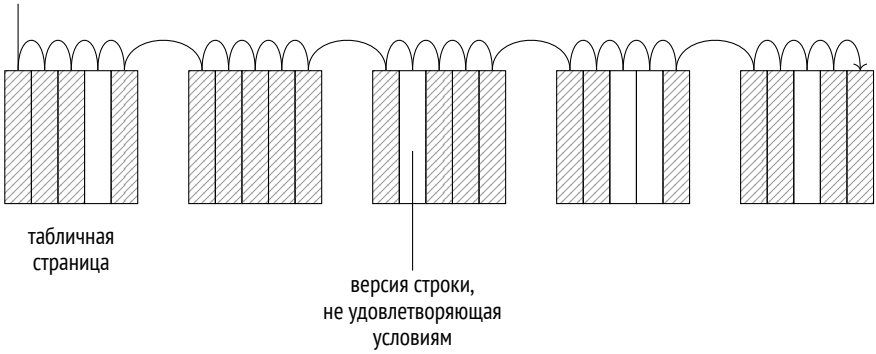
Из-за недостаточно проработанного интерфейса табличных методов доступа установка OrioleDB требует определенных изменений в ядре PostgreSQL.

Citus Columnar реализует классическое колоночное хранение данных со сжатием[1]. Этот метод доступа предназначен для аналитической нагрузки и хранилищ данных, он не поддерживает обновление и имеет другие ограничения.

18.2. Последовательное сканирование

Движок хранения отвечает за физическую организацию табличных данных и предоставляет метод доступа к ним — последовательное сканирование, при котором полностью читается файл (или файлы) основного слоя таблицы. На каждой прочитанной странице проверяется видимость каждой версии строки; версии, не удовлетворяющие условиям запроса, отбрасываются.

с. 97



Чтение происходит через буферный кеш; чтобы большие таблицы не вытесняли полезные данные, для последовательного сканирования используется буферное кольцо небольшого размера. При этом другие процессы, одновременно сканирующие ту же таблицу, присоединяются к кольцу и тем самым экономят операции дисковых чтений. Поэтому в общем случае сканирование может стартовать не с начала файла.

с. 198

[1] github.com/citusdata/citus/tree/main/src/backend/columnar.

Последовательное сканирование — самый эффективный способ прочитать всю таблицу или значительную ее часть. Иными словами, последовательное сканирование хорошо работает при низкой селективности. (При высокой селективности, когда из всей таблицы нужна только небольшая часть строк, более предпочтительным будет использование индекса.) *с. 394*

Оценка стоимости

В плане выполнения запроса последовательное сканирование представляется узлом Seq Scan:

```
=> EXPLAIN SELECT * FROM flights;
                          QUERY PLAN
---------------------------------------------------------------
 Seq Scan on flights  (cost=0.00..4772.67 rows=214867 width=63)
(1 row)
```

Оценка количества строк (rows) является частью базовой статистики:

```
=> SELECT reltuples FROM pg_class WHERE relname = 'flights';
 reltuples
-----------
    214867
(1 row)
```

В оценке стоимости оптимизатор учитывает две составляющие: дисковый ввод-вывод и ресурсы процессора[1].

Стоимость ввода-вывода рассчитывается как произведение числа страниц в таблице на стоимость чтения одной страницы, *при условии что страницы читаются последовательно*. Когда буферный менеджер запрашивает страницу у операционной системы, физически с диска за один раз читается больший объем данных, так что с высокой вероятностью несколько следующих страниц уже окажутся в кеше ОС. За счет этого стоимость последовательного чтения одной страницы (которая для

[1] backend/optimizer/path/costsize.c, функция cost_seqscan.

1 планировщика определяется значением параметра *seq_page_cost*) полу-
4 чается меньше, чем стоимость при случайном доступе (определяемая значением параметра *random_page_cost*).

Значения по умолчанию подходят для HDD-дисков; для накопителей SSD имеет смысл существенно уменьшить *random_page_cost* (значение *seq_page_cost*, как правило, не трогают, оставляя единицу в качестве опорного значения). Поскольку соотношение зависит от характеристик оборудования, параметры обычно задают на уровне табличных пространств (`ALTER TABLESPACE ... SET`).

```
=> SELECT relpages,
  current_setting('seq_page_cost') AS seq_page_cost,
  relpages * current_setting('seq_page_cost')::real AS total
FROM pg_class WHERE relname = 'flights';
 relpages | seq_page_cost | total
----------+---------------+-------
     2624 | 1             |  2624
(1 row)
```

с. 170 Приведенная формула отчетливо показывает последствия раздувания таблиц из-за несвоевременной очистки: чем больший объем занимает основной слой таблицы, тем больше страниц придется сканировать, независимо от количества актуальных версий строк в них.

Оценка ресурсов процессора учитывает стоимость обработки каждой версии строки (которая определяется для планировщика значением параметра
0.01 *cpu_tuple_cost*):

```
=> SELECT reltuples,
  current_setting('cpu_tuple_cost') AS cpu_tuple_cost,
  reltuples * current_setting('cpu_tuple_cost')::real AS total
FROM pg_class WHERE relname = 'flights';
 reltuples | cpu_tuple_cost |  total
-----------+----------------+---------
    214867 | 0.01           | 2148.67
(1 row)
```

Сумма двух приведенных оценок и составляет полную стоимость плана. Начальная стоимость равна нулю, поскольку последовательное сканирование не требует выполнения подготовительных действий.

Если на сканируемую таблицу наложены условия, они отображаются в плане запроса под узлом Seq Scan в секции Filter. Оценка числа строк будет учитывать селективность этих условий, а оценка стоимости — затраты на их вычисления. с. 346

Команда EXPLAIN ANALYZE выведет и реально полученное количество строк, и количество строк, отфильтрованных условиями:

```
=> EXPLAIN (analyze, timing off, summary off, buffers off)
SELECT * FROM flights
WHERE status = 'Scheduled';
                    QUERY PLAN
-----------------------------------------------
 Seq Scan on flights
   (cost=0.00..5309.84 rows=15383 width=63)
   (actual rows=15383.00 loops=1)
   Filter: ((status)::text = 'Scheduled'::text)
   Rows Removed by Filter: 199484
(5 rows)
```

Рассмотрим чуть более сложный пример плана выполнения с агрегацией:

```
=> EXPLAIN SELECT count(*) FROM seats;
                           QUERY PLAN
-----------------------------------------------------------------
 Aggregate  (cost=28.74..28.75 rows=1 width=8)
   -> Seq Scan on seats  (cost=0.00..25.39 rows=1339 width=0)
(2 rows)
```

План состоит из двух узлов: верхний узел Aggregate, в котором происходит вычисление функции count, получает данные от нижнего узла Seq Scan, выполняющего сканирование таблицы.

Начальная стоимость узла Aggregate включает саму агрегацию: невозможно выдать первую (и единственную) строку результата, не получив все строки от нижестоящего узла. Оценка вычисляется исходя из стоимости выполнения условной операции *cpu_operator_cost* над каждой входной строкой[1]: 0.0025

[1] backend/optimizer/path/costsize.c, функция cost_agg.

```
=> SELECT reltuples,
  current_setting('cpu_operator_cost') AS cpu_operator_cost,
  round((
    reltuples * current_setting('cpu_operator_cost')::real
  )::numeric, 2) AS cpu_cost
FROM pg_class WHERE relname = 'seats';
 reltuples | cpu_operator_cost | cpu_cost
-----------+-------------------+----------
      1339 | 0.0025            |     3.35
(1 row)
```

Полученная оценка добавляется к полной стоимости узла Seq Scan.

Полная стоимость узла Aggregate дополнительно включает стоимость обра-
0.01 ботки одной строки результата *cpu_tuple_cost*:

```
=> WITH t(cpu_cost) AS (
  SELECT round((
    reltuples * current_setting('cpu_operator_cost')::real
  )::numeric, 2)
  FROM pg_class WHERE relname = 'seats'
)
SELECT 21.39 + t.cpu_cost AS startup_cost,
  round((
    21.39 + t.cpu_cost +
    1 * current_setting('cpu_tuple_cost')::real
  )::numeric, 2) AS total_cost
FROM t;
 startup_cost | total_cost
--------------+------------
        24.74 |      24.75
(1 row)
```

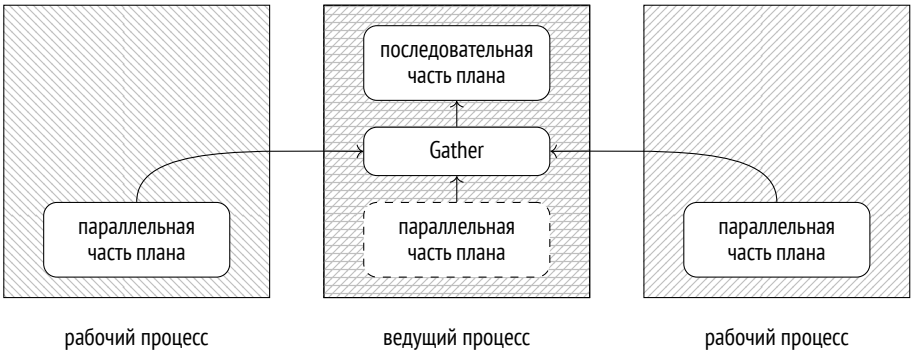
Зависимости между оценками стоимостей в этом примере можно представить себе следующим образом:

```
                  QUERY PLAN
-----------------------------------------------
 Aggregate
   (cost=28.74..28.75 rows=1 width=8)
   -> Seq Scan on seats
        (cost=0.00..25.39 rows=1339 width=0)
(4 rows)
```

18.3. Параллельные планы выполнения

PostgreSQL поддерживает параллельное выполнение запросов [1]. Идея состоит в том, что ведущий процесс, выполняющий запрос, порождает (с помощью postmaster) несколько рабочих процессов, которые одновременно выполняют одну и ту же параллельную часть плана. Результаты этого выполнения передаются ведущему процессу, который собирает их в узле Gather[2]. В свободное от приема данных время ведущий процесс также может выполнять параллельную часть плана.

При необходимости можно отключить ведущий процесс от выполнения параллельной части плана с помощью параметра *parallel_leader_participation*. on



Разумеется, запуск процессов и пересылка данных требуют определенных ресурсов, поэтому далеко не каждый запрос имеет смысл выполнять параллельно.

Кроме того, даже при параллельном выполнении не над всеми шагами плана запроса можно работать одновременно. Часть операций может выполняться ведущим процессом в одиночку, последовательно.

> В PostgreSQL не реализован другой теоретически возможный режим распараллеливания, при котором несколько процессов составляют конвейер для обработки данных (грубо говоря, отдельные узлы плана выполняются отдельными процессами). Разработчики PostgreSQL сочли такой режим неэффективным.

[1] postgrespro.ru/docs/postgresql/18/parallel-query; backend/access/transam/README.parallel.
[2] backend/executor/nodeGather.c.

18.4. Параллельное последовательное сканирование

Примером узла, предназначенного для параллельного выполнения, является Parallel Seq Scan — «параллельное последовательное сканирование».

Название звучит несколько противоречиво (все-таки параллельное или последовательное?), но тем не менее отражает суть операции. С точки зрения обращений к файлу страницы таблицы читаются последовательно, в том же порядке, в котором они читались бы при обычном последовательном сканировании. Однако чтение выполняется несколькими параллельно работающими процессами. Процессы синхронизируются между собой с помощью специально отведенного участка общей памяти, чтобы не прочитать одну и ту же страницу дважды.

v. 14 Тонкий момент состоит в том, что вместо общей картины последовательного сканирования операционная система видит несколько процессов, выполняющих случайное чтение. Из-за этого предвыборка данных, обычно ускоряющая последовательное чтение, работает плохо. Поэтому каждому процессу выделяется для чтения не одна, а несколько страниц, идущих подряд[1].

Сама по себе операция параллельного сканирования не имеет большого смысла, поскольку к обычным затратам на чтение страниц добавляются накладные расходы на пересылку данных от процесса к процессу. Но если рабочие процессы выполняют какую-то обработку прочитанных строк (например, агрегацию), то суммарное время выполнения запроса может оказаться существенно меньшим.

Оценка стоимости

Рассмотрим простой запрос с агрегацией над большой таблицей. План выполнения будет использовать параллелизм:

[1] backend/access/heap/heapam.c, функции table_block_parallelscan_startblock_init и table_block_parallelscan_nextpage.

```
=> EXPLAIN SELECT count(*) FROM bookings;
                                QUERY PLAN
-------------------------------------------------------------------
 Finalize Aggregate  (cost=25483.58..25483.59 rows=1 width=8)
   -> Gather  (cost=25483.36..25483.57 rows=2 width=8)
        Workers Planned: 2
        ->  Partial Aggregate
              (cost=24483.36..24483.37 rows=1 width=8)
              -> Parallel Seq Scan on bookings
                    (cost=0.00..22284.29 rows=879629 width=0)
(7 rows)
```

Узлы ниже Gather составляют параллельную часть плана. Она выполняется в каждом из рабочих процессов (которых в данном случае запланировано две штуки) и, возможно, в ведущем процессе (если это не отключено параметром *parallel_leader_participation*). Сам узел Gather и узлы выше него составляют последовательную часть плана и выполняются только в ведущем процессе.

Узел Parallel Seq Scan представляет сканирование таблицы в параллельном режиме. В поле rows показана оценка числа строк, которые *в среднем* выдаст *один* процесс. Всего над выполнением должно трудиться три процесса (ведущий и два рабочих), но ведущий процесс будет задействован не полностью: его вклад уменьшается с ростом числа рабочих процессов[1]. В данном случае используется коэффициент 2,4.

```
=> SELECT reltuples::numeric, round(reltuples / 2.4) AS per_process
FROM pg_class WHERE relname = 'bookings';
 reltuples | per_process
-----------+-------------
   2111110 |      879629
(1 row)
```

Стоимость узла Parallel Seq Scan оценивается почти так же, как и стоимость последовательного сканирования. Выигрыш получается за счет того, что каждый из процессов обрабатывает меньшее количество строк; составляющая ввода-вывода при этом учитывается полностью, поскольку таблицу все равно придется прочитать целиком, страница за страницей:

[1] backend/optimizer/path/costsize.c, функция get_parallel_divisor.

```
=> SELECT round((
  relpages        * current_setting('seq_page_cost')::real +
  reltuples / 2.4 * current_setting('cpu_tuple_cost')::real
)::numeric, 2)
FROM pg_class WHERE relname = 'bookings';
  round
----------
 22284.29
(1 row)
```

Следующий узел — Partial Aggregate — выполняет агрегацию данных, полученных рабочим процессом, то есть в данном случае подсчитывает количество строк.

Оценка стоимости агрегации выполняется уже известным образом и добавляется к оценке сканирования таблицы:

```
=> WITH t(startup_cost) AS (
  SELECT 22284.29 + round((
    reltuples / 2.4 * current_setting('cpu_operator_cost')::real
  )::numeric, 2)
  FROM pg_class WHERE relname = 'bookings'
)
SELECT startup_cost,
  startup_cost + round((
    1 * current_setting('cpu_tuple_cost')::real
  )::numeric, 2) AS total_cost
FROM t;
 startup_cost | total_cost
--------------+------------
     24483.36 |   24483.37
(1 row)
```

Следующий узел — Gather — выполняется ведущим процессом. Этот узел отвечает за запуск рабочих процессов и получение от них данных.

Оценка стоимости запуска процессов (независимо от их количества) опре-
1000 деляется для планировщика значением параметра *parallel_setup_cost*, а стои-
мость пересылки каждой строки данных между процессами — значением
0.1 *parallel_tuple_cost*.

В данном случае преобладает начальная стоимость (запуск процессов), и это значение добавляется к начальной стоимости узла Partial Aggregate. Полная

стоимость учитывает пересылку двух строк, и это значение складывается с полной стоимостью узла Partial Aggregate[1]:

```
=> SELECT
  24483.36 + round(
    current_setting('parallel_setup_cost')::numeric,
  2) AS setup_cost,
  24483.37 + round(
    current_setting('parallel_setup_cost')::numeric +
    2 * current_setting('parallel_tuple_cost')::numeric,
  2) AS total_cost;
 setup_cost | total_cost
------------+------------
   25483.36 |   25483.57
(1 row)
```

Последний узел — Finalize Aggregate — агрегирует частичные суммы, полученные узлом Gather от параллельных процессов.

Начальная стоимость этой окончательной агрегации учитывает подсчет трех строк; полученное значение складывается с полной стоимостью узла Gather (поскольку для вычисления результата нужны все строки). К полной стоимости добавляется стоимость выдачи одной строки результата.

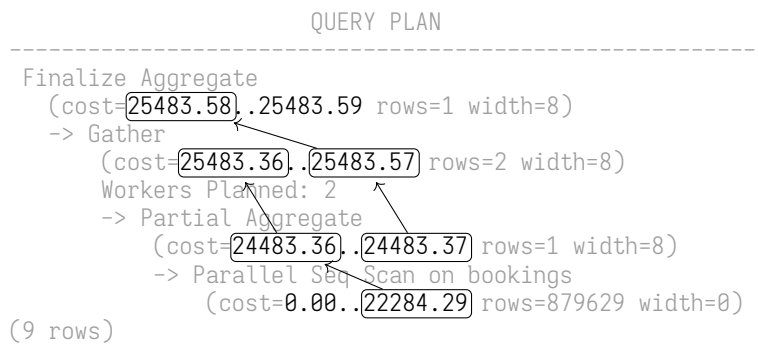
```
=> WITH t(startup_cost) AS (
  SELECT 25483.57 + round((
    3 * current_setting('cpu_operator_cost')::real
  )::numeric, 2)
  FROM pg_class WHERE relname = 'bookings'
)
SELECT startup_cost,
  startup_cost + round((
    1 * current_setting('cpu_tuple_cost')::real
  )::numeric, 2) AS total_cost
FROM t;
 startup_cost | total_cost
--------------+------------
     25483.58 |   25483.59
(1 row)
```

Зависимости между оценками стоимостей определяются тем, накапливает ли узел данные перед выдачей результатов вышестоящему узлу. Агрегация не может возвращать результаты до тех пор, пока не получит все входные

[1] backend/optimizer/path/costsize.c, функция cost_gather.

строки. Поэтому в основу *начальной* стоимости агрегации ложится *полная* стоимость нижестоящего узла. А узел Gather выдает строки наверх сразу, как только получает их снизу. Поэтому начальная стоимость этой операции зависит от начальной стоимости нижестоящего узла, а полная — от полной.

Схематически это можно представить себе следующим образом:



18.5. Ограничения параллельного выполнения

Количество рабочих процессов

Количество процессов ограничивают три параметра, образующих иерархию. Максимальное число одновременно выполняющихся рабочих процес-
8 сов определяется значением параметра *max_worker_processes*.

Но механизм рабочих процессов используется не только для параллельного выполнения запросов. Например, фоновые рабочие процессы задействованы в логической репликации, ими могут пользоваться расширения. Количество рабочих процессов, занимающихся именно параллельными плана-
8 ми, ограничено значением параметра *max_parallel_workers*.

2 Из этого числа не более *max_parallel_workers_per_gather* процессов могут обслуживать один ведущий процесс.

На выбор значений для этих параметров влияют:

- возможности аппаратуры — система должна располагать свободными ядрами, не занятыми другими задачами;
- наличие в базе данных больших таблиц;
- нагрузка — должны выполняться запросы, потенциально выигрывающие от параллельного выполнения.

В большинстве случаев таким критериям удовлетворяют OLAP-, а не OLTP-системы.

Планировщик вообще не будет рассматривать параллельное сканирование, если по его оценке размер прочитанных из таблицы данных не превысит значение *min_parallel_table_scan_size*. 8MB

Если число процессов не указано для таблицы явно в параметре хранения *parallel_workers*, оно вычисляется по формуле

$$1 + \left\lfloor \log_3 \left(\frac{\text{размер таблицы}}{min_parallel_table_scan_size} \right) \right\rfloor .$$

Она означает, что каждое увеличение таблицы в три раза будет добавлять еще один параллельный процесс. Например, значения параметров по умолчанию дают следующие цифры:

Таблица, Мбайт	Количество процессов
8	1
24	2
72	3
216	4
648	5
1944	6

В любом случае число процессов не может превышать значение параметра *max_parallel_workers_per_gather*.

Если запросить информацию из небольшой таблицы размером 19 Мбайт, будет запланирован и запущен один рабочий процесс:

```
=> EXPLAIN (analyze, costs off, timing off, summary off, buffers off)
SELECT count(*) FROM flights;
                              QUERY PLAN
-----------------------------------------------------------------------
 Finalize Aggregate (actual rows=1.00 loops=1)
   -> Gather (actual rows=2.00 loops=1)
        Workers Planned: 1
        Workers Launched: 1
        -> Partial Aggregate (actual rows=1.00 loops=2)
            -> Parallel Seq Scan on flights (actual rows=107433.50...
(6 rows)
```

При запросе данных из таблицы размером 105 Мбайт будет запланировано
2 только два процесса из-за ограничения *max_parallel_workers_per_gather*:

```
=> EXPLAIN (analyze, costs off, timing off, summary off, buffers off)
SELECT count(*) FROM bookings;
                              QUERY PLAN
-----------------------------------------------------------------------
 Finalize Aggregate (actual rows=1.00 loops=1)
   -> Gather (actual rows=3.00 loops=1)
        Workers Planned: 2
        Workers Launched: 2
        -> Partial Aggregate (actual rows=1.00 loops=3)
            -> Parallel Seq Scan on bookings (actual rows=703703.3...
(6 rows)
```

Сняв ограничение, получим расчетные три процесса:

```
=> ALTER SYSTEM SET max_parallel_workers_per_gather = 4;
=> SELECT pg_reload_conf();
=> EXPLAIN (analyze, costs off, timing off, summary off, buffers off)
SELECT count(*) FROM bookings;
                              QUERY PLAN
-----------------------------------------------------------------------
 Finalize Aggregate (actual rows=1.00 loops=1)
   -> Gather (actual rows=4.00 loops=1)
        Workers Planned: 3
        Workers Launched: 3
        -> Partial Aggregate (actual rows=1.00 loops=4)
            -> Parallel Seq Scan on bookings (actual rows=527777.5...
(6 rows)
```

Если при выполнении запроса число свободных слотов окажется меньше запланированного количества процессов, будет запущено только доступное количество рабочих процессов.

Ограничим общее количество параллельных процессов пятью и выполним два запроса одновременно:

```
=> ALTER SYSTEM SET max_parallel_workers = 5;
=> SELECT pg_reload_conf();
=> EXPLAIN (analyze, costs off, timing off, summary off, buffers off)
SELECT count(*) FROM bookings;
```

```
   => EXPLAIN (analyze, costs off, timing off, summary off, buffers off)
   SELECT count(*) FROM bookings;
                              QUERY PLAN
   ------------------------------------------------------------------
    Finalize Aggregate (actual rows=1.00 loops=1)
      -> Gather (actual rows=3.00 loops=1)
          Workers Planned: 3
          Workers Launched: 2
          -> Partial Aggregate (actual rows=1.00 loops=3)
              -> Parallel Seq Scan on bookings (actual rows=7037...
   (6 rows)
```

```
                             QUERY PLAN
------------------------------------------------------------------------
 Finalize Aggregate (actual rows=1.00 loops=1)
   -> Gather (actual rows=4.00 loops=1)
       Workers Planned: 3
       Workers Launched: 3
       -> Partial Aggregate (actual rows=1.00 loops=4)
           -> Parallel Seq Scan on bookings (actual rows=527777.5...
(6 rows)
```

Хотя в каждом случае было запланировано три рабочих процесса, одному запросу досталось только два свободных слота.

Восстановим значения параметров по умолчанию:

```
=> ALTER SYSTEM RESET ALL;
=> SELECT pg_reload_conf();
```

Нераспараллеливаемые запросы

Не каждый запрос может выполняться в параллельном режиме[1]. Не могут распараллеливаться:

- Запросы, изменяющие или блокирующие данные (UPDATE, DELETE, SELECT FOR UPDATE и подобные им).

  Это не касается запросов, которые используются в командах:

  - CREATE TABLE AS, SELECT INTO, CREATE MATERIALIZED VIEW;
  - REFRESH MATERIALIZED VIEW. v. 14

  Но вставка строк и в этих случаях выполняется последовательно.

- Запросы, выполнение которых может быть приостановлено. Это относится к запросам в курсорах, в том числе в циклах FOR PL/pgSQL.

- Запросы, содержащие вызовы *небезопасных* функций, помеченных как PARALLEL UNSAFE. К таковым по умолчанию относятся все пользовательские функции и небольшая часть стандартных. Список небезопасных функций можно получить из системного каталога:

```
SELECT * FROM pg_proc WHERE proparallel = 'u';
```

- Запросы внутри функций, когда эти функции вызываются из распараллеленного запроса (чтобы не допустить рекурсивного разрастания количества рабочих процессов).

Часть этих ограничений может быть снята в следующих версиях PostgreSQL. Так, например, появилась возможность распараллеливания запросов на уровне изоляции Serializable. v. 12

Запрос может не выполняться в параллельном режиме по нескольким причинам:

- данный запрос в принципе не может быть распараллелен ни при каких обстоятельствах;

[1] postgrespro.ru/docs/postgresql/18/when-can-parallel-query-be-used.

- параллельный план запрещен значениями конфигурационных параметров (в том числе из-за ограничения на размер таблиц);
- параллельный план имеет более высокую стоимость по сравнению с последовательным.

Чтобы проверить, может ли запрос быть распараллелен в принципе, можно на время включить параметр *debug_parallel_query*. При этом планировщик будет строить параллельные планы во всех случаях, когда это возможно:

v. 16
off

```
=> EXPLAIN SELECT * FROM flights;
                            QUERY PLAN
------------------------------------------------------------------
 Seq Scan on flights  (cost=0.00..4772.67 rows=214867 width=63)
(1 row)

=> SET debug_parallel_query = on;

=> EXPLAIN SELECT * FROM flights;
                               QUERY PLAN
------------------------------------------------------------------------
 Gather  (cost=1000.00..27259.37 rows=214867 width=63)
   Workers Planned: 1
   Single Copy: true
   -> Seq Scan on flights  (cost=0.00..4772.67 rows=214867 width=63)
(4 rows)
```

Теперь мы понимаем, что планировщик рассматривал параллельный план, но выбрал более эффективное в данном случае последовательное сканирование.

Ограниченно распараллеливаемые запросы

Чем большую часть плана удается выполнить параллельно, тем больший возможен эффект. Однако есть ряд операций, которые в целом не препятствуют распараллеливанию, но сами могут выполняться только последовательно в ведущем процессе[1]. Иными словами, они не могут появиться в дереве плана ниже узла Gather.

[1] postgrespro.ru/docs/postgresql/18/parallel-safety.

Общие табличные выражения. Не распараллеливается чтение результата общего табличного выражения[1] (узел плана CTE Scan):

```
=> EXPLAIN (costs off)
WITH t AS MATERIALIZED (
  SELECT * FROM flights
)
SELECT count(*) FROM t;
         QUERY PLAN
----------------------------
 Aggregate
   CTE t
     -> Seq Scan on flights
   -> CTE Scan on t
(4 rows)
```

v. 12 Если общее табличное выражение не материализуется, то план не содержит узла CTE Scan, и это ограничение не действует.

При этом само общее табличное выражение вполне может вычисляться в параллельном режиме, если это выгодно:

```
=> EXPLAIN (costs off)
WITH t AS MATERIALIZED (
  SELECT count(*) FROM flights
)
SELECT * FROM t;
                    QUERY PLAN
-----------------------------------------------
 CTE Scan on t
   CTE t
     -> Finalize Aggregate
         -> Gather
             Workers Planned: 1
             -> Partial Aggregate
                 -> Parallel Seq Scan on flights
(7 rows)
```

Коррелированные подзапросы. Некоррелированные подзапросы, вычисляv. 17 емые один раз и представленные в плане узлом InitPlan, могут участвовать в параллельном плане:

[1] backend/optimizer/plan/subselect.c.

```
=> EXPLAIN (costs off)
SELECT * FROM flights f1
WHERE f1.aircraft_code = (
  SELECT '733' -- InitPlan 1
)
UNION ALL
SELECT * FROM flights f2
WHERE f2.aircraft_code = (
  SELECT '320' -- InitPlan 2
);
                          QUERY PLAN
---------------------------------------------------------------
 Gather
   Workers Planned: 2
   -> Parallel Append
       -> Parallel Seq Scan on flights f2
           Filter: ((aircraft_code)::text = (InitPlan 2).col1)
           InitPlan 2
             -> Result
       -> Parallel Seq Scan on flights f1
           Filter: ((aircraft_code)::text = (InitPlan 1).col1)
           InitPlan 1
             -> Result
(11 rows)
```

Узел Parallel Append реализует здесь параллельное выполнение команды UNION ALL. Рабочие процессы разбирают себе по подзапросу (если процессов больше, чем подзапросов, несколько могут работать над одной задачей). Когда все данные от подзапроса уже получены, освободившийся процесс подключается к другому подзапросу, пока все данные не будут объединены.

А вот коррелированные подзапросы (узел плана SubPlan) не участвуют в параллельных планах:

```
=> EXPLAIN (costs off)
SELECT * FROM flights f1
WHERE f1.aircraft_code = (
  SELECT f1.aircraft_code -- SubPlan 1
)
UNION ALL
SELECT * FROM flights f2
WHERE f2.aircraft_code = (
  SELECT '320' -- InitPlan 2
);
```

```
                          QUERY PLAN
---------------------------------------------------------------
 Append
   -> Seq Scan on flights f1
        Filter: (aircraft_code = (SubPlan 1))
        SubPlan 1
          -> Result
   -> Gather
        Workers Planned: 1
        InitPlan 2
          -> Result
        -> Parallel Seq Scan on flights f2
            Filter: ((aircraft_code)::text = (InitPlan 2).col1)
(11 rows)
```

Вместо Parallel Append в плане появился узел Append, в котором операция UNION ALL выполняется последовательно, а параллельная часть плана охватывает теперь только второй подзапрос, в котором остался узел InitPlan.

Временные таблицы. Временные таблицы не могут сканироваться параллельно, поскольку доступны исключительно создавшему их процессу. Работа со страницами временных таблиц ведется в локальном буферном кеше. Чтобы разрешить обращаться к нему нескольким процессам, потребовалось бы ввести блокировки, как в разделяемом кеше, а это уменьшило бы остальные преимущества.

с. 209

с. 305

```
=> CREATE TEMPORARY TABLE flights_tmp
AS SELECT * FROM flights;
=> EXPLAIN (costs off)
SELECT count(*) FROM flights_tmp;
          QUERY PLAN
------------------------------
 Aggregate
   -> Seq Scan on flights_tmp
(2 rows)
```

Ограниченно распараллеливаемые функции. Вызовы функций, помеченных как PARALLEL RESTRICTED, могут выполняться только в последовательной части плана.

Список таких функций можно получить из системного каталога запросом

```
SELECT * FROM pg_proc WHERE proparallel = 'r';
```

Помечать собственные функции как PARALLEL RESTRICTED (и тем более как PARALLEL SAFE) нужно с большой осторожностью, внимательно изучив имеющиеся ограничения[1].

[1] postgrespro.ru/docs/postgresql/18/parallel-safety#PARALLEL-LABELING.

19

Индексные методы доступа

19.1. Индексы и расширяемость

Индексы — объекты базы данных, предназначенные в основном для ускорения доступа к данным. Это вспомогательные структуры: любой индекс можно удалить и восстановить заново по информации в таблице. Но, кроме ускорения, индексы служат и для поддержки некоторых ограничений целостности.

В ядро встроены шесть индексных методов доступа (типов индексов):

```
\dA   => SELECT amname FROM pg_am WHERE amtype = 'i';
  amname
--------
 btree
 hash
 gist
 gin
 spgist
 brin
(6 rows)
```

Следование идее расширяемости предполагает возможность добавлять и новые методы доступа без изменения ядра. Один такой пример (метод `bloom`) включен в стандартный набор расширений.

Несмотря на все различия между типами индексов, в конечном счете любой из них устанавливает соответствие между ключом (например, значением
с. 428 проиндексированного столбца) и версиями строк таблицы, в которых этот ключ встречается. В качестве ссылки используется шестибайтный *идентификатор версии строки* (tuple id, tid). Зная ключ или некоторую информацию

о нем, можно быстро прочитать те версии строк, в которых может находиться нужная информация, не просматривая всю таблицу полностью.

Чтобы новый метод доступа можно было добавить как расширение, выделен общий *механизм индексирования*. Его основной задачей является получение идентификаторов версий строк от метода доступа и работа с ними:

- чтение данных из соответствующих версий строк таблицы;
- проверка видимости версий строк в снимке данных; *с. 97*
- перепроверка условий, если метод не гарантирует их выполнения.

Механизм индексирования участвует в исполнении планов, построенных на этапе оптимизации. Оптимизатор, перебирая и оценивая различные пути выполнения запроса, должен знать свойства всех потенциально применимых методов доступа: может ли метод отдавать данные сразу в нужном порядке или надо отдельно предусмотреть сортировку? может ли метод вернуть несколько первых значений или только всю выборку сразу? и так далее.

Информация о методе доступа нужна не только оптимизатору. При создании индекса надо решить: допускает ли метод создание составного индекса? может ли данный индекс обеспечить уникальность?

Механизм индексирования позволяет работать с самыми разными методами доступа; для этого метод доступа должен реализовать определенный интерфейс, сообщив о своих возможностях и особенностях.

В задачи метода доступа входят:

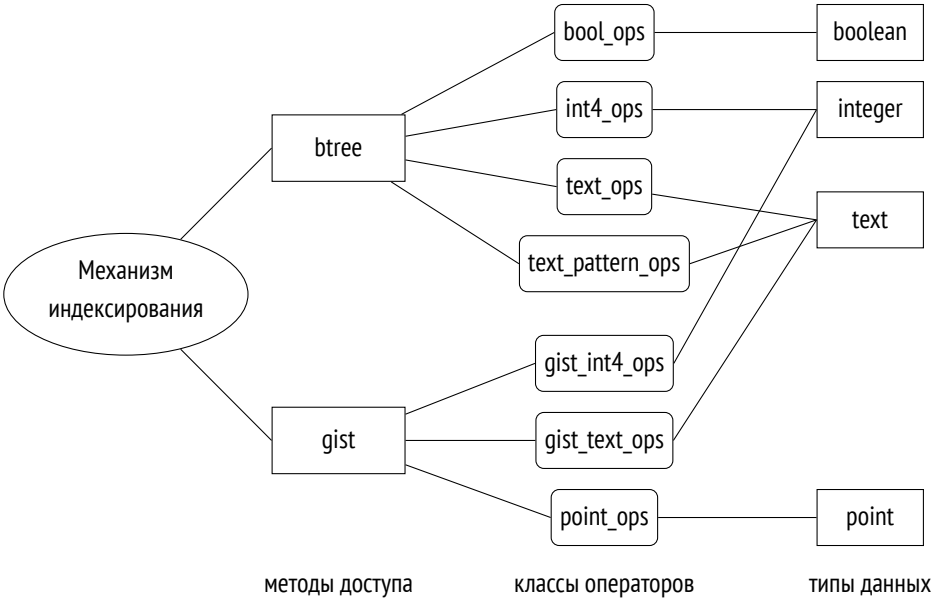
- реализация алгоритмов построения индекса, вставки и удаления строк;
- организация разбиения данных индекса по страницам (для работы с менеджером буферного кеша); *с. 183*
- реализация алгоритма очистки; *с. 125*
- установка блокировок для корректного конкурентного выполнения; *с. 305*
- формирование журнала предзаписи; *с. 211*
- поиск информации в индексе по ключу;
- оценка стоимости использования индекса.

Расширяемость проявляется и в том, что в систему могут добавляться новые типы данных, про которые методу доступа заранее ничего неизвестно. Поэтому методы доступа определяют собственные интерфейсы для подключения произвольных типов данных.

Чтобы значения определенного типа можно было использовать с определенным методом доступа, нужно реализовать этот интерфейс, предоставив операторы, для которых применим индекс и, возможно, некоторые вспомогательные *опорные* функции. Такой набор операторов и функций называется *классом операторов*.

Часть логики индексирования при этом находится в самом методе доступа, а часть оказывается вынесенной в класс операторов. Это довольно произвольное деление: если для B-деревьев вся логика «зашита» в методе доступа, то некоторые методы могут предоставлять лишь основной каркас, отдавая все существенные детали на откуп конкретным классам операторов. Во многих случаях для одного и того же типа данных существует несколько классов операторов с отличающимся поведением, среди которых пользователь может выбрать наиболее подходящий.

Небольшой фрагмент общей картины показан на рисунке:



19.2. Классы и семейства операторов

Класс операторов

Интерфейс метода доступа[1] реализуется *классом операторов*[2] — набором операторов и опорных функций, который используется методом доступа для работы с конкретным типом данных.

Классы операторов хранятся в системном каталоге в таблице pg_opclass. Полные данные для рисунка выше можно было бы получить запросом:

```
=> SELECT amname, opcname, opcintype::regtype                    \dAc
FROM pg_am am
  JOIN pg_opclass opc ON opcmethod = am.oid;
 amname |           opcname            |          opcintype
--------+------------------------------+-----------------------------
 btree  | array_ops                    | anyarray
 hash   | array_ops                    | anyarray
 btree  | bit_ops                      | bit
 btree  | bool_ops                     | boolean
 ...
 brin   | pg_lsn_minmax_multi_ops      | pg_lsn
 brin   | pg_lsn_bloom_ops             | pg_lsn
 brin   | box_inclusion_ops            | box
(177 rows)
```

В большинстве случаев про классы операторов не требуется ничего знать. Обычно мы просто создаем индекс, и при этом используется некоторый класс операторов по умолчанию.

Например, вот классы операторов для B-дерева и типа text. Один из имеющихся классов обязательно помечен для использования по умолчанию:

```
=> SELECT opcname, opcdefault
FROM pg_am am
  JOIN pg_opclass opc ON opcmethod = am.oid
WHERE amname = 'btree'
  AND opcintype = 'text'::regtype;
```

[1] postgrespro.ru/docs/postgresql/18/xindex.
[2] postgrespro.ru/docs/postgresql/18/indexes-opclass.

```
       opcname       | opcdefault
---------------------+------------
 text_ops            | t
 varchar_ops         | f
 text_pattern_ops    | f
 varchar_pattern_ops | f
(4 rows)
```

Привычная команда создания индекса выглядит так:

```
CREATE INDEX ON aircrafts_data(model, range);
```

Но это просто сокращение для развернутого варианта:

```
CREATE INDEX ON aircrafts_data
USING btree -- метод доступа по умолчанию
(
  model text_ops, -- класс операторов по умолчанию для text
  range int4_ops  -- класс операторов по умолчанию для integer
);
```

Если нужен индекс какого-то другого типа или необходимо нестандартное поведение, потребуется явно указать желаемый метод доступа или класс операторов.

Класс операторов, созданный для некоторого метода доступа и некоторого типа данных, должен содержать набор операторов, которые принимают параметры этого типа и имеют семантику, предусмотренную этим методом доступа.

Например, метод доступа `btree` определяет пять обязательных операторов сравнения. Любой класс операторов для него должен содержать все пять:

```
\dAo  => SELECT opcname, amopstrategy, amopopr::regoperator
      FROM pg_am am
        JOIN pg_opfamily opf ON opfmethod = am.oid
        JOIN pg_opclass opc ON opcfamily = opf.oid
        JOIN pg_amop amop ON amopfamily = opcfamily
      WHERE amname = 'btree'
        AND opcname IN ('text_ops', 'text_pattern_ops')
        AND amoplefttype = 'text'::regtype
        AND amoprighttype = 'text'::regtype
      ORDER BY opcname, amopstrategy;
```

```
     opcname      | amopstrategy |     amopopr
------------------+--------------+-----------------
 text_ops         |            1 | <(text,text)
 text_ops         |            2 | <=(text,text)
 text_ops         |            3 | =(text,text)
 text_ops         |            4 | >=(text,text)
 text_ops         |            5 | >(text,text)
 text_pattern_ops |            1 | ~<~(text,text)
 text_pattern_ops |            2 | ~<=~(text,text)
 text_pattern_ops |            3 | =(text,text)
 text_pattern_ops |            4 | ~>=~(text,text)
 text_pattern_ops |            5 | ~>~(text,text)
(10 rows)
```

Семантика оператора, подразумеваемая методом доступа, отражена в номере стратегии `amopstrategy`[1]. Например, для `btree` стратегия 1 означает «меньше», 2 — «меньше или равно» и так далее. Сами операторы при этом могут иметь произвольные названия.

В приведенном примере обычные операторы и операторы с тильдой отличаются тем, что последние не учитывают *правила сортировки* (collation)[2] и сравнивают строки побайтово. Тем не менее оба варианта реализуют одни и те же смысловые операции сравнения.

Класс операторов `text_pattern_ops` позволяет преодолеть ограничение на поддержку оператора `~~` (который соответствует конструкции `LIKE`). В базе данных с правилом сортировки, отличным от C, этот оператор не может использовать обычный индекс по текстовому полю:

```
=> SELECT datcollate FROM pg_database
WHERE datname = current_database();
 datcollate
-------------
 en_US.UTF-8
(1 row)
=> CREATE INDEX ON tickets(passenger_name);
=> EXPLAIN (costs off)
SELECT * FROM tickets WHERE passenger_name LIKE 'ELENA%';
```

[1] postgrespro.ru/docs/postgresql/18/xindex#XINDEX-STRATEGIES.

[2] postgrespro.ru/docs/postgresql/18/collation;
postgrespro.ru/docs/postgresql/18/indexes-collations.

```
                  QUERY PLAN
-----------------------------------------------
 Seq Scan on tickets
   Filter: (passenger_name ~~ 'ELENA%'::text)
(2 rows)
```

Другое дело — индекс с классом операторов `text_pattern_ops`:

```
=> CREATE INDEX tickets_passenger_name_pattern_idx
ON tickets(passenger_name text_pattern_ops);
=> EXPLAIN (costs off)
SELECT * FROM tickets WHERE passenger_name LIKE 'ELENA%';
                          QUERY PLAN
---------------------------------------------------------------
 Bitmap Heap Scan on tickets
   Filter: (passenger_name ~~ 'ELENA%'::text)
   -> Bitmap Index Scan on tickets_passenger_name_pattern_idx
        Index Cond: ((passenger_name ~>=~ 'ELENA'::text) AND
        (passenger_name ~<~ 'ELENB'::text))
(5 rows)
```

Обратите внимание, как изменилось выражение в условии Index Cond. Для поиска используется только префикс шаблона до символа **%**, а лишние совпадения отсеиваются при перепроверке условием Filter. Класс операторов для метода доступа `btree` не содержит оператора сравнения по шаблону, и единственный способ использовать B-дерево — переписать условие с помощью операторов сравнения. Операторы из класса `text_pattern_ops` действуют без учета правил сортировки, что и дает возможность заменить условие на эквивалентное[1].

Индекс может использоваться для ускорения доступа по условию, если

1) это условие имеет вид «*индексированный-столбец оператор выражение*» (если указан коммутирующий оператор[2], то допускается и «*выражение оператор индексированный-столбец*»)[3]

2) и при этом *оператор* входит в класс операторов, указанный для *индексированного-столбца* при создании индекса.

[1] backend/utils/adt/like_support.c.
[2] postgrespro.ru/docs/postgresql/18/xoper-optimization#id-1.8.3.18.6.
[3] backend/optimizer/path/indxpath.c, функция match_clause_to_indexcol.

Например, такой запрос может использовать индекс:

```
=> EXPLAIN (costs off)
SELECT * FROM tickets WHERE 'ELENA BELOVA' = passenger_name;
                        QUERY PLAN
-----------------------------------------------------------
 Index Scan using tickets_passenger_name_idx on tickets
   Index Cond: (passenger_name = 'ELENA BELOVA'::text)
(2 rows)
```

Обратите внимание на то, как переставлены аргументы в условии Index Cond: на этапе выполнения индексированное поле должно быть левым аргументом оператора. При перестановке оператор заменяется на коммутирующий; в данном случае это тот же самый оператор, поскольку отношение равенства коммутативно.

А следующий запрос в принципе не может использовать обычный индекс, поскольку вместо имени столбца в условии стоит вызов функции:

```
=> EXPLAIN (costs off)
SELECT * FROM tickets WHERE initcap(passenger_name) = 'Elena Belova';
                          QUERY PLAN
--------------------------------------------------------------
 Seq Scan on tickets
   Filter: (initcap(passenger_name) = 'Elena Belova'::text)
(2 rows)
```

В таком случае можно применить *индекс по выражению*[1], указав при его создании не столбец, а произвольное выражение:

```
=> CREATE INDEX ON tickets( (initcap(passenger_name)) );
=> EXPLAIN (costs off)
SELECT * FROM tickets WHERE initcap(passenger_name) = 'Elena Belova';
                             QUERY PLAN
--------------------------------------------------------------------
 Bitmap Heap Scan on tickets
   Recheck Cond: (initcap(passenger_name) = 'Elena Belova'::text)
   -> Bitmap Index Scan on tickets_initcap_idx
       Index Cond: (initcap(passenger_name) = 'Elena Belova'::text)
(4 rows)
```

[1] postgrespro.ru/docs/postgresql/18/indexes-expressional.

Выражение может зависеть только от значений полей в табличной строке и не должно зависеть ни от состояния других данных в базе, ни от настроек (например, от локали). Иными словами, если выражение содержит вызовы функций, то эти функции обязаны иметь и *соблюдать* категорию изменчивости IMMUTABLE[1]. В противном случае результат запроса, выполненного с индексным доступом, может отличаться от результата того же запроса, выполненного полным сканированием таблицы.

Кроме собственно операторов, класс операторов может предоставлять *опорные функции*[2], необходимые методу. Например, метод доступа btree определяет пять опорных функций[3], из которых только первая (сравнивающая два значения) является необходимой, а остальные могут отсутствовать:

```
\dAp  => SELECT amprocnum, amproc::regproc
      FROM pg_am am
        JOIN pg_opfamily opf ON opfmethod = am.oid
        JOIN pg_opclass opc ON opcfamily = opf.oid
        JOIN pg_amproc amproc ON amprocfamily = opcfamily
      WHERE amname = 'btree'
        AND opcname = 'text_ops'
        AND amproclefttype = 'text'::regtype
        AND amprocrighttype = 'text'::regtype
      ORDER BY amprocnum;
       amprocnum |       amproc
      -----------+--------------------
               1 | bttextcmp
               2 | bttextsortsupport
               4 | btvarstrequalimage
      (3 rows)
```

Семейство операторов

Класс операторов всегда входит в какое-либо *семейство операторов*[4] (таблица pg_opfamily системного каталога). При этом в одно общее семейство могут входить несколько классов, если они работают одинаковым образом с похожими типами данных.

[1] postgrespro.ru/docs/postgresql/18/xfunc-volatility.
[2] postgrespro.ru/docs/postgresql/18/xindex#XINDEX-SUPPORT.
[3] postgrespro.ru/docs/postgresql/18/btree-support-funcs.
[4] postgrespro.ru/docs/postgresql/18/xindex#XINDEX-OPFAMILY.

Например, семейство integer_ops включает несколько классов для разных по размеру, но одинаковых по смыслу числовых типов:

```
=> SELECT opcname, opcintype::regtype                                    \dAf
FROM pg_am am
  JOIN pg_opfamily opf ON opfmethod = am.oid
  JOIN pg_opclass opc ON opcfamily = opf.oid
WHERE amname = 'btree'
  AND opfname = 'integer_ops';
 opcname  | opcintype
----------+-----------
 int2_ops | smallint
 int4_ops | integer
 int8_ops | bigint
(3 rows)
```

А в семейство datetime_ops входят классы операторов для работы с датами:

```
=> SELECT opcname, opcintype::regtype
FROM pg_am am
  JOIN pg_opfamily opf ON opfmethod = am.oid
  JOIN pg_opclass opc ON opcfamily = opf.oid
WHERE amname = 'btree'
  AND opfname = 'datetime_ops';
     opcname     |          opcintype
-----------------+-----------------------------
 date_ops        | date
 timestamptz_ops | timestamp with time zone
 timestamp_ops   | timestamp without time zone
(3 rows)
```

Каждый класс операторов работает с каким-то одним типом. Семейство же включает и операторы, принимающие разные типы данных:

```
=> SELECT opcname, amopopr::regoperator
FROM pg_am am
  JOIN pg_opfamily opf ON opfmethod = am.oid
  JOIN pg_opclass opc ON opcfamily = opf.oid
  JOIN pg_amop amop ON amopfamily = opcfamily
WHERE amname = 'btree'
  AND opfname = 'integer_ops'
  AND amoplefttype = 'integer'::regtype
  AND amopstrategy = 1
ORDER BY opcname;
```

```
 opcname  |       amopopr
----------+---------------------
 int2_ops | <(integer,bigint)
 int2_ops | <(integer,smallint)
 int2_ops | <(integer,integer)
 int4_ops | <(integer,bigint)
 int4_ops | <(integer,smallint)
 int4_ops | <(integer,integer)
 int8_ops | <(integer,bigint)
 int8_ops | <(integer,smallint)
 int8_ops | <(integer,integer)
(9 rows)
```

За счет группировки операторов в семейство планировщик может использовать индекс для условий со значениями разных типов, не требуя явного приведения.

19.3. Интерфейс механизма индексирования

Как и для табличных методов доступа, столбец `amhandler` таблицы `pg_am` содержит имя функции, реализующей интерфейс[1]:

```
=> SELECT amname, amhandler FROM pg_am WHERE amtype = 'i';
 amname |  amhandler
--------+-------------
 btree  | bthandler
 hash   | hashhandler
 gist   | gisthandler
 gin    | ginhandler
 spgist | spghandler
 brin   | brinhandler
(6 rows)
```

Функция заполняет интерфейсную структуру[2] необходимыми значениями. Часть предоставляемой информации — это функции, выполняющие отдельные элементы общей работы (например, сканирующие индекс и возвращающие идентификаторы версий строк), а часть — свойства индексного метода, о которых должен знать механизм индексирования.

[1] postgrespro.ru/docs/postgresql/18/indexam.
[2] include/access/amapi.h.

Все свойства разделены на три уровня[1]:

- свойства метода доступа;
- свойства конкретного индекса;
- свойства отдельных столбцов индекса.

Выделение уровней метода доступа и индекса сделано с прицелом на будущее: в настоящее время все индексы, созданные на основе одного метода доступа, всегда будут иметь одинаковые свойства этих двух уровней.

Свойства метода доступа

К свойствам метода доступа относятся следующие пять (показаны на примере B-дерева):

```
=> SELECT a.amname, p.name, pg_indexam_has_property(a.oid, p.name)
FROM pg_am a, unnest(array[
  'can_order', 'can_unique', 'can_multi_col',
  'can_exclude', 'can_include'
]) p(name)
WHERE a.amname = 'btree';
 amname |     name      | pg_indexam_has_property
--------+---------------+-------------------------
 btree  | can_order     | t
 btree  | can_unique    | t
 btree  | can_multi_col | t
 btree  | can_exclude   | t
 btree  | can_include   | t
(5 rows)
```

CAN ORDER Возможность сразу получить отсортированные данные[2]. В настоящее время поддерживается только B-деревьями.

Чтобы получить результаты в нужном порядке, всегда можно просканировать таблицу и отсортировать полученные данные:

[1] backend/utils/adt/amutils.c, функция indexam_property.
[2] postgrespro.ru/docs/postgresql/18/indexes-ordering.

```
=> EXPLAIN (costs off)
SELECT * FROM seats ORDER BY seat_no;
       QUERY PLAN
------------------------
 Sort
   Sort Key: seat_no
   -> Seq Scan on seats
(3 rows)
```

Но при наличии индекса, поддерживающего это свойство, можно получить данные сразу в нужном порядке:

```
=> EXPLAIN (costs off)
SELECT * FROM seats ORDER BY aircraft_code;
              QUERY PLAN
-------------------------------------
 Index Scan using seats_pkey on seats
(1 row)
```

CAN UNIQUE Поддержка ограничения уникальности и первичного ключа[1]. Применимо только к B-деревьям.

При объявлении уникального или первичного ключа автоматически создается уникальный индекс, поддерживающий это ограничение целостности.

```
=> INSERT INTO bookings(book_ref, book_date, total_amount)
VALUES ('000004', now(), 100.00);
ERROR:  duplicate key value violates unique constraint
"bookings_pkey"
DETAIL:  Key (book_ref)=(000004) already exists.
```

С другой стороны, если не объявлять ограничение целостности явно, но создать уникальный индекс, видимый эффект будет таким же: столбец не будет допускать дубликатов. В чем разница?

Ограничение целостности — декларация свойства, которое должно выполняться, а индекс — тот механизм, который это обеспечивает. В принципе, ограничение может гарантироваться и другими средствами.

[1] postgrespro.ru/docs/postgresql/18/indexes-unique.

Например, PostgreSQL не позволяет создавать глобальные индексы на секционированных таблицах, но тем не менее позволяет объявить уникальный ключ (если в него входит ключ секционирования). В этом случае глобальная уникальность обеспечивается локальными уникальными индексами на каждой секции и тем фактом, что в двух секциях не может быть одинаковых значений ключа секционирования.

CAN MULTI COL Возможность построить *составной индекс* по нескольким столбцам[1].

Составной индекс позволяет ускорить поиск по нескольким условиям на разные столбцы таблицы. Например, таблица `ticket_flights` имеет составной первичный ключ и, соответственно, индекс:

```
=> \d ticket_flights_pkey
     Index "bookings.ticket_flights_pkey"
  Column   |     Type      | Key? | Definition
-----------+---------------+------+------------
 ticket_no | character(13) | yes  | ticket_no
 flight_id | integer       | yes  | flight_id
primary key, btree, for table "bookings.ticket_flights"
```

Поиск перелета по номеру билета и идентификатору рейса выполняется с помощью этого индекса:

```
=> EXPLAIN (costs off)
SELECT * FROM ticket_flights
WHERE ticket_no = '0005432001355' AND flight_id = 51618;
                        QUERY PLAN
-----------------------------------------------------------
 Index Scan using ticket_flights_pkey on ticket_flights
   Index Cond: ((ticket_no = '0005432001355'::bpchar) AND
   (flight_id = 51618))
(3 rows)
```

В случае B-дерева возможность и эффективность поиска зависят от порядка столбцов в индексе. Для индексов других типов это может быть и не так. *с. 536*, *с. 628*

[1] postgrespro.ru/docs/postgresql/18/indexes-multicolumn.

CAN EXCLUDE Поддержка ограничения исключения `EXCLUDE`[1].

Ограничение исключения гарантирует, что условие, определяемое оператором, не будет выполняться для любых пар строк таблицы. Для проверки ограничения автоматически создается индекс; класс операторов метода доступа должен содержать выбранный оператор.

с. 571

В качестве оператора обычно используется «пересечение» `&&`. Например, таким образом можно декларативно объявить, что аудитория не может быть дважды забронирована на одно и то же время или что здания на карте не могут накладываться друг на друга.

С оператором «равно» ограничение исключения приобретает смысл ограничения уникальности: запрещается наличие в таблице двух строк с совпадающими значениями ключевых столбцов. Тем не менее это не то же самое, что ограничение уникальности: в частности, на ключ ограничения исключения нельзя ссылаться во внешних ключах, его нельзя использовать во фразе `ON CONFLICT`.

CAN INCLUDE Возможность добавить к индексу столбцы, не являющиеся частью индексного ключа, чтобы сделать индекс покрывающим.

с. 425

Так, например, можно добавить дополнительные столбцы к уникальному индексу. Такой индекс будет гарантировать уникальность значений в ключевых столбцах, позволяя при этом получать значения дополнительных столбцов без обращения к таблице:

```
=> CREATE UNIQUE INDEX ON flights(flight_id) INCLUDE (status);
=> EXPLAIN (costs off)
SELECT status FROM flights WHERE flight_id = 51618;
                          QUERY PLAN
---------------------------------------------------------------
 Index Only Scan using flights_flight_id_status_idx on flights
   Index Cond: (flight_id = 51618)
(2 rows)
```

Свойств на самом деле гораздо больше, но многие недоступны на уровне SQL[2]. Вот некоторые из них:

[1] postgrespro.ru/docs/postgresql/18/ddl-constraints#DDL-CONSTRAINTS-EXCLUSION.
[2] include/access/amapi.h, структура IndexAmRoutine.

CAN PRED LOCKS Поддержка предикатных блокировок. *с. 298*

Отсутствует у методов доступа spgist и brin; при сканировании индексов таких типов на уровне изоляции Serializable предикатные блокировки будут устанавливаться на всю таблицу, как при последовательном сканировании, что приведет к снижению производительности.

CAN BUILD PARALLEL Возможность построения индекса несколькими процессами параллельно. Поддерживается методами доступа btree, brin и gin. v. 17 v. 18

Свойства индекса

Свойства, относящиеся к индексу (показаны на примере существующего):

```
=>  SELECT p.name, pg_index_has_property('seats_pkey', p.name)
FROM unnest(array[
  'clusterable', 'index_scan', 'bitmap_scan', 'backward_scan'
]) p(name);
     name      | pg_index_has_property
---------------+-----------------------
 clusterable   | t
 index_scan    | t
 bitmap_scan   | t
 backward_scan | t
(4 rows)
```

CLUSTERABLE Возможность физически переместить версии строк таблицы в соответствии с порядком, в котором данный индекс выдает их идентификаторы при индексном сканировании. *с. 361*

Это свойство определяет, применима ли команда CLUSTER.

INDEX SCAN Поддержка *индексного сканирования*. *с. 414*

Свойство означает, что метод доступа умеет выдавать идентификаторы версий строк по одному. Может показаться странным, но не все индексы поддерживают это свойство.

BITMAP SCAN Поддержка *сканирования по битовой карте*. *с. 427*

Свойство означает, что метод доступа может построить и вернуть битовую карту сразу по всем идентификаторам версий строк.

BACKWARD SCAN Возможность возвращать результат в порядке, обратном указанному при создании индекса.

Это свойство актуально, только если метод доступа поддерживает индексное сканирование.

Свойства столбцов

Наконец, свойства столбцов:

```
=> SELECT p.name,
  pg_index_column_has_property('seats_pkey', 1, p.name)
FROM unnest(array[
  'asc', 'desc', 'nulls_first', 'nulls_last', 'orderable',
  'distance_orderable', 'returnable', 'search_array', 'search_nulls'
]) p(name);
        name        | pg_index_column_has_property
--------------------+------------------------------
 asc                | t
 desc               | f
 nulls_first        | f
 nulls_last         | t
 orderable          | t
 distance_orderable | f
 returnable         | t
 search_array       | t
 search_nulls       | t
(9 rows)
```

ASC, DESC, NULLS FIRST, NULLS LAST Упорядоченность значений столбца.

Эти свойства определяют, в каком порядке хранятся значения (по возрастанию или по убыванию) и где находятся неопределенные значения (до обычных значений или после). Все свойства из этой группы применимы только к B-деревьям.

с. 350

ORDERABLE Возможность упорядочивать значения данного столбца в предложении `ORDER BY`.

Это свойство также применимо только к B-деревьям.

DISTANCE ORDERABLE Поддержка *операторов упорядочивания*[1].

с. 410

[1] postgrespro.ru/docs/postgresql/18/xindex#XINDEX-ORDERING-OPS.

В отличие от обычных индексных операторов, возвращающих логическое значение, операторы упорядочивания возвращают вещественное число — «расстояние» от одного аргумента до другого. Индекс поддерживает такие операторы в предложении ORDER BY.

Например, с помощью оператора упорядочивания <-> можно найти аэропорты, ближайшие к заданной точке:

```
=> CREATE INDEX ON airports_data USING gist(coordinates);
=> EXPLAIN (costs off)
SELECT * FROM airports
ORDER BY coordinates <-> point (43.578,57.593)
LIMIT 3;
                              QUERY PLAN
-----------------------------------------------------------------------
 Limit
   -> Index Scan using airports_data_coordinates_idx on airpo...
        Order By: (coordinates <-> '(43.578,57.593)'::point)
(3 rows)
```

RETURNABLE Возможность получения данных из индекса без обращения к таблице, то есть поддержка *сканирования только индекса*. *с. 422*

Свойство определяет, позволяет ли индексная структура восстановить проиндексированные значения. Это возможно не всегда; например, некоторые индексы могут сохранять не сами значения, а их хеш-коды. В таких случаях не будет работать и свойство CAN INCLUDE.

SEARCH ARRAY Поддержка поиска нескольких значений из массива.

Такая необходимость возникает не только при использовании массивов как таковых. Например, планировщик преобразует конструкцию IN (список) в поиск по массиву: *с.* **??**

```
=> EXPLAIN (costs off)
SELECT * FROM bookings
WHERE book_ref IN ('C7C821', 'A5D060', 'DDE1BB');
                 QUERY PLAN
-------------------------------------------
 Index Scan using bookings_pkey on bookings
   Index Cond: (book_ref = ANY
   ('{C7C821,A5D060,DDE1BB}'::bpchar[]))
(3 rows)
```

Если индексный метод не умеет работать с такими операторами, при исполнении плана потребуется несколько итераций для поиска одиночных значений (что может сказаться на эффективности).

SEARCH NULLS Возможность поиска по условиям `IS NULL` и `IS NOT NULL`.

Надо ли индексировать неопределенные значения? Это дает возможность использовать индекс для условий `IS [NOT] NULL`, а также в качестве покрывающего индекса при полном отсутствии условий на таблицу (поскольку в этом случае индекс должен вернуть данные всех строк таблицы, в том числе и с неопределенными значениями). Зато без `NULL` индекс может получиться компактнее.

Решение принимают разработчики метода доступа, но обычно неопределенные значения все-таки индексируются.

Если неопределенные значения в индексе не нужны, их можно исключить, построив *частичный индекс*[1] только по тем строкам таблицы, которые представляют интерес. Например:

```
=> CREATE INDEX ON flights(actual_arrival)
WHERE actual_arrival IS NOT NULL;
CREATE INDEX


=> EXPLAIN (costs off)
SELECT * FROM flights
WHERE actual_arrival = '2017-06-13 10:33:00+03';
                           QUERY PLAN
-----------------------------------------------------------------
 Index Scan using flights_actual_arrival_idx on flights
   Index Cond: (actual_arrival = '2017-06-13 10:33:00+03'::ti...
(2 rows)
```

Частичный индекс будет меньше полного по размеру и не будет обновляться при изменении строк, не входящих в индекс. В некоторых случаях это может дать ощутимый выигрыш. Конечно, условие в предложении `WHERE` может быть любым (удовлетворяющим требованиям категории изменчивости `IMMUTABLE`), а не только проверкой неопределенных значений.

[1] postgrespro.ru/docs/postgresql/18/indexes-partial.

Построение частичных индексов не зависит от метода доступа и обеспечивается механизмом индексирования.

Разумеется, интерфейсом охвачены не все возможности индексных методов доступа, а только те, о которых необходимо знать заранее, чтобы принять правильное решение. Например, нет свойства, которое отражает возможность неблокирующего создания индекса (`CONCURRENTLY`). Такие свойства определяются кодом функций, реализующих интерфейс.

20

Индексное сканирование

20.1. Простое индексное сканирование

Существует два базовых варианта работы с идентификаторами версий строк, поставляемыми индексом. Первый из них — *индексное сканирование*. Большинство индексных методов (но не все) обладают свойством INDEX SCAN и поддерживают этот способ.

с. 409

Операция представляется в плане запроса узлом Index Scan[1]:

```
=> EXPLAIN SELECT * FROM bookings
WHERE book_ref = '9AC0C6' AND total_amount = 48500.00;
                 QUERY PLAN
--------------------------------------------
 Index Scan using bookings_pkey on bookings
   (cost=0.43..8.45 rows=1 width=21)
   Index Cond: (book_ref = '9AC0C6'::bpchar)
   Filter: (total_amount = 48500.00)
(4 rows)
```

При индексном сканировании метод доступа возвращает идентификаторы версий строк по одному за раз[2]. Механизм индексирования получает очередной идентификатор, обращается к табличной странице, на которую он указывает, получает версию строки и, если она удовлетворяет правилам видимости, возвращает необходимый набор полей. Процесс продолжается, пока у метода доступа не закончатся идентификаторы, подходящие под условия запроса.

[1] backend/executor/nodeIndexscan.c.

[2] backend/access/index/indexam.c, функция index_getnext_tid.

В строке Index Cond указываются только те условия, которые могут быть проверены с помощью индекса. Дополнительные условия, требующие перепроверки по таблице, отображаются отдельно в строке Filter.

Таким образом, обращения к индексу и к таблице не выделены в два отдельных узла плана, а выполняются в общем узле Index Scan. Хотя и существует специальный узел Tid Scan[1], читающий из таблицы версии строк по известным идентификаторам:

```
=> EXPLAIN SELECT * FROM bookings WHERE ctid = '(0,1)'::tid;
                        QUERY PLAN
-----------------------------------------------------------
 Tid Scan on bookings  (cost=0.00..4.01 rows=1 width=21)
   TID Cond: (ctid = '(0,1)'::tid)
(2 rows)
```

Оценка стоимости

Оценка индексного сканирования складывается из оценки доступа к индексу и оценки чтения табличных страниц.

Очевидно, что индексная часть оценки полностью зависит от конкретного метода доступа. В случае B-дерева основные расходы приходятся на чтение индексных страниц и обработку строк в этих страницах. Сколько именно страниц и строк будет прочитано, можно определить, зная общий объем данных и селективность условий. Доступ к индексной странице носит случайный характер (страницы, соседствующие друг с другом логически, физически могут располагаться непредсказуемым образом). К оценкам также добавляются ресурсы процессора на спуск от корня до листовой страницы и на вычисление необходимых выражений[2].

с. 335

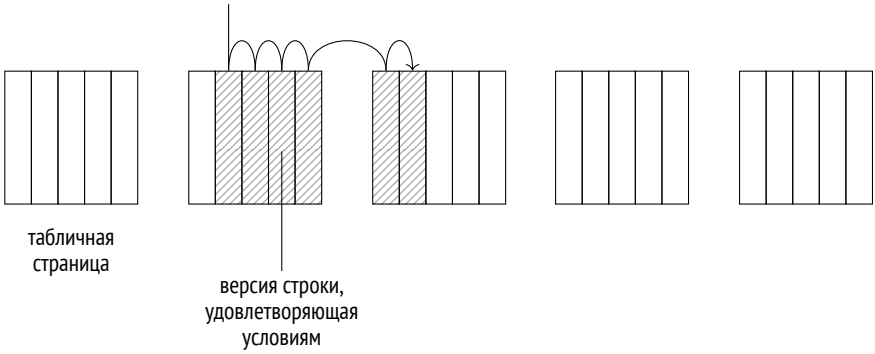
Табличная часть оценки учитывает стоимость доступа к страницам и процессорное время на обработку всех прочитанных версий. Важно, что оценка ввода-вывода зависит не только от селективности индексного доступа, но и от *корреляции* физического расположения версий строк на диске с тем порядком, в котором метод доступа выдает их идентификаторы.

[1] backend/executor/nodeTidscan.c.

[2] backend/utils/adt/selfuncs.c, функция btcostestimate; postgrespro.ru/docs/postgresql/18/index-cost-estimation.

Хороший случай: высокая корреляция

Если физический порядок версий на диске идеально коррелирует с логическим порядком идентификаторов в индексе, то, читая версии строк одну за другой, узел Index Scan будет *последовательно* переходить от страницы к странице и обратится к каждой из них *только один раз*.



с. 361 Информация о корреляции собирается как часть статистики:

```
=> SELECT attname, correlation
FROM pg_stats WHERE tablename = 'bookings'
ORDER BY abs(correlation) DESC;
   attname    | correlation
--------------+--------------
 book_ref     |            1
 total_amount | 0.0026738467
 book_date    |  8.02188e-05
(3 rows)
```

Значения, близкие по модулю к единице (как для столбца `book_ref`), говорят о высокой упорядоченности, а близкие к нулю — наоборот, о хаотичном распределении.

> В данном случае высокая корреляция для столбца `book_ref` вызвана, конечно, тем, что данные были загружены в таблицу в порядке возрастания значений этого столбца и не изменялись. К такой же картине приведет выполнение команды `CLUSTER` для индекса по этому столбцу.
>
> Но даже наличие полной корреляции не дает никаких гарантий, что любой запрос будет возвращать данные именно в порядке возрастания `book_ref`. Во-первых, изменение любой строки приведет к перемещению актуальной версии строки в конец

> таблицы. Во-вторых, план, использующий индексный доступ по какому-либо другому столбцу, выдаст данные в другом порядке. И даже последовательное сканирование может начаться не с начала таблицы. Поэтому если требуется определенный порядок, его необходимо указать в предложении ORDER BY. *с. 198*

Вот пример индексного сканирования большого количества строк:

```
=> EXPLAIN SELECT * FROM bookings WHERE book_ref < '100000';
                  QUERY PLAN
-----------------------------------------------
 Index Scan using bookings_pkey on bookings
   (cost=0.43..4640.91 rows=132999 width=21)
   Index Cond: (book_ref < '100000'::bpchar)
(3 rows)
```

Селективность условия по оценке планировщика составляет:

```
=> SELECT round(132999::numeric/reltuples::numeric, 4)
FROM pg_class WHERE relname = 'bookings';
 round
--------
 0.0630
(1 row)
```

Это близко к оценке $\frac{1}{16}$, которую можно дать из общих соображений, зная диапазон значений book_ref от 000000 до FFFFFF. *с. 357*

Индексная часть оценки ввода-вывода для B-дерева учитывает стоимость чтения необходимого количества страниц. Индексные строки, соответствующие любому условию, которое поддерживается B-деревом, упорядочены и находятся в логически связанных листовых страницах, поэтому количество прочитанных индексных страниц оценивается произведением размера индекса на селективность. Но физически эти страницы не упорядочены, поэтому чтение носит *случайный* характер.

Ресурсы процессора тратятся на обработку всех прочитанных индексных строк (стоимость обработки одной строки оценивается значением параметра *cpu_index_tuple_cost*) и вычисление условия для каждой из этих строк (в данном случае условие содержит один оператор; стоимость его вычисления оценивается значением параметра *cpu_operator_cost*). 0.005 0.0025

Табличный ввод-вывод рассматривается как *последовательное* чтение необходимого количества страниц. При идеальной корреляции табличные версии строк будут физически соседствовать друг с другом, поэтому количество страниц оценивается произведением размера таблицы на селективность.

К стоимости ввода-вывода добавляются затраты на обработку всех прочитанных версий строк; стоимость обработки одной версии оценивается зна-
0.01 чением параметра *cpu_tuple_cost*.

```
=> WITH costs(idx_cost, tbl_cost) AS (
  SELECT
    (
      SELECT round(
        current_setting('random_page_cost')::real * pages +
        current_setting('cpu_index_tuple_cost')::real * tuples +
        current_setting('cpu_operator_cost')::real * tuples
      )
      FROM (
        SELECT relpages * 0.0630 AS pages, reltuples * 0.0630 AS tuples
        FROM pg_class WHERE relname = 'bookings_pkey'
      ) c
    ),
    (
      SELECT round(
        current_setting('seq_page_cost')::real * pages +
        current_setting('cpu_tuple_cost')::real * tuples
      )
      FROM (
        SELECT relpages * 0.0630 AS pages, reltuples * 0.0630 AS tuples
        FROM pg_class WHERE relname = 'bookings'
      ) c
    )
)
SELECT idx_cost, tbl_cost, idx_cost + tbl_cost AS total
FROM costs;
```

```
 idx_cost | tbl_cost | total
----------+----------+-------
     2457 |     2180 |  4637
(1 row)
```

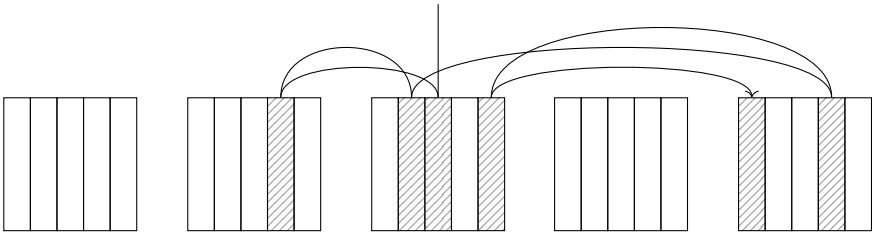
Формула показывает логику вычисления стоимости, и результат хорошо соответствует оценке планировщика, хотя и не является точным. Получение точного значения потребовало бы учета деталей, на которых мы не будем останавливаться.

Плохой случай: низкая корреляция

Ситуация с доступом к таблице меняется, когда корреляция оказывается низкой. Создадим индекс по столбцу book_date, имеющему практически нулевую корреляцию с индексом, и посмотрим на запрос, выбирающий приблизительно ту же долю строк, что и в предыдущем примере. Индексный доступ оказывается настолько дорогим, что планировщик выбирает его, только если запретить ему все альтернативы:

```
=> CREATE INDEX ON bookings(book_date);
=> SET enable_seqscan = off;
=> SET enable_bitmapscan = off;
=> EXPLAIN SELECT * FROM bookings
WHERE book_date < '2016-08-23 12:00:00+03';
                              QUERY PLAN
-----------------------------------------------------------------------
 Index Scan using bookings_book_date_idx on bookings
   (cost=0.43..57121.48 rows=132403 width=21)
   Index Cond: (book_date < '2016-08-23 12:00:00+03'::timestamp w...
(3 rows)
```

Дело в том, что чем ниже корреляция, тем выше вероятность того, что следующая версия строки, идентификатор которой выдает метод доступа, окажется на другой странице. Поэтому вместо последовательного чтения узел Index Scan «скачет» со страницы на страницу, а количество обращений к страницам в предельном случае может достигать количества выбираемых версий строк.



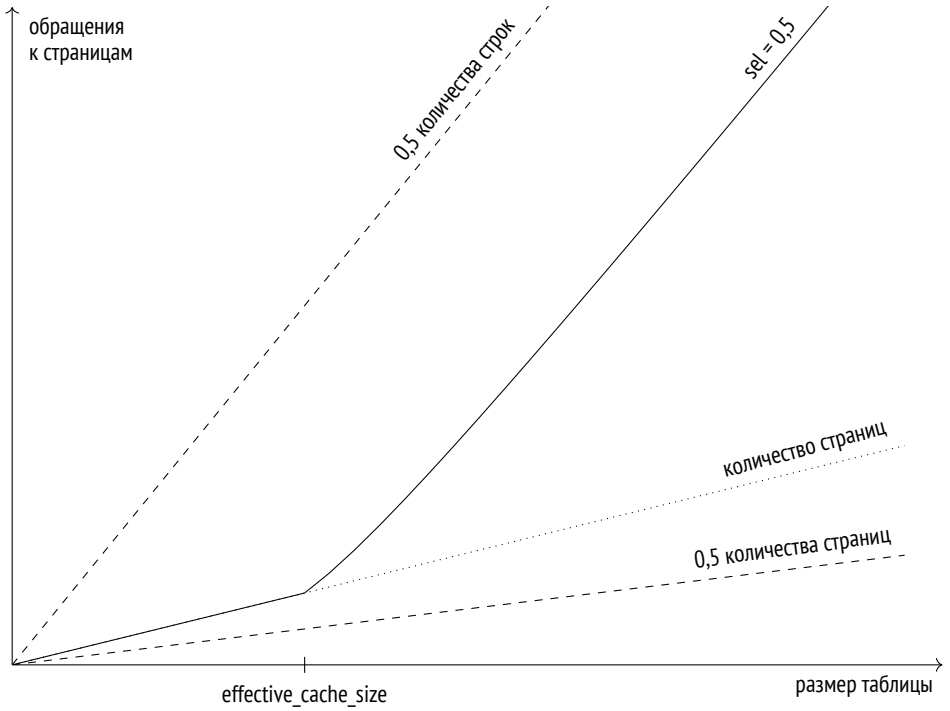
Однако было бы некорректным просто заменить в приведенной выше формуле *seq_page_cost* на *random_page_cost*, а relpages — на reltuples. Стоимость, которую мы видим в плане, на порядок меньше полученной таким образом оценки:

```
=> WITH costs(idx_cost, tbl_cost) AS (
  SELECT
    (
      SELECT round(
        current_setting('random_page_cost')::real * pages +
        current_setting('cpu_index_tuple_cost')::real * tuples +
        current_setting('cpu_operator_cost')::real * tuples
      )
      FROM (
        SELECT relpages * 0.0630 AS pages, reltuples * 0.0630 AS tuples
        FROM pg_class WHERE relname = 'bookings_pkey'
      ) c
    ),
    (
      SELECT round(
        current_setting('random_page_cost')::real * tuples +
        current_setting('cpu_tuple_cost')::real * tuples
      )
      FROM (
        SELECT relpages * 0.0630 AS pages, reltuples * 0.0630 AS tuples
        FROM pg_class WHERE relname = 'bookings'
      ) c
    )
)
SELECT idx_cost, tbl_cost, idx_cost + tbl_cost AS total
FROM costs;
 idx_cost | tbl_cost | total
----------+----------+--------
     2457 |   533330 | 535787
(1 row)
```

Дело в том, что модель дополнительно учитывает эффект кеширования. Часто используемые страницы задерживаются в буферном кеше (и в кеше операционной системы), поэтому чем больше кеш, тем больше вероятность найти нужную страницу в нем и избежать дисковой операции. Для целей планирования размер кеша определяется значением параметра
4GB *effective_cache_size*. Чем оно меньше, тем выше оценка количества страниц, которые придется прочитать.

График показывает зависимость оценки количества прочитанных страниц от размера таблицы (для селективности $\frac{1}{2}$ и случая, когда на странице помещается десять строк)[1].

[1] backend/optimizer/path/costsize.c, функция index_pages_fetched.



Пунктиром на графике показано количество обращений, равное половине количества страниц (лучший теоретически возможный случай при идеальной корреляции) и половине количества строк (худший случай при нулевой корреляции и отсутствии кеша).

Предполагается, что значение параметра *effective_cache_size* должно соответствовать общему объему памяти, доступному для кеширования (включая и буферный кеш PostgreSQL, и кеш ОС). Но поскольку параметр служит только для оценки и не приводит к реальному выделению памяти, при необходимости его можно изменять и без оглядки на реальные цифры.

Установив минимальное значение параметра, получим оценку плана, близкую к наихудшей, вычисленной выше без учета эффекта кеширования:

```
=> SET effective_cache_size = '8kB';
```

```
=> EXPLAIN SELECT * FROM bookings
WHERE book_date < '2016-08-23 12:00:00+03';
                              QUERY PLAN
----------------------------------------------------------------------
 Index Scan using bookings_book_date_idx on bookings
   (cost=0.43..532745.48 rows=132403 width=21)
   Index Cond: (book_date < '2016-08-23 12:00:00+03'::timestamp w...
(3 rows)


=> RESET effective_cache_size;
=> RESET enable_seqscan;
=> RESET enable_bitmapscan;
```

Стоимость табличного ввода-вывода вычисляется для двух рассмотренных случаев — хорошего и плохого, а затем берется промежуточное значение в зависимости от реальной корреляции[1].

Итак, индексное сканирование эффективно, когда требуется прочитать некоторую часть строк таблицы. Если расположение версий строк в таблице коррелирует с порядком, в котором метод доступа выдает идентификаторы, эта часть может быть весьма существенной. Если же корреляция слабая, то с уменьшением селективности индексный доступ очень быстро теряет свою привлекательность.

20.2. Сканирование только индекса

Индекс, содержащий все данные из таблицы, необходимые для выполнения запроса, называется *покрывающим* (covering) для данного запроса. При наличии покрывающего индекса можно избежать лишних обращений к таблице, получая от метода доступа не идентификаторы версий строк, а сами данные. Такая вариация индексного сканирования называется *сканированием только индекса*[2]. Она может использоваться теми методами доступа, которые поддерживают свойство RETURNABLE.

с. 411

[1] backend/optimizer/path/costsize.c, функция cost_index.

[2] postgrespro.ru/docs/postgresql/18/indexes-index-only-scans.

Операция представляется в плане запроса узлом Index Only Scan[1]:

```
=> EXPLAIN SELECT book_ref FROM bookings WHERE book_ref < '100000';
                    QUERY PLAN
-----------------------------------------------
 Index Only Scan using bookings_pkey on bookings
   (cost=0.43..4640.91 rows=132999 width=7)
   Index Cond: (book_ref < '100000'::bpchar)
(3 rows)
```

Название может навести на мысль, что узел Index Only Scan никогда не обращается к таблице, но это не так. Индексы в PostgreSQL не содержат информации, позволяющей судить о видимости строк, поэтому метод доступа возвращает данные из *всех* версий, попадающих под условие поиска, независимо от того, видны они текущей транзакции или нет. Видимость затем проверяется механизмом индексирования.

с. 89

Но если бы приходилось каждый раз заглядывать в таблицу для определения видимости, этот метод сканирования ничем не отличался бы от обычного индексного сканирования. Проблема решается тем, что PostgreSQL поддерживает для таблиц *карту видимости*, в которой процесс очистки отмечает страницы, содержащие только такие версии строк, которые видны всем транзакциям, независимо от их снимков. Если идентификатор версии строки, возвращенный индексным методом, относится к такой странице, то видимость версии можно не проверять.

с. 32

На оценку сканирования только индекса влияет доля табличных страниц, отмеченных в карте видимости. Эта информация входит в собираемую статистику:

```
=> SELECT relpages, relallvisible
FROM pg_class WHERE relname = 'bookings';
 relpages | relallvisible
----------+---------------
    13488 |         13446
(1 row)
```

[1] backend/executor/nodeIndexonlyscan.c.

Оценка стоимости сканирования только индекса отличается от оценки обычного индексного сканирования тем, что стоимость ввода-вывода, связанная с доступом к таблице, берется пропорционально доле страниц, не включенных в карту видимости. (Стоимость обработки версий строк процессором остается без изменений.)

Поскольку в данном примере все версии строк на всех страницах видны всем транзакциям, фактически из оценки стоимости полностью исключается стоимость табличного ввода-вывода:

```
=> WITH costs(idx_cost, tbl_cost) AS (
  SELECT
    (
      SELECT round(
        current_setting('random_page_cost')::real * pages +
        current_setting('cpu_index_tuple_cost')::real * tuples +
        current_setting('cpu_operator_cost')::real * tuples
      )
      FROM (
        SELECT relpages * 0.0630 AS pages,
               reltuples * 0.0630 AS tuples
        FROM pg_class WHERE relname = 'bookings_pkey'
      ) c
    ) AS idx_cost,
    (
      SELECT round(
        (1 - frac_visible) * -- доля страниц вне карты видимости
        current_setting('seq_page_cost')::real * pages +
        current_setting('cpu_tuple_cost')::real * tuples
      )
      FROM (
        SELECT relpages * 0.0630 AS pages,
               reltuples * 0.0630 AS tuples,
          relallvisible::real/relpages::real AS frac_visible
        FROM pg_class WHERE relname = 'bookings'
      ) c
    ) AS tbl_cost
)
SELECT idx_cost, tbl_cost, idx_cost + tbl_cost AS total
FROM costs;
 idx_cost | tbl_cost | total
----------+----------+-------
     2457 |     1333 |  3790
(1 row)
```

Наличие изменений, еще не попавших за горизонт базы данных и не обработанных очисткой, увеличивает оценку стоимости плана (и, соответственно, уменьшает привлекательность плана для оптимизатора). Чтобы узнать реальное количество вынужденных обращений к таблице, можно использовать команду EXPLAIN ANALYZE. *с. 106*

В только что созданной таблице перепроверяется видимость всех версий:

```
=> CREATE TEMP TABLE bookings_tmp WITH (autovacuum_enabled = off)
  AS SELECT * FROM bookings ORDER BY book_ref;
=> ALTER TABLE bookings_tmp ADD PRIMARY KEY(book_ref);
=> ANALYZE bookings_tmp;
=> EXPLAIN (analyze, timing off, summary off, buffers off)
SELECT book_ref FROM bookings_tmp WHERE book_ref < '100000';
                              QUERY PLAN
-----------------------------------------------------------------------
 Index Only Scan using bookings_tmp_pkey on bookings_tmp
   (cost=0.43..4641.91 rows=132999 width=7) (actual rows=132109.0...
   Index Cond: (book_ref < '100000'::bpchar)
   Heap Fetches: 132109
   Index Searches: 1
(5 rows)
```

Очистка создает карту видимости, и необходимость в проверке отпадает:

```
=> VACUUM bookings_tmp;
=> EXPLAIN (analyze, timing off, summary off, buffers off)
SELECT book_ref FROM bookings_tmp WHERE book_ref < '100000';
                              QUERY PLAN
-----------------------------------------------------------------------
 Index Only Scan using bookings_tmp_pkey on bookings_tmp
   (cost=0.43..3794.91 rows=132999 width=7) (actual rows=132109.0...
   Index Cond: (book_ref < '100000'::bpchar)
   Heap Fetches: 0
   Index Searches: 1
(5 rows)
```

Include-индексы

Возможна ситуация, когда индекс нельзя расширить так, чтобы он содержал все необходимые запросу столбцы:

- для уникального индекса добавление столбца не будет гарантировать уникальность исходных столбцов;
- тип данных дополнительного столбца может не иметь класса операторов для индексного метода доступа.

В этом случае можно добавить к индексу столбцы, не делая их частью индексного ключа. Поиск по дополнительным столбцам, конечно, будет невозможен, но для запросов, включающих эти столбцы, индекс сможет работать как покрывающий.

Пример показывает замену индекса, автоматически созданного для поддержки первичного ключа, на другой, с дополнительным столбцом:

```
=> CREATE UNIQUE INDEX ON bookings(book_ref) INCLUDE (book_date);
=> BEGIN;
=> ALTER TABLE bookings DROP CONSTRAINT bookings_pkey CASCADE;
NOTICE:  drop cascades to constraint tickets_book_ref_fkey on table
tickets
ALTER TABLE
=> ALTER TABLE bookings ADD CONSTRAINT bookings_pkey PRIMARY KEY
  USING INDEX bookings_book_ref_book_date_idx; -- новый индекс
NOTICE:  ALTER TABLE / ADD CONSTRAINT USING INDEX will rename index
"bookings_book_ref_book_date_idx" to "bookings_pkey"
ALTER TABLE
=> ALTER TABLE tickets
  ADD FOREIGN KEY (book_ref) REFERENCES bookings(book_ref);
=> COMMIT;
=> EXPLAIN SELECT book_ref, book_date
FROM bookings WHERE book_ref < '100000';
                            QUERY PLAN
-------------------------------------------------------------------
 Index Only Scan using bookings_pkey on bookings  (cost=0.43..438...
   Index Cond: (book_ref < '100000'::bpchar)
(2 rows)
```

> Очень часто *покрывающими* называют именно такие include-индексы, но это неверно. Индекс является покрывающим, если его набор столбцов *покрывает* столбцы, необходимые для конкретного запроса. Используются ли при этом ключевые поля или поля, добавленные с помощью предложения INCLUDE, — не играет никакой роли. Более того, один и тот же индекс может быть покрывающим для одного запроса и не быть таковым для другого.

20.3. Сканирование по битовой карте

При индексном сканировании уменьшение корреляции увеличивает количество обращений к страницам и меняет характер чтения с последовательного на случайный. Это ограничение можно преодолеть, получив перед обращением к таблице *все* идентификаторы и расположив их в порядке возрастания номеров страниц[1]. Именно так устроен второй базовый способ работы с идентификаторами — *сканирование по битовой карте*. Его могут использовать методы доступа, поддерживающие свойство BITMAP SCAN. *с. 409*

В отличие от обычного индексного сканирования, операция представляется в плане запроса двумя узлами:

```
=> CREATE INDEX ON bookings(total_amount);
=> EXPLAIN
SELECT * FROM bookings WHERE total_amount = 48500.00;
                              QUERY PLAN
-----------------------------------------------------------------------
 Bitmap Heap Scan on bookings  (cost=54.63..7045.60 rows=2865 wid...
   Recheck Cond: (total_amount = 48500.00)
   -> Bitmap Index Scan on bookings_total_amount_idx
       (cost=0.00..53.92 rows=2865 width=0)
       Index Cond: (total_amount = 48500.00)
(5 rows)
```

Узел Bitmap Index Scan[2] обращается к методу доступа за *битовой картой* всех идентификаторов версий строк[3].

Битовая карта состоит из фрагментов, каждый из которых соответствует одной табличной странице. Все фрагменты имеют одинаковый размер, достаточный для того, чтобы охватить все версии на странице, сколько бы их ни было. Это количество ограничено из-за довольно крупного заголовка; на странице стандартного размера помещается не больше 256 версий, для представления которых хватает 32 байт[4].

[1] backend/access/index/indexam.c, функция index_getbitmap.
[2] backend/executor/nodeBitmapIndexscan.c.
[3] backend/access/index/indexam.c, функция index_getbitmap.
[4] backend/nodes/tidbitmap.c.

Узел Bitmap Heap Scan[1] затем просматривает битовую карту фрагмент за фрагментом, читает соответствующую очередному фрагменту страницу и проверяет на этой странице все версии строк, отмеченные в битовой карте. Таким образом, страницы читаются в порядке возрастания номеров, и каждая — только один раз.

Поскольку список необходимых страниц известен уже на этапе построения
с. 183 битовой карты, сканирование задействует асинхронный ввод-вывод. Не все операции будут многоблочными, из-за того что нужные страницы не всегда располагаются подряд друг за другом, но в любом случае асинхронная предвыборка значительно повышает эффективность этого метода доступа.

Точность карты

Чем больше страниц охвачено версиями строк, соответствующих условию запроса, тем больше места занимает битовая карта. Она строится в локальной памяти обслуживающего процесса и ограничена размером, указанным
4MB в параметре *work_mem*. Если при построении карты размер достигает предельного значения, некоторые фрагменты карты «загрубляются» таким образом, чтобы каждый бит фрагмента соответствовал целой странице, а сам фрагмент охватывал не одну страницу, а диапазон[2]. Это позволяет уменьшить размер битовой карты, пожертвовав при этом точностью.

Команда EXPLAIN ANALYZE показывает точность построенной карты. В данном случае объема памяти хватило для точной (exact) битовой карты:

```
=> EXPLAIN (analyze, costs off, timing off, summary off, buffers off)
SELECT * FROM bookings WHERE total_amount > 150000.00;
                            QUERY PLAN
------------------------------------------------------------------
 Bitmap Heap Scan on bookings (actual rows=242691.00 loops=1)
   Recheck Cond: (total_amount > 150000.00)
   Heap Blocks: exact=13447
   -> Bitmap Index Scan on bookings_total_amount_idx (actual rows...
        Index Cond: (total_amount > 150000.00)
        Index Searches: 1
(6 rows)
```

[1] backend/executor/nodeBitmapHeapscan.c.
[2] backend/nodes/tidbitmap.c, функция tbm_lossify.

Если уменьшить значение *work_mem*, часть фрагментов карты будет храниться с потерей точности (lossy):

```
=> SET work_mem = '512kB';
=> EXPLAIN (analyze, costs off, timing off, summary off, buffers off)
SELECT * FROM bookings WHERE total_amount > 150000.00;
                            QUERY PLAN
------------------------------------------------------------------
 Bitmap Heap Scan on bookings (actual rows=242691.00 loops=1)
   Recheck Cond: (total_amount > 150000.00)
   Rows Removed by Index Recheck: 1145721
   Heap Blocks: exact=5178 lossy=8269
   -> Bitmap Index Scan on bookings_total_amount_idx (actual rows...
       Index Cond: (total_amount > 150000.00)
       Index Searches: 1
(7 rows)
=> RESET work_mem;
```

При чтении табличной страницы, соответствующей грубому фрагменту битовой карты, необходимо перепроверять условия выборки для каждой версии строки на странице. Условие перепроверки всегда отображается в плане как Recheck Cond — независимо от того, выполняется ли перепроверка на самом деле. А количество версий строк, отфильтрованных перепроверкой, показывается отдельно (Rows Removed by Index Recheck).

> При очень большой выборке может получиться, что битовая карта, даже целиком состоящая из грубых фрагментов, все равно не помещается в память размером *work_mem*. В этом случае ограничение не соблюдается, и карта занимает столько места, сколько необходимо. Никакого дополнительного уменьшения точности или сброса части фрагментов на диск не предусмотрено.

Действия с битовыми картами

Если в запросе условия наложены на несколько полей таблицы и для этих полей созданы разные индексы, сканирование битовой карты позволяет использовать несколько индексов одновременно[1]. Для каждого из индексов строятся битовые карты версий строк, которые затем либо побитово логически умножаются (если выражения соединены условием AND), либо логически складываются (если выражения соединены условием OR).

[1] postgrespro.ru/docs/postgresql/18/indexes-ordering.

Например:

```
=> EXPLAIN (costs off)
SELECT *
FROM bookings
WHERE book_date < '2016-08-28'
  AND total_amount > 250000;
                              QUERY PLAN
-----------------------------------------------------------------------
 Bitmap Heap Scan on bookings
   Recheck Cond: ((total_amount > '250000'::numeric) AND (book_da...
   -> BitmapAnd
        -> Bitmap Index Scan on bookings_total_amount_idx
             Index Cond: (total_amount > '250000'::numeric)
        -> Bitmap Index Scan on bookings_book_date_idx
             Index Cond: (book_date < '2016-08-28 00:00:00+03'::tim...
(7 rows)
```

Здесь узел BitmapAnd объединяет две битовые карты с помощью битовой операции «и».

При объединении двух битовых карт в одну[1] точные фрагменты остаются точными (если для новой карты хватает места *work_mem* в памяти), но если хотя бы один из фрагментов грубый, то и результирующий фрагмент тоже будет иметь низкую точность.

Оценка стоимости

Возьмем пример запроса со сканированием по битовой карте:

```
=> EXPLAIN
SELECT * FROM bookings WHERE total_amount = 28000.00;
                              QUERY PLAN
-----------------------------------------------------------------------
 Bitmap Heap Scan on bookings  (cost=599.48..14485.96 rows=31878 ...
   Recheck Cond: (total_amount = 28000.00)
   -> Bitmap Index Scan on bookings_total_amount_idx
        (cost=0.00..591.51 rows=31878 width=0)
        Index Cond: (total_amount = 28000.00)
(5 rows)
```

[1] backend/nodes/tidbitmap.c, функции tbm_union и tbm_intersect.

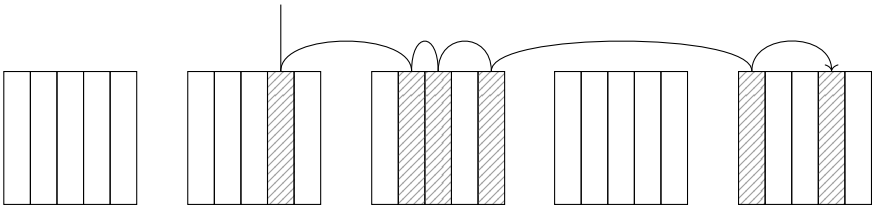
Здесь селективность условия по оценке планировщика равна примерно

```
=> SELECT round(31878::numeric/reltuples::numeric, 4)
FROM pg_class WHERE relname = 'bookings';
 round
--------
 0.0151
(1 row)
```

Оценка полной стоимости узла Bitmap Index Scan вычисляется так же, как стоимость обычного индексного доступа без учета обращений к таблице:

```
=> SELECT round(
  current_setting('random_page_cost')::real * pages +
  current_setting('cpu_index_tuple_cost')::real * tuples +
  current_setting('cpu_operator_cost')::real * tuples
)
FROM (
  SELECT relpages * 0.0151 AS pages, reltuples * 0.0151 AS tuples
  FROM pg_class WHERE relname = 'bookings_total_amount_idx'
) c;
 round
-------
   589
(1 row)
```

Для узла Bitmap Heap Scan оценка ввода-вывода отличается от аналогичной оценки в случае простого индексного сканирования при идеальной корреляции. Битовая карта позволяет читать табличные страницы в порядке возрастания номеров и без повторных обращений, но версии строк, соответствующие условию, больше не располагаются по соседству друг с другом. Вместо строго последовательного компактного диапазона, скорее всего, потребуется прочитать намного больше страниц.



Количество прочитанных страниц оценивается формулой[1]

$$\min\left(\frac{2\,\texttt{relpages}\cdot\texttt{reltuples}\cdot sel}{2\,\texttt{relpages}+\texttt{reltuples}\cdot sel},\texttt{relpages}\right),$$

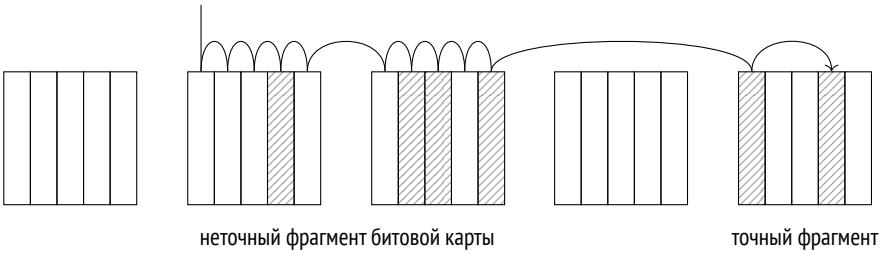
а стоимость чтения одной страницы — значением от *seq_page_cost* до *random_page_cost* в зависимости от отношения числа прочитанных страниц к их общему количеству в таблице:

```
=> WITH t AS (
  SELECT relpages,
    least(
      (2 * relpages * reltuples * 0.0151) /
      (2 * relpages + reltuples * 0.0151),
      relpages
    ) AS pages_fetched,
    round(reltuples * 0.0151) AS tuples_fetched,
    current_setting('random_page_cost')::real AS rnd_cost,
    current_setting('seq_page_cost')::real AS seq_cost
  FROM pg_class WHERE relname = 'bookings'
)
SELECT pages_fetched,
  rnd_cost - (rnd_cost - seq_cost) *
  sqrt(pages_fetched / relpages) AS cost_per_page,
  tuples_fetched
FROM t;
 pages_fetched | cost_per_page | tuples_fetched
---------------+---------------+----------------
         13488 |             1 |          31878
(1 row)
```

Как обычно, к оценке ввода-вывода добавляется оценка обработки каждой прочитанной узлом версии строки. При точной битовой карте количество строк оценивается произведением общего количества строк в таблице на селективность условий. Но когда некоторые фрагменты битовой карты неточны, приходится проверять все версии строк на соответствующих страницах. Поэтому в оценке учитывается предполагаемая доля неточных фрагментов битовой карты (которую можно посчитать, зная общее количество строк в выборке и ограничение на размер битовой карты *work_mem*)[2].

К оценке также добавляется полная стоимость перепроверки условий (независимо от точности битовой карты).

[1] backend/optimizer/path/costsize.c, функция compute_bitmap_pages.
[2] backend/optimizer/path/costsize.c, функция compute_bitmap_pages.



В нашем примере битовая карта полностью точна, и оценка вычисляется примерно следующим образом[1]:

```
=> WITH t AS (
  SELECT 1 AS cost_per_page,
         13488 AS pages_fetched,
         31878 AS tuples_fetched
),
costs(startup_cost, run_cost) AS (
  SELECT
    (
      SELECT round(
        589 /* оценка нижележащего узла */ +
        0.1 * current_setting('cpu_operator_cost')::real *
        reltuples * 0.0151
      )
      FROM pg_class
      WHERE relname = 'bookings_total_amount_idx'
    ),
    (
      SELECT round(
        cost_per_page * pages_fetched +
        current_setting('cpu_tuple_cost')::real * tuples_fetched +
        current_setting('cpu_operator_cost')::real * tuples_fetched
      )
      FROM t
    )
)
SELECT startup_cost, run_cost,
  startup_cost + run_cost AS total_cost
FROM costs;
 startup_cost | run_cost | total_cost
--------------+----------+------------
          597 |    13886 |      14483
(1 row)
```

[1] backend/optimizer/path/costsize.c, функция cost_bitmap_heap_scan.

Здесь оценка начальной стоимости узла Bitmap Heap Scan опирается на оценку полной стоимости узла Bitmap Index Scan, к которой добавляется стоимость работы с битовыми картами:

```
                        QUERY PLAN
-----------------------------------------------------
 Bitmap Heap Scan on bookings
   (cost=599.48..14485.96 rows=31878 width=21)
   Recheck Cond: (total_amount = 28000.00)
   -> Bitmap Index Scan on bookings_total_amount_idx
        (cost=0.00..591.51 rows=31878 width=0)
        Index Cond: (total_amount = 28000.00)
(6 rows)
```

Если план запроса использует объединение нескольких битовых карт, сумма стоимостей доступа к отдельным индексам увеличивается на (небольшую) стоимость собственно объединения[1].

20.4. Параллельные версии индексного сканирования

Все способы индексного сканирования — обычное, сканирование только ин-
с. 379 декса, сканирование по битовой карте — имеют версии для параллельных планов.

Стоимость параллельного выполнения оценивается аналогично последовательному, но (как и в случае параллельного последовательного сканирования) ресурсы процессора распределяются между всеми параллельными процессами, уменьшая итоговую стоимость. Составляющая ввода-вывода не распределяется, поскольку чтение страниц синхронизируется между процессами и происходит последовательно.

Дальше я просто покажу примеры параллельных планов без разбора оценок стоимости.

[1] backend/optimizer/path/costsize.c, функции cost_bitmap_and_node и cost_bitmap_or_node.

Параллельное индексное сканирование Parallel Index Scan:

```
=> EXPLAIN SELECT sum(total_amount)
FROM bookings WHERE book_ref < '400000';
                              QUERY PLAN
-----------------------------------------------------------------------
 Finalize Aggregate  (cost=19202.81..19202.82 rows=1 width=32)
   -> Gather  (cost=19202.59..19202.80 rows=2 width=32)
        Workers Planned: 2
        -> Partial Aggregate  (cost=18202.59..18202.60 rows=1 widt...
            -> Parallel Index Scan using bookings_pkey on bookings
                (cost=0.43..17652.82 rows=219907 width=6)
                Index Cond: (book_ref < '400000'::bpchar)
(7 rows)
```

При параллельном сканировании B-дерева в общей памяти сервера хранится номер текущей индексной страницы. Начальное значение устанавливается процессом, который начинает сканирование: он спускается от корня дерева к листовой странице и запоминает ее. Рабочие процессы обращаются по мере необходимости за следующей индексной страницей, изменяя сохраненный номер. Получив страницу, процесс обрабатывает все подходящие строки в ней и читает соответствующие версии строк из таблицы. Сканирование завершается, когда прочитан весь диапазон значений, удовлетворяющих условию запроса.

с. 524

Параллельное сканирование только индекса Parallel Index Only Scan:

```
=> EXPLAIN SELECT sum(total_amount)
FROM bookings WHERE total_amount < 50000.00;
                              QUERY PLAN
-----------------------------------------------------------------------
 Finalize Aggregate  (cost=23538.60..23538.61 rows=1 width=32)
   -> Gather  (cost=23538.38..23538.59 rows=2 width=32)
        Workers Planned: 2
        -> Partial Aggregate  (cost=22538.38..22538.39 rows=1 widt...
            -> Parallel Index Only Scan using bookings_total_amoun...
                (cost=0.43..21555.27 rows=393244 width=6)
                Index Cond: (total_amount < 50000.00)
(7 rows)
```

Параллельное сканирование только индекса отличается лишь тем, что не обращается к табличным страницам, если это позволяет карта видимости.

И наконец, параллельное сканирование по битовой карте:

```
=> EXPLAIN SELECT sum(total_amount)
FROM bookings WHERE book_date < '2016-10-01';
                              QUERY PLAN
-----------------------------------------------------------------------
 Finalize Aggregate  (cost=21533.21..21533.22 rows=1 width=32)
   -> Gather  (cost=21532.99..21533.20 rows=2 width=32)
        Workers Planned: 2
        -> Partial Aggregate  (cost=20532.99..20533.00 rows=1 widt...
            -> Parallel Bitmap Heap Scan on bookings
                (cost=4891.17..20174.01 rows=143588 width=6)
                Recheck Cond: (book_date < '2016-10-01 00:00:00+03...
                -> Bitmap Index Scan on bookings_book_date_idx
                    (cost=0.00..4805.01 rows=344611 width=0)
                    Index Cond: (book_date < '2016-10-01 00:00:00+...
(10 rows)
```

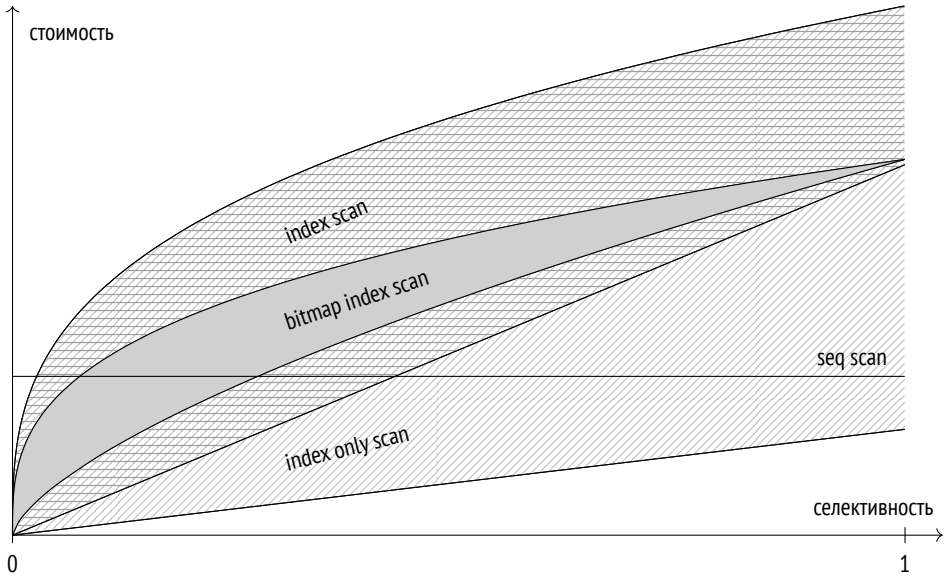
При сканировании по битовой карте построение карты всегда выполняется последовательно, одним ведущим процессом; поэтому к названию узла Bitmap Index Scan не добавляется слово Parallel. Когда битовая карта готова, сканирование таблицы выполняется параллельно в узле Parallel Bitmap Heap Scan. Рабочие процессы обращаются за очередной табличной страницей и обрабатывают ее.

20.5. Сравнение методов доступа

График зависимости стоимости различных методов доступа от селективности условий показан на следующей странице. Он имеет качественный характер; конкретные числа, разумеется, будут зависеть от таблицы и от параметров сервера.

Последовательное сканирование не зависит от селективности и, начиная с некоторой доли выбираемых строк, обычно работает лучше остальных методов.

Стоимость индексного сканирования сильно зависит от корреляции между физическим расположением версий строк и порядком, в котором индексный метод доступа выдает их идентификаторы. При идеальной корреляции индексное сканирование эффективно даже при довольно большой доле



выбираемых строк. Но при слабой корреляции (что чаще встречается в реальности) стоимость высока и очень быстро начинает превышать даже стоимость последовательного сканирования. При всем этом индексное сканирование — безусловный лидер в очень важном случае, когда индекс (часто уникальный) используется для выборки единственной строки.

Сканирование только индекса (если оно применимо) может давать прекрасные результаты и выигрывать у последовательного сканирования даже при выборке всех строк. Его производительность, однако, очень сильно зависит от карты видимости, и в худшем случае сканирование только индекса деградирует до обычного индексного сканирования.

Стоимость сканирования по битовой карте зависит от объема доступной памяти, но в гораздо меньшей степени, чем стоимость индексного сканирования зависит от корреляции. При слабой корреляции сканирование по битовой карте существенно выигрывает.

Каждый из методов доступа может превосходить остальные в одних ситуациях и проигрывать им в других. Планировщик выполняет серьезную работу по оценке эффективности всех методов в каждом конкретном случае. Конечно, близость этих оценок к реальности сильно зависит от актуальности статистической информации.

21

Вложенный цикл

21.1. Виды и способы соединений

Соединения — ключевая возможность языка SQL, основа его гибкости и мощности. Наборы строк (полученные непосредственно из таблиц или как результат выполнения других операций) всегда соединяются попарно.

Существует несколько *видов* соединений.

Внутренние соединения. *Внутреннее соединение* (`INNER JOIN`, или просто `JOIN`) включает такие пары строк из двух наборов, для которых выполняется *условие соединения*. Условие соединения связывает некоторые столбцы одного набора строк с некоторыми столбцами другого; все участвующие столбцы составляют *ключ соединения*.

Если условие требует равенства значений в столбцах одного и другого наборов, соединение называют *эквисоединением*; это наиболее частый случай.

Декартово произведение (`CROSS JOIN`) двух наборов включает все возможные пары строк — это частный случай внутреннего соединения с истинным условием.

Внешние соединения. *Левое внешнее соединение* (`LEFT OUTER JOIN`, или просто `LEFT JOIN`) добавляет к внутреннему соединению строки из левого набора, для которых не нашлось соответствия в правом наборе (столбцы отсутствующего правого набора получают неопределенные значения).

То же верно и для *правого внешнего соединения* (`RIGHT JOIN`), с точностью до перестановки наборов.

Полное внешнее соединение (`FULL JOIN`) объединяет левое и правое внешние соединения, добавляя строки как из левого, так и из правого наборов, для которых не нашлось соответствия.

Полусоединения и антисоединения. *Полусоединение* похоже на внутреннее соединение, но включает строки одного набора, для которых нашлось соответствие в другом наборе (строка будет включена в результат только один раз, даже если соответствий несколько).

Антисоединение включает строки одного набора, для которых не нашлось пары в другом наборе.

В языке SQL нет явных операций полу- и антисоединения, но к ним, например, приводят такие конструкции, как `EXISTS` и `NOT EXISTS`.

Все это — логические операции. Например, внутреннее соединение часто описывается как декартово произведение, в котором оставлены только строки, удовлетворяющие условию соединения. Но физически выполнить внутреннее соединение обычно можно другими, более экономными средствами.

PostgreSQL предоставляет несколько *способов* соединения:

- соединение вложенным циклом (nested loop);
- соединение хешированием (hash join);
- соединение слиянием (merge join).

Способы соединения — алгоритмы, реализующие логические операции соединения SQL. Эти базовые алгоритмы часто имеют вариации, приспособленные для конкретных видов соединений, хотя могут и не поддерживать все из них. Например, вложенный цикл может применяться и для внутреннего соединения (в этом случае он будет представлен узлом Nested Loop), и для левого внешнего соединения (что будет отражено в плане узлом Nested Loop Left Join), но не может использоваться для полного соединения.

Более того, варианты тех же алгоритмов используются и для выполнения других операций, например агрегации.

Как и методы доступа, способы соединения по-разному показывают себя в разных ситуациях; планировщик выбирает лучший по стоимости вариант.

21.2. Соединение вложенным циклом

Базовый алгоритм соединения вложенным циклом устроен следующим образом. Во внешнем цикле перебираются строки первого набора (который называется *внешним*). Для каждой такой строки во вложенном цикле перебираются строки второго набора (который называется *внутренним*), удовлетворяющие условию соединения. Каждая найденная пара немедленно возвращается как часть результата[1].

Алгоритм обращается к внутреннему набору столько раз, сколько строк содержит внешний набор. Поэтому на эффективность соединения вложенным циклом влияет несколько условий:

- кардинальность внешнего набора строк;
- наличие метода доступа ко внутреннему набору, позволяющего эффективно получить нужные строки;
- повторные обращения к одним и тем же строкам внутреннего набора.

Декартово произведение

Соединение вложенным циклом — наиболее эффективный способ выполнения декартова произведения, независимо от количества строк в наборах:

```
=> EXPLAIN SELECT * FROM aircrafts_data a1
  CROSS JOIN aircrafts_data a2
WHERE a2.range > 5000;
                        QUERY PLAN
-----------------------------------------------------
 Nested Loop  (cost=0.00..2.78 rows=45 width=144)
   -> Seq Scan on aircrafts_data a1                      внешний набор
        (cost=0.00..1.09 rows=9 width=72)
   -> Materialize  (cost=0.00..1.14 rows=5 width=72)
        -> Seq Scan on aircrafts_data a2                 внутренний набор
             (cost=0.00..1.11 rows=5 width=72)
             Filter: (range > 5000)
(7 rows)
```

[1] backend/executor/nodeNestloop.c.

Узел Nested Loop выполняет соединение алгоритмом вложенного цикла. Он всегда имеет два дочерних узла. Тот, что находится выше в выводе плана, представляет внешний набор строк; тот, что ниже, — внутренний.

В данном случае внутренний набор представлен узлом Materialize [1]. Этот узел материализует строки, полученные от нижестоящего узла, что позволяет при повторном обращении не сканировать заново всю таблицу, а сразу прочитать сохраненные строки, удовлетворяющие условию. *с. 338*

К плану такого же вида может привести и запрос с обычным эквисоединением:

```
=> EXPLAIN SELECT *
FROM tickets t
  JOIN ticket_flights tf ON tf.ticket_no = t.ticket_no
WHERE t.ticket_no = '0005432000284';
                            QUERY PLAN
------------------------------------------------------------------
 Nested Loop  (cost=0.99..25.05 rows=3 width=136)
   -> Index Scan using tickets_pkey on tickets t
       (cost=0.43..8.45 rows=1 width=104)
       Index Cond: (ticket_no = '0005432000284'::bpchar)
   -> Index Scan using ticket_flights_pkey on ticket_flights tf
       (cost=0.56..16.57 rows=3 width=32)
       Index Cond: (ticket_no = '0005432000284'::bpchar)
(7 rows)
```

Здесь планировщик, понимая равенство двух значений, заменил условие соединения `tf.ticket_no = t.ticket_no` на условие `tf.ticket_no =` *константа*, фактически сведя эквисоединение к декартову произведению[2].

Оценка кардинальности. Кардинальность декартова произведения равна произведению кардинальностей соединяемых наборов: $3 = 1 \times 3$.

Оценка стоимости. Начальная стоимость соединения равна сумме начальных стоимостей дочерних узлов.

Полная стоимость соединения в данном случае складывается:

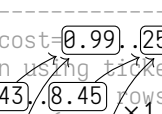
- из стоимости получения всех строк внешнего набора;

[1] backend/executor/nodeMaterial.c.
[2] backend/optimizer/path/equivclass.c.

- однократной стоимости получения всех строк внутреннего набора (поскольку оценка кардинальности внешнего набора равна единице);
- стоимости обработки каждой строки результата.

Схема зависимостей при вычислении оценок выглядит так:

```
                                QUERY PLAN
-----------------------------------------------------------------------
 Nested Loop  (cost=0.99..25.05 rows=3 width=136)
   -> Index Scan using tickets_pkey on tickets t
        (cost=0.43..8.45 rows=1 width=104)
        Index Cond: (ticket_no = '0005432000284'::bpchar)
   -> Index Scan using ticket_flights_pkey on ticket_flights tf
        (cost=0.56..16.57 rows=3 width=32)
        Index Cond: (ticket_no = '0005432000284'::bpchar)
(7 rows)
```



Вот как вычисляется стоимость соединения в этом случае:

```
=> SELECT 0.43 + 0.56 AS startup_cost,
  round((
    8.45 + 16.57 +
    3 * current_setting('cpu_tuple_cost')::real
  )::numeric, 2) AS total_cost;
 startup_cost | total_cost
--------------+------------
         0.99 |      25.05
(1 row)
```

Вернемся теперь к предыдущему примеру с узлом Materialize:

```
=> EXPLAIN SELECT *
FROM aircrafts_data a1
  CROSS JOIN aircrafts_data a2
WHERE a2.range > 5000;
                       QUERY PLAN
-------------------------------------------------------
 Nested Loop  (cost=0.00..2.78 rows=45 width=144)
   -> Seq Scan on aircrafts_data a1
        (cost=0.00..1.09 rows=9 width=72)
   -> Materialize  (cost=0.00..1.14 rows=5 width=72)
        -> Seq Scan on aircrafts_data a2
             (cost=0.00..1.11 rows=5 width=72)
             Filter: (range > 5000)
(7 rows)
```

В общем случае полная стоимость соединения складывается[1]:

- из стоимости получения всех строк внешнего набора;
- однократной стоимости первоначального получения всех строк внутреннего набора (в ходе которого выполняется материализация);
- $(N-1)$-кратной стоимости повторного получения строк внутреннего набора (где N — число строк во внешнем наборе);
- стоимости обработки каждой строки результата.

Схема зависимостей здесь получается такая:

```
                        QUERY PLAN
-----------------------------------------------------
 Nested Loop  (cost=0.00..2.78 rows=45 width=144)
   -> Seq Scan on aircrafts_data a1
        (cost=0.00..1.09 rows=9 width=72)
   -> Materialize                          ×9
        (cost=0.00..1.14 rows=5 width=72)
        -> Seq Scan on aircrafts_data a2
             (cost=0.00..1.11 rows=5 width=72)
             Filter: (range > 5000)
(8 rows)
```

Благодаря материализации повторное получение данных обошлось дешевле. Стоимость первого обращения к узлу Materialize указана в плане, но стоимость повторного обращения не выводится. Я не буду разбирать, как вычисляется эта оценка[2], но в данном случае она составляет 0,0125.

Таким образом, стоимость соединения для этого примера вычисляется так:

```
=> SELECT 0.00 + 0.00 AS startup_cost,
  round((
    1.09 + (1.14 + 8 * 0.0125) +
    45 * current_setting('cpu_tuple_cost')::real
  )::numeric, 2) AS total_cost;
 startup_cost | total_cost
--------------+------------
         0.00 |       2.78
(1 row)
```

[1] backend/optimizer/path/costsize.c, функции initial_cost_nestloop и final_cost_nestloop.

[2] backend/optimizer/path/costsize.c, функция cost_rescan.

Параметризованное соединение

Рассмотрим другой, более типичный пример, который не сводится к простому декартову произведению:

```
=> CREATE INDEX ON tickets(book_ref);
=> EXPLAIN SELECT *
FROM tickets t
  JOIN ticket_flights tf ON tf.ticket_no = t.ticket_no
WHERE t.book_ref = '03A76D';
                            QUERY PLAN
------------------------------------------------------------------
 Nested Loop  (cost=0.99..45.67 rows=6 width=136)
   -> Index Scan using tickets_book_ref_idx on tickets t
       (cost=0.43..12.46 rows=2 width=104)
       Index Cond: (book_ref = '03A76D'::bpchar)
   -> Index Scan using ticket_flights_pkey on ticket_flights tf
       (cost=0.56..16.57 rows=3 width=32)
       Index Cond: (ticket_no = t.ticket_no)
(7 rows)
```

Здесь узел Nested Loop перебирает строки внешнего набора (билеты) и для каждой такой строки обращается к строкам внутреннего набора (перелеты), передавая в условие доступа номер билета `t.ticket_no` *как параметр*. Когда вызывается внутренний узел Index Scan, он имеет дело с условием `ticket_no` = *константа*.

Оценка кардинальности. По оценке планировщика, условию на номер бронирования удовлетворяют две строки внешнего набора (rows=2), и для каждой во внутреннем наборе будет найдено в среднем три строки (rows=3).

Селективностью соединения называется доля строк от декартова произведения двух наборов, которая остается после соединения. Конечно, из учета надо сразу исключить строки обоих наборов, содержащие неопределенные значения в столбцах, по значениям которых происходит соединение (для таких строк условие равенства заведомо не будет выполняться).

Кардинальность оценивается как кардинальность декартова произведения (то есть произведение кардинальностей двух наборов), умноженная на селективность[1].

[1] backend/optimizer/path/costsize.c, функция calc_joinrel_size_estimate.

В данном случае известна оценка кардинальности первого (внешнего) набора — две строки. Никаких условий на второй (внутренний) набор, кроме самого условия соединения, нет. Поэтому за кардинальность второго набора принимается кардинальность таблицы ticket_flights.

Поскольку соединяемые таблицы связаны внешним ключом, оценка селективности дается на основании того, что каждая строка дочерней таблицы имеет ровно одну пару в родительской таблице. За селективность в этом случае принимается величина, обратная размеру таблицы, на которую ссылается внешний ключ[1].

Таким образом (учитывая, что столбцы ticket_no не имеют неопределенных значений), оценка составляет

```
=> SELECT round(2 * tf.reltuples * (1.0 / t.reltuples)) AS rows
FROM pg_class t, pg_class tf
WHERE t.relname = 'tickets'
  AND tf.relname = 'ticket_flights';
 rows
------
    6
(1 row)
```

Разумеется, таблицы можно соединять и без внешних ключей. Тогда в качестве селективности соединения будет использоваться оценка селективности конкретных условий соединения[2].

Для нашего случая эквисоединения «базовая» формула расчета селективности, предполагающая равномерное распределение значений, выглядит как $\min\left(\frac{1}{nd_1}, \frac{1}{nd_2}\right)$, где nd_1 — число уникальных значений ключа соединения в первом наборе строк, а nd_2 — во втором[3]. *с. 351*

Статистика по количеству уникальных значений показывает, что в таблице tickets номера билетов уникальны (что естественно, поскольку столбец ticket_no является первичным ключом), а в таблице ticket_flights на каждый билет в среднем приходится примерно три строки:

[1] backend/optimizer/path/costsize.c, функция get_foreign_key_join_selectivity.
[2] backend/optimizer/path/clausesel.c, функция clauselist_selectivity.
[3] backend/utils/adt/selfuncs.c, функция eqjoinsel.

```
=> SELECT t.n_distinct, tf.n_distinct
FROM pg_stats t, pg_stats tf
WHERE t.tablename = 'tickets' AND t.attname = 'ticket_no'
  AND tf.tablename = 'ticket_flights' AND tf.attname = 'ticket_no';
 n_distinct | n_distinct
------------+-------------
         -1 | -0.30457819
(1 row)
```

В итоге оценка совпала бы с оценкой на основе внешнего ключа:

```
=> SELECT round(2 * tf.reltuples *
  least(1.0/t.reltuples, 1.0/tf.reltuples/0.30457819)
) AS rows
FROM pg_class t, pg_class tf
WHERE t.relname = 'tickets' AND tf.relname = 'ticket_flights';
 rows
------
    6
(1 row)
```

Планировщик старается по возможности уточнить базовую оценку. В настоящее время он не использует гистограммы, но учитывает списки наиболее
с. 353 частых значений, если такая статистика собрана по ключу соединения для обеих таблиц[1]. В этом случае можно относительно точно рассчитать селективность соединения той части строк, которая попадает в списки, и только оставшуюся часть оценивать исходя из равномерного распределения.

Тем не менее в общем случае оценка селективности соединения в отсутствие внешнего ключа может оказаться хуже оценки при его наличии. Особенно велик риск получить сильно заниженную оценку для составных ключей соединения.

С помощью команды EXPLAIN ANALYZE можно посмотреть не только реальное число строк, но и количество обращений к внутреннему циклу:

```
=> EXPLAIN (analyze, timing off, summary off, buffers off) SELECT *
FROM tickets t
  JOIN ticket_flights tf ON tf.ticket_no = t.ticket_no
WHERE t.book_ref = '03A76D';
```

[1] backend/utils/adt/selfuncs.c, функция eqjoinsel.

```
                                QUERY PLAN
-----------------------------------------------------------------------
 Nested Loop  (cost=0.99..45.67 rows=6 width=136)
   (actual rows=8.00 loops=1)
   -> Index Scan using tickets_book_ref_idx on tickets t
       (cost=0.43..12.46 rows=2 width=104) (actual rows=2.00 loop...
       Index Cond: (book_ref = '03A76D'::bpchar)
       Index Searches: 1
   -> Index Scan using ticket_flights_pkey on ticket_flights tf
       (cost=0.56..16.57 rows=3 width=32) (actual rows=4.00 loops=2)
       Index Cond: (ticket_no = t.ticket_no)
       Index Searches: 2
(10 rows)
```

Оценка подтвердилась: во внешнем наборе оказалось две строки (actual rows=2). Внутренний узел Index Scan выполнялся поэтому два раза (loops=2) и каждый раз выбирал в среднем четыре строки (actual rows=4). Отсюда общее количество найденных строк: actual rows=8.

> Я выключаю вывод времени выполнения каждого шага плана (`timing off`), чтобы не увеличивать ширину вывода, сильно ограниченную размером книжной страницы; к тому же на некоторых платформах такой вывод может существенно замедлять выполнение запроса. Но если время все-таки оставить, выведенное значение тоже будет усредненным, как и количество строк. Чтобы получить полное время, среднее надо умножить на число итераций (loops). *с. 166*

Оценка стоимости. Стоимость рассчитывается так же, как в уже рассмотренных примерах. Повторное сканирование внутреннего набора строк в данном случае не отличается по стоимости от первого.

В итоге получаем:

```
=> SELECT 0.43 + 0.56 AS startup_cost,
  round((
    12.46 + 2 * 16.58 +
    6 * current_setting('cpu_tuple_cost')::real
  )::numeric, 2) AS total_cost;
 startup_cost | total_cost
--------------+------------
         0.99 |      45.68
(1 row)
```

v. 14 Кеширование (мемоизация) строк

Если повторное сканирование внутреннего набора строк часто выполняется с одними и теми же значениями параметра и, соответственно, приводит к одним и тем же результатам, может оказаться выгодным закешировать строки внутреннего набора.

Такую функцию выполняет узел Memoize[1]. Он схож с узлом материализации Materialize, но рассчитан на параметризованное соединение и устроен значительно сложнее:

- узел Materialize просто материализует все строки дочернего узла, а Memoize отдельно запоминает строки для каждого значения параметра;
- при переполнении хранилище строк узла Materialize начинает сбрасывать данные на диск, а хранилище узла Memoize — нет (это лишило бы кеширование всякого смысла).

Вот пример плана запроса, который использует узел Memoize:

```
=> EXPLAIN SELECT *
FROM flights f
  JOIN aircrafts_data a ON f.aircraft_code = a.aircraft_code
WHERE f.flight_no = 'PG0003';
                                QUERY PLAN
------------------------------------------------------------------------
 Nested Loop  (cost=5.44..387.10 rows=113 width=135)
   -> Bitmap Heap Scan on flights f
       (cost=5.30..382.22 rows=113 width=63)
       Recheck Cond: (flight_no = 'PG0003'::bpchar)
       -> Bitmap Index Scan on flights_flight_no_scheduled_depart...
           (cost=0.00..5.27 rows=113 width=0)
           Index Cond: (flight_no = 'PG0003'::bpchar)
   -> Memoize  (cost=0.15..0.27 rows=1 width=72)
       Cache Key: f.aircraft_code
       Cache Mode: logical
       -> Index Scan using aircrafts_pkey on aircrafts_data a
           (cost=0.14..0.26 rows=1 width=72)
           Index Cond: (aircraft_code = f.aircraft_code)
(13 rows)
```

[1] backend/executor/nodeMemoize.c.

Для кеширования строк выделяется память процесса размером *work_mem* × 4MB
× *hash_mem_multiplier*. Как следует из названия второго параметра, внутри 2.0
для поиска строк используется хеш-таблица (вариант с открытой адресацией [1]). Ключом кеширования (Cache Key) служит значение параметра (или набор значений параметров, если их несколько). В логическом режиме (Cache Mode: logical) ключи сравниваются по значению (обычным оператором равенства), а в двоичном — побитово. *с. 520*

Кроме того, все ключи кеширования связаны в список, один конец которого считается «холодным» (давно не использовавшиеся ключи), а другой — «горячим» (ключи, использованные недавно).

Если при обращении к узлу Memoize оказывается, что строки, соответствующие переданным значениям параметров, находятся в кеше, они возвращаются родительскому узлу (Nested Loop) без обращения к дочернему узлу. Использованный ключ кеширования передвигается в горячий конец списка.

Если же в кеше нет нужных строк, узел Memoize получает строки от дочернего узла, сохраняет их в кеше и возвращает узлу выше. Новый ключ кеширования также становится горячим.

Пока кеш заполняется новыми данными, доступная память может исчерпаться. Чтобы освободить ее, из кеша удаляются строки, соответствующие холодным ключам. Этот алгоритм вытеснения отличается от того, что используется в буферном кеше, но выполняет ту же задачу. *с. 196*

Может оказаться, что каким-то значениям параметров соответствует так много строк, что они не помещаются полностью в кеш, даже если все остальные строки уже вытеснены. Тогда такие параметры пропускаются — нет смысла запоминать лишь часть строк, поскольку в следующий раз все равно придется обращаться к дочернему узлу за полной выборкой.

Оценки кардинальности и стоимости. Вычисление оценок ничем радикально не отличается от того, что мы уже видели выше. Однако следует учесть, что

[1] include/lib/simplehash.h.

стоимость узла Memoize, показанная в плане, не имеет ничего общего с реальностью: это просто стоимость дочернего узла, увеличенная на значение
0.01 *cpu_tuple_cost*[1].

Но чтобы узел Memoize имел смысл, его стоимость должна быть не больше, а наоборот, меньше стоимости дочернего узла. С похожей ситуацией мы уже сталкивались на примере узла Materialize: «настоящая» стоимость вычисляется для *повторного сканирования* узла[2] и в плане не отображается.

Стоимость повторного сканирования узла Memoize учитывает размер памяти, доступной для кеширования, и предполагаемый характер обращений к кешу. Расчет очень сильно зависит от точности оценки количества различных значений параметров, с которыми будет сканироваться внутренний набор строк[3]. Получив ее, можно прикинуть вероятность обнаружения строки в кеше и вероятность вытеснения строк из кеша. Ожидаемые попадания в кеш уменьшают оценку стоимости, а потенциальные вытеснения — наоборот, увеличивают. В детали вычисления стоимости я вдаваться не буду.

Разобраться в том, что на самом деле происходит при выполнении запроса, как обычно, помогает команда `EXPLAIN ANALYZE`.

В нашем примере выбираются рейсы по одному маршруту, который обслуживается одним типом самолета; поэтому ключ хеширования совпадает для всех обращений к узлу Memoize. Первый раз за нужной строкой приходится сходить в таблицу (Misses: 1), но все повторные обращения обслуживаются кешем (Hits: 112). На все про все хватило одного килобайта памяти.

Еще два выведенных значения равны нулю: количество вытеснений из кеша (Evictions) и количество переполнений памяти с невозможностью сохранить все строки, относящиеся к одному набору параметров (Overflows). Большие цифры говорили бы о том, что выделенной под кеш памяти оказалось недостаточно, скорее всего, из-за некорректной оценки количества разных значений параметров. В таких условиях применение узла Memoize может оказаться весьма затратным. В крайнем случае можно запретить планировщи-
on ку использовать кеш, отключив параметр *enable_memoize*.

[1] backend/optimizer/util/pathnode.c, функция create_memoize_path.
[2] backend/optimizer/path/costsize.c, функция cost_memoize_rescan.
[3] backend/utils/adt/selfuncs.c, функция estimate_num_groups.

```
=> EXPLAIN (analyze, costs off, timing off, summary off, buffers off)
SELECT * FROM flights f
  JOIN aircrafts_data a ON f.aircraft_code = a.aircraft_code
WHERE f.flight_no = 'PG0003';
                              QUERY PLAN
-----------------------------------------------------------------------
 Nested Loop (actual rows=113.00 loops=1)
   -> Bitmap Heap Scan on flights f
       (actual rows=113.00 loops=1)
       Recheck Cond: (flight_no = 'PG0003'::bpchar)
       Heap Blocks: exact=2
       -> Bitmap Index Scan on flights_flight_no_scheduled_depart...
           (actual rows=113.00 loops=1)
           Index Cond: (flight_no = 'PG0003'::bpchar)
           Index Searches: 1
   -> Memoize (actual rows=1.00 loops=113)
       Cache Key: f.aircraft_code
       Cache Mode: logical
       Hits: 112  Misses: 1  Evictions: 0  Overflows: 0  Memory
       Usage: 1kB
       -> Index Scan using aircrafts_pkey on aircrafts_data a
           (actual rows=1.00 loops=1)
           Index Cond: (aircraft_code = f.aircraft_code)
           Index Searches: 1
(18 rows)
```

Внешние соединения

Соединение вложенным циклом может применяться для *левого внешнего соединения* (узел Nested Loop Left Join). В этом примере планировщик выбрал непараметризованное соединение:

```
=> EXPLAIN SELECT * FROM ticket_flights tf
  LEFT JOIN boarding_passes bp ON bp.ticket_no = tf.ticket_no
                              AND bp.flight_id = tf.flight_id
WHERE tf.ticket_no = '0005434026720' and tf.flight_id = 82977;
                              QUERY PLAN
-----------------------------------------------------------------------
 Nested Loop Left Join  (cost=1.12..17.17 rows=1 width=57)
   -> Index Scan using ticket_flights_pkey on ticket_flights tf
       (cost=0.56..8.58 rows=1 width=32)
       Index Cond: ((ticket_no = '0005434026720'::bpchar) AND (fl...
   -> Index Scan using boarding_passes_pkey on boarding_passes bp
       (cost=0.56..8.58 rows=1 width=25)
       Index Cond: ((ticket_no = '0005434026720'::bpchar) AND (fl...
(7 rows)
```

Кардинальность внешнего соединения оценивается так же, как и кардинальность внутреннего, но в качестве результата берется максимум из полученной оценки и кардинальности внешнего набора строк[1]. Иными словами, внешнее соединение никогда не уменьшает количество строк (но увеличить может).

Стоимость оценивается аналогично внутреннему соединению.

Правое соединение не поддерживается[2], поскольку для алгоритма вложенного цикла внешний и внутренний наборы строк неравнозначны. Внешний набор строк просматривается полностью, а из внутреннего при индексном доступе читаются только строки, удовлетворяющие условию соединения. При этом часть строк может остаться непросмотренной.

Полное соединение не поддерживается по тем же соображениям.

Анти- и полусоединения

Антисоединения и полусоединения похожи тем, что для каждой строки первого (внешнего) набора во втором (внутреннем) наборе достаточно найти лишь *одну* подходящую строку.

Антисоединение возвращает строки первого набора, если только для них не нашлось соответствия в другом наборе. Иными словами, если во втором наборе нашлась одна подходящая строка, строка из первого набора уже не попадет в результат, и дальше можно не проверять.

Антисоединение может использоваться для вычисления предиката `NOT EXISTS`. Найдем, например, модели самолетов, для которых не задана конфигурация салона. Антисоединение вложенным циклом отображается в плане как узел Nested Loop Anti Join:

```
=> EXPLAIN SELECT *
FROM aircrafts a
WHERE NOT EXISTS (
  SELECT * FROM seats s WHERE s.aircraft_code = a.aircraft_code
);
```

[1] backend/optimizer/path/costsize.c, функция calc_joinrel_size_estimate.
[2] backend/optimizer/path/joinpath.c, функция match_unsorted_outer.

```
                              QUERY PLAN
-----------------------------------------------------------------------
 Nested Loop Anti Join  (cost=0.28..5.26 rows=1 width=40)
   -> Seq Scan on aircrafts_data ml  (cost=0.00..1.09 rows=9 widt...
   -> Index Only Scan using seats_pkey on seats s
       (cost=0.28..10.64 rows=149 width=4)
       Index Cond: (aircraft_code = ml.aircraft_code)
(5 rows)
```

Тот же план будет построен и для эквивалентного запроса без NOT EXISTS:

```
=> EXPLAIN SELECT a.*
FROM aircrafts a
  LEFT JOIN seats s ON a.aircraft_code = s.aircraft_code
WHERE s.aircraft_code IS NULL;
                              QUERY PLAN
-----------------------------------------------------------------------
 Nested Loop Anti Join  (cost=0.28..5.26 rows=1 width=40)
   -> Seq Scan on aircrafts_data ml  (cost=0.00..1.09 rows=9 widt...
   -> Index Only Scan using seats_pkey on seats s
       (cost=0.28..10.64 rows=149 width=4)
       Index Cond: (aircraft_code = ml.aircraft_code)
(5 rows)
```

Полусоединение возвращает те строки первого набора, для которых нашлось хотя бы одно соответствие во втором наборе (и снова последующие совпадения можно не проверять — результат уже известен).

Полусоединение может использоваться для вычисления предиката EXISTS. Найдем теперь модели самолетов, в салоне которых установлены кресла:

```
=> EXPLAIN SELECT *
FROM aircrafts a
WHERE EXISTS (
  SELECT * FROM seats s WHERE s.aircraft_code = a.aircraft_code
);
                              QUERY PLAN
-----------------------------------------------------------------------
 Nested Loop Semi Join  (cost=0.28..7.28 rows=9 width=40)
   -> Seq Scan on aircrafts_data ml  (cost=0.00..1.09 rows=9 widt...
   -> Index Only Scan using seats_pkey on seats s
       (cost=0.28..10.64 rows=149 width=4)
       Index Cond: (aircraft_code = ml.aircraft_code)
(5 rows)
```

Полусоединение вложенным циклом представлено узлом Nested Loop Semi Join. В этом плане (и в планах выше для антисоединения) для таблицы `seats` указана обычная оценка строк (rows=149), хотя на самом деле достаточно получить всего одну. При выполнении запроса цикл, конечно, останавливается после первой строки:

```
=> EXPLAIN (analyze, costs off, timing off, summary off, buffers off)
SELECT * FROM aircrafts a
WHERE EXISTS (
  SELECT * FROM seats s WHERE s.aircraft_code = a.aircraft_code
);
                          QUERY PLAN
---------------------------------------------------------------
 Nested Loop Semi Join (actual rows=9.00 loops=1)
   -> Seq Scan on aircrafts_data ml (actual rows=9.00 loops=1)
   -> Index Only Scan using seats_pkey on seats s
       (actual rows=1.00 loops=9)
       Index Cond: (aircraft_code = ml.aircraft_code)
       Heap Fetches: 9
       Index Searches: 9
(7 rows)
```

Оценка кардинальности. Для полусоединения дается обычным образом, но кардинальность внутреннего набора строк считается равной единице. А для антисоединения рассчитанная селективность вычитается из единицы, как для отрицания[1].

Оценка стоимости. Для анти- и полусоединений оценка стоимости учитывает, что второй набор читается не полностью, а только пока не будет найдена пара к строке первого набора[2].

Неэквисоединения

Алгоритм вложенного цикла позволяет соединять наборы строк по любому условию соединения.

[1] backend/optimizer/path/costsize.c, функция calc_joinrel_size_estimate.
[2] backend/optimizer/path/costsize.c, функция final_cost_nestloop.

Конечно, если внутренний набор строк является базовой таблицей, на этой таблице создан индекс и оператор условия соединения входит в класс операторов этого индекса, то к внутреннему набору строк возможен эффективный доступ. Но всегда остается вариант декартова произведения строк с фильтрацией по условию — и в этом случае условие может быть совершенно произвольным. Как, например, в этом запросе, выбирающем пары аэропортов, расположенных недалеко друг от друга:

с. 397

```
=> CREATE EXTENSION earthdistance CASCADE;
=> EXPLAIN (costs off) SELECT *
FROM airports a1
  JOIN airports a2 ON a1.airport_code != a2.airport_code
                  AND a1.coordinates <@> a2.coordinates < 100;
                              QUERY PLAN
-----------------------------------------------------------------------
 Nested Loop
   Join Filter: ((ml.airport_code <> ml_1.airport_code) AND
   ((ml.coordinates <@> ml_1.coordinates) < '100'::double precisi...
   -> Seq Scan on airports_data ml
   -> Materialize
       -> Seq Scan on airports_data ml_1
(6 rows)
```

Параллельный режим

Соединение вложенным циклом может использоваться в параллельном режиме[1].

с. 379

Распараллеливание происходит только на уровне внешнего набора строк, который может одновременно читаться несколькими рабочими процессами. Получив очередную строку внешнего набора, каждый процесс затем сам перебирает соответствующие ей строки внутреннего набора — уже последовательно.

Ниже приведен пример запроса с несколькими соединениями, который находит пассажиров, купивших билеты на определенный рейс:

[1] backend/optimizer/path/joinpath.c, функция consider_parallel_nestloop.

```
=> EXPLAIN (costs off) SELECT t.passenger_name
FROM tickets t
  JOIN ticket_flights tf ON tf.ticket_no = t.ticket_no
  JOIN flights f ON f.flight_id = tf.flight_id
WHERE f.flight_id = 12345;
                              QUERY PLAN
-----------------------------------------------------------------------
 Gather
   Workers Planned: 2
   -> Nested Loop
       -> Nested Loop
           -> Parallel Seq Scan on ticket_flights tf
               Filter: (flight_id = 12345)
           -> Index Scan using tickets_pkey on tickets t
               Index Cond: (ticket_no = tf.ticket_no)
       -> Materialize
           -> Index Only Scan using flights_flight_id_status_idx ...
               Index Cond: (flight_id = 12345)
(11 rows)
```

Для получения внутреннего набора используется параллельный план. Каж-
с. 380 дый из процессов читает свои строки таблицы перелетов `ticket_flights` и соединяет их вложенным циклом с билетами `tickets`.

Затем каждый процесс соединяет полученный набор строк с одной строкой таблицы рейсов `flights`, полученной по уникальному ключу и материализо-
v. 18 ванной в узле Materialize, поэтому вложенный цикл оправдан даже для большого внутреннего набора строк.

22

Хеширование

22.1. Соединение хешированием

Однопроходное соединение хешированием

Идея соединения хешированием состоит в поиске подходящих строк с помощью заранее подготовленной хеш-таблицы. Вот пример плана, использующего такое соединение:

```
=> EXPLAIN (costs off) SELECT *
FROM tickets t
  JOIN ticket_flights tf ON tf.ticket_no = t.ticket_no;
               QUERY PLAN
-----------------------------------------
 Hash Join
   Hash Cond: (tf.ticket_no = t.ticket_no)
   -> Seq Scan on ticket_flights tf
   -> Hash
       -> Seq Scan on tickets t
(5 rows)
```

На **первом этапе** узел Hash Join[1] обращается к узлу Hash[2]. Тот получает от своего дочернего узла весь внутренний набор строк и помещает его в *хеш-таблицу*.

Хеш-таблица позволяет сохранять пары, составленные из *ключа хеширования* и *значения*, а затем искать значения по ключу за фиксированное время, не зависящее от размера хеш-таблицы. Для этого ключи хеширования

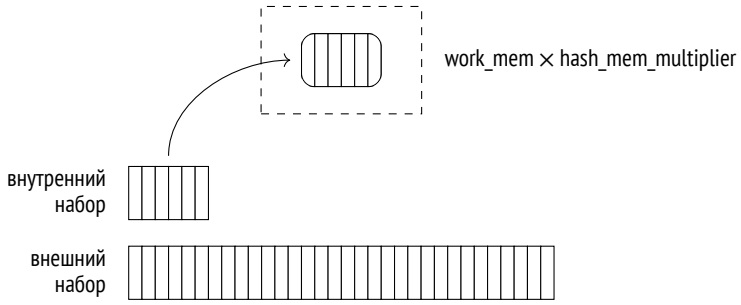
[1] backend/executor/nodeHashjoin.c.
[2] backend/executor/nodeHash.c.

распределяются более или менее равномерно по ограниченному количеству *корзин* (bucket). Номер корзины хеш-таблицы определяется значением *хеш-функции* от ключа хеширования; поскольку число корзин всегда является степенью двойки, из вычисленного значения берется нужное количество двоичных разрядов.

Реализация использует динамически расширяемую хеш-таблицу с разреше-
с. 190 нием коллизий с помощью цепочек[1], как для буферного кеша.

Итак, на первом этапе последовательно читаются строки внутреннего набора, и для каждой из них вычисляется хеш-функция. Ключом хеширования в данном случае являются поля, участвующие в условии соединения (Hash Cond), а в самой хеш-таблице сохраняются все поля строки из внутреннего набора, необходимые для запроса.

v. 13 Наиболее эффективно — за один проход по данным — соединение хе-
4MB шированием работает, если хеш-таблица целиком помещается в опера-
2.0 тивную память. Отведенный ей размер ограничен значением *work_mem* × × *hash_mem_multiplier*.



Вот пример, в котором запрос выполнен с помощью команды EXPLAIN ANALYZE, чтобы получить информацию об использовании памяти:

```
=> SET work_mem = '128MB';
=> EXPLAIN (analyze, costs off, timing off, summary off, buffers off)
SELECT *
FROM bookings b
  JOIN tickets t ON b.book_ref = t.book_ref;
```

[1] backend/utils/hash/dynahash.c.

```
                           QUERY PLAN
-----------------------------------------------------------------
 Hash Join (actual rows=2949857.00 loops=1)
   Hash Cond: (t.book_ref = b.book_ref)
   -> Seq Scan on tickets t (actual rows=2949857.00 loops=1)
   -> Hash (actual rows=2111110.00 loops=1)
       Buckets: 4194304  Batches: 1  Memory Usage: 145986kB
       -> Seq Scan on bookings b (actual rows=2111110.00 loops=1)
(6 rows)
```

В отличие от соединения вложенным циклом, для которого внутренний и внешний наборы существенно различаются, соединение на основе хеширования позволяет переставлять наборы местами. Как правило, в качестве внутреннего используется меньший набор, поскольку и хеш-таблица в таком случае получается меньше.

Здесь объема памяти хватило для размещения всей хеш-таблицы, которая занимает около 143 Мбайт (Memory Usage) и содержит $4\,\text{M} = 2^{22}$ корзин (Buckets). Поэтому соединение выполняется за один проход (Batches).

Однако если бы в запросе использовался один столбец, то для хеш-таблицы хватило бы 111 Мбайт:

```
=> EXPLAIN (analyze, costs off, timing off, summary off, buffers off)
SELECT b.book_ref
FROM bookings b
  JOIN tickets t ON b.book_ref = t.book_ref;
                           QUERY PLAN
-----------------------------------------------------------------
 Hash Join (actual rows=2949857.00 loops=1)
   Hash Cond: (t.book_ref = b.book_ref)
   -> Seq Scan on tickets t (actual rows=2949857.00 loops=1)
   -> Hash (actual rows=2111110.00 loops=1)
       Buckets: 4194304  Batches: 1  Memory Usage: 113172kB
       -> Seq Scan on bookings b (actual rows=2111110.00 loops=1)
(6 rows)
=> RESET work_mem;
```

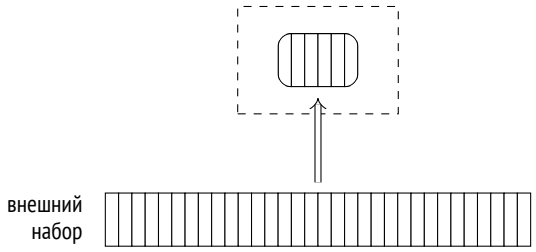
Это еще одна причина не использовать в запросах лишние поля, в том числе «звездочку».

Количество корзин хеш-таблицы выбирается так, чтобы в полностью заполненной данными таблице каждая корзина содержала в среднем одну строку. Более плотное заполнение повысило бы вероятность хеш-коллизий и, следовательно, снизило бы эффективность поиска, а более разреженная хеш-таблица слишком неэкономно расходовала бы память. Рассчитанное количество корзин увеличивается до первой подходящей степени двойки[1].

(Если, исходя из оценки средней «ширины» одной строки, размер хеш-таблицы с расчетным количеством корзин превышает ограничение по памяти, используется двухпроходное хеширование.)

Пока хеш-таблица не построена полностью, соединение хешированием не может начать возвращать результаты.

На **втором этапе** (хеш-таблица к этому моменту уже готова) узел Hash Join обращается ко второму дочернему узлу за внешним набором строк. Для каждой прочитанной строки проверяется наличие соответствующих ей строк в хеш-таблице. Для этого хеш-функция вычисляется от значений полей внешнего набора, входящих в условие соединения.



Найденные соответствия возвращаются вышестоящему узлу.

с. 444 **Оценка стоимости.** Оценку кардинальности я уже рассматривал, и она не зависит от способа соединения, поэтому дальше я буду говорить только об оценке стоимости.

[1] backend/executor/nodeHash.c, функция ExecChooseHashTableSize.

В качестве стоимости узла Hash берется полная стоимость его дочернего узла. Это фиктивная цифра, просто чтобы было что показать в плане запроса[1]. Все реальные оценки включены в стоимость узла Hash Join[2].

Рассмотрим пример:

```
=> EXPLAIN (analyze, timing off, summary off, buffers off)
SELECT * FROM flights f
  JOIN seats s ON s.aircraft_code = f.aircraft_code;
                              QUERY PLAN
-----------------------------------------------------------------------
 Hash Join  (cost=42.13..278511.28 rows=16518865 width=78)
   (actual rows=16518865.00 loops=1)
   Hash Cond: (f.aircraft_code = s.aircraft_code)
   -> Seq Scan on flights f  (cost=0.00..4772.67 rows=214867 widt...
       (actual rows=214867.00 loops=1)
   -> Hash  (cost=25.39..25.39 rows=1339 width=15)
       (actual rows=1339.00 loops=1)
       Buckets: 2048  Batches: 1  Memory Usage: 79kB
       -> Seq Scan on seats s  (cost=0.00..25.39 rows=1339 width=15)
           (actual rows=1339.00 loops=1)
(10 rows)
```

Начальная стоимость соединения отражает в основном создание хеш-таблицы и складывается:

- из полной стоимости получения внутреннего набора строк, который необходим для построения хеш-таблицы;
- стоимости вычисления хеш-функции от всех полей, входящих в ключ соединения, для каждой строки внутреннего набора (одна операция оценивается значением параметра *cpu_operator_cost*); 0.0025
- стоимости вставки всех строк внутреннего набора в хеш-таблицу (вставка отдельной строки оценивается значением параметра *cpu_tuple_cost*); 0.01
- начальной стоимости получения внешнего набора строк, без которого нельзя приступить к выполнению соединения.

[1] backend/optimizer/plan/createplan.c, функция create_hashjoin_plan.
[2] backend/optimizer/path/costsize.c, функции initial_cost_hashjoin и final_cost_hashjoin.

Полная стоимость добавляет к начальной оценке стоимость собственно соединения:

- стоимость вычисления хеш-функции от всех полей, входящих в ключ соединения, для каждой строки внешнего набора (*cpu_operator_cost*);
- стоимость перепроверок условий соединения, которые необходимы из-за возможных хеш-коллизий (вычисление каждого оператора оценивается значением параметра *cpu_operator_cost*);
- стоимость обработки каждой результирующей строки (*cpu_tuple_cost*).

Наиболее сложной частью оценки здесь является определение количества перепроверок, которые потребуются в ходе соединения. Оно оценивается произведением числа строк внешнего набора на некоторую долю числа строк внутреннего набора (находящегося в хеш-таблице). Оценка этой доли учитывает в том числе и возможное неравномерное распределение значений. Я не буду вдаваться в подробности[1]; в данном случае эта доля оценивается как 0,150112.

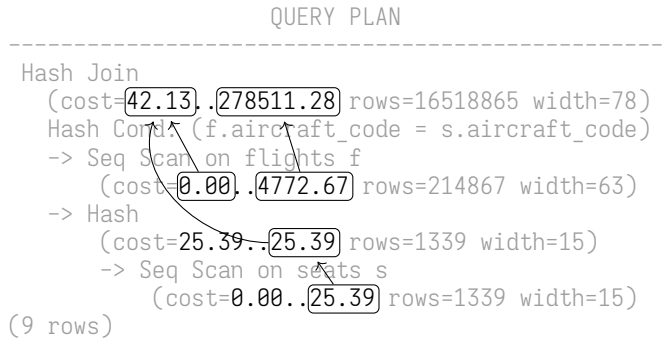
Итак, для нашего примера оценка вычисляется следующим образом:

```
=> WITH cost(startup) AS (
  SELECT round((
    21.39 +
    current_setting('cpu_operator_cost')::real * 1339 +
    current_setting('cpu_tuple_cost')::real * 1339 +
    0.00
  )::numeric, 2)
)
SELECT startup,
  startup + round((
    4772.67 +
    current_setting('cpu_operator_cost')::real * 214867 +
    current_setting('cpu_operator_cost')::real * 214867 * 1339 *
      0.150112 +
    current_setting('cpu_tuple_cost')::real * 16518865
  )::numeric, 2) AS total
FROM cost;
```

[1] backend/utils/adt/selfuncs.c, функция estimate_hash_bucket_stats.

```
 startup |   total
---------+-----------
   38.13 | 278507.26
(1 row)
```

Схему зависимостей расчета стоимости можно представить так:



Двухпроходное соединение хешированием

Если на этапе планирования оценки показывают, что хеш-таблица не поместится в отведенные рамки, внутренний набор строк разбивается на отдельные *пакеты* (batch), каждый из которых обрабатывается отдельно. Количество пакетов (как и корзин) всегда является степенью двойки; номер пакета определяется соответствующим количеством битов хеш-значения[1].

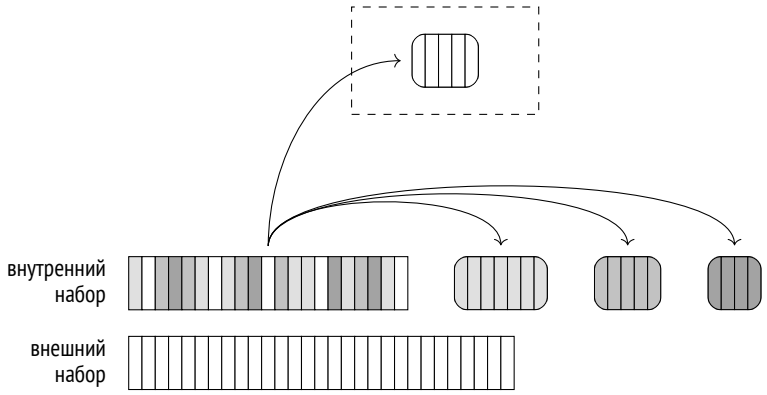
Любые две строки, соответствующие друг другу при соединении, принадлежат одному и тому же пакету, поскольку у строк из разных пакетов не могут совпасть хеш-коды.

К каждому пакету относится одинаковое количество хеш-значений. Если данные распределены равномерно, то и размеры всех пакетов будут примерно одинаковыми. Планировщик может управлять потреблением памяти, выбирая подходящее количество пакетов[2].

[1] backend/executor/nodeHash.c, функция ExecHashGetBucketAndBatch.
[2] backend/executor/nodeHash.c, функция ExecChooseHashTableSize.

На **первом этапе** выполнения читается внутренний набор строк и строится хеш-таблица. Если очередная строка внутреннего набора относится к первому пакету, она добавляется к хеш-таблице и остается в оперативной памяти. Если же строка относится к какому-либо другому пакету, она записывается во *временный файл* — свой для каждого из пакетов[1].

-1 Напомню, что объем используемых сеансом временных файлов на диске можно ограничить, установив предельное значение в параметре *temp_file_limit* (временные таблицы в это ограничение не входят). Если сеанс исчерпает ограничение, запрос будет аварийно прерван.

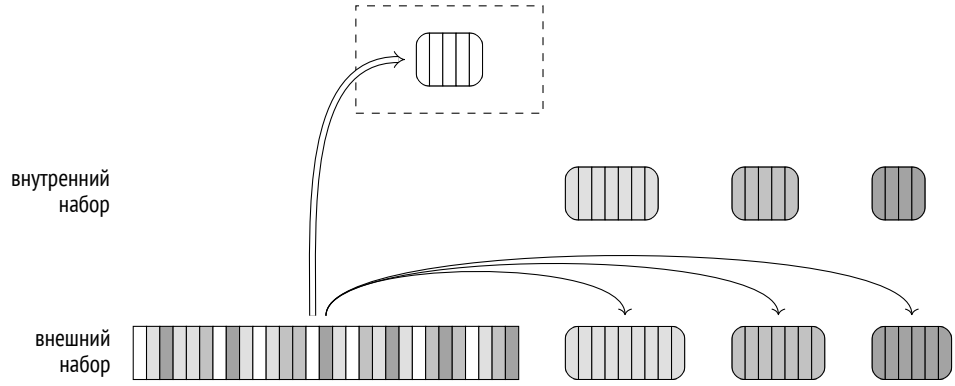


На **втором этапе** читается внешний набор строк. Если очередная строка принадлежит первому пакету, она сопоставляется с хеш-таблицей, которая как раз содержит строки первого пакета внутреннего набора (а в других пакетах соответствий быть не может).

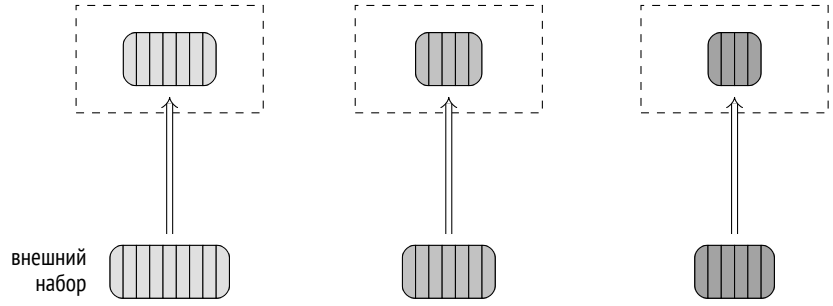
Если же строка принадлежит другому пакету, она сбрасывается во временный файл — опять же свой для каждого пакета. Таким образом, при N пакетах будет использоваться $2(N-1)$ файлов (или меньше, если некоторые пакеты окажутся пустыми).

После окончания второго этапа память, занимаемая хеш-таблицей, освобождается. На этот момент уже имеется частичный результат соединения по одному из имеющихся пакетов.

[1] backend/executor/nodeHash.c, функция ExecHashTableInsert.



Далее оба этапа повторяются поочередно для каждого из сохраненных на диск пакетов: из временного файла в хеш-таблицу переносятся строки внутреннего набора; строки внешнего набора, соответствующие этому же пакету, считываются из другого временного файла и сопоставляются с хеш-таблицей. Использованные временные файлы удаляются.



Двухпроходное соединение в выводе команды EXPLAIN отличается от однопроходного количеством пакетов, большим единицы. Кроме того, с ключевым словом buffers команда покажет статистику обмена с диском:

```
=> EXPLAIN (analyze, costs off, timing off, summary off)
SELECT *
FROM bookings b
  JOIN tickets t ON b.book_ref = t.book_ref;
```

```
                              QUERY PLAN
-----------------------------------------------------------------------
 Hash Join (actual rows=2949857.00 loops=1)
   Hash Cond: (t.book_ref = b.book_ref)
   Buffers: shared hit=12759 read=50169, temp read=54218 writ...
   -> Seq Scan on tickets t (actual rows=2949857.00 loops=1)
       Buffers: shared hit=385 read=49055
   -> Hash (actual rows=2111110.00 loops=1)
       Buckets: 131072  Batches: 32  Memory Usage: 4551kB
       Buffers: shared hit=12374 read=1114, temp written=10701
       -> Seq Scan on bookings b (actual rows=2111110.00 loop...
           Buffers: shared hit=12374 read=1114
(10 rows)
```

Я уже приводил этот пример выше, но с увеличенным значением *work_mem*. В отведенные по умолчанию 8 Мбайт вся хеш-таблица не помещается; здесь задействовано 32 пакета, хеш-таблица использует 128 K $= 2^{17}$ корзин. На этапе построения хеш-таблицы (узел Hash) выполняется запись во временные файлы; на этапе соединения (узел Hash Join) файлы и записываются, и читаются.

-1 Параметр *log_temp_files* позволяет получать более детальную информацию о временных файлах в журнале сообщений сервера. При нулевом значении в журнале будет отмечен каждый файл и его размер (на момент удаления).

Динамические корректировки плана

Запланированный ход событий могут нарушить две проблемы: некорректная статистика и неравномерное распределение.

При неравномерном распределении значений в столбцах, входящих в ключ соединения, разные пакеты будут иметь разное количество строк.

Если какой-нибудь пакет (кроме самого первого) окажется большим, все его строки придется сначала записать на диск, а затем прочитать с диска. В основном неприятность доставляет внешний набор данных, потому что обычно он больше. Поэтому если для внешнего набора строк доступна обыч-
с. 369 ная, не многовариантная статистика по наиболее частым значениям (то есть внешний набор представлен таблицей и соединение выполняется по одному столбцу), то строки с хеш-кодами, соответствующими нескольким наиболее

частым значениям, считаются принадлежащими первому пакету[1]. Эта оптимизация (skew optimization) позволяет несколько уменьшить ввод-вывод при двухпроходном соединении.

Обе проблемы могут привести к тому, что размер некоторых (или всех) пакетов окажется больше расчетного. Тогда хеш-таблица для них не поместится в запланированный размер и выйдет за рамки ограничений.

Поэтому если в процессе построения хеш-таблицы выясняется, что ее размер не укладывается в ограничения, количество пакетов увеличивается (удваивается) на лету. Фактически каждый пакет разделяется на два новых: примерно половина строк (если предполагать равномерное распределение) остается в хеш-таблице, а другая половина сбрасывается на диск в новый временный файл[2].

Это может произойти и в том случае, когда планировалось однопроходное соединение. По сути, одно- и двухпроходное соединения — один и тот же алгоритм, реализуемый одним и тем же кодом. Я разделяю их только для удобства изложения.

Количество пакетов может только увеличиваться. Если оказывается, что планировщик ошибся в бóльшую сторону, пакеты не объединяются.

Однако при неравномерном распределении увеличение числа пакетов может не помочь. Например, ключевой столбец может содержать *одно и то же* значение *во всех* строках: очевидно, что все они попадут в один пакет, поскольку хеш-функция будет возвращать одно и то же значение. Увы, в таком случае хеш-таблица будет просто расти, невзирая на значения ограничивающих параметров.

> Теоретически для такой ситуации можно было бы применить многопроходное соединение, рассматривая за один раз только часть пакета, но это не реализовано.

Для демонстрации динамического увеличения количества пакетов придется приложить некоторые усилия, чтобы обмануть планировщик. *с. 349*

[1] backend/executor/nodeHash.c, функция ExecHashBuildSkewHash.
[2] backend/executor/nodeHash.c, функция ExecHashIncreaseNumBatches.

```
=> CREATE TABLE bookings_copy (LIKE bookings INCLUDING INDEXES)
WITH (autovacuum_enabled = off);
=> INSERT INTO bookings_copy SELECT * FROM bookings;
INSERT 0 2111110
=> DELETE FROM bookings_copy WHERE random() < 0.9;
DELETE 1899898
=> ANALYZE bookings_copy;
=> INSERT INTO bookings_copy SELECT * FROM bookings
ON CONFLICT DO NOTHING;
INSERT 0 1899898
=> SELECT reltuples FROM pg_class WHERE relname = 'bookings_copy';
 reltuples
-----------
    211212
(1 row)
```

В результате этих манипуляций мы получили таблицу bookings_copy — полную копию bookings, но планировщик считает, что в ней в десять раза меньше строк, чем на самом деле. В реальности похожая ситуация может возникнуть, например, когда хеш-таблица строится по набору строк, полученному в результате другого соединения, для которого в таком случае нет достоверной статистики.

Из-за этой ошибки планировщик полагает, что будет достаточно 4 пакетов, но в процессе выполнения соединения их число возрастает до 16:

```
=> EXPLAIN (analyze, costs off, timing off, summary off, buffers off)
SELECT *
FROM bookings_copy b
  JOIN tickets t ON b.book_ref = t.book_ref;
                              QUERY PLAN
-----------------------------------------------------------------------
 Hash Join (actual rows=2949857.00 loops=1)
   Hash Cond: (t.book_ref = b.book_ref)
   -> Seq Scan on tickets t (actual rows=2949857.00 loops=1)
   -> Hash (actual rows=2111110.00 loops=1)
       Buckets: 131072 (originally 131072)  Batches: 16 (originally
       4)  Memory Usage: 8085kB
       -> Seq Scan on bookings_copy b (actual rows=2111110.00 loo...
(7 rows)
```

Оценка стоимости. Вот тот же пример, на котором я показывал расчет стоимости для однопроходного соединения, но теперь я предельно уменьшаю размер доступной памяти, и планировщик вынужден использовать два пакета. Стоимость соединения при этом увеличивается:

```
=> SET work_mem = '64kB';
=> SET hash_mem_multiplier = 1.0;
=> EXPLAIN (analyze, timing off, summary off, buffers off)
SELECT *
FROM flights f
  JOIN seats s ON s.aircraft_code = f.aircraft_code;
                              QUERY PLAN
-----------------------------------------------------------------------
 Hash Join  (cost=49.13..283143.28 rows=16518865 width=78)
   (actual rows=16518865.00 loops=1)
   Hash Cond: (f.aircraft_code = s.aircraft_code)
   -> Seq Scan on flights f  (cost=0.00..4772.67 rows=214867 widt...
       (actual rows=214867.00 loops=1)
   -> Hash  (cost=25.39..25.39 rows=1339 width=15)
       (actual rows=1339.00 loops=1)
       Buckets: 2048  Batches: 2  Memory Usage: 55kB
       -> Seq Scan on seats s  (cost=0.00..25.39 rows=1339 width=15)
           (actual rows=1339.00 loops=1)
(10 rows)
=> RESET work_mem;
=> RESET hash_mem_multiplier;
```

Расходы второго прохода связаны с записью строк во временные файлы и чтением их из файлов.

Начальная стоимость, рассчитанная для однопроходного соединения, увеличивается в этом случае на оценку записи такого количества страниц, которого хватит для сохранения нужных полей[1] *всех* строк внутреннего набора. Хотя первый пакет и не записывается на диск при построении хеш-таблицы, это не учитывается в оценке, поэтому она не зависит от количества пакетов.

Полная стоимость увеличивается на оценку чтения записанных ранее строк внутреннего набора и оценку записи и чтения строк внешнего набора.

Как запись, так и чтение одной страницы оцениваются значением параметра *seq_page_cost*, исходя из последовательного характера ввода-вывода.

[1] backend/optimizer/path/costsize.c, функция page_size.

В данном примере количество страниц для строк внутреннего набора оценено как 7, а для внешнего — как 2309. Добавив оценки к стоимости, которая была получена выше для однопроходного соединения, получаем цифры, совпадающие со стоимостью в плане запроса:

```
=> SELECT 38.13 + -- начальная стоимость однопроходного соединения
  current_setting('seq_page_cost')::real * 7
  AS startup,
278507.28 + -- полная стоимость однопроходного соединения
  current_setting('seq_page_cost')::real * 2 * (7 + 2309)
  AS total;
 startup |   total
---------+-----------
   45.13 | 283139.28
(1 row)
```

Таким образом, при нехватке оперативной памяти алгоритм соединения становится двухпроходным, и эффективность его падает. Поэтому важно, чтобы:

- в хеш-таблицу попадали только действительно нужные поля (обязанность автора запроса);
- хеш-таблица строилась по меньшему набору строк (обязанность планировщика).

Соединение хешированием в параллельных планах

Соединение хешированием может участвовать в параллельных планах в том виде, в котором я описывал его выше. Это означает, что сначала несколько параллельных процессов независимо друг от друга строят собственные (совершенно одинаковые) хеш-таблицы по внутреннему набору данных, а затем используют параллельный доступ к внешнему набору строк. Выигрыш здесь достигается за счет того, что каждый из процессов просматривает только часть внешнего набора.

Вот пример плана с участием обычного (в данном случае однопроходного) соединения хешированием:

```
=> SET work_mem = '128MB';
=> SET enable_parallel_hash = off;

=> EXPLAIN (analyze, costs off, timing off, summary off, buffers off)
SELECT count(*)
FROM bookings b
  JOIN tickets t ON t.book_ref = b.book_ref;
                              QUERY PLAN
-----------------------------------------------------------------------
 Finalize Aggregate (actual rows=1.00 loops=1)
   -> Gather (actual rows=3.00 loops=1)
       Workers Planned: 2
       Workers Launched: 2
       -> Partial Aggregate (actual rows=1.00 loops=3)
           -> Hash Join (actual rows=983285.67 loops=3)
               Hash Cond: (t.book_ref = b.book_ref)
               -> Parallel Seq Scan on tickets t (actual rows=983...
               -> Hash (actual rows=2111110.00 loops=3)
                   Buckets: 4194304  Batches: 1  Memory Usage:
                   113172kB
                   -> Seq Scan on bookings b (actual rows=2111110...
(12 rows)

=> RESET enable_parallel_hash;
```

Здесь каждый процесс хеширует таблицу bookings, а затем сопоставляет с хеш-таблицей свою часть строк, полученную параллельным сканированием (Parallel Seq Scan).

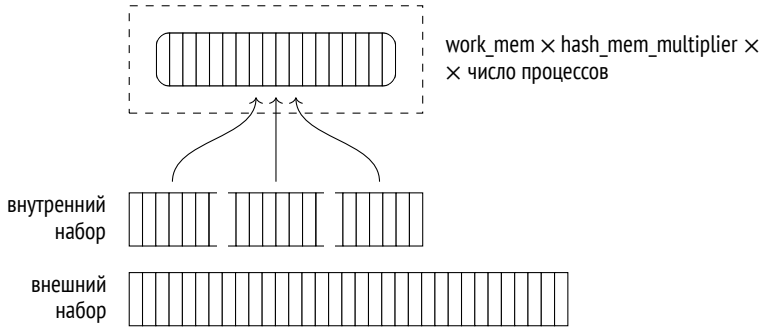
Ограничение на память под хеш-таблицу применяется к каждому параллельному процессу, так что суммарно будет выделено в три раза больше памяти, чем это указано в плане (Memory Usage).

Параллельное однопроходное хеш-соединение

Несмотря на то что и обычное соединение хешированием может давать определенную выгоду в параллельных планах (особенно в случае небольшого внутреннего набора, который нет смысла обрабатывать параллельно), для больших наборов данных лучше работает специальный параллельный алгоритм хеш-соединения.

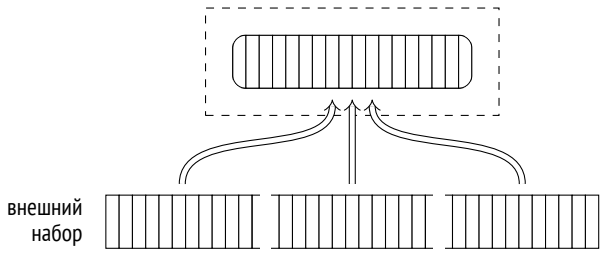
Важное отличие от непараллельной версии алгоритма состоит в том, что хеш-таблица создается не в локальной памяти процесса, а в *общей* динамически выделяемой памяти и доступна каждому параллельному процессу, участвующему в соединении. Это позволяет вместо нескольких отдельных хеш-таблиц создать одну общую, используя суммарный объем памяти, выделяемый всем процессам-участникам. Благодаря этому увеличивается вероятность выполнения соединения за один проход.

На **первом этапе**, представляемом в плане узлом Parallel Hash, все параллельные процессы строят общую хеш-таблицу, используя параллельный доступ к внутреннему набору строк[1].



Чтобы можно было двигаться дальше, каждый из параллельных процессов должен завершить свою часть первого этапа[2].

На **втором этапе** (узел Parallel Hash Join), когда хеш-таблица построена, каждый процесс сопоставляет с ней свою часть строк внешнего набора, используя параллельный доступ[3].



[1] backend/executor/nodeHash.c, функция MultiExecParallelHash.
[2] backend/storage/ipc/barrier.c.
[3] backend/executor/nodeHashjoin.c, функция ExecParallelHashJoin.

Вот пример такого плана:

```
=> SET work_mem = '64MB';

=> EXPLAIN (analyze, costs off, timing off, summary off, buffers off)
SELECT count(*)
FROM bookings b
  JOIN tickets t ON t.book_ref = b.book_ref;
                              QUERY PLAN
-----------------------------------------------------------------------
 Finalize Aggregate (actual rows=1.00 loops=1)
   -> Gather (actual rows=3.00 loops=1)
       Workers Planned: 2
       Workers Launched: 2
       -> Partial Aggregate (actual rows=1.00 loops=3)
           -> Parallel Hash Join (actual rows=983285.67 loops=3)
               Hash Cond: (t.book_ref = b.book_ref)
               -> Parallel Seq Scan on tickets t (actual rows=983...
               -> Parallel Hash (actual rows=703703.33 loops=3)
                   Buckets: 4194304  Batches: 1  Memory Usage:
                   115424kB
                   -> Parallel Seq Scan on bookings b (actual row...
(12 rows)

=> RESET work_mem;
```

Это тот же запрос, что я показывал в предыдущем разделе, но там параллельная версия хеш-соединения была специально отключена параметром *enable_parallel_hash*. on

Несмотря на то что я уменьшил объем памяти под хеш-таблицу вдвое по сравнению с обычным хеш-соединением из предыдущего раздела, соединение осталось однопроходным за счет совместного использования памяти всех параллельных процессов (Memory Usage). Хеш-таблица немного выросла, но существует теперь в единственном экземпляре, так что суммарное использование памяти уменьшилось.

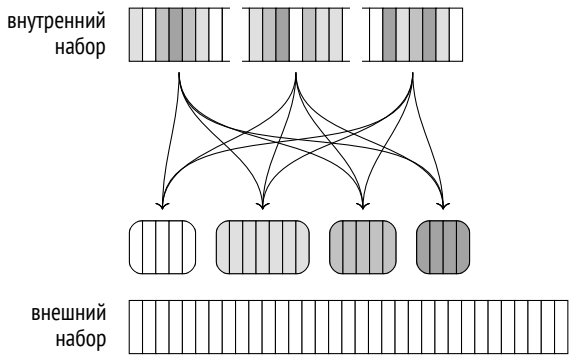
Параллельное двухпроходное хеш-соединение

Даже совместной памяти всех параллельных процессов может не хватить для размещения всей хеш-таблицы. Это может стать понятно как на этапе

планирования, так и позже, во время выполнения. В таком случае используется двухпроходный алгоритм, который существенно отличается от всех уже рассмотренных.

Важное отличие состоит в том, что используется не одна большая общая хеш-таблица, а несколько таблиц меньшего размера. Каждый процесс работает со своей таблицей и обрабатывает пакеты независимо от других процессов. (Однако отдельные хеш-таблицы тоже располагаются в общей памяти, так что доступ к ним может получить любой процесс.) Если уже на этапе планирования становится ясно, что одним пакетом не обойтись[1], для каждого процесса сразу создается своя хеш-таблица. Если решение принимается во время выполнения, таблица перестраивается[2].

Итак, **на первом этапе** процессы параллельно читают внутренний набор строк, разделяя его на пакеты и записывая эти пакеты во временные файлы[3]. Поскольку каждый процесс читает только свою часть строк внутреннего набора, ни один из них не построит полную хеш-таблицу ни для одного из пакетов (даже для первого). Полный набор строк любого пакета собирается только в файле, запись в который ведут все параллельные процессы, синхронизируясь друг с другом[4]. Поэтому, в отличие от непараллельной версии алгоритма и от параллельной однопроходной версии, в данном случае на диск сбрасываются все пакеты, в том числе и первый.

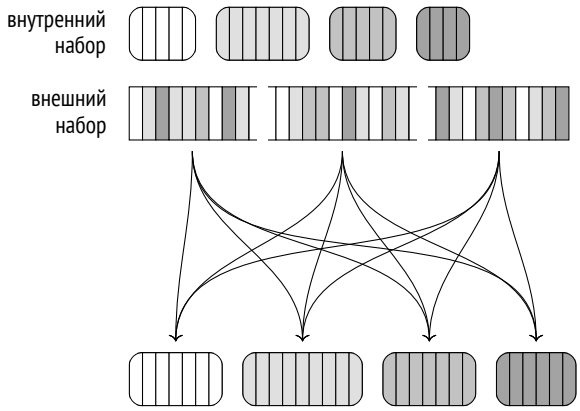


[1] backend/executor/nodeHash.c, функция ExecChooseHashTableSize.
[2] backend/executor/nodeHash.c, функция ExecParallelHashIncreaseNumBatches.
[3] backend/executor/nodeHash.c, функция MultiExecParallelHash.
[4] backend/utils/sort/sharedtuplestore.c.

Когда все процессы закончили хеширование внутреннего набора, начинается **второй этап**[1].

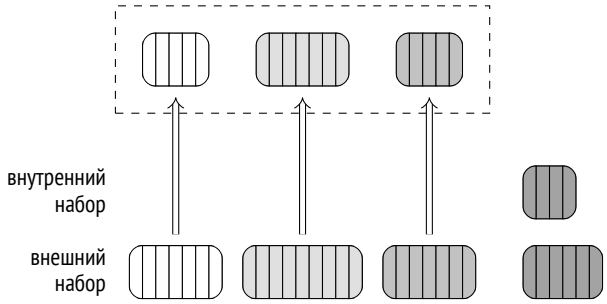
В случае непараллельной версии алгоритма строки внешнего набора, относящиеся к первому пакету, сразу же сопоставлялись бы с хеш-таблицей. Но в параллельной версии готовой хеш-таблицы еще нет в памяти, и пакеты обрабатываются процессами независимо. Поэтому в начале второго этапа внешний набор строк читается параллельно, распределяется по пакетам, и каждый пакет записывается в свой временный файл[2]. Прочитанные строки не попадают в хеш-таблицу (как это происходит на первом этапе), так что увеличение количества пакетов невозможно.

Когда все процессы закончили чтение внешнего набора данных, на диске оказывается 2*N* временных файлов, содержащих пакеты внутреннего и внешнего наборов.

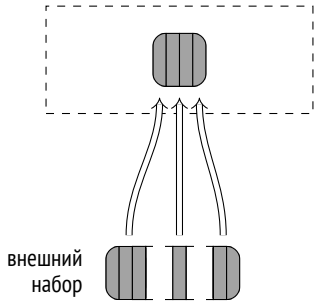


Затем каждый процесс выбирает один из пакетов и выполняет соединение: загружает внутренний набор строк в свою хеш-таблицу в памяти, читает строки внешнего набора и сопоставляет их со строками в хеш-таблице. Завершив обработку одного пакета, процесс выбирает следующий, еще не обработанный[3].

[1] backend/executor/nodeHashjoin.c, функция ExecParallelHashJoin.
[2] backend/executor/nodeHashjoin.c, функция ExecParallelHashJoinPartitionOuter.
[3] backend/executor/nodeHashjoin.c, функция ExecParallelHashJoinNewBatch.



Когда необработанные пакеты заканчиваются, освободившийся процесс подключается к обработке одного из еще не завершенных пакетов, пользуясь тем, что все хеш-таблицы находятся в разделяемой памяти.



Такая схема работает лучше, чем одна большая хеш-таблица, общая для всех процессов: проще организовать совместную работу, меньше ресурсов тратится на синхронизацию.

Модификации

Алгоритм соединения хешированием может использоваться не только для внутренних, но и для любых других типов соединений: левых, правых и полных внешних соединений, для полу- и антисоединений. Однако, повторюсь, в качестве условия соединения допускается только равенство.

с. 451 Часть операций я уже показывал, когда говорил о соединении вложенным циклом. Вот пример *правого внешнего соединения*:

```
=> EXPLAIN (costs off) SELECT *
FROM bookings b
  LEFT OUTER JOIN tickets t ON t.book_ref = b.book_ref;
               QUERY PLAN
----------------------------------------
 Hash Right Join
   Hash Cond: (t.book_ref = b.book_ref)
   -> Seq Scan on tickets t
   -> Hash
        -> Seq Scan on bookings b
(5 rows)
```

Обратите внимание, как логическая операция левого соединения в SQL-запросе превратилась в физическую операцию правого соединения в плане выполнения.

На логическом уровне внешней таблицей (стоящей слева от операции соединения) являются бронирования (bookings), а внутренней — таблица билетов (tickets). Поэтому в результат соединения должны попасть в том числе и бронирования без билетов.

На физическом уровне внешний и внутренний наборы данных определяются не по положению в тексте запроса, а исходя из стоимости соединения. Обычно это означает, что внутренним набором будет тот, чья хеш-таблица меньше. Так происходит и здесь: в качестве внутреннего набора выступает таблица бронирований, и вид соединения меняется с левого на правый.

И наоборот, если в запросе указать правое внешнее соединение (желая вывести билеты, не связанные с бронированиями), то в плане выполнения соединение поменяется на левое:

```
=> EXPLAIN (costs off) SELECT *
FROM bookings b
  RIGHT OUTER JOIN tickets t ON t.book_ref = b.book_ref;
               QUERY PLAN
----------------------------------------
 Hash Left Join
   Hash Cond: (t.book_ref = b.book_ref)
   -> Seq Scan on tickets t
   -> Hash
        -> Seq Scan on bookings b
(5 rows)
```

Для полноты картины — пример плана с полным внешним соединением:

```
=> EXPLAIN (costs off)
SELECT *
FROM bookings b
  FULL OUTER JOIN tickets t ON t.book_ref = b.book_ref;
               QUERY PLAN
---------------------------------------
 Hash Full Join
   Hash Cond: (t.book_ref = b.book_ref)
   -> Seq Scan on tickets t
   -> Hash
       -> Seq Scan on bookings b
(5 rows)
```

Параллельное соединение хешированием поддерживается для всех видов соединений: не только для внутренних и внешних левых, но также и для
v. 16 правых и полных. В следующем примере планировщик отдает предпочтение правому соединению, чтобы построить хеш-таблицу меньшего размера:

```
=> EXPLAIN (costs off)
SELECT sum(b.total_amount)
FROM bookings b
  LEFT OUTER JOIN tickets t ON t.book_ref = b.book_ref;
                      QUERY PLAN
------------------------------------------------------
 Finalize Aggregate
   -> Gather
       Workers Planned: 2
       -> Partial Aggregate
           -> Parallel Hash Right Join
               Hash Cond: (t.book_ref = b.book_ref)
               -> Parallel Seq Scan on tickets t
               -> Parallel Hash
                   -> Parallel Seq Scan on bookings b
(9 rows)
```

22.2. Группировка и уникальные значения

Группировка значений для агрегации и устранение дубликатов могут выполняться алгоритмами, схожими с алгоритмами соединения. Один из способов состоит в том, чтобы построить хеш-таблицу по нужным столбцам.

Значения помещаются в хеш-таблицу, только если они отсутствуют в ней. Таким образом, в конечном итоге в хеш-таблице собираются все уникальные значения.

Узел, который отвечает за агрегацию методом хеширования, обозначается в плане выполнения как HashAggregate[1].

Приведу несколько примеров ситуаций, в которых может использоваться такой узел.

Количество мест для каждого класса обслуживания (GROUP BY):

```
=> EXPLAIN (costs off)
SELECT fare_conditions, count(*)
FROM seats
GROUP BY fare_conditions;
          QUERY PLAN
------------------------------
 HashAggregate
   Group Key: fare_conditions
   -> Seq Scan on seats
(3 rows)
```

Список классов обслуживания (DISTINCT):

```
=> EXPLAIN (costs off)
SELECT DISTINCT fare_conditions
FROM seats;
          QUERY PLAN
------------------------------
 HashAggregate
   Group Key: fare_conditions
   -> Seq Scan on seats
(3 rows)
```

Классы обслуживания и еще одно значение (UNION):

```
=> EXPLAIN (costs off) SELECT fare_conditions
FROM seats
UNION
SELECT NULL;
```

[1] backend/executor/nodeAgg.c.

```
                QUERY PLAN
-----------------------------------
 HashAggregate
   Group Key: seats.fare_conditions
   -> Append
       -> Seq Scan on seats
       -> Result
(5 rows)
```

Узел Append соответствует объединению двух наборов строк, но не удаляет дубликаты, как того требует операция UNION. Удаление выполняется отдельно узлом HashAggregate.

4MB
2.0

Память, выделяемая под хеш-таблицу, ограничена значением *work_mem* × × *hash_mem_multiplier*, как и в случае хеш-соединения.

Если хеш-таблица помещается в отведенную память, агрегация выполняется за один проход по набору строк, как в этом примере:

```
=> EXPLAIN (analyze, costs off, timing off, summary off, buffers off)
SELECT DISTINCT amount FROM ticket_flights;
                              QUERY PLAN
-----------------------------------------------------------------------
 HashAggregate (actual rows=338.00 loops=1)
   Group Key: amount
   Batches: 1  Memory Usage: 56kB
   -> Seq Scan on ticket_flights (actual rows=8391852.00 loops=1)
(4 rows)
```

Уникальных значений стоимости не так много, поэтому хеш-таблица заняла всего 45 Кбайт (Memory Usage).

v. 13

Как только во время построения хеш-таблицы новые значения перестают помещаться в отведенный объем, они сбрасываются во временные файлы, распределяясь по разделам на основании нескольких битов хеш-значения. Количество разделов является степенью двойки и выбирается планировщиком так, чтобы хеш-таблица для каждого из них поместилась целиком в оперативную память. Конечно, эта оценка зависит от качества статистики, поэтому расчетное значение умножается на полтора, чтобы дополнительно уменьшить размер разделов и увеличить вероятность того, что каждый из них можно будет обработать за один раз[1].

[1] backend/executor/nodeAgg.c, функция hash_choose_num_partitions.

После того как весь набор данных прочитан, узел возвращает результаты агрегации по тем значениям, которые попали в хеш-таблицу.

Затем хеш-таблица очищается, и каждый из разделов, записанных на предыдущем шаге во временные файлы, читается и обрабатывается точно так же, как обычный набор строк. При неудачном стечении обстоятельств хеш-таблица раздела может снова не поместиться в память; тогда «лишние» строки опять будут разбиты на разделы и записаны на диск для последующей обработки.

Рассмотрим другой пример. Здесь количество уникальных идентификаторов слишком велико, чтобы хеш-таблица поместилась в память целиком, поэтому планируется использовать четыре раздела:

```
=> SET hash_mem_multiplier = 1.1;
=> EXPLAIN
SELECT DISTINCT flight_id FROM ticket_flights;
                              QUERY PLAN
-----------------------------------------------------------------------
 HashAggregate  (cost=174858.15..176000.70 rows=114255 width=4)
   Group Key: flight_id
   -> Seq Scan on ticket_flights  (cost=0.00..153878.52 rows=8391...
(3 rows)
```

В процессе выполнения запроса потребовалось пять итераций: одна по начальному набору данных и еще четыре по каждому из записанных на диск разделов.

```
=> EXPLAIN (analyze, costs off, timing off, summary off, buffers off)
SELECT DISTINCT flight_id FROM ticket_flights;
                              QUERY PLAN
-----------------------------------------------------------------------
 HashAggregate (actual rows=150588.00 loops=1)
   Group Key: flight_id
   Batches: 5  Memory Usage: 4665kB  Disk Usage: 71536kB
   -> Seq Scan on ticket_flights (actual rows=8391852.00 loops=1)
(4 rows)


=> RESET hash_mem_multiplier;
```

В двухпроходном алгоритме соединения хешированием наиболее частые значения специальным образом перемещаются в первый пакет, для того чтобы избежать лишнего ввода-вывода. Для агрегации такая оптимизация не нужна, поскольку на разделы разбиваются не все строки, а только те, которым не хватило отведенной памяти. Частые значения с большой вероятностью встретятся в наборе строк достаточно рано, чтобы успеть занять место.

23

Сортировка и слияние

23.1. Соединение слиянием

Соединение слиянием работает для наборов данных, отсортированных по ключу соединения, и возвращает отсортированный же результат. Входной набор может оказаться уже отсортированным в результате индексного сканирования, или он может быть отсортирован явно [1].

Слияние отсортированных наборов

Вот пример соединения слиянием; оно представлено в плане выполнения узлом Merge Join [2]:

```
=> EXPLAIN (costs off) SELECT *
FROM tickets t
  JOIN ticket_flights tf ON tf.ticket_no = t.ticket_no
ORDER BY t.ticket_no;
                          QUERY PLAN
---------------------------------------------------------------
 Merge Join
   Merge Cond: (t.ticket_no = tf.ticket_no)
   -> Index Scan using tickets_pkey on tickets t
   -> Index Scan using ticket_flights_pkey on ticket_flights tf
(4 rows)
```

[1] backend/optimizer/path/joinpath.c, функция generate_mergejoin_paths.
[2] backend/executor/nodeMergejoin.c.

Здесь оптимизатор предпочел именно этот способ соединения, поскольку он возвращает результат как раз в том порядке, который указан в предложении ORDER BY. Работая с планами, оптимизатор учитывает порядок сортировки наборов данных и не выполняет явную сортировку, если в ней нет необходимости. В частности, набор строк, полученный в результате соединения слиянием, можно использовать для следующего соединения слиянием, если подходит имеющийся порядок сортировки:

```
=> EXPLAIN (costs off) SELECT *
FROM tickets t
  JOIN ticket_flights tf ON t.ticket_no = tf.ticket_no
  JOIN boarding_passes bp ON bp.ticket_no = tf.ticket_no
                         AND bp.flight_id = tf.flight_id
ORDER BY t.ticket_no;
                              QUERY PLAN
-----------------------------------------------------------------------
 Merge Join
   Merge Cond: ((t.ticket_no = tf.ticket_no) AND (bp.flight_id = ...
   -> Merge Join
       Merge Cond: (bp.ticket_no = t.ticket_no)
       -> Index Scan using boarding_passes_pkey on boarding_passe...
       -> Index Scan using tickets_pkey on tickets t
   -> Index Scan using ticket_flights_pkey on ticket_flights tf
(7 rows)
```

Сначала соединяются строки таблиц посадочных талонов boarding_passes и билетов tickets, прочитанные с помощью индексного доступа в порядке ticket_no; результат тоже отсортирован по столбцу ticket_no. Полученный набор соединяется со строками таблицы перелетов ticket_flights, выдаваемыми индексным сканированием в том же порядке.

Соединение выполняется за один проход по обоим наборам данных и не требует дополнительной памяти. Используются два указателя на текущие (изначально — первые) строки внутреннего и внешнего наборов.

Если ключи двух текущих строк не совпадают, один из указателей — тот, что ссылается на строку с меньшим ключом, — продвигается на одну позицию вперед до тех пор, пока не будет найдено совпадение. Соответствующие друг другу строки возвращаются вышестоящему узлу, а вперед продвигается указатель внутреннего набора данных. Соединение продолжается до исчерпания одного из наборов.

Такой алгоритм справляется с дубликатами ключей во внутреннем наборе данных, но поскольку дубликаты могут быть и во внешнем наборе, алгоритм приходится немного усложнить: если после продвижения внешнего указателя ключ остается прежним, внутренний указатель возвращается назад на первую строку с тем же значением ключа. Таким образом, каждой строке из внешнего набора данных будут сопоставлены все строки с тем же ключом из внутреннего набора данных[1].

Для внешнего соединения алгоритм еще немного меняется, но общая идея остается той же самой.

Единственный оператор, на который рассчитано соединение слиянием, — равенство, то есть поддерживаются только эквисоединения (хотя работа над поддержкой других условий тоже ведется[2]).

Оценка стоимости. Рассмотрим приведенный выше пример:

```
=> EXPLAIN SELECT *
FROM tickets t
  JOIN ticket_flights tf ON tf.ticket_no = t.ticket_no
ORDER BY t.ticket_no;
                              QUERY PLAN
-----------------------------------------------------------------------
 Merge Join  (cost=0.99..822507.74 rows=8391852 width=136)
   Merge Cond: (t.ticket_no = tf.ticket_no)
   -> Index Scan using tickets_pkey on tickets t
       (cost=0.43..139136.22 rows=2949857 width=104)
   -> Index Scan using ticket_flights_pkey on ticket_flights tf
       (cost=0.56..571098.73 rows=8391852 width=32)
(6 rows)
```

В начальную стоимость соединения входят как минимум начальные стоимости дочерних узлов.

В общем случае для нахождения первого соответствия может потребоваться прочитать некоторую долю внешнего или внутреннего набора данных.

[1] backend/executor/nodeMergejoin.c, функция ExecMergeJoin.

[2] Например, commitfest.postgresql.org/33/3160.

с. 356 Чтобы оценить эту долю, можно сравнить (с помощью гистограммы) минимальные значения ключа соединения в двух наборах[1]. Но в данном случае диапазоны номеров билетов в обеих таблицах совпадают.

Полная стоимость соединения складывается из стоимости получения данных от дочерних узлов и стоимости вычислений.

Поскольку алгоритм соединения останавливается, когда заканчивается один из наборов данных (конечно, кроме случая внешнего соединения), другой набор может быть прочитан не полностью. Оценку доли прочитанных строк можно получить, сравнив максимальные значения ключа в двух наборах. В нашем случае оба набора будут просмотрены до конца, так что в полную стоимость соединения войдет сумма полных стоимостей обоих дочерних узлов.

Кроме того, при наличии дубликатов часть строк внутреннего набора может быть прочитана несколько раз. Количество повторных чтений оценивается разностью кардинальностей результата соединения и внутреннего набора[2]. В нашем запросе эти кардинальности совпадают, что говорит об отсутствии дубликатов.

Алгоритм сравнивает ключи соединений двух наборов. Стоимость одного
0.0025 сравнения оценивается значением параметра *cpu_operator_cost*, а их количество можно оценить суммой числа строк в обоих наборах (добавив к ней количество повторных чтений, вызванных дубликатами). Стоимость обработки каждой результирующей строки оценивается, как обычно, значением
0.01 параметра *cpu_tuple_cost*.

В итоге для нашего примера стоимость соединения вычисляется следующим образом[3]:

```
=> SELECT 0.43 + 0.56 AS startup,
  round((
    139110.29 + 571098.73 +
    current_setting('cpu_tuple_cost')::real * 8391852 +
    current_setting('cpu_operator_cost')::real * (2949857 + 8391852)
  )::numeric, 2) AS total;
```

[1] backend/utils/adt/selfuncs.c, функция mergejoinscansel.
[2] backend/optimizer/path/costsize.c, функция final_cost_mergejoin.
[3] backend/optimizer/path/costsize.c, функции initial_cost_mergejoin и final_cost_mergejoin.

```
 startup |   total
---------+-----------
    0.99 | 822481.81
(1 row)
```

Параллельный режим

Соединение слиянием не имеет специальной параллельной реализации, но может использоваться в параллельных планах[1].

Сканирование внешнего набора строк выполняется рабочими процессами параллельно, но внутренний набор строк каждый рабочий процесс всегда читает самостоятельно.

Поскольку параллельное хеш-соединение почти всегда выигрывает по стои- *с. 471*
мости, я временно отключу его:

```
=> SET enable_hashjoin = off;
```

Вот пример параллельного плана, использующего соединение слиянием:

```
=> EXPLAIN (costs off)
SELECT count(*), sum(tf.amount)
FROM tickets t
  JOIN ticket_flights tf ON tf.ticket_no = t.ticket_no;
                            QUERY PLAN
-------------------------------------------------------------------
 Finalize Aggregate
   -> Gather
       Workers Planned: 2
       -> Partial Aggregate
           -> Merge Join
               Merge Cond: (tf.ticket_no = t.ticket_no)
               -> Parallel Index Scan using ticket_flights_pkey o...
               -> Index Only Scan using tickets_pkey on tickets t
(8 rows)
```

Полные и правые внешние соединения слиянием в параллельных планах не поддерживаются.

[1] backend/optimizer/path/joinpath.c, функция consider_parallel_mergejoin.

Модификации

Соединение слиянием поддерживает любые виды соединений. Для полного и правого внешних соединений есть единственное ограничение: условие соединения должно содержать только выражения, подходящие для слияния (равенство столбцов из внешнего и внутреннего наборов или равенство столбца константе)[1]. При внутреннем и левом внешнем соединениях результат слияния просто фильтруется по неподходящим условиям, но для полного и правого соединений такая фильтрация невозможна.

Вот пример полного соединения, использующего алгоритм слияния:

```
=> EXPLAIN (costs off) SELECT *
FROM tickets t
  FULL JOIN ticket_flights tf ON tf.ticket_no = t.ticket_no
ORDER BY t.ticket_no;
                              QUERY PLAN
-----------------------------------------------------------------------
 Sort
   Sort Key: t.ticket_no
   -> Merge Full Join
       Merge Cond: (t.ticket_no = tf.ticket_no)
       -> Index Scan using tickets_pkey on tickets t
       -> Index Scan using ticket_flights_pkey on ticket_flights tf
(6 rows)
```

Внутреннее и левое соединения слиянием сохраняют порядок сортировки. Но для полного и правого внешних соединений это не так, поскольку между упорядоченными значениями внешнего набора данных могут быть вставлены неопределенные значения — а это нарушает сортировку[2]. Поэтому здесь появляется узел Sort, восстанавливающий нужный порядок. Это, конечно, увеличивает стоимость плана и делает хеш-соединение более привлекательным, и планировщик выбрал этот план только потому, что хеш-соединения были отключены.

Но в следующем примере нет иного способа выполнить операцию: вложенный цикл в принципе не поддерживает полное соединение, а слияние здесь неприменимо из-за условия неподходящего вида. Поэтому хеш-соединение

[1] backend/optimizer/path/joinpath.c, функция select_mergejoin_clauses.
[2] backend/optimizer/path/pathkeys.c, функция build_join_pathkeys.

используется, даже несмотря на запрет (о чем напоминает строка Disabled: true под узлом Hash Full Join): v. 18

```
=> EXPLAIN (costs off) SELECT *
FROM tickets t
  FULL JOIN ticket_flights tf ON tf.ticket_no = t.ticket_no
                             AND tf.amount > 0
ORDER BY t.ticket_no;
                  QUERY PLAN
-----------------------------------------------
 Sort
   Sort Key: t.ticket_no
   -> Hash Full Join
       Disabled: true
       Hash Cond: (tf.ticket_no = t.ticket_no)
       Join Filter: (tf.amount > '0'::numeric)
       -> Seq Scan on ticket_flights tf
       -> Hash
           -> Seq Scan on tickets t
(9 rows)
```

Восстановим отключенную ранее возможность использования хеш-соединений:

```
=> RESET enable_hashjoin;
```

23.2. Сортировка

Если какой-то из наборов строк (а возможно, и оба) не отсортирован по ключу соединения, перед выполнением слияния он должен быть переупорядочен. Такая явная сортировка представляется в плане выполнения узлом Sort[1]:

```
=> EXPLAIN (costs off)
SELECT *
FROM flights f
  JOIN airports_data dep ON f.departure_airport = dep.airport_code
ORDER BY dep.airport_code;
```

[1] backend/executor/nodeSort.c.

```
                        QUERY PLAN
-----------------------------------------------------------
 Merge Join
   Merge Cond: (f.departure_airport = dep.airport_code)
   -> Sort
       Sort Key: f.departure_airport
       -> Seq Scan on flights f
   -> Sort
       Sort Key: dep.airport_code
       -> Seq Scan on airports_data dep
(8 rows)
```

Такая же сортировка может применяться и вне контекста соединений при использовании предложения ORDER BY, как самого по себе, так и в составе оконных функций:

```
=> EXPLAIN (costs off)
SELECT flight_id,
  row_number() OVER (PARTITION BY flight_no ORDER BY flight_id)
FROM flights f;
                              QUERY PLAN
-----------------------------------------------------------------------
 WindowAgg
   Window: w1 AS (PARTITION BY flight_no ORDER BY flight_id ROWS ...
   -> Sort
       Sort Key: flight_no, flight_id
       -> Seq Scan on flights f
(5 rows)
```

Здесь узел WindowAgg[1] вычисляет оконную функцию по набору данных, предварительно отсортированному узлом Sort.

В арсенале планировщика имеется несколько способов сортировки данных. В примере, который я уже показывал, используются два из них (Sort Method). Как обычно, эти детали позволяет узнать команда EXPLAIN ANALYZE:

```
=> EXPLAIN (analyze, costs off, timing off, summary off, buffers off)
SELECT *
FROM flights f
  JOIN airports_data dep ON f.departure_airport = dep.airport_code
ORDER BY dep.airport_code;
```

[1] backend/executor/nodeWindowAgg.c.

```
                              QUERY PLAN
-----------------------------------------------------------------------
 Merge Join (actual rows=214867.00 loops=1)
   Merge Cond: (f.departure_airport = dep.airport_code)
   -> Sort (actual rows=214867.00 loops=1)
       Sort Key: f.departure_airport
       Sort Method: external merge  Disk: 17120kB
       -> Seq Scan on flights f (actual rows=214867.00 loops=1)
   -> Sort (actual rows=104.00 loops=1)
       Sort Key: dep.airport_code
       Sort Method: quicksort  Memory: 42kB
       -> Seq Scan on airports_data dep (actual rows=104.00 loops=1)
(10 rows)
```

Быстрая сортировка

Если сортируемый набор данных помещается в память, ограниченную значением параметра *work_mem*, применяется традиционная *быстрая сортировка* (quick sort). Этот алгоритм описан во всех учебниках, так что я не буду его повторять.

4MB

С точки зрения реализации сортировка выполняется специальным компонентом[1], который выбирает наиболее подходящий алгоритм в зависимости от доступной памяти и других факторов.

Оценка стоимости. В качестве примера возьмем сортировку небольшой таблицы. В этом случае выполняется быстрая сортировка в памяти:

```
=> EXPLAIN SELECT *
FROM airports_data
ORDER BY airport_code;
                              QUERY PLAN
-----------------------------------------------------------------------
 Sort  (cost=7.52..7.78 rows=104 width=145)
   Sort Key: airport_code
   -> Seq Scan on airports_data  (cost=0.00..4.04 rows=104 width=...
(3 rows)
```

[1] backend/utils/sort/tuplesort.c.

Известно, что сортировка n значений имеет вычислительную сложность $O(n \log_2 n)$. Одна операция сравнения оценивается удвоенным значением
0.0025 параметра *cpu_operator_cost*. Поскольку результат можно получить, только прочитав и отсортировав *весь* набор данных, начальная стоимость сортировки определяется полной стоимостью дочернего узла и стоимостью всех операций сравнения.

В полную стоимость сортировки добавляется обработка каждой строки результата, которая оценивается значением параметра *cpu_operator_cost* (а не *cpu_tuple_cost*, как обычно, поскольку для узла Sort накладные расходы невелики)[1].

В нашем примере стоимость вычисляется так:

```
=> WITH costs(startup) AS (
  SELECT 4.04 + round((
    current_setting('cpu_operator_cost')::real * 2 *
      104 * log(2, 104)
  )::numeric, 2)
)
SELECT startup,
  startup + round((
    current_setting('cpu_operator_cost')::real * 104
  )::numeric, 2) AS total
FROM costs;
 startup | total
---------+-------
    7.52 |  7.78
(1 row)
```

Частичная пирамидальная сортировка

Если нужно отсортировать не весь набор данных, а только его часть (что определяется предложением LIMIT), может применяться *частичная пирамидальная сортировка* (top-N heapsort). Точнее, этот алгоритм используется, если количество строк после сортировки уменьшается как минимум вдвое или если входной набор строк не помещается целиком в отведенную оперативную память (но для выходного набора при этом достаточно места).

[1] backend/optimizer/path/costsize.c, функция cost_sort.

```
=> EXPLAIN (analyze, timing off, summary off, buffers off)
SELECT * FROM seats
ORDER BY seat_no LIMIT 100;
                            QUERY PLAN
------------------------------------------------------------------
 Limit  (cost=72.57..72.82 rows=100 width=15)
   (actual rows=100.00 loops=1)
   -> Sort  (cost=72.57..75.91 rows=1339 width=15)
       (actual rows=100.00 loops=1)
       Sort Key: seat_no
       Sort Method: top-N heapsort  Memory: 32kB
       -> Seq Scan on seats  (cost=0.00..21.39 rows=1339 width=15)
           (actual rows=1339.00 loops=1)
(8 rows)
```

Чтобы найти k максимальных (минимальных) значений из n, в структуру данных, называемую кучей, добавляются k первых строк. Затем по одной добавляются и все остальные строки, но после добавления каждой следующей строки из кучи изымается одно наименьшее (наибольшее) значение. В результате в куче остаются k искомых значений.

> Куча (heap), используемая в этом алгоритме, является структурой данных и не имеет ничего общего с таблицами базы данных, которые часто называют этим же термином.

Оценка стоимости. Сложность алгоритма оценивается как $O(n \log_2 k)$, однако каждая операция обходится дороже, чем в случае быстрой сортировки. Поэтому формула расчета стоимости [1] использует $n \log_2 2k$.

```
=> WITH costs(startup) AS (
  SELECT 21.39 + round((
    current_setting('cpu_operator_cost')::real * 2 *
      1339 * log(2, 2 * 100)
  )::numeric, 2)
)
SELECT startup,
  startup + round((
    current_setting('cpu_operator_cost')::real * 100
  )::numeric, 2) AS total
FROM costs;
```

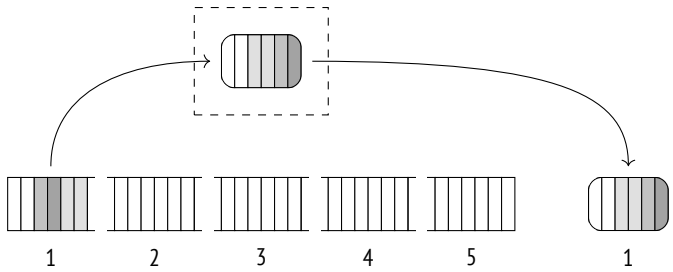
[1] backend/optimizer/path/costsize.c, функция cost_sort.

```
 startup | total
---------+-------
   72.57 | 72.82
(1 row)
```

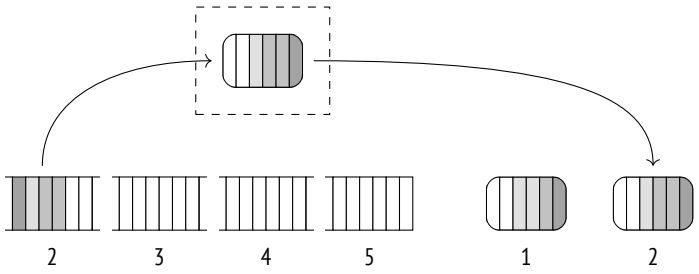
Внешняя сортировка

Если при чтении набора данных выясняется, что он слишком велик для сортировки в оперативной памяти, узел сортировки переключается на *внешнюю сортировку слиянием* (external merge).

Уже прочитанные строки сортируются в памяти алгоритмом быстрой сортировки и записываются во временный файл.



В освобожденную память читаются следующие строки, и процедура повторяется до тех пор, пока все данные не будут записаны в несколько файлов, каждый из которых в отдельности отсортирован.



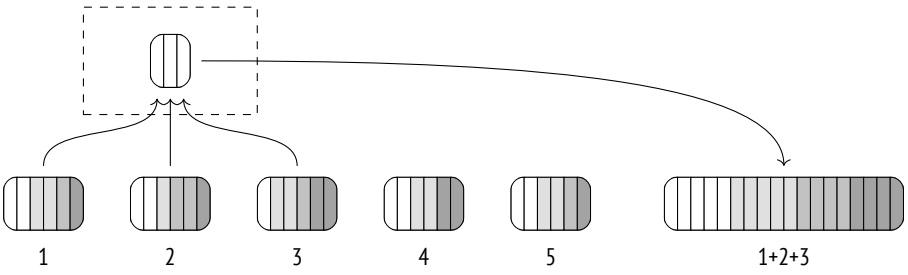
Далее несколько файлов объединяются в один примерно тем же алгоритмом, что используется и при соединении слиянием. Основное отличие состоит в том, что объединяться могут более двух файлов одновременно.

Для слияния не требуется много памяти. В принципе, достаточно располагать местом под одну строку для каждого файла. Из файлов читаются первые строки, среди них выбирается минимальная (или максимальная, в зависимости от направления сортировки) и возвращается как часть результата, а на ее место читается новая строка из того же файла.

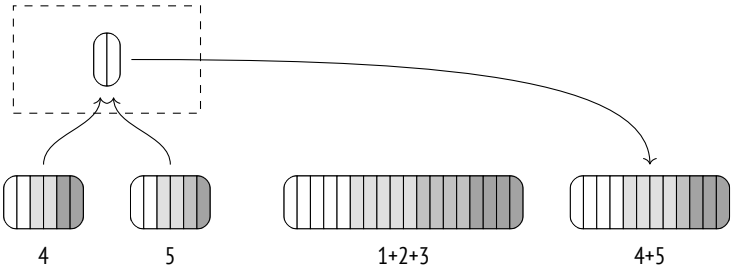
На практике строки читаются не по одной, а порциями по 32 страницы; это уменьшает количество операций ввода-вывода. Количество файлов, которые объединяются за одну итерацию, определяется доступным местом в памяти, но меньше шести не используется никогда. Сверху это количество тоже ограничено (числом 500), поскольку эффективность теряется, когда файлов становится слишком много[1].

> Для алгоритмов сортировки в английском языке исторически сложилась специальная терминология. Внешняя сортировка изначально использовала накопители на магнитной ленте, и компонент PostgreSQL, управляющий временными файлами, сохранил это название[2]. Частично отсортированные наборы данных называются словом run (*серия* в русском переводе книги Кнута[3]). Количество серий, участвующих в слиянии, называется merge order (*порядок слияния*). Я не использовал эти термины, но их полезно знать, чтобы понимать код и комментарии.

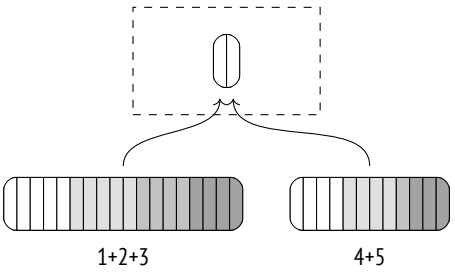
Если объединить все отсортированные временные файлы за одну итерацию не получается, приходится выполнять слияние файлов по частям и записывать результат в новые временные файлы. Каждая такая итерация увеличивает объем записываемых и читаемых данных, поэтому чем больше оперативной памяти доступно, тем эффективнее будет выполняться внешняя сортировка.



[1] backend/utils/sort/tuplesort.c, функция tuplesort_merge_order.
[2] backend/utils/sort/logtape.c.
[3] *Дональд Кнут*. Искусство программирования. Т. 3 : Сортировка и поиск.



На следующей итерации слияние продолжается уже с новыми файлами.



Финальное слияние обычно откладывается и выполняется на лету, когда вышестоящий узел плана запрашивает данные.

Команда EXPLAIN ANALYZE показывает объем дисковой памяти, который потребовался внешней сортировке. Добавив ключевое слово buffers, можно получить и статистику использования буферов временных файлов (temp read и written). Количество записанных буферов будет (примерно) равно количеству прочитанных, и именно это значение, пересчитанное в килобайты, показано в плане в позиции Disk:

```
=> EXPLAIN (analyze, costs off, timing off, summary off)
SELECT * FROM flights
ORDER BY scheduled_departure;
                        QUERY PLAN
-----------------------------------------------------------
 Sort (actual rows=214867.00 loops=1)
   Sort Key: scheduled_departure
   Sort Method: external merge  Disk: 17120kB
   Buffers: shared hit=2627, temp read=2140 written=2145
   -> Seq Scan on flights (actual rows=214867.00 loops=1)
       Buffers: shared hit=2624
(6 rows)
```

Более подробную статистику использования временных файлов можно получить в журнале сообщений, установив параметр *log_temp_files*.

Оценка стоимости. В качестве примера возьмем тот же план с внешней сортировкой:

```
=> EXPLAIN SELECT *
FROM flights
ORDER BY scheduled_departure;
                              QUERY PLAN
-----------------------------------------------------------------------
 Sort  (cost=31883.96..32421.12 rows=214867 width=63)
   Sort Key: scheduled_departure
   -> Seq Scan on flights  (cost=0.00..4772.67 rows=214867 width=63)
(3 rows)
```

Здесь к обычной стоимости сравнений (количество которых остается таким же, как и в случае быстрой сортировки в памяти) добавляется стоимость ввода-вывода[1]. Все входные данные придется сначала записать на диск во временные файлы, а затем прочитать с диска при слиянии (причем, возможно, несколько раз, если количество созданных файлов будет превышать количество одновременно объединяемых наборов).

Обращение к диску (и запись, и чтение) считается на три четверти последовательным и на одну четверть случайным.

Объем данных, попадающих на диск, определяется количеством сортируемых строк и числом столбцов, используемых в запросе[2]. В данном случае запрос выводит все столбцы таблицы `flights`, поэтому на диск попадет объем, почти равный размеру всей таблицы, за вычетом служебной информации в версиях строк и страницах (2309 страниц вместо 2624).

В нашем примере для сортировки хватает одной итерации.

Таким образом, стоимость сортировки в нашем плане вычисляется так:

[1] backend/optimizer/path/costsize.c, функция cost_sort.
[2] backend/optimizer/path/costsize.c, функция relation_byte_size.

```
=> WITH costs(startup) AS (
  SELECT 4772.67 + round((
    current_setting('cpu_operator_cost')::real * 2 *
      214867 * log(2, 214867) +
    (current_setting('seq_page_cost')::real * 0.75 +
     current_setting('random_page_cost')::real * 0.25) *
    2 * 2309 * 1 -- одна итерация
  )::numeric, 2)
)
SELECT startup,
  startup + round((
    current_setting('cpu_operator_cost')::real * 214867
  )::numeric, 2) AS total
FROM costs;
 startup  |  total
----------+----------
 31883.96 | 32421.13
(1 row)
```

v. 13 Инкрементальная сортировка

Если набор данных требуется отсортировать по ключам $K_1 \dots K_m \dots K_n$ и при этом известно, что набор уже отсортирован по первым m ключам, то не обязательно пересортировывать весь набор заново. Можно разбить набор данных на группы, имеющие одинаковые значения начальных ключей $K_1 \dots K_m$ (значения таких групп следуют друг за другом), и затем отсортировать отдельно каждую из групп по оставшимся ключам $K_{m+1} \dots K_n$. Такой способ называется *инкрементальной сортировкой*.

Инкрементальная сортировка уменьшает требования к памяти, разбивая весь набор на несколько меньших групп, и позволяет выдавать результаты уже после обработки первой группы, не дожидаясь сортировки всего набора.

Реализация[1] действует более тонко: отдельно обрабатываются только относительно крупные группы строк, а небольшие объединяются и сортируются полностью. Это уменьшает накладные расходы на вызов сортировки[2].

В плане выполнения инкрементальная сортировка представлена узлом Incremental Sort:

[1] backend/executor/nodeIncrementalSort.c.
[2] backend/utils/sort/tuplesort.c.

```
=> EXPLAIN (analyze, costs off, timing off, summary off, buffers off)
SELECT *
FROM bookings
ORDER BY total_amount, book_date;
                              QUERY PLAN
-----------------------------------------------------------------------
 Incremental Sort (actual rows=2111110.00 loops=1)
   Sort Key: total_amount, book_date
   Presorted Key: total_amount
   Full-sort Groups: 2823  Sort Method: quicksort  Average
   Memory: 27kB  Peak Memory: 27kB
   Pre-sorted Groups: 2624  Sort Method: quicksort  Average
   Memory: 1953kB  Peak Memory: 2014kB
   -> Index Scan using bookings_total_amount_idx on bookings (ac...
       Index Searches: 1
(9 rows)
```

Как видно из плана, набор строк уже отсортирован по `total_amount`, поскольку получен сканированием индекса, построенного по этому столбцу (Presorted Key). Команда `EXPLAIN ANALYZE` показывает также статистику времени выполнения. Строка Full-sort Groups относится к небольшим группам, которые были объединены и отсортированы полностью, а строка Presorted Groups — к крупным группам, которые досортировывались по столбцу `book_date`. В обоих случаях использовалась быстрая сортировка в памяти. Наличие групп разного размера вызвано неравномерным распределением стоимости бронирований.

Инкрементальная сортировка может использоваться и при вычислении оконных функций: v. 14

```
=> EXPLAIN (costs off)
SELECT row_number() OVER (ORDER BY total_amount, book_date)
FROM bookings;
                              QUERY PLAN
-----------------------------------------------------------------------
 WindowAgg
   Window: w1 AS (ORDER BY total_amount, book_date ROWS UNBOUNDED...
   -> Incremental Sort
       Sort Key: total_amount, book_date
       Presorted Key: total_amount
       -> Index Scan using bookings_total_amount_idx on bookings
(6 rows)
```

Оценка стоимости. Расчет стоимости инкрементальной сортировки[1] опирается на оценку количества групп[2] и оценку сортировки группы среднего размера (которую мы уже рассмотрели).

Начальная стоимость отражает оценки сортировки одной (первой) группы, после которой узел уже может выдавать отсортированные строки, а полная стоимость учитывает сортировку всех групп.

Подробно останавливаться на вычислении оценок я не буду.

Параллельный режим

Сортировка может выполняться параллельно. Но хотя рабочие процессы выдают свою часть данных в отсортированном виде, узел Gather ничего про это не знает и может объединять данные только в порядке поступления. Чтобы сохранить сортировку, применяется другой узел — Gather Merge[3].

```
=> EXPLAIN (analyze, costs off, timing off, summary off, buffers off)
SELECT *
FROM flights
ORDER BY scheduled_departure
LIMIT 10;
                              QUERY PLAN
-----------------------------------------------------------------------
 Limit (actual rows=10.00 loops=1)
   -> Gather Merge (actual rows=10.00 loops=1)
       Workers Planned: 1
       Workers Launched: 1
       -> Sort (actual rows=8.50 loops=2)
           Sort Key: scheduled_departure
           Sort Method: top-N heapsort  Memory: 27kB
           Worker 0:  Sort Method: top-N heapsort  Memory: 27kB
           -> Parallel Seq Scan on flights (actual rows=107433.50...
(9 rows)
```

Узел Gather Merge использует двоичную кучу[4] для упорядочения строк, поступающих от нескольких процессов. По сути, он выполняет слияние

[1] backend/optimizer/path/costsize.c, функция cost_incremental_sort.
[2] backend/utils/adt/selfuncs.c, функция estimate_num_groups.
[3] backend/executor/nodeGatherMerge.c.
[4] backend/lib/binaryheap.c.

нескольких отсортированных наборов строк, как при внешней сортировке, но алгоритм рассчитан на другие условия работы: на небольшое фиксированное число источников и получение строк по одной, без блочного доступа.

Оценка стоимости. Начальная стоимость узла Gather Merge опирается на начальную стоимость дочернего узла. К ней (как и для узла Gather) добавляется стоимость запуска процессов, которая оценивается значением параметра *parallel_setup_cost*. *с. 382* 1000

Сюда же добавляется оценка построения двоичной кучи, что требует сортировки n значений по числу параллельных процессов (то есть $n \log_2 n$). Одна операция сравнения оценивается удвоенным значением параметра *cpu_operator_cost*, и общая сумма обычно пренебрежимо мала, поскольку n невелико. 0.0025

В полную стоимость входят получение всех данных дочерним узлом, который выполняется несколькими параллельными процессами, и стоимость пересылки строк от этих процессов. Пересылка одной строки оценивается значением параметра *parallel_tuple_cost*, увеличенным на 5 %, чтобы учесть возможные потери при ожидании очередных значений. 0.1

В полную стоимость входит также обновление двоичной кучи. Для каждой входящей строки данных это требует $\log_2 n$ операций сравнения и определенных вспомогательных действий (которые оцениваются значением *cpu_operator_cost*)[1].

Вот еще один пример плана с узлом Gather Merge. Он интересен тем, что рабочие процессы выполняют частичную агрегацию с помощью хеширования, затем полученные результаты сортируются узлом Sort (это дешево, поскольку после агрегации остается немного строк) и передаются ведущему процессу, который собирает полный результат в узле Gather Merge. Окончательная же агрегация выполняется по отсортированному списку значений: *с. 478*

```
=> EXPLAIN SELECT amount, count(*)
FROM ticket_flights
GROUP BY amount;
```

[1] backend/optimizer/path/costsize.c, функция cost_gather_merge.

```
                                QUERY PLAN
-------------------------------------------------------------------------
 Finalize GroupAggregate  (cost=123426.68..123528.57 rows=338 wid...
   Group Key: amount
   -> Gather Merge  (cost=123426.68..123521.13 rows=811 width=14)
       Workers Planned: 2
       -> Sort  (cost=122426.65..122427.50 rows=338 width=14)
           Sort Key: amount
           -> Partial HashAggregate  (cost=122409.07..122412.46 r...
               Group Key: amount
               -> Parallel Seq Scan on ticket_flights  (cost=0.00...
(9 rows)
```

В данном случае количество параллельных процессов равно трем (включая основной), и стоимость узла Gather Merge вычисляется так:

```
=> WITH costs(startup, run) AS (
  SELECT round((
    -- запуск процессов
    current_setting('parallel_setup_cost')::real +
    -- построение кучи
    current_setting('cpu_operator_cost')::real * 2 * 3 * log(2, 3)
  )::numeric, 2),
  round((
    -- передача строк
    current_setting('parallel_tuple_cost')::real * 1.05 * 676 +
    -- обновление кучи
    current_setting('cpu_operator_cost')::real * 2 * 676 * log(2, 3) +
    current_setting('cpu_operator_cost')::real * 676
  )::numeric, 2)
)
SELECT 122426.65 + startup AS startup,
  122427.50 + startup + run AS total
FROM costs;
  startup  |   total
-----------+-----------
 123426.67 | 123505.55
(1 row)
```

23.3. Группировка и уникальные значения

Как мы только что видели, группировка значений для агрегации (и устра-
с. 479 нения дубликатов) может выполняться не только хешированием, но также

и с помощью сортировки. В отсортированном списке группы повторяющихся значений элементарно выделяются за один проход.

Выбор уникальных значений из отсортированного списка представляется в плане очень простым узлом Unique[1]:

```
=> EXPLAIN (costs off) SELECT DISTINCT book_ref
FROM bookings
ORDER BY book_ref;
                      QUERY PLAN
------------------------------------------------------
 Unique
   -> Index Only Scan using bookings_pkey on bookings
(2 rows)
```

Узел Unique может участвовать в параллельном плане, устраняя дубликаты в списке, полученном от рабочих процессов: v. 15

```
=> EXPLAIN (costs off) SELECT DISTINCT fare_conditions
FROM ticket_flights;
                      QUERY PLAN
------------------------------------------------------
 Unique
   -> Gather Merge
       Workers Planned: 2
       -> Sort
           Sort Key: fare_conditions
           -> HashAggregate
               Group Key: fare_conditions
               -> Parallel Seq Scan on ticket_flights
(8 rows)
```

В этом примере рабочие процессы выделяют уникальные значения в своих наборах данных с помощью хеширования в узле HashAggregate; общий список, собранный узлом Gather, сортируется в узле Sort и передается узлу Unique для окончательного устранения дубликатов.

Для агрегации используется другой узел, GroupAggregate[2]:

[1] backend/executor/nodeUnique.c.

[2] backend/executor/nodeAgg.c, функция agg_retrieve_direct.

```
=> EXPLAIN (costs off) SELECT book_ref, count(*)
FROM bookings
GROUP BY book_ref
ORDER BY book_ref;
                      QUERY PLAN
------------------------------------------------------
 GroupAggregate
   Group Key: book_ref
   -> Index Only Scan using bookings_pkey on bookings
(3 rows)
```

В параллельных планах такой узел будет называться Partial GroupAggregate, а узел, завершающий агрегацию, — Finalize GroupAggregate.

Обе стратегии — хеширование и сортировка — могут совмещаться в одном узле при группировке по нескольким наборам (в предложениях GROUPING SETS, CUBE или ROLLUP). Я не буду углубляться в весьма непростые детали алгоритма. Приведу лишь один пример, в котором группировка должна вычисляться по трем разным столбцам:

```
=> EXPLAIN (costs off) SELECT count(*)
FROM ticket_flights
GROUP BY GROUPING SETS (fare_conditions, ticket_no, amount);
             QUERY PLAN
------------------------------------
 MixedAggregate
   Hash Key: amount
   Group Key: fare_conditions
   Sort Key: ticket_no
     Group Key: ticket_no
   -> Sort
       Sort Key: fare_conditions
       -> Seq Scan on ticket_flights
(8 rows)
```

Вот что происходит при выполнении этого запроса. Узел агрегации, который обозначен в плане как MixedAggregate, получает набор данных, отсортированных по столбцу `fare_conditions`.

На первом этапе этот набор сканируется, и значения группируются по столбцу `fare_conditions` (Group Key). По мере сканирования строки переупорядочиваются по столбцу `ticket_no` (так, как это делает обычный узел Sort: либо

быстрой сортировкой в памяти, если ее достаточно, либо внешней сортировкой на диске) и одновременно с этим записываются в хеш-таблицу с ключом amount (так, как это делает агрегация хешированием: либо в памяти, либо с использованием временных файлов).

На втором этапе сканируется набор строк, отсортированный на предыдущем этапе по столбцу ticket_no, и значения группируются по этому же столбцу (Sort Key и вложенный Group Key). Если бы требовалась группировка сортировкой по еще одному столбцу, строки были бы пересортированы в необходимом дальше порядке.

Наконец, сканируется хеш-таблица, подготовленная на первом этапе, и значения группируются по столбцу amount (Hash Key).

23.4. Сравнение способов соединения

Итак, для соединения двух наборов данных могут использоваться три разных способа, каждый со своими достоинствами и недостатками.

Соединение вложенным циклом не требует никакой подготовительной работы и сразу начинает возвращать результирующие строки. Это единственный из способов соединения, которому не требуется просматривать внутренний набор полностью, если для него есть эффективный индексный доступ. Эти свойства делают алгоритм вложенного цикла (в сочетании с индексами) идеальным механизмом для коротких OLTP-запросов, которые строятся на небольшой выборке строк.

Недостаток вложенного цикла проявляется с ростом объема данных. Для декартова произведения этот алгоритм имеет квадратичную сложность — затраты пропорциональны произведению размеров соединяемых наборов данных. Декартово произведение нечасто встречается на практике; обычно для каждой строки внешнего набора данных с помощью индекса просматривается некоторое количество строк внутреннего набора, и это среднее количество не зависит от размера всего набора данных (например, среднее количество билетов в одном бронировании не меняется с ростом числа бронирований и купленных билетов). Поэтому часто рост сложности будет линейным, а не квадратичным, хотя и с большим коэффициентом.

Важная особенность вложенного цикла состоит в его универсальности: он поддерживает любые условия соединения, в то время как остальные способы работают только с эквисоединениями. Это дает возможность выполнять любые запросы с любыми условиями (кроме полного соединения, которое не реализуется вложенным циклом), но надо помнить о том, что неэквисоединение больших объемов почти наверняка будет выполняться медленнее, чем хотелось бы.

Соединение хешированием очень эффективно для больших наборов данных. При наличии достаточного объема оперативной памяти оно требует однократного просмотра двух наборов данных, то есть имеет линейную сложность. В сочетании с последовательным сканированием таблиц соединение хешированием часто встречается в OLAP-запросах, вычисляющих результат на основании большого объема данных.

Для ситуаций, в которых время отклика важнее пропускной способности, хеш-соединение подходит хуже, поскольку результирующие строки не могут возвращаться, пока хеш-таблица не построена полностью.

Применение хеш-соединения ограничено эквисоединениями. Кроме того, тип данных должен допускать хеширование (это выполняется почти всегда).

v. 14 Вложенный цикл может иногда составить конкуренцию соединению хешированием за счет кеширования строк внутреннего набора в узле Memoize (также основанного на хеш-таблице). Выигрыш может достигаться благодаря тому, что соединение хешированием всегда просматривает внутренний набор строк полностью, а алгоритм вложенного цикла — нет.

Соединение слиянием отлично подходит и для коротких OLTP-запросов, и для длинных запросов OLAP. Оно имеет линейную сложность (требуется однократный просмотр соединяемых наборов строк), не требовательно к памяти и выдает результаты без предварительной подготовки — правда, наборы данных должны быть отсортированы в правильном порядке. Наиболее эффективный способ добиться этого — получать данные от индексного сканирования. Это естественный вариант для небольшого количества строк; при большом объеме данных индексный доступ тоже может быть эффективен, но только если обращений к таблице немного или их вовсе нет.

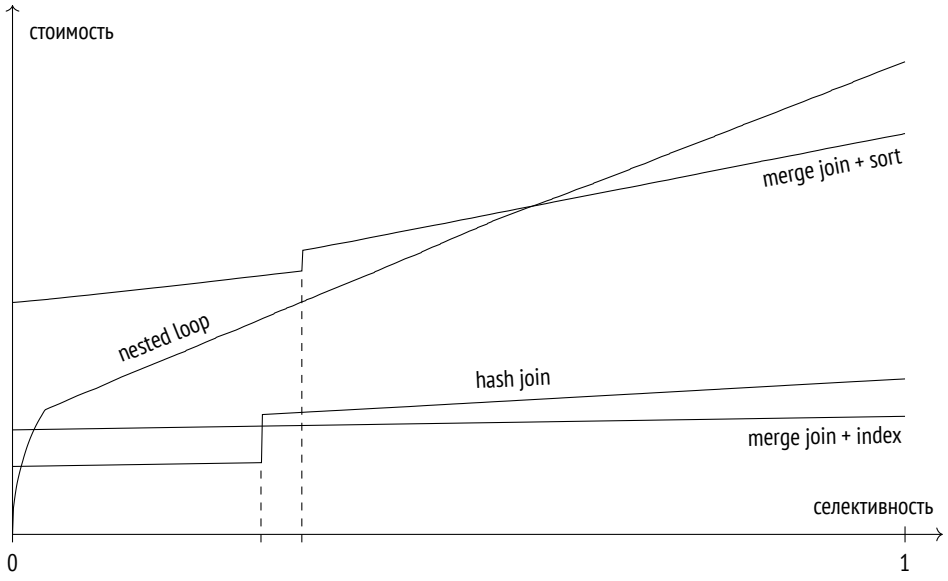
Если подходящих индексов нет, то наборы данных придется сортировать, а сортировка требует памяти и имеет сложность выше линейной: $O(n \log_2 n)$.

В таком случае соединение слиянием почти всегда проигрывает соединению хешированием — за исключением ситуации, когда результат нужен отсортированным.

Приятным свойством соединения слиянием является равноценность внешнего и внутреннего наборов строк. Эффективность и вложенного цикла, и хеш-соединения сильно зависит от того, правильно ли планировщик выберет, какой из наборов данных поставить внешним, а какой — внутренним.

Применение соединения слиянием ограничено эквисоединениями. Кроме того, тип данных должен иметь класс операторов для B-дерева. *с. 546*

Ниже показан примерный график зависимости стоимостей различных способов соединений от доли соединяемых строк.



Соединение вложенным циклом при высокой селективности использует индексный доступ к обеим таблицам; затем планировщик переключается на полное сканирование внешней таблицы, и график становится линейным.

Соединение хешированием использует в этом примере полное сканирование обеих таблиц. «Ступенька» на графике возникает в тот момент, когда хеш-таблица перестает помещаться в оперативной памяти и пакеты начинают сбрасываться на диск.

Соединение слиянием с использованием индекса показывает небольшой линейный рост стоимости. При достаточном объеме *work_mem* соединение хешированием обычно оказывается эффективнее, но когда дело доходит до временных файлов, соединение слиянием выигрывает.

Верхний график соединения слиянием с сортировкой показывает рост стоимости в ситуации, когда индексы недоступны и данные приходится сортировать. Как и в случае соединения хешированием, «ступенька» на графике вызвана недостатком памяти и необходимостью использовать для сортировки временные файлы.

Это только пример; в каждом конкретном случае соотношения стоимостей, разумеется, будут отличаться.

Часть V

Типы индексов

24

Хеш-индекс

24.1. Общий принцип

Хеш-индекс [1] позволяет по ключу индексирования быстро найти идентификатор версии строки (tid). В первом приближении это просто хеш-таблица, хранящаяся на диске. Единственная операция, которую поддерживает хеш-индекс, — поиск по условию равенства.

При вставке в индекс [2] вычисляется хеш-функция от значения ключа индексирования. Хеш-функции в PostgreSQL возвращают 32-битные или 64-битные целые, а в качестве номера корзины используются несколько младших битов этого значения. В выбранную корзину добавляется идентификатор версии вместе с хеш-кодом ключа. Само значение ключа не сохраняется в индексе, поскольку реализации удобнее иметь дело с небольшими значениями фиксированной длины.

Хеш-таблица индекса динамически расширяется [3]. Минимальное число корзин равно двум. По мере увеличения количества индексируемых строк одна из корзин расщепляется на две. Для этого задействуется дополнительный бит хеш-кода, так что элементы перераспределяются только между двумя корзинами, участвующими в расщеплении, а состав остальных корзин хеш-таблицы не меняется [4].

[1] postgrespro.ru/docs/postgresql/18/hash-index; backend/access/hash/README.
[2] backend/access/hash/hashinsert.c.
[3] backend/access/hash/hashpage.c, функция _hash_expandtable.
[4] backend/access/hash/hashpage.c, функция _hash_getbucketbuf_from_hashkey.

При поиске в индексе[1] вычисляется хеш-функция от ключа индексирования и соответствующий этому значению номер корзины. Из всего содержимого корзины возвращаются только те идентификаторы версий, которые соответствуют хеш-коду ключа. Благодаря тому что элементы корзин упорядочены по хеш-кодам ключей, подходящие идентификаторы эффективно находятся двоичным поиском.

Поскольку само значение ключа не сохраняется в хеш-таблице, индексный метод может вернуть лишние идентификаторы версий строк из-за хеш-
с. 411 коллизий. Поэтому механизм индексирования перепроверяет все полученные от метода доступа результаты по таблице. По этой же причине невозможно сканирование только индекса.

24.2. Страничная организация

В отличие от обычной хеш-таблицы, индекс хранится на диске. Поэтому вся необходимая информация должна быть разложена по страницам, причем желательно, чтобы операциям над индексом (поиск, вставка, удаление) требовался доступ к как можно меньшему количеству страниц.

Хеш-индекс использует страницы четырех видов:

- метастраница (meta page) — нулевая страница с «оглавлением» индекса;
- страницы корзин (bucket page) — основные страницы индекса, по одной на каждую корзину;
- страницы переполнения (overflow page) — дополнительные страницы, которые используются, когда основной страницы корзины не хватает для размещения всех элементов;
- страницы битовой карты (bitmap page) — страницы с битовым массивом, в котором отмечены освободившиеся и доступные для повторного использования страницы переполнения.

[1] backend/access/hash/hashsearch.c.

Заглянуть внутрь индексных страниц позволяет расширение pageinspect.

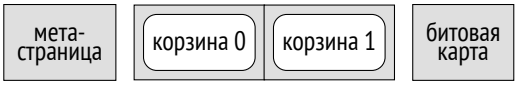
Начнем с пустой таблицы:

```
=> CREATE EXTENSION pageinspect;
=> CREATE TABLE t(n integer);
=> ANALYZE t;
=> CREATE INDEX ON t USING hash(n);
```

Я проанализировал таблицу, чтобы получить индекс минимального размера; иначе количество корзин соответствовало бы таблице из 10 страниц[1]. v. 14

Сейчас в индексе четыре страницы: метастраница, две основные страницы корзин и одна страница битовой карты (сразу созданная «про запас»):

```
=> SELECT page, hash_page_type(get_raw_page('t_n_idx', page))
FROM generate_series(0,3) page;
 page | hash_page_type
------+----------------
    0 | metapage
    1 | bucket
    2 | bucket
    3 | bitmap
(4 rows)
```



Метастраница содержит всю управляющую информацию об индексе. Пока нас интересует несколько значений:

```
=> SELECT ntuples, ffactor, maxbucket
FROM hash_metapage_info(get_raw_page('t_n_idx', 0));
 ntuples | ffactor | maxbucket
---------+---------+-----------
       0 |     307 |         1
(1 row)
```

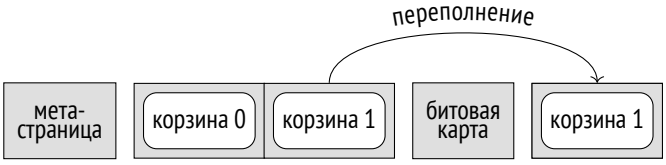
[1] backend/access/table/tableam.c, функция table_block_relation_estimate_size.

Расчетное количество строк на одну корзину показано в поле `ffactor`. Это значение вычисляется исходя из вместимости блока и установленного зна-
75 чения параметра хранения *fillfactor*. При совершенно равномерном распределении данных и отсутствии хеш-коллизий значение *fillfactor* можно было бы и увеличить, но в реальности это повышает вероятность того, что элементы одной корзины не поместятся на одну страницу.

Самый плохой для хеш-индекса случай — сильный перекос в распределении данных, когда один ключ повторяется много раз. Поскольку хеш-функция будет выдавать одно и то же значение, все данные будут попадать в одну и ту же корзину, и увеличение количества корзин ничем не поможет.

Сейчас индекс пуст, о чем говорит значение поля `ntuples`. Переполним страницу корзины, вставив в таблицу определенное количество строк с одинаковым значением индексируемого столбца. В индексе появится страница переполнения:

```
=> INSERT INTO t(n)
  SELECT 0 FROM generate_series(1,500); -- одно значение
=> SELECT page, hash_page_type(get_raw_page('t_n_idx', page))
FROM generate_series(0,4) page;
 page | hash_page_type
------+----------------
    0 | metapage
    1 | bucket
    2 | bucket
    3 | bitmap
    4 | overflow
(5 rows)
```



Сводка информации по страницам показывает, что корзина 0 пуста, а все значения попали в корзину 1: часть на основную страницу, а те, что не поместились, — на страницу переполнения.

```
=> SELECT page, live_items, free_size, hasho_bucket
FROM (VALUES (1), (2), (4)) p(page),
  hash_page_stats(get_raw_page('t_n_idx', page));
 page | live_items | free_size | hasho_bucket
------+------------+-----------+--------------
    1 |          0 |      8148 |            0
    2 |        407 |         8 |            1
    4 |         93 |      6288 |            1
(3 rows)
```

Очевидно, что разброс элементов одной корзины по нескольким страницам плохо сказывается на производительности. Лучшие результаты хеш-индекс показывает на равномерно распределенных данных.

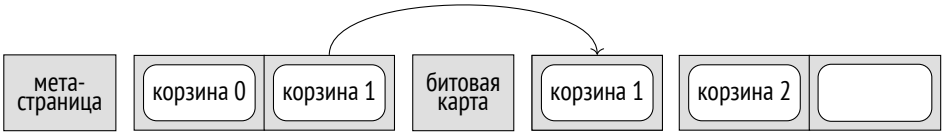
Посмотрим теперь, как расщепляется корзина. Это происходит, когда количество строк в индексе превышает расчетное число `ffactor` для имеющихся корзин. В данном случае при двух корзинах и `ffactor` = 307 это произойдет, когда в индекс будет вставлена 615-я строка:

```
=> SELECT ntuples, ffactor, maxbucket, ovflpoint
FROM hash_metapage_info(get_raw_page('t_n_idx', 0));
 ntuples | ffactor | maxbucket | ovflpoint
---------+---------+-----------+-----------
     500 |     307 |         1 |         1
(1 row)
=> INSERT INTO t(n)
  SELECT n FROM generate_series(1,115) n; -- теперь разные
=> SELECT ntuples, ffactor, maxbucket, ovflpoint
FROM hash_metapage_info(get_raw_page('t_n_idx', 0));
 ntuples | ffactor | maxbucket | ovflpoint
---------+---------+-----------+-----------
     615 |     307 |         2 |         2
(1 row)
```

Значение `maxbucket` увеличилось до двух: теперь у нас есть три корзины с номерами от 0 до 2. Но хотя добавилась всего одна корзина, количество страниц удваивается:

```
=> SELECT page, hash_page_type(get_raw_page('t_n_idx', page))
FROM generate_series(0,6) page;
```

```
 page | hash_page_type
------+----------------
    0 | metapage
    1 | bucket
    2 | bucket
    3 | bitmap
    4 | overflow
    5 | bucket
    6 | unused
(7 rows)
```



Одна из новых страниц задействована под корзину 2, а другая остается незанятой и будет использована для корзины 3, когда та появится.

```
=> SELECT page, live_items, free_size, hasho_bucket
FROM (VALUES (1), (2), (4), (5)) p(page),
  hash_page_stats(get_raw_page('t_n_idx', page));
 page | live_items | free_size | hasho_bucket
------+------------+-----------+--------------
    1 |         27 |      7608 |            0
    2 |        407 |         8 |            1
    4 |        158 |      4988 |            1
    5 |         23 |      7688 |            2
(4 rows)
```

Таким образом, с точки зрения операционной системы размер хеш-индекса растет скачкообразно, хотя логически хеш-таблица увеличивается постепенно.

Чтобы немного сгладить этот рост и не выделять сразу слишком много страниц, начиная с 10-го удвоения страницы выделяются не все сразу, а частями, по $\frac{1}{4}$ от расчетного количества.

Еще два поля из метастраницы, представляющих собой битовые маски, раскрывают детали адресации корзин:

```
=> SELECT maxbucket, highmask::bit(4), lowmask::bit(4)
FROM hash_metapage_info(get_raw_page('t_n_idx', 0));
```

```
 maxbucket | highmask | lowmask
-----------+----------+---------
         2 | 0011     | 0001
(1 row)
```

Номер корзины определяется битами хеш-кода, соответствующими маске highmask. Но если полученный номер корзины не существует (превышает maxbucket), то надо взять биты, соответствующие маске lowmask[1]. В данном случае мы берем два младших бита, что дает значения от 0 до 3; но если выпадает 3, то берем только один младший бит, то есть вместо третьей корзины используем первую.

При каждом удвоении размера новые страницы корзин всегда выделяются одним непрерывным фрагментом, а страницы переполнения и битовой карты вставляются между этими фрагментами по необходимости. Метастраница хранит количество вставленных страниц для каждого из фрагментов в массиве spares, и это дает возможность по номеру корзины вычислить номер ее основной страницы[2].

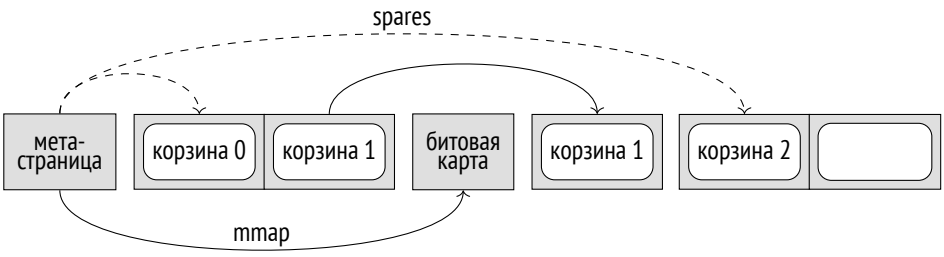
В данном случае после первого удвоения было вставлено две страницы (битовая карта и страница переполнения), а после второго пока не добавилось ничего нового:

```
=> SELECT spares[2], spares[3]
FROM hash_metapage_info(get_raw_page('t_n_idx', 0));
 spares | spares
--------+--------
      2 |      2
(1 row)
```

В метастранице также хранится массив ссылок на страницы битовой карты:

```
=> SELECT mapp[1]
FROM hash_metapage_info(get_raw_page('t_n_idx', 0));
 mapp
------
    3
(1 row)
```

[1] backend/access/hash/hashutil.c, функция _hash_hashkey2bucket.
[2] include/access/hash.h, макрос BUCKET_TO_BLKNO.



Место внутри индексных страниц освобождается при удалении ссылок на мертвые версии строк. Это происходит при самоочистке (она срабатывает при попытке вставки элемента в полностью заполненную страницу[1]) или при обычной очистке.

Но уменьшаться в размерах хеш-индекс не умеет, и однажды выделенные страницы уже не возвращаются операционной системе. Основные страницы всегда закреплены за своими корзинами, даже если в них нет ни одного элемента; опустевшие страницы переполнения отслеживаются в битовой карте и переиспользуются (возможно, уже для другой корзины). Единственный вариант уменьшить физический размер индекса — перестроить
с. 170 его командой `REINDEX` или `VACUUM FULL`.

В плане запроса тип индекса никак не отмечается:

```
=> CREATE INDEX ON flights USING hash(flight_no);

=> EXPLAIN (costs off)
SELECT *
FROM flights
WHERE flight_no = 'PG0001';
                    QUERY PLAN
--------------------------------------------------
 Bitmap Heap Scan on flights
   Recheck Cond: (flight_no = 'PG0001'::bpchar)
   -> Bitmap Index Scan on flights_flight_no_idx
       Index Cond: (flight_no = 'PG0001'::bpchar)
(4 rows)
```

[1] backend/access/hash/hashinsert.c, функция _hash_vacuum_one_page.

24.3. Класс операторов

До версии PostgreSQL 10 хеш-индексы не журналировались, то есть не были защищены от сбоев и не реплицировались, и, как следствие, не рекомендовались к использованию. Но даже в таком виде они представляли определенную ценность. Дело в том, что алгоритм хеширования применяется очень широко (в частности, для хеш-соединений и группировок), и системе требуется знать, какая хеш-функция предназначена для каждого типа данных. Но это соответствие не статично: его нельзя задать раз и навсегда, поскольку PostgreSQL позволяет добавлять новые типы данных на лету. Поэтому соответствие поддерживается классом операторов для хеш-индекса и конкретного типа данных. Собственно функция хеширования представлена опорной функцией класса:

с. 457

с. 397

```
=> SELECT opfname AS opfamily_name,                                    \dAp
  amproc::regproc AS opfamily_procedure
FROM pg_am am
  JOIN pg_opfamily opf ON opfmethod = am.oid
  JOIN pg_amproc amproc ON amprocfamily = opf.oid
WHERE amname = 'hash'
AND amprocnum = 1
ORDER BY opfamily_name, opfamily_procedure;
   opfamily_name    | opfamily_procedure
--------------------+--------------------
 aclitem_ops        | hash_aclitem
 array_ops          | hash_array
 bool_ops           | hashbool
 bpchar_ops         | hashbpchar
 bpchar_pattern_ops | hashbpchar
 bytea_ops          | hashbytea
 ...
 timestamptz_ops    | timestamptz_hash
 timetz_ops         | timetz_hash
 uuid_ops           | uuid_hash
 xid8_ops           | hashxid8
 xid_ops            | hashxid
(38 rows)
```

Эти функции возвращают 32-битные целые числа. Хотя они и не документированы, их можно использовать для вычисления хеш-кода значения соответствующего типа.

Например, для семейства text_ops используется функция hashtext:

```
=> SELECT hashtext('раз'), hashtext('два');
 hashtext  | hashtext
-----------+-----------
 127722028 | 345620034
(1 row)
```

В класс операторов хеш-индекса входит один оператор — «равно». Обычно именно он используется для системных проверок значений на равенство.

\dAo

```
=> SELECT opfname AS opfamily_name,
  left(amopopr::regoperator::text, 20) AS opfamily_operator
FROM pg_am am
  JOIN pg_opfamily opf ON opfmethod = am.oid
  JOIN pg_amop amop ON amopfamily = opf.oid
WHERE amname = 'hash'
ORDER BY opfamily_name, opfamily_operator;
   opfamily_name    |  opfamily_operator
--------------------+----------------------
 aclitem_ops        | =(aclitem,aclitem)
 array_ops          | =(anyarray,anyarray)
 bool_ops           | =(boolean,boolean)
 ...
 xid8_ops           | =(xid8,xid8)
 xid_ops            | =(xid,xid)
(48 rows)
```

24.4. Свойства

с. 404 Посмотрим свойства хеш-индекса, которые этот метод доступа сообщает о себе системе.

Свойства уровня метода доступа

```
=> SELECT a.amname, p.name, pg_indexam_has_property(a.oid, p.name)
FROM pg_am a, unnest(array[
  'can_order', 'can_unique', 'can_multi_col',
  'can_exclude', 'can_include'
]) p(name)
WHERE a.amname = 'hash';
```

```
 amname |     name      | pg_indexam_has_property
--------+---------------+-------------------------
 hash   | can_order     | f
 hash   | can_unique    | f
 hash   | can_multi_col | f
 hash   | can_exclude   | t
 hash   | can_include   | f
(5 rows)
```

Очевидно, что хеш-индексы не позволяют упорядочивать строки: хеш-функция перемешивает данные более или менее случайным образом.

Не поддерживается и ограничение уникальности. Правда, хеш-индексы можно использовать для ограничения исключения, а с единственной функцией «равно» исключение приобретает смысл уникальности: *с. 408*

```
=> ALTER TABLE aircrafts_data
  ADD CONSTRAINT unique_range EXCLUDE USING hash(range WITH =);
=> INSERT INTO aircrafts_data
  VALUES ('744','{"ru": "Боинг 747-400"}',11100);
ERROR:  conflicting key value violates exclusion constraint
"unique_range"
DETAIL:  Key (range)=(11100) conflicts with existing key
(range)=(11100).
```

Также не поддерживаются многоколоночные хеш-индексы и возможность добавлять к индексу дополнительные include-столбцы.

Свойства уровня индекса

```
=>  SELECT p.name, pg_index_has_property('flights_flight_no_idx', p.name)
FROM unnest(array[
  'clusterable', 'index_scan', 'bitmap_scan', 'backward_scan'
]) p(name);
     name      | pg_index_has_property
---------------+-----------------------
 clusterable   | f
 index_scan    | t
 bitmap_scan   | t
 backward_scan | t
(4 rows)
```

Хеш-индекс работает как с обычным индексным сканированием, так и со сканированием по битовой карте.

Кластеризация таблицы по хеш-индексу не предусмотрена. Это логично, поскольку сложно представить, зачем может понадобиться физически упорядочить данные в таблице по значению хеш-функции.

Свойства уровня столбца

Фактически свойства столбцов полностью определены на уровне метода доступа и всегда принимают одни и те же значения.

```
=> SELECT p.name,
  pg_index_column_has_property('flights_flight_no_idx', 1, p.name)
FROM unnest(array[
  'asc', 'desc', 'nulls_first', 'nulls_last', 'orderable',
  'distance_orderable', 'returnable', 'search_array', 'search_nulls'
]) p(name);
```

```
        name        | pg_index_column_has_property
--------------------+------------------------------
 asc                | f
 desc               | f
 nulls_first        | f
 nulls_last         | f
 orderable          | f
 distance_orderable | f
 returnable         | f
 search_array       | f
 search_nulls       | f
(9 rows)
```

Поскольку хеш-функция не сохраняет отношение порядка, к хеш-индексу не применимы свойства, касающиеся упорядоченности.

Хеш-индекс не может участвовать в сканировании только индекса, поскольку не сохраняет ключ индексации и требует перепроверки по таблице.

Хеш-индекс не работает с неопределенными значениями: операция «равно» не имеет смысла для `NULL`.

Поиск значений из массива не реализован.

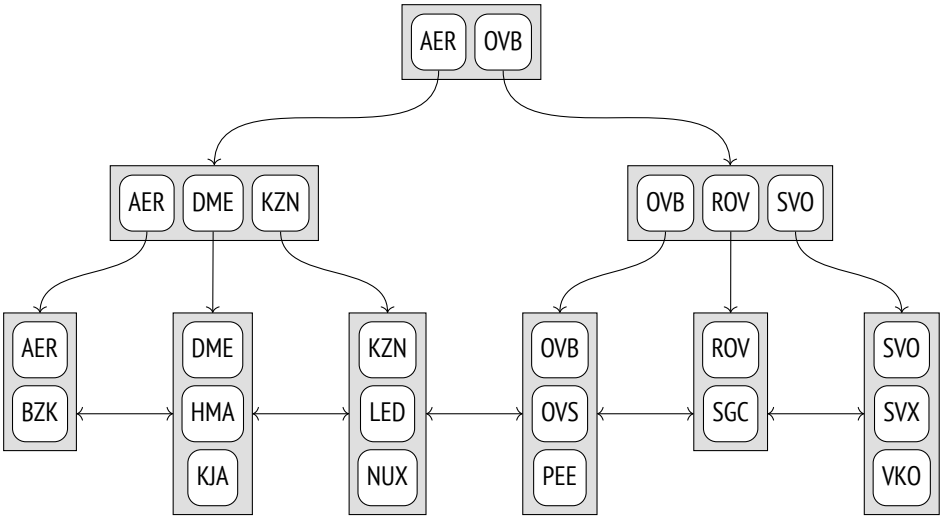
25

B-дерево

25.1. Общий принцип

B-дерево (метод доступа `btree`) — структура данных, которая позволяет быстро найти нужный элемент в листьях дерева, спускаясь к нему от корня [1]. Чтобы путь поиска был однозначно определен, элементы в дереве должны быть упорядочены. B-дерево предназначено для порядковых типов данных, значения которых можно сравнивать и сортировать.

На схематично показанном индексе, построенном по кодам аэропортов, внутренние узлы изображены горизонтально, а листовые — вертикально:



[1] postgrespro.ru/docs/postgresql/18/btree; backend/access/nbtree/README.

Каждый узел дерева содержит несколько элементов, состоящих из ключа индексирования и ссылки. Элементы внутренних узлов дерева ссылаются на узлы следующего уровня, а элементы листовых узлов — на версии строк таблицы (на рисунке эти ссылки не показаны).

B-деревья обладают несколькими важными свойствами:

- Они сбалансированы, то есть все листья находятся на одной глубине. Поэтому поиск любого значения занимает одинаковое время.
- Они сильно ветвисты, то есть каждый узел содержит много элементов, часто сотни (на рисунке узлы состоят всего из трех элементов исключительно для наглядности). За счет этого глубина B-деревьев получается небольшой даже для очень больших таблиц.

  > Доподлинно неизвестно, что означает буква «B» в названии дерева. Одинаково хорошо подходит и balanced (сбалансированное), и bushy (ветвистое). Но на удивление часто встречается расшифровка binary (двоичное), которая, конечно, неверна.

- Данные в индексе упорядочены по возрастанию (или по убыванию) как между узлами, так и внутри каждого узла. Узлы одного уровня связаны между собой двунаправленным списком, поэтому можно получить упорядоченный набор данных, просто пройдя по списку в одну или в другую сторону, не повторяя каждый раз путь от корня.

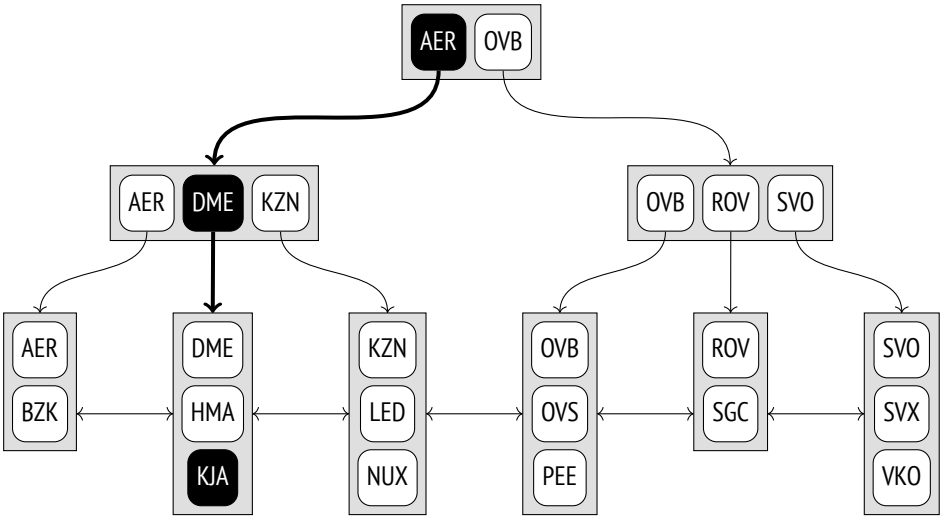
25.2. Поиск и вставка

Поиск по равенству

Рассмотрим поиск[1] значения в дереве по условию «*индексированный-столбец* = *выражение*». Допустим, нас интересует аэропорт KJA (Красноярск).

Поиск начинается с корневого узла, и метод доступа должен определить, в какой из дочерних узлов спускаться. Выбирается ключ K_i, для которого верно $K_i \leqslant$ *выражение* $< K_{i+1}$.

[1] backend/access/nbtree/nbtsearch.c, функция _bt_search.



В корневом узле находятся ключи AER и OVB. Выполняется условие AER $\leqslant$ $\leqslant$ KJA < OVB, поэтому спускаться надо в дочерний узел, на который ссылается элемент с ключом AER.

Эта процедура повторяется рекурсивно до тех пор, пока мы не дойдем до листового узла, из которого уже можно получить идентификатор версии строки. В данном случае в дочернем узле выполняется условие DME $\leqslant$ KJA < < KZN, поэтому надо спуститься в листовой узел, на который ссылается элемент с ключом DME.

Можно заметить, что во внутренних узлах дерева крайний левый ключ избыточен: чтобы выбрать дочерний узел корня, достаточно выполнения условия KJA < OVB. Такие ключи не хранятся в B-дереве, и на следующих рисунках я буду оставлять соответствующие элементы пустыми.

В листовом узле нужный элемент быстро находится простым двоичным поиском.

В реальности процедура поиска вовсе не так проста, как кажется на первый взгляд. Нужно учитывать, что значения в индексе могут быть упорядочены не только по возрастанию, как на рисунке, но и по убыванию. Даже в уникальном индексе может оказаться несколько подходящих значений, и поиск должен вернуть их все. Причем одинаковых значений может оказаться так

с. 533

много, что они не поместятся в один узел, и тогда поиск придется продолжить в соседней листовой странице.

> Из-за того, что в индексе могут находиться неуникальные значения, правильнее было бы говорить не «по возрастанию», а «по неубыванию» (и не «по убыванию», а «по невозрастанию»). Я все-таки буду придерживаться более простого варианта. К тому же идентификатор версии строки является частью индексного ключа, что позволяет считать элементы индекса уникальными, даже когда значения совпадают.

v. 12

Кроме всего прочего, во время поиска другие процессы могут изменять данные, страницы могут расщепляться на две, и дерево может перестраиваться. Все алгоритмы построены таким образом, чтобы эти одновременные действия по возможности не мешали друг другу и не требовали лишних блокировок. Но в эти детали я вдаваться не буду.

Поиск по неравенству

При поиске по условию «*индексированный-столбец* ⩽ *выражение*» (или «*индексированный-столбец* ⩾ *выражение*») сначала надо найти в индексе значение по условию равенства, а затем двигаться по листовым узлам в нужную сторону до самого конца.

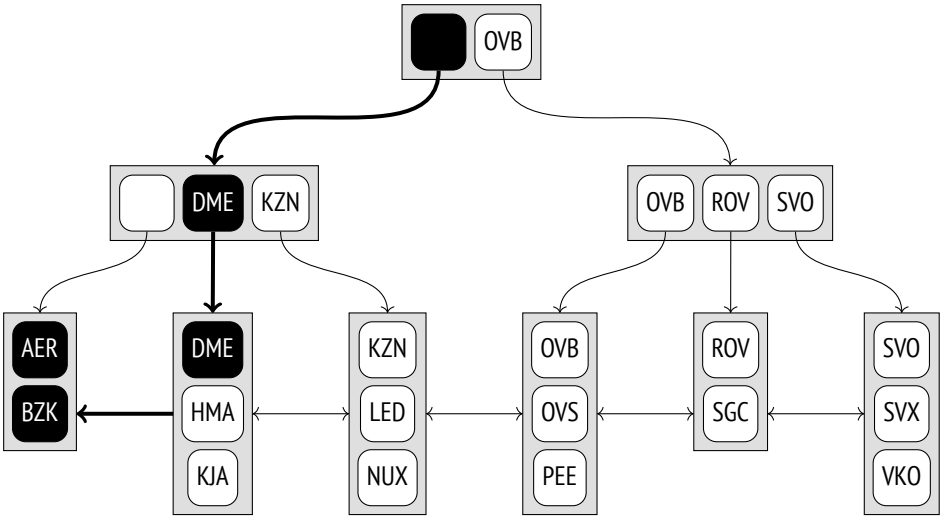


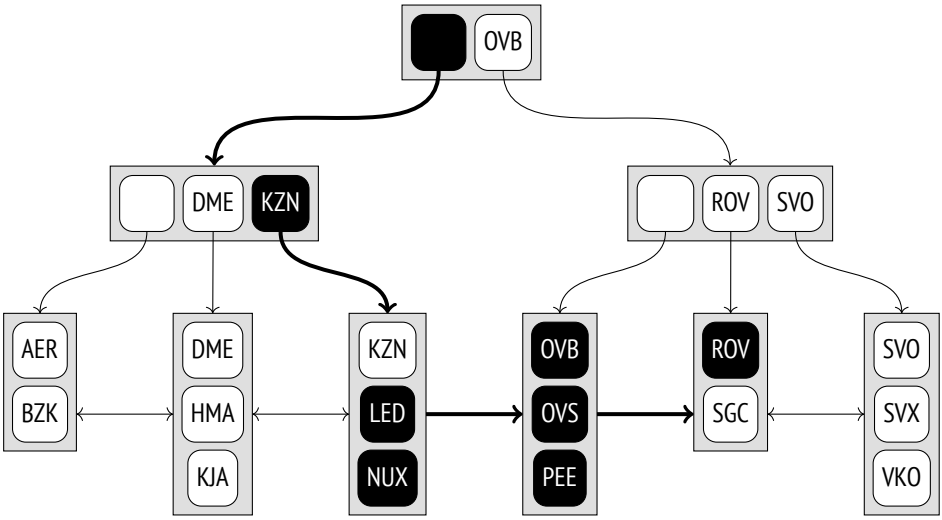
Рисунок показывает процесс поиска аэропортов с кодами, меньшими или равными DME (Домодедово).

Для операторов «больше» и «меньше» поиск выполняется так же, надо только исключить исходно найденное значение.

Поиск по диапазону

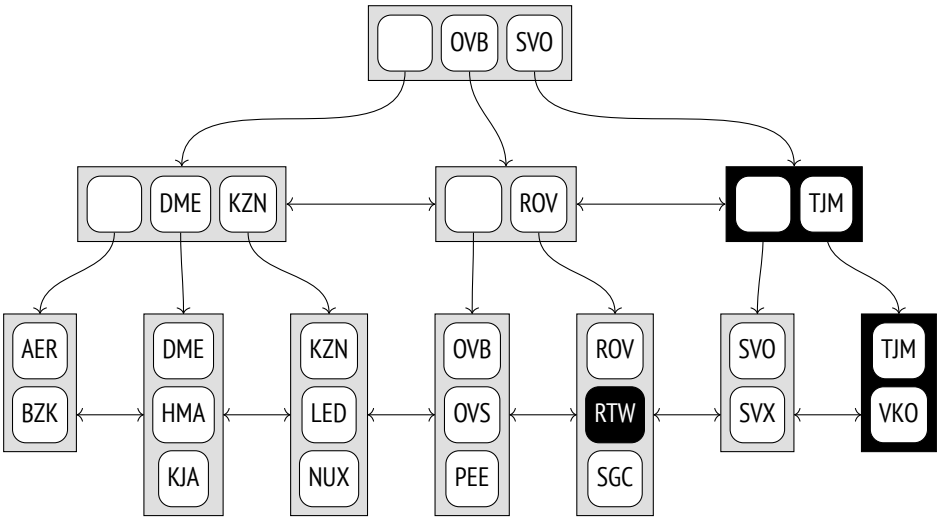
При поиске по диапазону «*выражение*$_1$ ⩽ *индексированный-столбец* ⩽ *выражение*$_2$» сначала надо найти *выражение*$_1$, а затем двигаться вправо по листовым узлам до *выражения*$_2$.

На рисунке показан процесс поиска аэропортов с кодами от LED (Санкт-Петербург) до ROV (Ростов-на-Дону) включительно.



Вставка

Место нового элемента в листовых узлах однозначно определяется порядком расположения ключей. Например, при вставке в таблицу аэропорта с кодом RTW (Саратов) элемент будет добавлен в предпоследний листовой узел между имеющимися ROV и SGC.



Но может получиться так, что в листовом узле не хватит места для нового элемента. Например (если считать, что узел вмещает максимум три элемента), при вставке аэропорта с кодом TJM (Тюмень) последний листовой узел переполнится. В этом случае узел *расщепляется* на два, часть элементов старого узла переносится в новый, а в родительский узел добавляется ссылка на нового потомка. Конечно, родительский узел тоже может переполниться. В таком случае и он расщепляется на два узла, и так далее. Если дело доходит до расщепления корня, над образовавшимися узлами создается еще один узел, который становится новым корнем дерева. В этом случае глубина дерева увеличивается на единицу.

В данном примере добавление аэропорта TJM привело к расщеплению двух узлов; новые узлы выделены на рисунке. Чтобы можно было расщеплять любые узлы дерева, двунаправленным списком связаны не только узлы нижнего уровня, но и узлы остальных уровней тоже.

Описанная процедура вставок и расщеплений гарантирует сбалансированность дерева, а поскольку в реальности количество элементов в узле обычно велико, наращивание дополнительного уровня происходит редко.

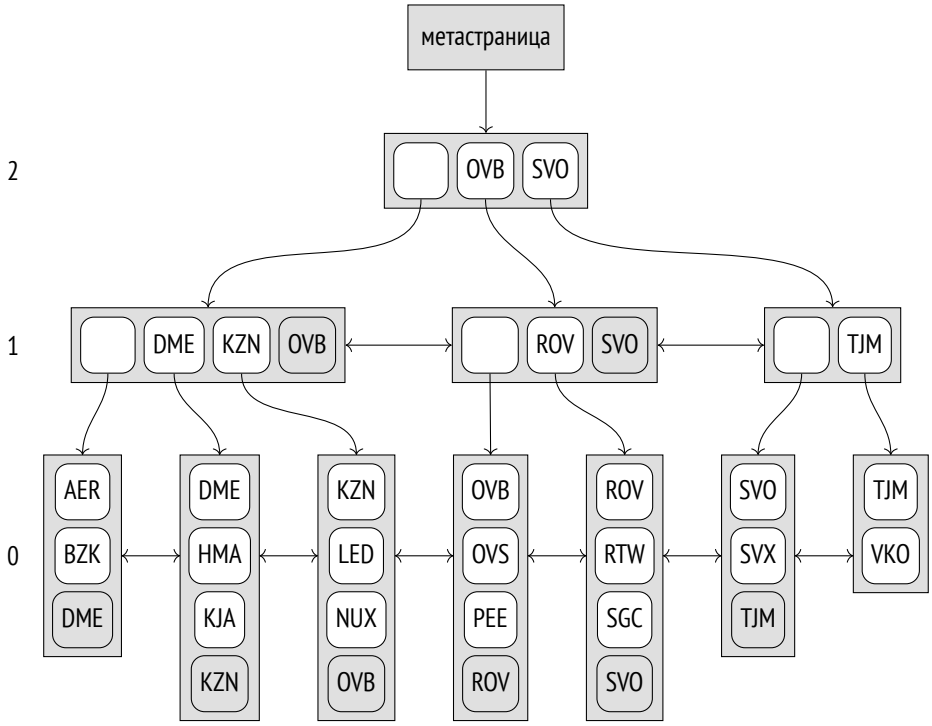
Проблема в том, что расщепленные узлы никогда не объединяются, даже если после очистки в них остается мало элементов. Это ограничение не B-дерева как структуры данных, а реализации PostgreSQL. Поэтому если при

вставке узел оказывается заполнен, метод доступа старается в первую очередь очистить узел от лишней информации, чтобы освободить место и не допустить лишнее расщепление. с. 123

25.3. Страничная организация

Каждый узел B-дерева занимает одну страницу. Размер страницы и определяет вместимость узла.

Расщепления приводят к тому, что в разные моменты времени корнем дерева могут выступать разные страницы. Но алгоритм поиска должен начинать работу именно с корня. Он находит номер текущей корневой страницы в нулевой странице индекса (называемой метастраницей). Метастраница содержит и некоторую другую служебную информацию.



Представление информации на индексных страницах немного отличается от того, что было изображено ранее. Все страницы, кроме самых правых на своем уровне, содержат дополнительный «верхний ключ», гарантированно не меньший любого значения на данной странице. На предыдущем рисунке верхние ключи выделены серым.

Заглянем в страницы настоящего индекса по шестизначным номерам бронирований с помощью расширения pageinspect. В метастранице нас интересует номер корневой страницы и максимальный уровень дерева (уровни нумеруются с нуля, начиная с листьев):

```
=> SELECT root, level
FROM bt_metap('bookings_pkey');
 root | level
------+-------
  290 |     2
(1 row)
```

Ключи в индексных записях выводятся как последовательность байтов, что не очень-то удобно. Вот пример:

```
=> SELECT data
FROM bt_page_items('bookings_pkey',290)
WHERE itemoffset = 2;
          data
------------------------
 0f 30 43 39 41 42 31 00
(1 row)
```

Для расшифровки значений придется написать adhoc-функцию. Она будет работать не во всех случаях и не на всех платформах, но годится для примеров в этой главе:

```
=> CREATE FUNCTION data_to_text(data text)
RETURNS text
AS $$
DECLARE
  raw bytea := ('\x'||replace(data,' ',''))::bytea;
  pos integer := 0;
  len integer;
  res text := '';
```

```
BEGIN
  WHILE (octet_length(raw) > pos) LOOP
    len := (get_byte(raw,pos) - 3) / 2;
    EXIT WHEN len <= 0;
    IF pos > 0 THEN
      res := res || ', ';
    END IF;
    res := res || (
      SELECT string_agg( chr(get_byte(raw, i)),'')
      FROM generate_series(pos+1,pos+len) i
    );
    pos := pos + len + 1;
  END LOOP;
  RETURN res;
END;
$$ LANGUAGE plpgsql;
```

Теперь мы можем посмотреть на содержимое корневой страницы:

```
=> SELECT itemoffset, ctid, data_to_text(data)
FROM bt_page_items('bookings_pkey',290);
 itemoffset |   ctid   | data_to_text
------------+----------+--------------
          1 | (3,0)    |
          2 | (289,1)  | 0C9AB1
          3 | (575,1)  | 192F03
          4 | (860,1)  | 25D715
          5 | (1145,1) | 32785C
          6 | (1430,1) | 3F0EB8
 ...
         17 | (4565,1) | C993F6
         18 | (4850,1) | D63931
         19 | (5135,1) | E2CB14
         20 | (5420,1) | EF6FEA
         21 | (5705,1) | FC147D
(21 rows)
```

Первая запись, как я и говорил, не хранит ключ. Столбец `ctid` содержит ссылки на дочерние страницы.

Допустим, мы ищем бронирование с номером E2D725. В этом случае мы выбираем запись 19, поскольку E2CB14 $\leqslant$ E2D725 $<$ EF6FEA, и спускаемся на страницу 5135.

```
=> SELECT itemoffset, ctid, data_to_text(data)
FROM bt_page_items('bookings_pkey',5135);
 itemoffset |   ctid   | data_to_text
------------+----------+--------------
          1 | (5417,1) | EF6FEA     ⟵ верхний ключ
          2 | (5132,0) |
          3 | (5133,1) | E2D71D
          4 | (5134,1) | E2E2F4
          5 | (5136,1) | E2EDE7
 ...
        282 | (5413,1) | EF41BE
        283 | (5414,1) | EF4D69
        284 | (5415,1) | EF58D4
        285 | (5416,1) | EF6410
(285 rows)
```

Первая запись этой страницы — несколько неожиданно — содержит верхний ключ. Логически он должен был бы располагаться в конце страницы, но с точки зрения реализации удобнее держать его в начале, чтобы не переносить каждый раз, когда содержимое страницы меняется.

На этой странице мы выбираем запись 3, поскольку E2D71D $\leqslant$ E2D725 $<$ $<$ E2E2F4, и спускаемся на страницу 5133.

```
=> SELECT itemoffset, ctid, data_to_text(data)
FROM bt_page_items('bookings_pkey',5133);
 itemoffset |    ctid     | data_to_text
------------+-------------+--------------
          1 | (11921,1)   | E2E2F4
          2 | (11919,76)  | E2D71D
          3 | (11919,77)  | E2D725
          4 | (11919,78)  | E2D72D
          5 | (11919,79)  | E2D733
 ...
        364 | (11921,124) | E2E2DB
        365 | (11921,125) | E2E2DF
        366 | (11921,126) | E2E2E5
        367 | (11921,127) | E2E2ED
(367 rows)
```

Это листовая страница индекса. Первая запись — верхний ключ, а все остальные записи ссылаются на табличные версии строк.

Вот и наше бронирование:

```
=> SELECT * FROM bookings WHERE ctid = '(11919,77)';
 book_ref |       book_date        | total_amount
----------+------------------------+--------------
 E2D725   | 2017-01-25 04:10:00+03 |     28000.00
(1 row)
```

Примерно такая работа и выполняется на низком уровне, когда мы запрашиваем бронирование по коду:

```
=> EXPLAIN (costs off) SELECT *
FROM bookings WHERE book_ref = 'E2D725';
                 QUERY PLAN
--------------------------------------------
 Index Scan using bookings_pkey on bookings
   Index Cond: (book_ref = 'E2D725'::bpchar)
(2 rows)
```

Компактное хранение дубликатов

v. 13

Неуникальные индексы могут содержать множество одинаковых ключей со ссылками на разные табличные строки. Поскольку повторяющиеся ключи занимают много места, дубликаты «схлопываются» в одну индексную запись, содержащую ключ и список табличных идентификаторов[1]. В ряде случаев эта процедура, называемая *исключением дубликатов* (deduplication), позволяет существенно сократить размер индекса.

Но и уникальные индексы могут содержать дубликаты ключей из-за многоверсионности, поскольку индекс хранит ссылки на все версии табличных строк. С раздуванием индекса из-за ссылок на неактуальные и обычно короткоживущие версии помогает справиться механизм hot-обновлений, но *с. 111* он работает не всегда. В таком случае исключение дубликатов позволяет выиграть время, необходимое для очистки таблицы от неактуальных версий, и сократить количество вынужденных расщеплений страниц.

Чтобы не тратить ресурсы на исключение дубликатов, когда в этом нет необходимости, «схлопывание» выполняется, только когда на листовой странице

[1] postgrespro.ru/docs/postgresql/18/btree#BTREE-DEDUPLICATION.

не хватает места для вставки новой строки. В этом случае[1] самоочистка и исключение дубликатов[2] могут освободить место и предотвратить ненужное расщепление. Впрочем, если дубликаты возникают редко, механизм исключения можно отключить с помощью параметра хранения *deduplicate_items*.

Исключение дубликатов может применяться не для всех индексов. Основное ограничение состоит в том, что равенство ключей должно проверяться простым двоичным сравнением внутреннего представления. Далеко не все типы данных можно сравнивать таким образом. Числа с плавающей точкой (`float` и `double precision`) имеют два разных представления нуля. Числа с произвольной точностью (`numeric`) допускают представление одного и того же числа в разных масштабах, а тип `jsonb` использует такие числа. Не годятся также текстовые типы при использовании недетерминированных правил сортировки[3], позволяющих представлять одинаковые символы разными последовательностями байтов (стандартные правила сортировки детерминированы).

Кроме того, в настоящее время исключение дубликатов невозможно для составных типов, диапазонов и массивов, а также не работает с include-индексами.

Чтобы проверить, поддерживается ли исключение дубликатов для конкретного индекса, можно заглянуть в поле `allequalimage` метастраницы:

```
=> CREATE INDEX ON tickets(book_ref);
=> SELECT allequalimage FROM bt_metap('tickets_book_ref_idx');
 allequalimage
---------------
 t
(1 row)
```

В данном случае исключение дубликатов поддерживается. И действительно, на одной из листовых страниц мы видим, что некоторые индексные записи содержат единственный табличный идентификатор (`htid`), а некоторые — список идентификаторов (`tids`):

[1] backend/access/nbtree/nbtinsert.c, функция _bt_delete_or_dedup_one_page.
[2] backend/access/nbtree/nbtdedup.c, функция _bt_dedup_pass.
[3] postgrespro.ru/docs/postgresql/18/collation.

```
=> SELECT itemoffset, htid, left(tids::text,27) tids,
  data_to_text(data) AS data
FROM bt_page_items('tickets_book_ref_idx',1)
WHERE itemoffset > 1;
 itemoffset |    htid    |            tids             |  data
------------+------------+-----------------------------+--------
          2 | (32965,40) |                             | 000004
          3 | (47429,51) |                             | 00000F
          4 | (3648,56)  | {"(3648,56)","(3648,57)"}   | 000010
          5 | (6498,47)  |                             | 000012
 ...
        271 | (21492,46) |                             | 000890
        272 | (26601,55) | {"(26601,55)","(26601,56)"} | 0008AC
        273 | (25669,37) |                             | 0008B6
(272 rows)
```

Компактное хранение внутренних индексных записей

v. 12

Исключение дубликатов позволяет разместить больше записей в листовых страницах индекса. Но для предотвращения лишних расщеплений уплотнение информации во внутренних страницах играет не меньшую роль. Основной объем индекса составляют листовые страницы, но эффективность поиска в индексе напрямую зависит от глубины дерева.

Индексные записи во внутренних страницах содержат ключи индексирования, но их значения используются только для определения поддерева, в которое надо спускаться при поиске. В составных индексах для этого часто достаточно только первого атрибута ключа или нескольких первых. Остальные атрибуты могут быть отброшены, чтобы не занимать место на странице.

Процедура *исключения некоторых атрибутов* (suffix truncation) выполняется, когда происходит расщепление листовой страницы и во внутренней странице необходимо разместить новую ссылку[1].

> В принципе, можно пойти еще дальше и оставлять только значимую *часть* атрибута, например несколько первых символов строки, достаточных для различения поддеревьев. Но это пока не реализовано: атрибут или целиком остается в индексной записи, или полностью исключается из нее.

[1] backend/access/nbtree/nbtinsert.c, функция _bt_split.

Для примера — несколько записей корневой страницы индекса, построенного на таблице билетов по номеру бронирования и имени пассажира:

```
=> CREATE INDEX tickets_bref_name_idx
ON tickets(book_ref, passenger_name);
=> SELECT itemoffset, ctid, data_to_text(data)
FROM bt_page_items('tickets_bref_name_idx',229)
WHERE itemoffset BETWEEN 8 AND 13;
 itemoffset |   ctid   |        data_to_text
------------+----------+----------------------------
          8 | (1607,1) | 1A98A0
          9 | (1833,2) | 1E57D1, SVETLANA MAKSIMOVA
         10 | (2054,1) | 220797
         11 | (2282,1) | 25DB06
         12 | (2509,2) | 299FE4, YURIY AFANASEV
         13 | (2736,1) | 2D62C9
(6 rows)
```

Видно, что в некоторых индексных записях второй атрибут исключен.

В листовых страницах, конечно, должны храниться все ключевые атрибуты, в том числе и значения дополнительных include-столбцов. В противном случае индекс не мог бы использоваться в сканировании только индекса. Исключение составляет «верхний ключ», который может храниться не полностью.

25.4. Сортировка и составные индексы

Порядок столбцов

Порядок, в котором столбцы перечисляются при создании индекса, крайне важен, поскольку данные внутри страниц будут отсортированы сначала по первому столбцу, затем по второму и так далее. Как правило, эффективный поиск в таком индексе возможен по условиям на непрерывную последовательность столбцов, начиная с самого первого: по одному первому, по первому и второму, с первого по третий и так далее. При этом все остальные условия используются только для фильтрации значений, найденных по основным критериям[1].

[1] postgrespro.ru/docs/postgresql/18/indexes-multicolumn.

Вот порядок индексных записей в первой листовой странице индекса, созданного ранее на таблице билетов по номеру бронирования и по имени пассажира:

```
=> SELECT itemoffset, data_to_text(data)
FROM bt_page_items('tickets_bref_name_idx',1)
WHERE itemoffset > 1;
 itemoffset |        data_to_text
------------+----------------------------
          2 | 000004, PETR MAKAROV
          3 | 00000F, ANNA ANTONOVA
          4 | 000010, ALEKSANDR SOKOLOV
          5 | 000010, LYUDMILA BOGDANOVA
          6 | 000012, TAMARA ZAYCEVA
 ...
        189 | 00040C, DMITRIY FEDOROV
        190 | 00041E, EGOR FEDOROV
        191 | 00041E, ILYA STEPANOV
        192 | 000447, VIKTOR VASILEV
        193 | 00044D, NADEZHDA KULIKOVA
(192 rows)
```

В этом случае эффективный поиск билетов возможен либо по номеру бронирования и имени пассажира, либо только по номеру бронирования:

```
=> EXPLAIN (costs off) SELECT *
FROM tickets
WHERE book_ref = '000010';
                    QUERY PLAN
-------------------------------------------------
 Index Scan using tickets_book_ref_idx on tickets
   Index Cond: (book_ref = '000010'::bpchar)
(2 rows)
=> EXPLAIN (costs off) SELECT *
FROM tickets
WHERE book_ref = '000010' AND passenger_name = 'LYUDMILA BOGDANOVA';
                             QUERY PLAN
--------------------------------------------------------------------
 Index Scan using tickets_bref_name_idx on tickets
   Index Cond: ((book_ref = '000010'::bpchar) AND (passenger_name...
(2 rows)
```

Но для поиска только по имени пассажира придется перебрать все строки:

```
=> EXPLAIN (costs off) SELECT *
FROM tickets
WHERE passenger_name = 'LYUDMILA BOGDANOVA';
                         QUERY PLAN
------------------------------------------------------------
 Gather
   Workers Planned: 2
   -> Parallel Seq Scan on tickets
       Filter: (passenger_name = 'LYUDMILA BOGDANOVA'::text)
(4 rows)
```

И наоборот, если создать индекс по именам и по номерам бронирования, эффективно будут поддерживаться запросы по имени или одновременно по имени и по номеру бронирования:

```
=> CREATE INDEX tickets_name_bref_idx
ON tickets(passenger_name, book_ref);
=> SELECT itemoffset, data_to_text(data)
FROM bt_page_items('tickets_name_bref_idx',1)
WHERE itemoffset > 1;
 itemoffset |          data_to_text
------------+-----------------------------
          2 | ADELINA ABRAMOVA, E37EDB
          3 | ADELINA AFANASEVA, 1133B7
          4 | ADELINA AFANASEVA, 4F3370
          5 | ADELINA AKIMOVA, 7D2881
 ...
        185 | ADELINA LEBEDEVA, 0A00E3
        186 | ADELINA LEBEDEVA, DAEADE
        187 | ADELINA LEBEDEVA, DFD7E5
        188 | ADELINA LOGINOVA, 8022F3
        189 | ADELINA LOGINOVA, EE67B9
        190 | ADELINA LUKYANOVA, 292786
        191 | ADELINA LUKYANOVA, 54D3F9
(190 rows)
=> EXPLAIN (costs off) SELECT * FROM tickets
WHERE passenger_name = 'LYUDMILA BOGDANOVA';
                            QUERY PLAN
------------------------------------------------------------------
 Bitmap Heap Scan on tickets
   Recheck Cond: (passenger_name = 'LYUDMILA BOGDANOVA'::text)
   -> Bitmap Index Scan on tickets_name_bref_idx
       Index Cond: (passenger_name = 'LYUDMILA BOGDANOVA'::text)
(4 rows)
```

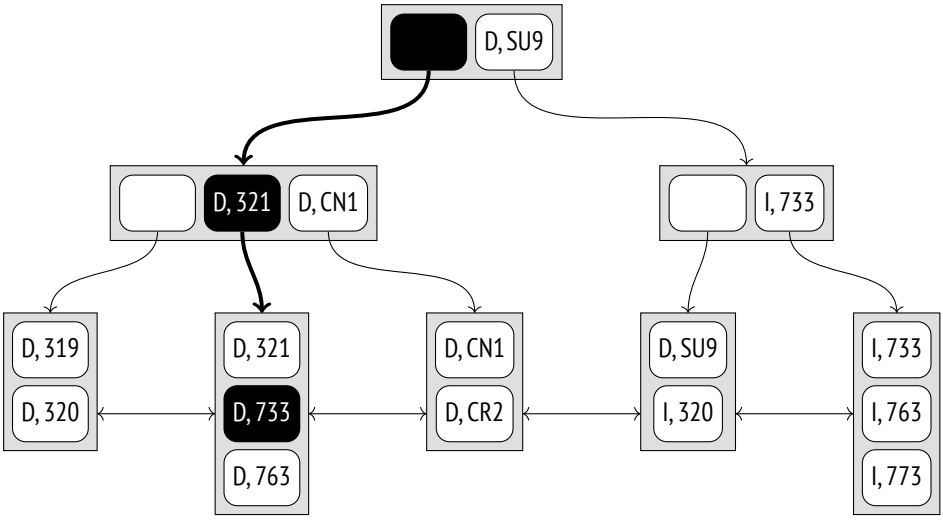
Оптимизация пропуска

v. 18

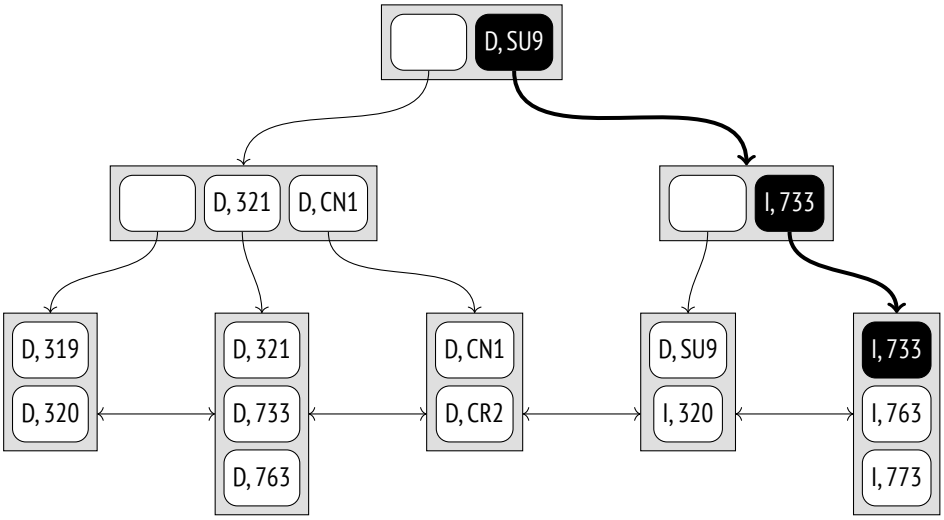
Важное исключение представляет собой случай, когда первый столбец содержит не слишком много уникальных значений. Если условие на такой столбец отсутствует, индекс все равно может значительно ускорить поиск. Для этого надо обратиться к индексу несколько раз, подставляя на место пропущенного условия проверку на равенство каждому из уникальных значений.

Представим себе таблицу, которая показывает, на каких авиалиниях эксплуатируются самолеты. Ее первый столбец `route_type` имеет всего два возможных значения: `D` (внутренние линии, domestic) и `I` (международные, international). Второй столбец `aircraft_code` хранит код модели. Некоторые самолеты летают только внутри страны, некоторые — только за рубеж, а часть используется на любых рейсах. На первом и втором столбцах построен индекс.

Чтобы разобраться с оптимизацией пропуска, предположим сначала, что мы сумели добавить к условию `aircraft_code = '733'` еще одно: `route_type = ANY('{D,I}')`. Индексное сканирование эффективно учтет такое условие благодаря тому, что B-дерево поддерживает свойство SEARCH ARRAY. *с. 411* Сначала поиск в индексе выполняется по условию `route_type = 'D' AND aircraft_code = '733'`:



Затем выполняется еще один поиск, теперь по условию `route_type = 'I' AND aircraft_code = '733'`:



С точки зрения планировщика оба поиска в индексе происходят в рамках одного индексного сканирования, поэтому в плане выполнения будет один узел Bitmap Index Scan. Посмотрим на реальном запросе:

```
=> CREATE INDEX ON flights(status, aircraft_code);
=> EXPLAIN (analyze, costs off, timing off, summary off)
SELECT * FROM flights
WHERE status IN ('Arrived', 'Cancelled', 'Delayed',
                 'Departed', 'On Time', 'Scheduled')
  AND aircraft_code = '733';
                              QUERY PLAN
-----------------------------------------------------------------------
 Bitmap Heap Scan on flights (actual rows=8263.00 loops=1)
   Recheck Cond: (((status)::text = ANY ('{Arrived,Cancelled,Dela...
   Heap Blocks: exact=133
   Buffers: shared hit=136 read=14
   -> Bitmap Index Scan on flights_status_aircraft_code_idx (actu...
        Index Cond: (((status)::text = ANY ('{Arrived,Cancelled,De...
        Index Searches: 3
        Buffers: shared hit=3 read=14
(8 rows)
```

Строка Index Searches показывает количество поисков в индексе. В данном случае сканирование потребовало всего три спуска от корня к листьям, а не

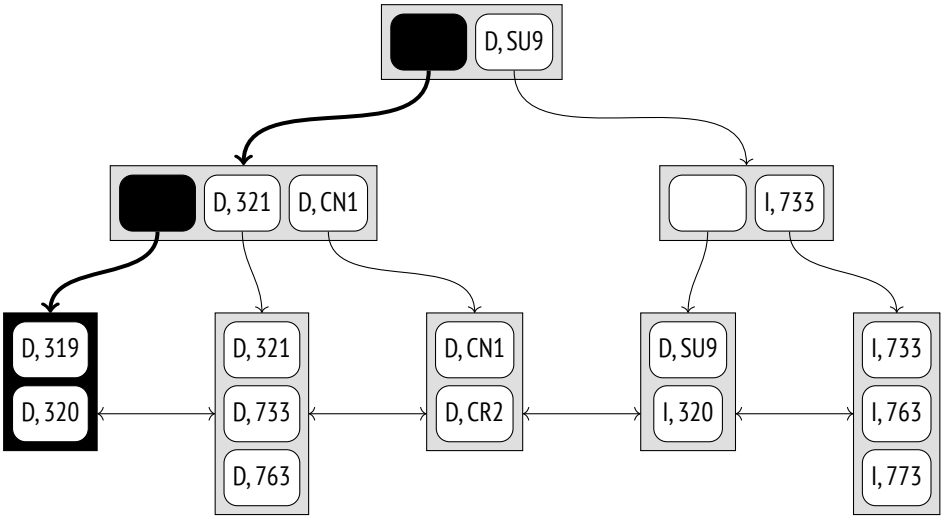
шесть, поскольку строки с четырьмя статусами оказались удачно локализованы на одной странице данных:

```
=> SELECT itemoffset, ctid, data_to_text(data)
FROM bt_page_items('flights_status_aircraft_code_idx',174)
WHERE data_to_text(data) LIKE '% 733';
 itemoffset |   ctid    |  data_to_text
------------+-----------+----------------
          6 | (24,8207) | Cancelled, 733
         15 | (1015,64) | Delayed, 733
         22 | (24,8194) | Departed, 733
         30 | (24,8214) | On Time, 733
(4 rows)
```

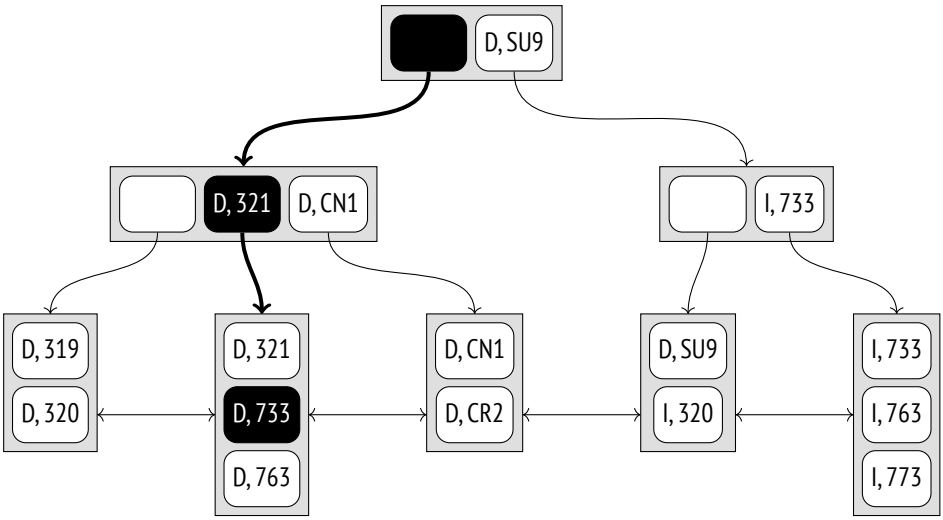
Нерелевантные части индекса при таком сканировании пропускаются, поэтому операция и называется *оптимизацией пропуска*, или skip scan. Это не отдельный вид сканирования, а именно оптимизация, которая работает с любым индексным доступом (Index Scan, Index Only Scan, Bitmap Index Scan).

Что же происходит, когда условие на первый столбец не указано? Оценить количество уникальных значений можно по статистике, но полагаться на статистику для получения точного списка, конечно, нельзя. *с. 351*

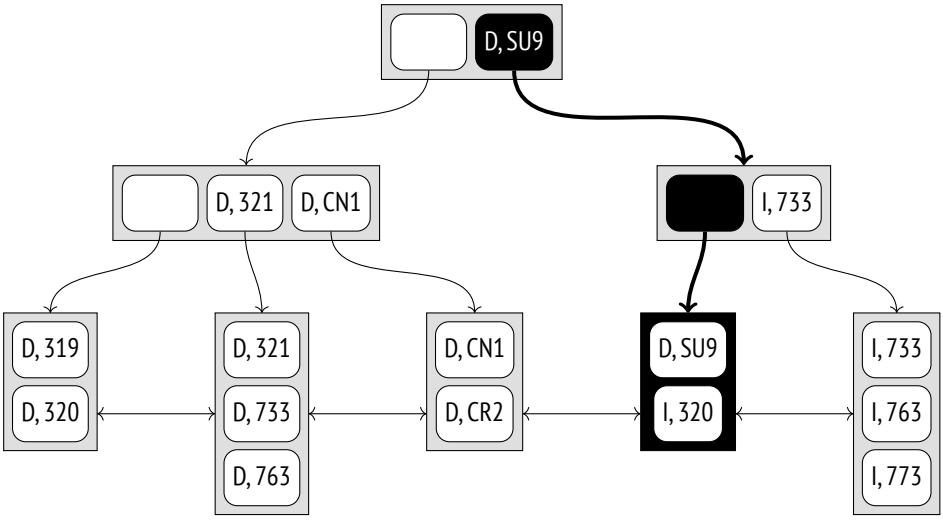
Итак, мы хотим выполнить запрос к таблице по условию `aircraft_code = '733'`. Нам неизвестно наименьшее значение `route_type`, но его можно получить из самой первой записи самой первой листовой страницы индекса:



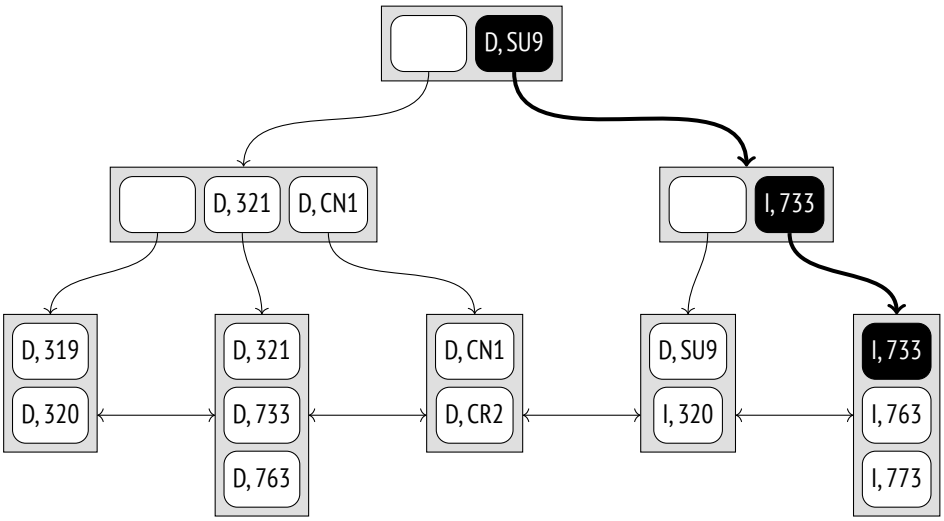
Теперь мы знаем, что первое значение столбца `route_type` равно `D`, и можем выполнить второй поиск по условию на оба столбца: `route_type = 'D' AND aircraft_code = '733'`.



Мы нашли первое подходящее значение. Если бы на прочитанной листовой странице оказалась запись с другим значением первого столбца, мы бы знали следующее точное условие. В данном случае нам не повезло, поэтому приходится выполнить поиск по условию `route_type > 'D'`:



На найденной листовой странице не оказалось нужной записи, но мы узнали второе значение столбца route_type. Поэтому выполняем следующий поиск по условию route_type = 'I' AND aircraft_code = '733':



Сканирование можно завершать, поскольку мы добрались до последней листовой страницы индекса. В противном случае пришлось бы выполнить еще один поиск — по условию route_type > 'I', — чтобы убедиться, что других значений нет.

Запрос без условия на первый столбец потребовал немного больше поисков в индексе и прочитал немного больше страниц, чем предыдущий вариант, но все же выполнился весьма эффективно:

```
=> EXPLAIN (analyze, costs off, timing off, summary off)
SELECT * FROM flights
WHERE aircraft_code = '733';
                              QUERY PLAN
-----------------------------------------------------------------------
 Bitmap Heap Scan on flights (actual rows=8263.00 loops=1)
   Recheck Cond: (aircraft_code = '733'::bpchar)
   Heap Blocks: exact=133
   Buffers: shared hit=154 read=4
   -> Bitmap Index Scan on flights_status_aircraft_code_idx (actu...
       Index Cond: (aircraft_code = '733'::bpchar)
       Index Searches: 4
       Buffers: shared hit=21 read=4
(8 rows)
```

Оптимизация пропуска может применяться и в других ситуациях, когда условия на часть столбцов отсутствуют или не являются проверкой равенства. Значение Index Searches, большее единицы, говорит о том, что при сканировании индекса выполнялись пропуски.

Конечно, приведенный пример не претендует на полноту описания. В частности, в некоторых случаях начатый поиск может продолжаться на соседних листовых страницах в надежде на то, что в них обнаружится подходящая индексная запись. Это позволяет укрупнить отдельные обращения к индексу и уменьшить их количество, экономя на спуске от корня к листовым узлам[1].

Другой пример: для дискретных типов данных в классе операторов B-дерева можно определить опорную функцию, которая сообщит минимальное и максимальное значения данного типа и функции для перехода к следующему и предыдущему значениям[2]. Благодаря этой информации отыскивать очередное значение в индексе не потребуется; сканирование сможет сразу перейти к его проверке.

Направление сортировки

При создании индекса нужно учитывать не только порядок столбцов, но и порядок сортировки. По умолчанию значения упорядочены по возрастанию (ASC), но можно указать и обратный порядок (DESC). В индексе по одному столбцу это не играет особой роли, поскольку индекс можно сканировать в любую сторону. Но в составном индексе порядок важен.

Только что созданный индекс можно использовать для получения данных, отсортированных по обоим столбцам по возрастанию или по обоим столбцам по убыванию:

```
=> EXPLAIN (costs off) SELECT *
FROM tickets ORDER BY passenger_name, book_ref;
                    QUERY PLAN
--------------------------------------------------
 Index Scan using tickets_name_bref_idx on tickets
(1 row)
```

[1] backend/access/nbtree/nbtutils.c, функция _bt_advance_array_keys.
[2] include/utils/skipsupport.h.

```
=> EXPLAIN (costs off) SELECT *
FROM tickets ORDER BY passenger_name DESC, book_ref DESC;
                          QUERY PLAN
---------------------------------------------------------------
 Index Scan Backward using tickets_name_bref_idx on tickets
(1 row)
```

Но из этого индекса невозможно сразу же получить данные, которые по одному столбцу будут отсортированы по возрастанию, а по другому — по убыванию. Индекс поставляет частично упорядоченные данные, которые затем сортируются по второму атрибуту (здесь и далее для простоты отключены параллельные планы):

с. 498

```
=> EXPLAIN (costs off) SELECT *
FROM tickets ORDER BY passenger_name ASC, book_ref DESC;
                        QUERY PLAN
-----------------------------------------------------------
 Incremental Sort
   Sort Key: passenger_name, book_ref DESC
   Presorted Key: passenger_name
   -> Index Scan using tickets_name_bref_idx on tickets
(4 rows)
```

На возможность использовать индекс для сортировки влияет и расположение неопределенных значений. По умолчанию для целей сортировки неопределенные значения считаются «больше» обычных значений, то есть располагаются с правого конца дерева при сортировке по возрастанию или с левого при сортировке по убыванию. Этот порядок можно менять с помощью предложений `NULLS LAST` и `NULLS FIRST`.

В следующем примере индекс не удовлетворяет предложению `ORDER BY`, и результат приходится сортировать:

```
=> EXPLAIN (costs off) SELECT *
FROM tickets ORDER BY passenger_name NULLS FIRST, book_ref DESC;
                       QUERY PLAN
-------------------------------------------------------
 Sort
   Sort Key: passenger_name NULLS FIRST, book_ref DESC
   -> Seq Scan on tickets
(3 rows)
```

Но если создать индекс, соответствующий желаемому порядку, он будет использоваться:

```
=> CREATE INDEX tickets_name_bref_idx2
ON tickets(passenger_name NULLS FIRST, book_ref DESC);
=> EXPLAIN (costs off) SELECT *
FROM tickets ORDER BY passenger_name NULLS FIRST, book_ref DESC;
                      QUERY PLAN
-------------------------------------------------
 Index Scan using tickets_name_bref_idx2 on tickets
(1 row)
```

25.5. Класс операторов

Семантика сравнения

Системе необходимо уметь не только хешировать значения, но и упорядочивать значения разных типов, в том числе пользовательских. Это требуется для сортировок и группировок, соединения слиянием и некоторых других операций. И так же, как для хеширования, соответствие между типом данных и операторами сравнения задается классом операторов[1].

Класс операторов позволяет не привязываться к именам (таким как >, <, =) и даже иметь несколько способов упорядочить значения одного и того же типа.

Вот какие операторы сравнения должны быть заданы в любом классе операторов метода `btree` (на примере семейства `bool_ops`):

```
\dAo  => SELECT amopopr::regoperator AS opfamily_operator,
        amopstrategy
      FROM pg_am am
        JOIN pg_opfamily opf ON opfmethod = am.oid
        JOIN pg_amop amop ON amopfamily = opf.oid
      WHERE amname = 'btree'
      AND opfname = 'bool_ops'
      ORDER BY amopstrategy;
```

[1] postgrespro.ru/docs/postgresql/18/btree#BTREE-BEHAVIOR.

```
  opfamily_operator  | amopstrategy
---------------------+--------------
 <(boolean,boolean)  |            1
 <=(boolean,boolean) |            2
 =(boolean,boolean)  |            3
 >=(boolean,boolean) |            4
 >(boolean,boolean)  |            5
(5 rows)
```

Здесь мы видим пять операторов сравнения. Каждый из них соответствует одной из *стратегий*[1], которая определяет его семантику:

1) меньше;
2) меньше либо равно;
3) равно;
4) больше либо равно;
5) больше.

Класс операторов B-дерева включает также несколько опорных функций[2]. Первая из них должна возвращать единицу, если ее первый аргумент больше второго, минус единицу — если меньше, и ноль — если аргументы равны.

Остальные опорные функции необязательны, но позволяют методу доступа работать более производительно.

Чтобы лучше разобраться в этом механизме, полезно создать новый тип данных и установить для него порядок сортировки, отличный от порядка по умолчанию. Документация приводит такой пример для комплексных чисел[3], но использует язык C. К счастью, класс операторов для B-дерева можно создать и с помощью интерпретируемых языков, чем я и воспользуюсь, чтобы сделать этот пример как можно более простым (хотя и заведомо неэффективным).

Создадим новый составной тип для единиц измерения информации:

```
=> CREATE TYPE capacity_units AS ENUM (
  'B', 'kB', 'MB', 'GB', 'TB', 'PB'
);
```

[1] postgrespro.ru/docs/postgresql/18/xindex#XINDEX-STRATEGIES.
[2] postgrespro.ru/docs/postgresql/18/btree-support-funcs.
[3] postgrespro.ru/docs/postgresql/18/xindex#XINDEX-EXAMPLE.

```
=> CREATE TYPE capacity AS (
  amount integer,
  unit capacity_units
);
```

Теперь создадим таблицу со столбцом нового типа и заполним ее случайными значениями:

```
=> CREATE TABLE test AS
  SELECT ( (random()*1023)::integer, u.unit )::capacity AS cap
  FROM generate_series(1,100),
    unnest(enum_range(NULL::capacity_units)) AS u(unit);
```

По умолчанию значения составных типов сортируются в лексикографическом порядке, который в данном случае не совпадает с естественным порядком:

```
=> SELECT * FROM test ORDER BY cap;
    cap
-----------
 (1,MB)
 (2,kB)
 (3,MB)
 (7,GB)
 ...
 (1018,TB)
 (1020,B)
 (1020,kB)
 (1021,PB)
(600 rows)
```

Создание класса операторов начнем с функции, которая пересчитывает объем в байты:

```
=> CREATE FUNCTION capacity_to_bytes(a capacity) RETURNS numeric
AS $$
SELECT a.amount::numeric *
  1024::numeric ^ ( array_position(enum_range(a.unit), a.unit) - 1 );
$$ LANGUAGE sql STRICT IMMUTABLE;
=> SELECT capacity_to_bytes('(1,kB)'::capacity);
 capacity_to_bytes
--------------------
 1024.0000000000000
(1 row)
```

Затем создадим опорную функцию для будущего класса операторов:

```
=> CREATE FUNCTION capacity_cmp(a capacity, b capacity)
RETURNS integer
AS $$
SELECT sign(bookings.capacity_to_bytes(a) - bookings.capacity_to_bytes(b));
$$ LANGUAGE sql STRICT IMMUTABLE;
```

Теперь с помощью опорной функции легко определить операторы сравнения. Я специально использую «странные» имена, чтобы показать, что они могут быть произвольными:

```
=> CREATE FUNCTION capacity_lt(a capacity, b capacity) RETURNS boolean
AS $$
BEGIN
  RETURN capacity_cmp(a,b) < 0;
END;
$$ LANGUAGE plpgsql IMMUTABLE STRICT;
=> CREATE OPERATOR #<# (
  LEFTARG = capacity,
  RIGHTARG = capacity,
  FUNCTION = capacity_lt
);
```

Остальные четыре оператора определяются по аналогии.

```
=> CREATE FUNCTION capacity_le(a capacity, b capacity) RETURNS boolean
AS $$
BEGIN
  RETURN capacity_cmp(a,b) <= 0;
END;
$$ LANGUAGE plpgsql IMMUTABLE STRICT;
=> CREATE OPERATOR #<=# (
  LEFTARG = capacity,
  RIGHTARG = capacity,
  FUNCTION = capacity_le
);

=> CREATE FUNCTION capacity_eq(a capacity, b capacity) RETURNS boolean
AS $$
BEGIN
  RETURN capacity_cmp(a,b) = 0;
END;
$$ LANGUAGE plpgsql IMMUTABLE STRICT;
```

```
=> CREATE OPERATOR #=# (
  LEFTARG = capacity,
  RIGHTARG = capacity,
  FUNCTION = capacity_eq,
  MERGES -- можно использовать в соединении слиянием
);

=> CREATE FUNCTION capacity_ge(a capacity, b capacity) RETURNS boolean
AS $$
BEGIN
  RETURN capacity_cmp(a,b) >= 0;
END;
$$ LANGUAGE plpgsql IMMUTABLE STRICT;
=> CREATE OPERATOR #>=# (
  LEFTARG = capacity,
  RIGHTARG = capacity,
  FUNCTION = capacity_ge
);

=> CREATE FUNCTION capacity_gt(a capacity, b capacity) RETURNS boolean
AS $$
BEGIN
  RETURN capacity_cmp(a,b) > 0;
END;
$$ LANGUAGE plpgsql IMMUTABLE STRICT;
=> CREATE OPERATOR #>#
  LEFTARG = capacity,
  RIGHTARG = capacity,
  FUNCTION = capacity_gt
);
```

На этом этапе мы уже можем правильно сравнивать объемы:

```
=> SELECT (1,'MB')::capacity #>#  (512, 'kB')::capacity;
 ?column?
----------
 t
(1 row)
```

А после создания класса операторов заработает как надо и сортировка:

```
=> CREATE OPERATOR CLASS capacity_ops
DEFAULT FOR TYPE capacity -- будет использоваться по умолчанию
USING btree AS
  OPERATOR 1 #<#,
  OPERATOR 2 #<=#,
```

```
  OPERATOR 3 #=#,
  OPERATOR 4 #>=#,
  OPERATOR 5 #>#,
  FUNCTION 1 capacity_cmp(capacity,capacity);
=> SELECT * FROM test ORDER BY cap;
    cap
-----------
 (9,B)
 (30,B)
 (42,B)
 (47,B)
 ...
 (999,PB)
 (1007,PB)
 (1015,PB)
 (1021,PB)
(600 rows)
```

Наш класс операторов используется по умолчанию при создании индексов, а созданный индекс возвращает значения в правильном порядке:

```
=> CREATE INDEX ON test(cap);
=> SELECT * FROM test WHERE cap #<# (100,'B')::capacity ORDER BY cap;
  cap
--------
 (9,B)
 (30,B)
 (42,B)
 (47,B)
 (66,B)
 (67,B)
 (69,B)
 (97,B)
 (99,B)
(9 rows)
=> EXPLAIN (costs off)
SELECT * FROM test WHERE cap #<# (100,'B')::capacity ORDER BY cap;
                QUERY PLAN
-------------------------------------------
 Index Only Scan using test_cap_idx on test
   Index Cond: (cap #<# '(100,B)'::capacity)
(2 rows)
```

Предложение MERGES, указанное при создании оператора равенства, разрешает использовать соединение слиянием для значений нашего типа. *с. 483*

25.6. Свойства

с. 404 Посмотрим интерфейсные свойства B-деревьев.

Свойства уровня метода доступа

```
=> SELECT a.amname, p.name, pg_indexam_has_property(a.oid, p.name)
FROM pg_am a, unnest(array[
  'can_order', 'can_unique', 'can_multi_col',
  'can_exclude', 'can_include'
]) p(name)
WHERE a.amname = 'btree';
 amname |     name      | pg_indexam_has_property
--------+---------------+-------------------------
 btree  | can_order     | t
 btree  | can_unique    | t
 btree  | can_multi_col | t
 btree  | can_exclude   | t
 btree  | can_include   | t
(5 rows)
```

B-дерево может упорядочивать данные и поддерживает уникальность. Это единственный метод доступа, который обеспечивает такие свойства.

Многие методы доступа позволяют создавать составные индексы, но в случае B-деревьев приходится тщательно следить за порядком столбцов в индексе, поскольку значения в нем упорядочены.

Поддержка ограничений исключения формально есть, но ограничивается оператором равенства, что соответствует требованию уникальности. Вместо такого ограничения гораздо лучше использовать настоящее полноценное ограничение уникальности.

B-дерево позволяет также добавлять в индекс дополнительные столбцы, не участвующие в поиске.

Индекс будет строиться в параллельном режиме, если таблица достаточно велика. Количество рабочих процессов определяется по той же формуле,
с. 384 что и для параллельного последовательного сканирования, но ограничено
2 сверху значением параметра *max_parallel_maintenance_workers*. На все рабо-
64MB чие процессы выделяется объем памяти размером *maintenance_work_mem*.

Рабочие процессы параллельно сканируют таблицу и сортируют полученные данные [1]; результат образует листовые страницы индекса, над которыми затем надстраиваются внутренние страницы вплоть до корня.

Свойства уровня индекса

```
=>  SELECT p.name, pg_index_has_property('flights_pkey', p.name)
FROM unnest(array[
  'clusterable', 'index_scan', 'bitmap_scan', 'backward_scan'
]) p(name);
     name      | pg_index_has_property
---------------+-----------------------
 clusterable   | t
 index_scan    | t
 bitmap_scan   | t
 backward_scan | t
(4 rows)
```

Индекс на основе B-дерева можно использовать для кластеризации.

B-деревья поддерживают оба способа получения значений: и индексное сканирование, и сканирование по битовой карте. Благодаря тому что листовые страницы связаны двунаправленным списком, индекс можно обходить не только в прямом, но и в обратном направлении, получая противоположный порядок сортировки:

```
=> EXPLAIN (costs off) SELECT *
FROM bookings ORDER BY book_ref DESC;
                      QUERY PLAN
------------------------------------------------------
 Index Scan Backward using bookings_pkey on bookings
(1 row)
```

Свойства уровня столбца

Свойство ORDERABLE говорит о том, что B-дерево хранит данные упорядоченно, и первые четыре свойства (ASC и DESC, NULLS FIRST и NULLS LAST)

[1] backend/access/nbtree/nbtsort.c, функция btbuild.

определяют, как именно они упорядочены для данного столбца. В следующем примере значения столбца отсортированы по возрастанию, а неопределенные значения находятся в конце.

```
=> SELECT p.name,
  pg_index_column_has_property('flights_pkey', 1, p.name)
FROM unnest(array[
  'asc', 'desc', 'nulls_first', 'nulls_last', 'orderable',
  'distance_orderable', 'returnable', 'search_array', 'search_nulls'
]) p(name);
        name        | pg_index_column_has_property
--------------------+------------------------------
 asc                | t
 desc               | f
 nulls_first        | f
 nulls_last         | t
 orderable          | t
 distance_orderable | f
 returnable         | t
 search_array       | t
 search_nulls       | t
(9 rows)
```

Свойство SEARCH NULLS показывает возможность поиска неопределенных значений.

Для B-деревьев не реализована поддержка операторов упорядочивания (DISTANCE ORDERABLE), хотя такие попытки предпринимались[1].

B-деревья поддерживают поиск нескольких значений из массива (свойство SEARCH ARRAY) и позволяют получать данные без обращения к таблице (RETURNABLE).

[1] commitfest.postgresql.org/27/1804.

26

Индекс GiST

26.1. Общий принцип

Метод доступа GiST (Generalized Search Tree)[1] — обобщение идеи сбалансированного дерева поиска для типов данных, допускающих взаимное расположение значений. B-деревья жестко привязаны к порядковым типам данных, значения которых можно сравнивать (но поддержка таких типов реализована максимально эффективно). А GiST позволяет задать произвольный принцип распределения данных по дереву в классе операторов. В GiST-индекс можно «уложить» R-дерево для пространственных данных или RD-дерево для множеств, а сигнатурное дерево можно применять для любых типов данных (например, для текстов или изображений).

Благодаря расширяемости в PostgreSQL можно с нуля создать новый метод доступа, реализовав интерфейс с механизмом индексирования. Но это требует не только продумывания логики индексации, но и организации страничной структуры, эффективной стратегии блокирования, поддержки журнала упреждающей записи. Все это подразумевает очень высокую квалификацию разработчика и большую трудоемкость. GiST упрощает задачу, беря на себя решение низкоуровневых проблем и реализуя общую идею поиска. Чтобы использовать метод GiST для нового типа данных, достаточно предоставить класс операторов, состоящий из десятка опорных функций. Такой класс содержит значительную часть логики индексирования, в отличие от совсем простого класса операторов B-дерева. В этом смысле можно говорить о том, что GiST является каркасом для построения новых методов доступа.

[1] postgrespro.ru/docs/postgresql/18/gist; backend/access/gist/README.

Каждая запись листового узла (листовая запись) содержит, если говорить в самом общем виде, некий *предикат* (логическое условие) и идентификатор табличной версии строки. Ключ индексирования обязан удовлетворять предикату; при этом не важно, входит сам ключ в эту запись или нет.

Каждая запись внутреннего узла (внутренняя запись) также содержит предикат и ссылку на дочерний узел, причем все индексированные данные дочернего поддерева должны удовлетворять этому предикату. Иными словами, предикат внутренней записи включает в себя предикаты всех дочерних записей. Это важное свойство, заменяющее индексу GiST простую упорядоченность B-дерева.

Поиск в дереве GiST использует *функцию согласованности* — одну из опорных функций, определяемых классом операторов.

Функция согласованности вызывается для индексной записи и определяет, «согласуется» ли предикат данной записи с условием поиска (вида «*индексированный-столбец оператор выражение*»). Для внутренней записи она показывает, надо ли спускаться в соответствующее поддерево, а для листовой записи — удовлетворяет ли условию ключ индексирования.

Поиск[1], как обычно в дереве, начинается с корневого узла. С помощью функции согласованности выясняется, в какие дочерние узлы имеет смысл заходить, а в какие — нет. Затем алгоритм повторяется для каждого из найденных дочерних узлов; в индексе GiST, в отличие от B-дерева, их может оказаться несколько. Записи, отобранные функцией согласованности в листовых узлах, возвращаются в качестве результатов.

Поиск всегда производится в глубину: алгоритм в первую очередь старается добраться до какого-нибудь листового узла. Это позволяет быстрее начать возвращать результаты, что может быть важно, если пользователя интересуют только несколько первых строк.

Для вставки нового значения в дерево GiST функция согласованности не годится, поскольку нужно выбрать один конкретный листовой узел[2]. Узел выбирается так, чтобы минимизировать стоимость вставки, определяемую *штрафной функцией* класса операторов.

[1] backend/access/gist/gistget.c, функция gistgettuple.
[2] backend/access/gist/gistutil.c, функция gistchoose.

Как и в случае B-дерева, в выбранном узле может не оказаться свободного места, что приводит к расщеплению[1]. Для этого требуются еще две функции. Одна решает, какие записи останутся в старом узле, а какие будут перенесены в новый; другая объединяет два предиката, чтобы обновить предикат родительского узла.

Добавление новых значений постепенно расширяет существующие предикаты, а сужение выполняется только при расщеплении страницы либо при полном перестроении индекса. Это может приводить к деградации производительности GiST-индекса при частых обновлениях.

Поскольку все эти рассуждения общего вида могут показаться непонятными, а значительная часть логики зависит от конкретного класса операторов, дальше я рассмотрю несколько конкретных примеров.

26.2. R-дерево для точек

Первый пример относится к индексации точек (или других геометрических объектов) на плоскости. Обычное B-дерево не подходит для такого типа данных, поскольку для точек не определены операторы сравнения. Конечно, такие операторы можно определить самостоятельно, но при работе с геометрическими данными нас интересует индексная поддержка совсем других операций, из которых я подробно рассмотрю две: поиск вхождения в заданную область и поиск ближайших соседей.

Идея R-дерева состоит в том, что плоскость разбивается на прямоугольники, которые в сумме покрывают все индексируемые точки. Индексная запись хранит ограничивающий прямоугольник, а предикат можно сформулировать так: *точка лежит внутри данного ограничивающего прямоугольника*.

Корень R-дерева содержит несколько самых крупных прямоугольников (возможно, даже пересекающихся). В дочерних узлах находятся меньшие по размеру прямоугольники, вложенные в родительские; в совокупности они охватывают все нижележащие точки.

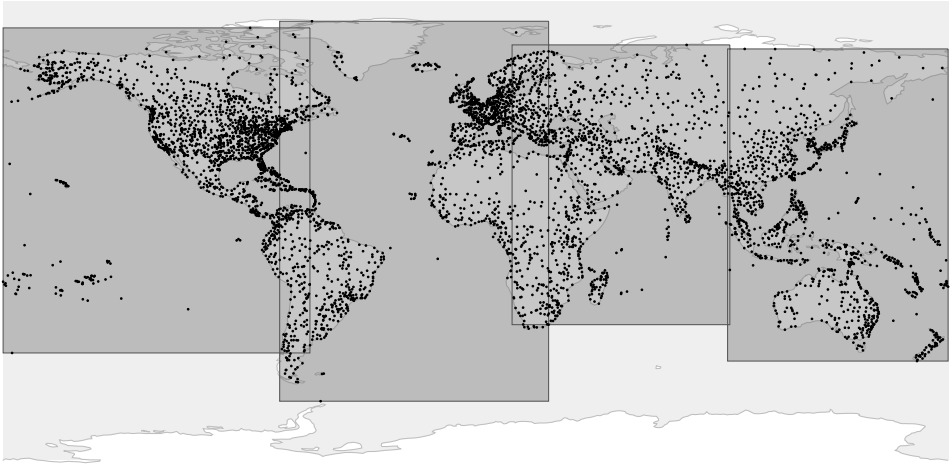
[1] backend/access/gist/gistsplit.c, функция gistSplitByKey.

Листовые узлы должны содержать сами индексируемые точки, однако GiST требует, чтобы тип данных совпадал во всех индексных записях; поэтому хранятся все те же прямоугольники, но «схлопнутые» в точки.

Представить эту структуру наглядно помогут рисунки трех уровней R-дерева, построенного по координатам аэропортов. Для этого примера я расширяю таблицу airports демобазы до пяти тысяч строк[1]. Для наглядности
90 индекс построен с низким значением параметра хранения *fillfactor*, чтобы дерево получилось глубоким; со значением по умолчанию хватает одного уровня.

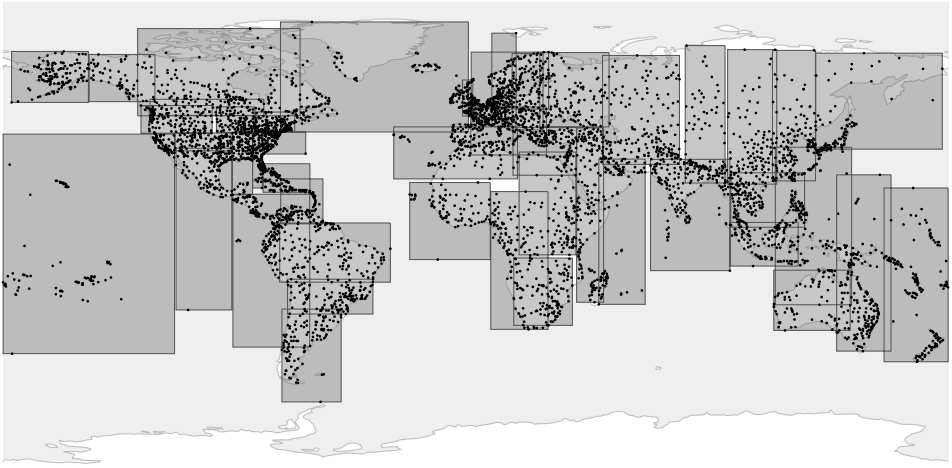
```
=> CREATE TABLE airports_big AS
  SELECT * FROM airports_data;
=> COPY airports_big FROM
  '/home/student/internals/airports/extra_airports.copy';
=> CREATE INDEX airports_gist_idx ON airports_big
  USING gist(coordinates) WITH (fillfactor=10);
```

На верхнем уровне все точки поделены между несколькими (отчасти пересекающимися) ограничивающими прямоугольниками:

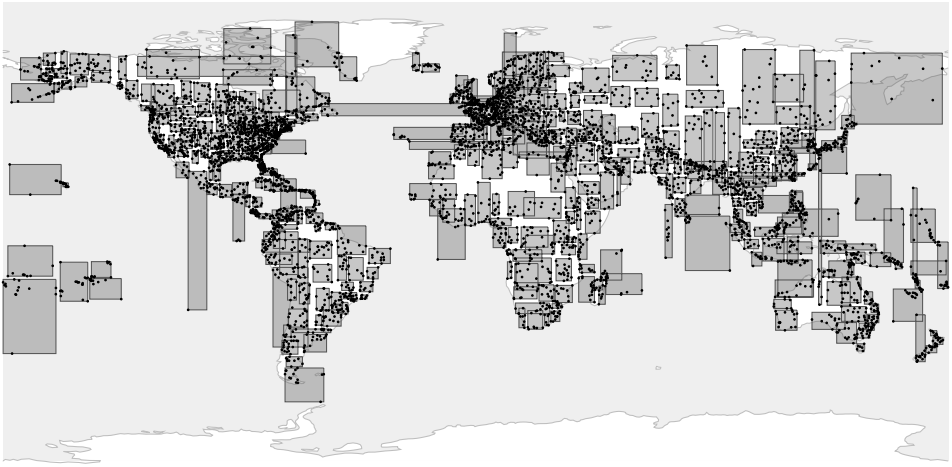


На следующем уровне большие прямоугольники распадаются на меньшие:

[1] Файл доступен по адресу edu.postgrespro.ru/internals-18/extra_airports.copy (использовались данные с сайта openflights.org).



На последнем, внутреннем уровне дерева каждый ограничивающий прямоугольник содержит столько точек, сколько помещается на одну страницу:



Рисунки показывают структуру индекса, созданного 13-й версией PostgreSQL. Версия 14 предварительно сортирует точки в z-порядке[1] — это дает значительный выигрыш в скорости построения индекса, но увеличивает перекрытия прямоугольников (и снижает эффективность поиска). В версии 15 алгоритм сортировки был существенно улучшен[2], но прежние рисунки остаются более наглядными.

[1] commitfest.postgresql.org/29/2276.
[2] commitfest.postgresql.org/37/3488.

В этом индексе используется класс операторов `point_ops`, единственный для точек.

Точно так же можно индексировать и прямоугольники, и любые другие геометрические фигуры, если вместо самой фигуры сохранять в индексе ограничивающий ее прямоугольник.

Страничная организация

v. 14 Содержимое страниц GiST-индекса можно изучать с помощью расширения `pageinspect`.

В отличие от B-дерева, индекс GiST не имеет метастраницы, а корень дерева всегда располагается в нулевой странице. Если корневая страница расщепляется, старый корень переносится в отдельную страницу, а новый занимает его место.

Вот содержимое корневой страницы:

```
=> SELECT ctid, left(keys,65) AS keys
FROM gist_page_items(
  get_raw_page('airports_gist_idx', 0), 'airports_gist_idx'
) \gx
-[ RECORD 1 ]----------------------------------------------------------
ctid | (216,65535)
keys | (coordinates)=("(27.588300704956055,42.55270004272461),(-179.8769
-[ RECORD 2 ]----------------------------------------------------------
ctid | (367,65535)
keys | (coordinates)=("(27.826700210571,82.51779937740001),(-176.6459960
-[ RECORD 3 ]----------------------------------------------------------
ctid | (217,65535)
keys | (coordinates)=("(50.84510040283203,70.871399),(13.922200202941895
-[ RECORD 4 ]----------------------------------------------------------
ctid | (526,65535)
keys | (coordinates)=("(179.951004028,73.51780700683594),(46.95610046386
```

Эти четыре строки соответствуют четырем прямоугольникам верхнего уровня, показанным на рисунке на предыдущем развороте.

Класс операторов

Список опорных функций, реализующих логику поиска и вставки в дерево[1], можно получить следующим запросом:

```
=> SELECT amprocnum, amproc::regproc                                    \dAp
FROM pg_am am
  JOIN pg_opclass opc ON opcmethod = am.oid
  JOIN pg_amproc amop ON amprocfamily = opcfamily
WHERE amname = 'gist'
AND opcname = 'point_ops'
ORDER BY amprocnum;
 amprocnum |         amproc
-----------+------------------------
         1 | gist_point_consistent
         2 | gist_box_union
         3 | gist_point_compress
         5 | gist_box_penalty
         6 | gist_box_picksplit
         7 | gist_box_same
         8 | gist_point_distance
         9 | gist_point_fetch
        11 | gist_point_sortsupport
(9 rows)
```

Основные обязательные функции я уже называл выше:

1) функция согласованности для обхода дерева при поиске;
2) функция объединения (обратите внимание на название: объединяются прямоугольники);
5) функция штрафа для спуска при вставке;
6) функция, распределяющая записи при расщеплении;
7) функция, проверяющая равенство двух ключей.

Необязательные функции позволяют управлять аспектами хранения данных и производительности.

Класс операторов `point_ops` поддерживает следующие операторы:

[1] postgrespro.ru/docs/postgresql/18/gist-extensibility.

```
\dAo   => SELECT amopopr::regoperator, amopstrategy AS st, oprcode::regproc,
         left(obj_description(opr.oid, 'pg_operator'), 19) description
       FROM pg_am am
         JOIN pg_opclass opc ON opcmethod = am.oid
         JOIN pg_amop amop ON amopfamily = opcfamily
         JOIN pg_operator opr ON opr.oid = amopopr
       WHERE amname = 'gist'
       AND opcname = 'point_ops'
       ORDER BY amopstrategy;
             amopopr      | st |       oprcode       |     description
       -------------------+----+---------------------+---------------------
        <<(point,point)   |  1 | point_left          | is left of
        >>(point,point)   |  5 | point_right         | is right of
        ~=(point,point)   |  6 | point_eq            | same as
        <<|(point,point)  | 10 | point_below         | is below
        |>>(point,point)  | 11 | point_above         | is above
        <->(point,point)  | 15 | point_distance      | distance between
        <@(point,box)     | 28 | on_pb               | point inside box
        <^(point,point)   | 29 | point_below         | deprecated, use <<|
        >^(point,point)   | 30 | point_above         | deprecated, use |>>
        <@(point,polygon) | 48 | pt_contained_poly   | is contained by
        <@(point,circle)  | 68 | pt_contained_circle | is contained by
       (11 rows)
```

По именам операторов бывает сложно догадаться об их значении, поэтому в запросе выведены названия реализующих их функций и описание. Все операторы так или иначе связаны со взаимным положением объектов («слева», «справа», «выше», «ниже», «содержит», «содержится в») и расстоянием между ними.

В отличие от B-деревьев, GiST располагает значительно бóльшим количеством стратегий. Часть номеров стратегий общая для нескольких типов индексов[1], а часть вычисляется по формулам (например, номера 28, 48 и 68 представляют, по сути, одну стратегию: «содержится в» для прямоугольников, полигонов и окружностей). Кроме того, поддерживаются некоторые устаревшие названия операторов (<<| и |>>).

Конкретный класс операторов может реализовывать не все из имеющихся стратегий. Например, класс для точек не реализует стратегию «содержит», но такая стратегия присутствует в классах фигур, имеющих площадь (`box_ops`, `poly_ops` и `circle_ops`).

[1] include/access/stratnum.h.

Поиск вхождения в область

Типичный запрос, ускоряемый индексом, получает все точки, входящие в заданную область.

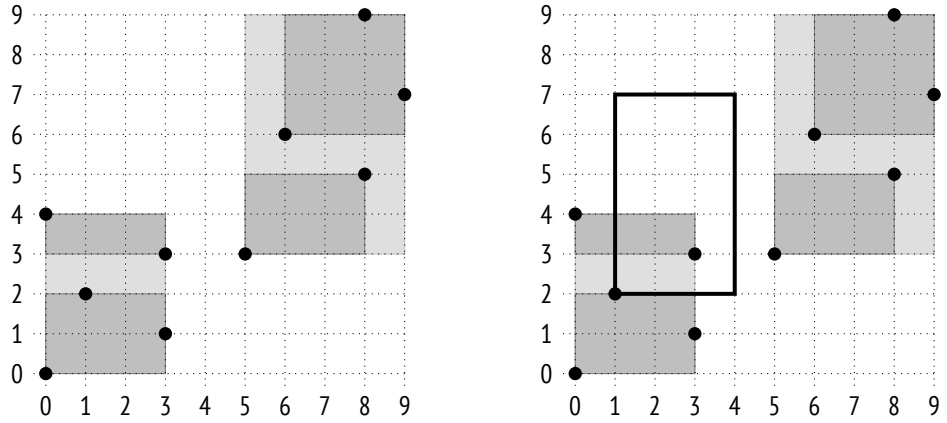
Например, найдем все аэропорты, находящиеся в пределах одного градуса от центра Москвы:

```
=> SELECT airport_code, airport_name->>'en'
FROM airports_big
WHERE coordinates <@ '<(37.622513,55.753220),1.0>'::circle;
 airport_code |              ?column?
--------------+-----------------------------------
 SVO          | Sheremetyevo International Airport
 VKO          | Vnukovo International Airport
 DME          | Domodedovo International Airport
 BKA          | Bykovo Airport
 ZIA          | Zhukovsky International Airport
 CKL          | Chkalovskiy Air Base
 OSF          | Ostafyevo International Airport
(7 rows)
=> EXPLAIN (costs off) SELECT airport_code
FROM airports_big
WHERE coordinates <@ '<(37.622513,55.753220),1.0>'::circle;
                            QUERY PLAN
------------------------------------------------------------------
 Bitmap Heap Scan on airports_big
   Recheck Cond: (coordinates <@ '<(37.622513,55.75322),1>'::circle)
   -> Bitmap Index Scan on airports_gist_idx
        Index Cond: (coordinates <@ '<(37.622513,55.75322),1>'::ci...
(4 rows)
```

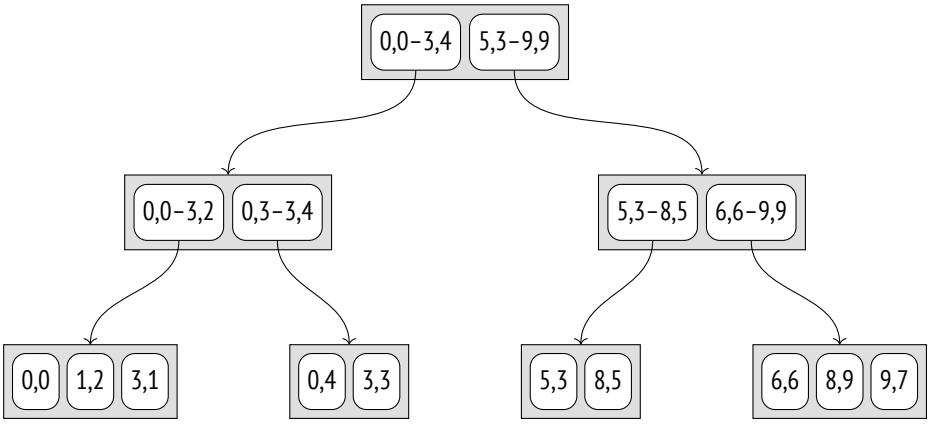
Оператор «содержится в» `<@` определяет, лежит ли точка внутри заданного прямоугольника. Функция согласованности для этого оператора[1] возвращает «да», если прямоугольник индексной записи имеет общие точки с заданным прямоугольником. Для листовых страниц, записи которых хранят «схлопнутые» в точки прямоугольники, это означает, что функция (как и сам оператор) определяет, содержится ли точка внутри заданного прямоугольника.

Рассмотрим подробнее работу оператора на совсем простом примере, показанном на рисунке:

[1] backend/access/gist/gistproc.c, функция gist_point_consistent.



Структура индекса при таком разбиении выглядит следующим образом:

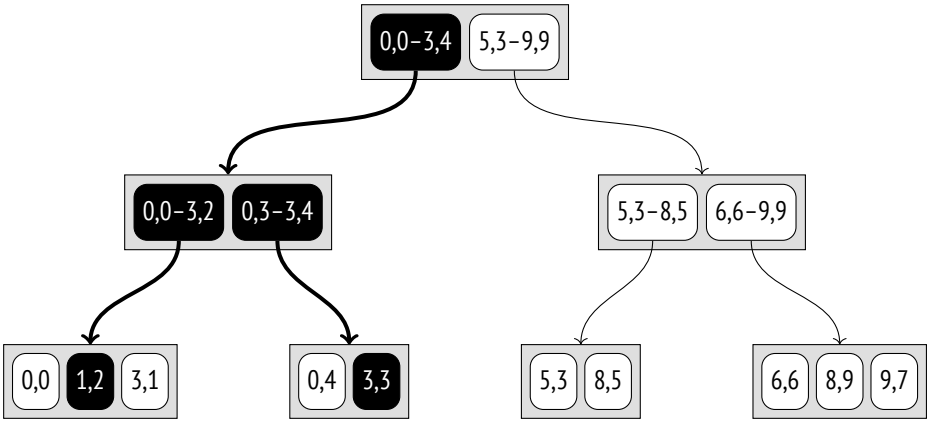


Давайте найдем внутренние точки прямоугольника (1,2)–(4,7), показанного справа на рисунке выше.

Поиск начинается с корневого узла. Заданный прямоугольник пересекается с (0,0)–(3,4), но не пересекается с (5,3)–(9,9). Поэтому во второе поддерево нам спускаться не нужно.

На следующем уровне заданный прямоугольник пересекается с (0,3)–(3,4) и касается (0,0)–(3,2), поэтому нам придется проверить оба поддерева.

Когда мы дойдем до листовых узлов, останется перебрать все содержащиеся там точки и вернуть те, для которых выполняется функция согласованности.



Поиск по B-дереву всегда однозначно выбирает дочерний узел; поиск по GiST же может потребовать обхода нескольких поддеревьев, особенно если их ограничивающие прямоугольники пересекаются.

Поиск ближайших соседей

Большинство операторов, поддерживаемых индексами (таких как = или <@ в предыдущем примере), можно назвать *поисковыми*, так как они задают условия поиска в запросе. Такие операторы являются предикатами, то есть возвращают логическое значение.

Есть и другой тип операторов — *упорядочивающие*, которые возвращают расстояние между аргументами. Такие операторы в предложении ORDER BY могут поддерживаться индексами, реализующими свойство DISTANCE ORDERABLE, что позволяет эффективно находить заданное количество ближайших соседей. Такой поиск известен как k-NN — k-nearest neighbor search. *с. 410*

Например, можно найти десять аэропортов, ближайших к Костроме:

```
=> SELECT airport_code, airport_name->>'en'
FROM airports_big
ORDER BY coordinates <-> '(40.926780,57.767943)'::point
LIMIT 10;
```

```
 airport_code |                    ?column?
--------------+-----------------------------------------------
 KMW          | Kostroma Sokerkino Airport
 IAR          | Tunoshna Airport
 IWA          | Ivanovo South Airport
 VGD          | Vologda Airport
 RYB          | Staroselye Airport
 GOJ          | Nizhny Novgorod Strigino International Airport
 CEE          | Cherepovets Airport
 CKL          | Chkalovskiy Air Base
 ZIA          | Zhukovsky International Airport
 BKA          | Bykovo Airport
(10 rows)
```

```
=> EXPLAIN (costs off) SELECT airport_code
FROM airports_big
ORDER BY coordinates <-> '(40.926780,57.767943)'::point
LIMIT 10;
```

```
                          QUERY PLAN
---------------------------------------------------------------
 Limit
   -> Index Scan using airports_gist_idx on airports_big
       Order By: (coordinates <-> '(40.92678,57.767943)'::point)
(3 rows)
```

За счет того, что индексное сканирование выдает результаты по одному и его можно остановить в любой момент, поиск нескольких значений получается очень быстрым.

> В отсутствие индексной поддержки построить эффективный поиск было бы непросто. Для этого пришлось бы искать точки, входящие в некоторую область, постепенно расширяя ее, пока не будет найдено заданное количество результатов. Это потребовало бы нескольких индексных сканирований, не говоря уже о проблеме выбора начального размера области и ее приращения.

В системном каталоге можно увидеть тип оператора (s — поисковый, o — упорядочивающий):

\dAo

```
=> SELECT amopopr::regoperator, amoppurpose, amopstrategy
FROM pg_am am
  JOIN pg_opclass opc ON opcmethod = am.oid
  JOIN pg_amop amop ON amopfamily = opcfamily
WHERE amname = 'gist'
AND opcname = 'point_ops'
ORDER BY amopstrategy;
```

```
      amopopr      | amoppurpose | amopstrategy
-------------------+-------------+--------------
 <<(point,point)   | s           |            1
 >>(point,point)   | s           |            5
 ~=(point,point)   | s           |            6
 <<|(point,point)  | s           |           10
 |>>(point,point)  | s           |           11
 <->(point,point)  | o           |           15
 <@(point,box)     | s           |           28
 <^(point,point)   | s           |           29
 >^(point,point)   | s           |           30
 <@(point,polygon) | s           |           48
 <@(point,circle)  | s           |           68
(11 rows)
```

Для поддержки поиска ближайших соседей класс операторов должен определить дополнительную опорную *функцию расстояния*. Функция расстояния вызывается для элемента индекса и вычисляет расстояние от значения в данном элементе до некоторого другого значения.

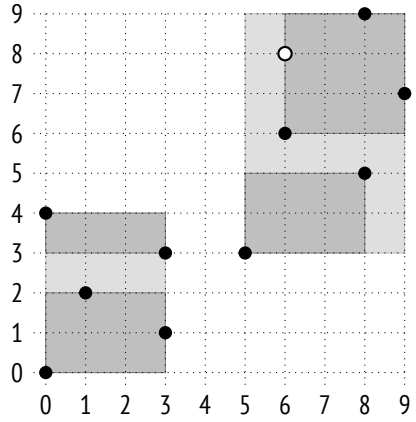
Для листового элемента, представляющего собой индексированное значение, функция должна вернуть расстояние от этого значения. В случае точек [1] это обычное евклидово расстояние, равное $\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$.

Для внутреннего элемента функция должна вернуть *минимальное* из расстояний от дочерних листовых элементов. Поскольку перебирать все дочерние записи весьма накладно, функция имеет право оптимистически приуменьшать расстояние (жертвуя при этом эффективностью), но не должна возвращать большее значение — это нарушит корректность поиска.

Поэтому для внутреннего элемента, представленного ограничивающим прямоугольником, расстояние до точки понимается в обычном математическом смысле, как минимальное расстояние между точкой и прямоугольником или ноль, если точка находится внутри [2]. Это значение легко вычислить, не обходя дочерние точки прямоугольника, и оно гарантированно не больше расстояния до любой из них.

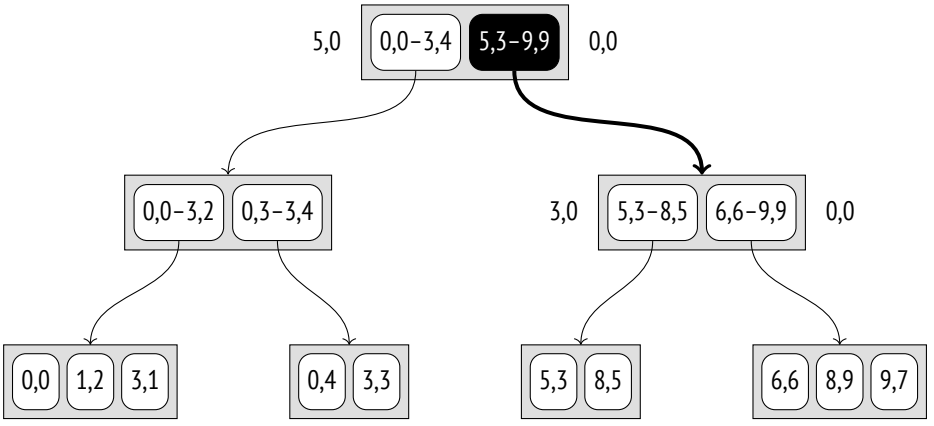
Рассмотрим на нашем простом примере алгоритм поиска трех ближайших соседей точки (6,8):

[1] backend/utils/adt/geo_ops.c, функция point_distance.
[2] backend/utils/adt/geo_ops.c, функция box_closest_point.

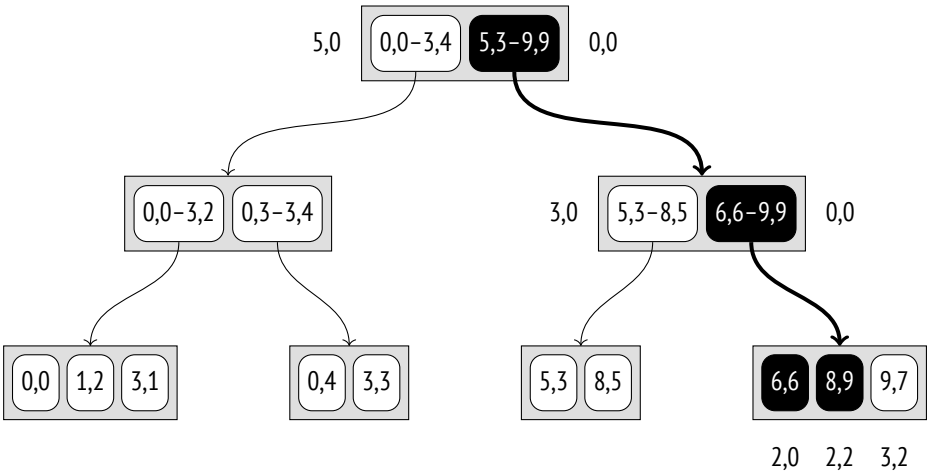


Поиск начинается с корневого узла, в котором имеется два ограничивающих прямоугольника. За расстояние от точки до (0,0)–(3,4) принимается расстояние до угла (3,4) прямоугольника, и оно составляет 5,0. Расстояние до (5,3)–(9,9) равно 0,0. (Я буду округлять значения до одного знака после запятой; в этом примере такой точности достаточно.)

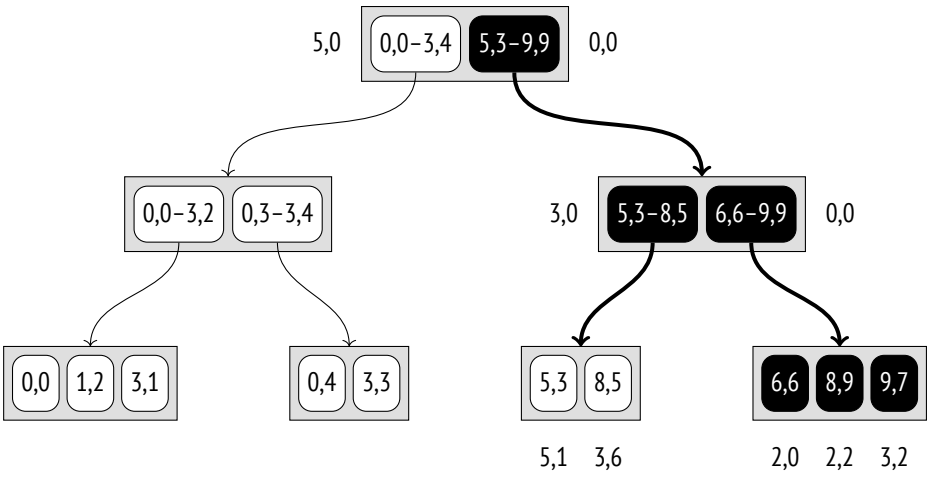
Обход дочерних узлов происходит в порядке увеличения расстояния. Таким образом, сначала мы спускаемся в правый дочерний узел, содержащий два прямоугольника: (5,3)–(8,5) и (6,6)–(9,9). Расстояние до первого составляет 3,0, а до второго — 0,0:



Снова спускаемся в правое поддерево и попадаем в листовой узел, содержащий три точки: (6,6) на расстоянии 2,0, (8,9) на расстоянии 2,2 и (9,7) на расстоянии 3,2. Таким образом, у нас есть первые две точки (6,6) и (8,9), но расстояние до третьей точки этого узла больше, чем расстояние до прямоугольника (5,3)–(8,5):



Поэтому теперь спускаемся в левый дочерний узел, содержащий две точки: (8,5) на расстоянии 3,6 и (5,3) на расстоянии 5,1. Оказывается, точка (9,7) в предыдущем дочернем узле ближе, чем любая из точек левого поддерева, и мы можем вернуть ее в качестве третьего результата:

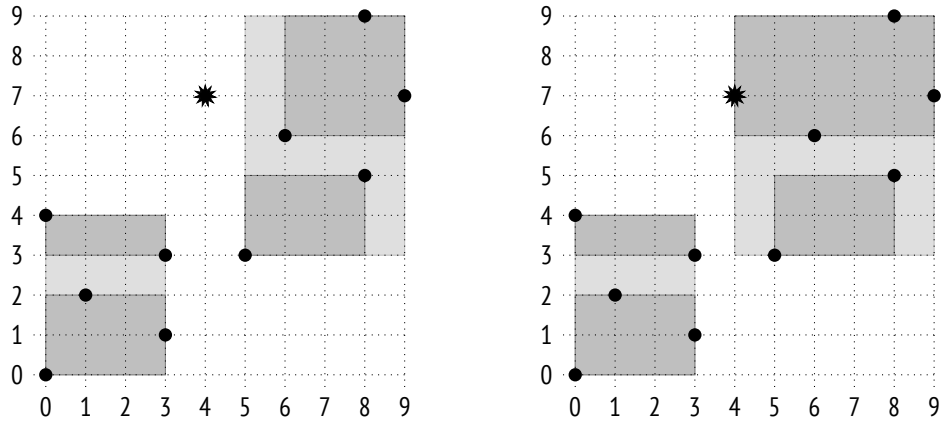


Этот пример поясняет требования к функции расстояния для внутренних записей. Уменьшенное расстояние (3,0 вместо реальных 3,6) до прямоугольника (5,3)–(8,5) ухудшило эффективность, поскольку пришлось напрасно просмотреть дополнительный узел, но не нарушило правильность работы алгоритма.

Вставка

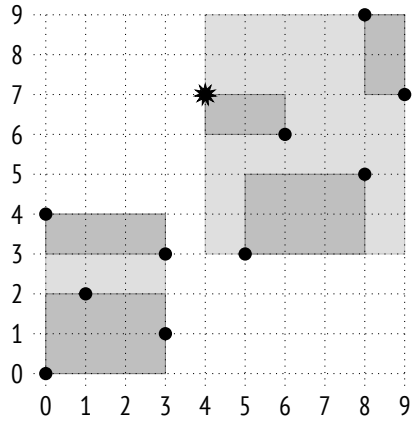
При вставке нового ключа в R-дерево узел для вставки выбирается на основании штрафной функции так, чтобы площадь охватывающего прямоугольника изменилась как можно меньше[1].

Например, точка (4,7) будет добавлена к прямоугольнику (5,3)–(9,9), поскольку его площадь при этом увеличится всего на 6 единиц, в то время как площадь прямоугольника (0,0)–(3,4) пришлось бы увеличить на 12. На следующем (листовом) уровне из этих же соображений точка будет добавлена к прямоугольнику (6,6)–(9,9).



Если предположить, что на странице помещается не более трех элементов, произойдет расщепление страницы на две, и элементы перераспределятся между ними:

[1] backend/access/gist/gistproc.c, функция gist_box_penalty.



В этом примере результат кажется очевидным, но в общем случае выбор распределения представляет весьма непростую задачу. Функция расщепления в первую очередь пытается минимизировать пересечение ограничивающих прямоугольников, отдавая предпочтение прямоугольникам с меньшей площадью и равномерному распределению точек между страницами[1].

Ограничение исключения

Индекс GiST может применяться для поддержки ограничений исключения (exclude).

Ограничение исключения гарантирует, что заданные поля любых двух строк таблицы не будут «соответствовать» друг другу в смысле некоторого *оператора*. Должны выполняться следующие условия:

- ограничение исключения должно поддерживаться индексным методом (свойство CAN EXCLUDE);
- оператор должен входить в класс операторов этого индексного метода;
- оператор должен быть коммутативным, то есть должно выполняться условие *a оператор b = b оператор a*.

[1] backend/access/gist/gistproc.c, функция gist_box_picksplit.

Методы доступа hash и btree, рассмотренные выше, предлагают единственный оператор, подходящий под условие, — равенство. С таким оператором
с. 521 ограничение исключения соответствует ограничению уникальности, что не особенно полезно.

Но метод gist имеет еще две подходящие стратегии:

- пересечение — оператор &&;
- примыкание — оператор -|- (определен для интервалов).

В качестве примера создадим ограничение, запрещающее располагать аэропорты слишком близко друг к другу. Это условие можно переформулировать так: окружности некоторого радиуса с центрами в координатах аэропортов не должны пересекаться:

```
=> ALTER TABLE airports_data ADD EXCLUDE
USING gist (circle(coordinates,0.2) WITH &&);
```

```
=> INSERT INTO airports_data(
  airport_code, airport_name, city, coordinates, timezone
) VALUES (
  'ZIA', '{}', '{"en": "Moscow"}', point(38.1517, 55.5533),
  'Europe/Moscow'
);
ERROR:  conflicting key value violates exclusion constraint
"airports_data_circle_excl"
DETAIL:  Key (circle(coordinates, 0.2::double
precision))=(<(38.1517,55.5533),0.2>) conflicts with existing key
(circle(coordinates, 0.2::double
precision))=(<(37.90629959106445,55.40879821777344),0.2>).
```

При создании ограничения исключения автоматически создается индекс для его проверки. В данном примере это GiST-индекс по выражению.

Усложним задачу. Пусть требуется разрешить близкое соседство аэропортов, но только в том случае, когда они относятся к одному городу. Решением будет создать новое ограничение целостности, которое можно сформулировать так: не допускаются пары строк, в которых *пересекаются* (&&) окружности с центрами в координатах аэропортов и при этом *не совпадают* (!=) названия городов.

Попытка создать такое ограничение приводит к ошибке, поскольку не существует класса операторов для текстового типа данных:

```
=> ALTER TABLE airports_data
DROP CONSTRAINT airports_data_circle_excl; -- удаляем старое
=> ALTER TABLE airports_data ADD EXCLUDE USING gist (
  circle(coordinates,0.2) WITH &&,
            (city->>'en') WITH !=
);
ERROR:  data type text has no default operator class for access method
"gist"
HINT:  You must specify an operator class for the index or define a
default operator class for the data type.
```

Но ведь GiST работает со стратегиями вроде «находится слева», «находится справа» и «совпадает», которые можно применить и к обычным порядковым типам данных, таким как числа или текстовые строки. Расширение `btree_gist` как раз и добавляет GiST-поддержку операций, характерных для B-деревьев:

```
=> CREATE EXTENSION btree_gist;
=> ALTER TABLE airports_data ADD EXCLUDE USING gist (
  circle(coordinates,0.2) WITH &&,
            (city->>'en') WITH !=
);
ALTER TABLE
```

Ограничение создано. Добавить аэропорт Жуковский в городе с тем же названием теперь не получится из-за близости московских аэропортов:

```
=> INSERT INTO airports_data(
  airport_code, airport_name, city, coordinates, timezone
) VALUES (
  'ZIA', '{}', '{"en": "Zhukovsky"}', point(38.1517, 55.5533),
  'Europe/Moscow'
);
ERROR:  conflicting key value violates exclusion constraint
"airports_data_circle_expr_excl"
DETAIL:  Key (circle(coordinates, 0.2::double precision), (city ->>
'en'::text))=(<(38.1517,55.5533),0.2>, Zhukovsky) conflicts with
existing key (circle(coordinates, 0.2::double precision), (city ->>
'en'::text))=(<(37.90629959106445,55.40879821777344),0.2>, Moscow).
```

Но это можно сделать, указав городом аэропорта Москву:

```
=> INSERT INTO airports_data(
  airport_code, airport_name, city, coordinates, timezone
) VALUES (
  'ZIA', '{}', '{"en": "Moscow"}', point(38.1517, 55.5533),
  'Europe/Moscow'
);
INSERT 0 1
```

Важно помнить, что хотя GiST и может работать с операциями «больше», «меньше» или «равно», B-дерево справляется с ними гораздо эффективнее, особенно при доступе к диапазону значений. Так что показанный в примере прием с расширением `btree_gist` стоит использовать, только если индекс GiST необходим по существу.

Свойства

Свойства уровня метода доступа

```
=> SELECT a.amname, p.name, pg_indexam_has_property(a.oid, p.name)
FROM pg_am a, unnest(array[
  'can_order', 'can_unique', 'can_multi_col',
  'can_exclude', 'can_include'
]) p(name)
WHERE a.amname = 'gist';
 amname |     name      | pg_indexam_has_property
--------+---------------+-------------------------
 gist   | can_order     | f
 gist   | can_unique    | f
 gist   | can_multi_col | t
 gist   | can_exclude   | t
 gist   | can_include   | t
(5 rows)
```

Поддержка сортировки значений и уникальности отсутствует.

v. 12 GiST-индекс можно создавать с дополнительными include-столбцами.

Индекс, как мы видели, можно строить по нескольким столбцам и использовать в ограничениях исключения.

Свойства уровня индекса

```
=>  SELECT p.name, pg_index_has_property('airports_gist_idx', p.name)
FROM unnest(array[
  'clusterable', 'index_scan', 'bitmap_scan', 'backward_scan'
]) p(name);
     name      | pg_index_has_property
---------------+-----------------------
 clusterable   | t
 index_scan    | t
 bitmap_scan   | t
 backward_scan | f
(4 rows)
```

GiST-индекс можно использовать для кластеризации таблицы.

Поддерживаются оба способа получения данных: и строка за строкой, и по битовой карте. Но обратный порядок сканирования для GiST не определен.

Свойства уровня столбца

Основная часть свойств столбцов определена на уровне метода доступа и не меняется:

```
=> SELECT p.name,
  pg_index_column_has_property('airports_gist_idx', 1, p.name)
FROM unnest(array[
  'orderable', 'search_array', 'search_nulls'
]) p(name);
     name     | pg_index_column_has_property
--------------+------------------------------
 orderable    | f
 search_array | f
 search_nulls | t
(3 rows)
```

Все свойства, связанные с сортировкой, отключены.

Неопределенные значения допускаются GiST-индексом, но обрабатываются не слишком эффективно. Считается, что неопределенное значение не расширяет ограничивающий прямоугольник; поэтому при вставке такие значения попадают в разные поддеревья индекса случайным образом, и их приходится искать во всем дереве.

Однако пара свойств уровня столбца может меняться в зависимости от класса операторов.

```
=> SELECT p.name,
  pg_index_column_has_property('airports_gist_idx', 1, p.name)
FROM unnest(array[
   'returnable', 'distance_orderable'
]) p(name);
        name        | pg_index_column_has_property
--------------------+------------------------------
 returnable         | t
 distance_orderable | t
(2 rows)
```

Индекс может использоваться в сканировании только индекса, поскольку ключи индексирования полностью хранятся в листовых узлах.

В данном случае, как мы видели, класс операторов содержит оператор расстояния для поиска ближайших соседей. Расстояние до неопределенного значения считается неопределенным; такие значения выдаются последними (аналогично указанию NULLS LAST в B-деревьях).

Но для диапазонных типов (которые представляют отрезки, то есть те же геометрические фигуры, только на прямой, а не на плоскости) оператор расстояния не реализован, поэтому для индекса по такому типу данных свойство будет отличаться:

```
=> CREATE TABLE reservations(during tsrange);

=> CREATE INDEX ON reservations USING gist(during);

=> SELECT p.name,
  pg_index_column_has_property('reservations_during_idx', 1, p.name)
FROM unnest(array[
   'returnable', 'distance_orderable'
]) p(name);
        name        | pg_index_column_has_property
--------------------+------------------------------
 returnable         | t
 distance_orderable | f
(2 rows)
```

26.3. RD-дерево для полнотекстового поиска

Про полнотекстовый поиск

Задача полнотекстового поиска[1] — выбрать из набора *документов* те, которые соответствуют *поисковому запросу*.

Для целей поиска документ приводится к специальному типу `tsvector`, который содержит *лексемы* и их позиции в документе. Лексемы — это слова, преобразованные к виду, удобному для поиска. Например, по умолчанию слова приводятся к нижнему регистру и лишаются изменяемых окончаний:

```
=> SET default_text_search_config = russian;
=> SELECT to_tsvector(
  'И встал Айболит, побежал Айболит. ' ||
  'По полям, по лесам, по лугам он бежит.'
);
                            to_tsvector
-------------------------------------------------------------------
 'айбол':3,5 'беж':13 'встал':2 'лес':9 'луг':11 'побежа':4 'пол':7
(1 row)
```

Так называемые стоп-слова («и», «по») удаляются, так как они предположительно встречаются слишком часто, чтобы поиск по ним был осмысленным. Разумеется, все эти преобразования настраиваются.

Поисковый запрос представляется другим типом — `tsquery`. Запрос состоит из одной или нескольких лексем, соединенных логическими связками: «и» `&`, «или» `|`, «не» `!`. Также можно использовать скобки для уточнения приоритета операций.

```
=> SELECT to_tsquery('Айболит & (побежал | пошел)');
            to_tsquery
----------------------------------
 'айбол' & ( 'побежа' | 'пошел' )
(1 row)
```

Для полнотекстового поиска используется единственный *оператор соответствия* `@@`:

[1] postgrespro.ru/docs/postgresql/18/textsearch.

```
=> SELECT amopopr::regoperator, oprcode::regproc, amopstrategy
FROM pg_am am
  JOIN pg_opclass opc ON opcmethod = am.oid
  JOIN pg_amop amop ON amopfamily = opcfamily
  JOIN pg_operator opr ON opr.oid = amopopr
WHERE amname = 'gist'
AND opcname = 'tsvector_ops'
ORDER BY amopstrategy;
       amopopr        |   oprcode   | amopstrategy
----------------------+-------------+--------------
 @@(tsvector,tsquery) | ts_match_vq |            1
(1 row)
```

Этот оператор определяет, соответствует ли документ поисковому запросу. Вот простой пример:

```
=> SELECT to_tsvector('И встал Айболит, побежал Айболит.') @@
  to_tsquery('Айболит & побежать');
 ?column?
----------
 t
(1 row)
```

с. 613 Это ни в коей мере не исчерпывающее описание полнотекстового поиска, но этих сведений будет достаточно для понимания индексирования.

Индексация tsvector

с. 613 Чтобы полнотекстовый поиск работал быстро, его нужно поддержать индексом[1]. Поскольку индексируются не сами документы, а значения типа `tsvector`, есть два варианта: построить индекс по выражению, с приведением типа, или создать отдельный столбец типа `tsvector` и индексировать его. Плюс первого варианта в том, что он не расходует место для хранения значений `tsvector`, которые сами по себе никому не нужны. Однако он проигрывает варианту со столбцом по скорости, когда механизм индексирования перепроверяет по таблице идентификаторы версий, возвращенные индексным методом. Для этого приходится повторно вычислять значение

[1] postgrespro.ru/docs/postgresql/18/textsearch-indexes.

`tsvector` для каждой проверяемой строки, а индекс GiST, как мы увидим, перепроверяет все строки.

Построим небольшой пример. Создадим таблицу с двумя столбцами: в первом будет храниться документ, во втором — значение `tsvector`. Второй столбец можно обновлять триггером[1], но удобнее просто объявить его как генерируемый[2]: v. 12

```
=> CREATE TABLE ts(
  doc text,
  doc_tsv tsvector GENERATED ALWAYS AS (
    to_tsvector('pg_catalog.russian', doc)
  ) STORED
);
=> CREATE INDEX ts_gist_idx ON ts USING gist(doc_tsv);
```

> В примерах выше я приводил функцию `to_tsvector` с одним параметром, предварительно устанавливая *конфигурацию* полнотекстового поиска в параметре *default_text_search_config*. Такой вариант функции имеет категорию изменчивости `STABLE`, поскольку неявно зависит от значения параметра. Здесь же используется вариант функции с явным указанием конфигурации, который имеет категорию `IMMUTABLE`, подходящую для использования в генерирующем выражении. english

Вставим в таблицу несколько строк:

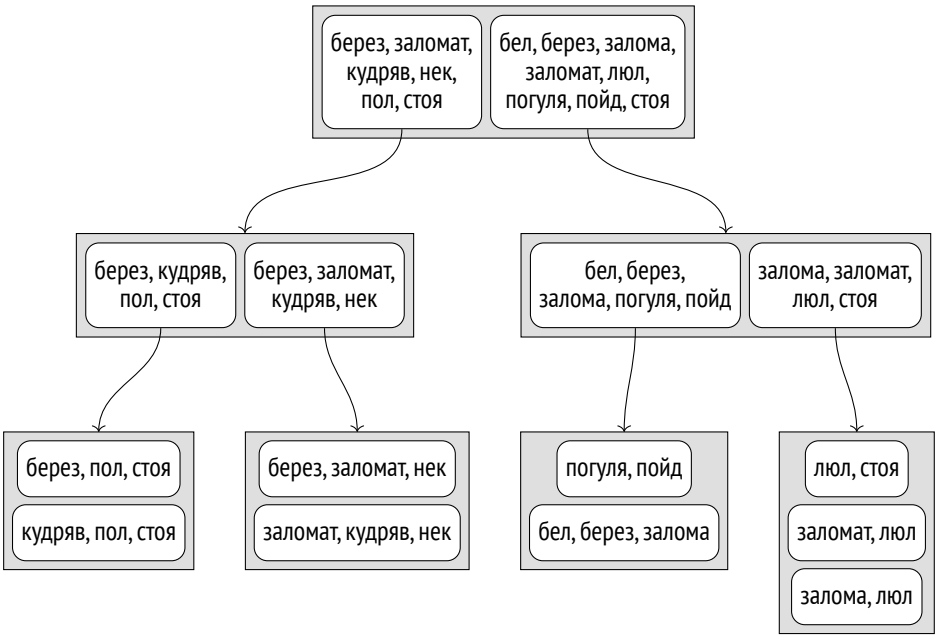
```
=> INSERT INTO ts(doc) VALUES
  ('Во поле береза стояла'),  ('Во поле кудрявая стояла'),
  ('Люли, люли, стояла'),     ('Люли, люли, стояла'),
  ('Некому березу заломати'), ('Некому кудряву заломати'),
  ('Люли, люли, заломати'),   ('Люли, люли, заломати'),
  ('Я пойду погуляю'),        ('Белую березу заломаю'),
  ('Люли, люли, заломаю'),    ('Люли, люли, заломаю')
RETURNING doc_tsv;
            doc_tsv
-------------------------------
 'берез':3 'пол':2 'стоя':4
 'кудряв':3 'пол':2 'стоя':4
 'люл':1,2 'стоя':3
 'люл':1,2 'стоя':3
 'берез':2 'заломат':3 'нек':1
 'заломат':3 'кудряв':2 'нек':1
```

[1] postgrespro.ru/docs/postgresql/18/textsearch-features#TEXTSEARCH-UPDATE-TRIGGERS.
[2] postgrespro.ru/docs/postgresql/18/ddl-generated-columns.

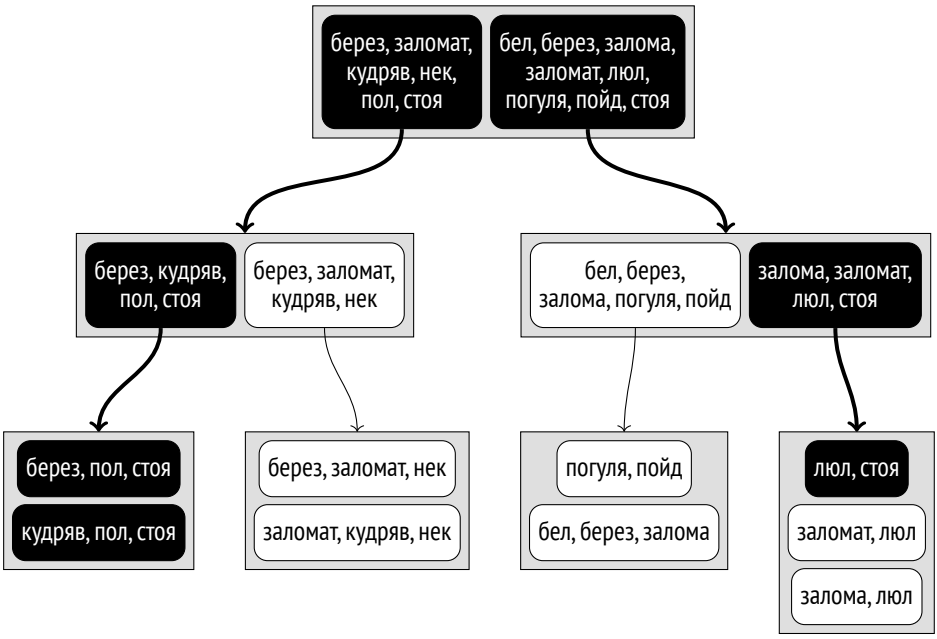
```
 'заломат':3 'люл':1,2
 'заломат':3 'люл':1,2
 'погуля':3 'пойд':2
 'бел':1 'берез':2 'залома':3
 'залома':3 'люл':1,2
 'залома':3 'люл':1,2
(12 rows)
INSERT 0 12
```

R-дерево как таковое не годится для индексации документов, поскольку к ним неприменимо понятие ограничивающего прямоугольника. Используется модификация этого подхода — RD-дерево (Russian Doll, матрешка). Вместо ограничивающего прямоугольника это дерево использует *ограничивающее множество*, то есть множество, содержащее все элементы дочерних множеств. Для полнотекстового поиска множество состоит из лексем документа, но в общем случае множество может быть произвольным.

Есть несколько способов представления множеств в индексных записях. Самый простой — непосредственно перечислить все элементы множества. Вот как это могло бы выглядеть:



Тогда, например, для поиска документов, удовлетворяющих условию `doc_tsv @@ to_tsquery('стояла')`, нужно спускаться в узлы, для которых известно, что дочерние записи содержат лексему «стоя»:



Проблемы такого представления очевидны. Количество лексем в документе может быть огромным, а место в странице ограничено. Даже если в каждом документе не так много уникальных лексем, объединение множеств на верхних уровнях дерева все равно может получиться очень большим.

Для полнотекстового поиска применяется другое, более компактное решение — *сигнатурное дерево*. Идея его хорошо знакома всем, кто имел дело с фильтром Блума. *с. 665*

Каждую лексему можно представить своей *сигнатурой*: битовой строкой определенной длины, в которой все биты равны нулю (сброшены), кроме одного, который равен единице (установлен). Номер установленного бита определяется значением хеш-функции от лексемы.

Сигнатурой документа называется побитовое «или» сигнатур всех лексем документа.

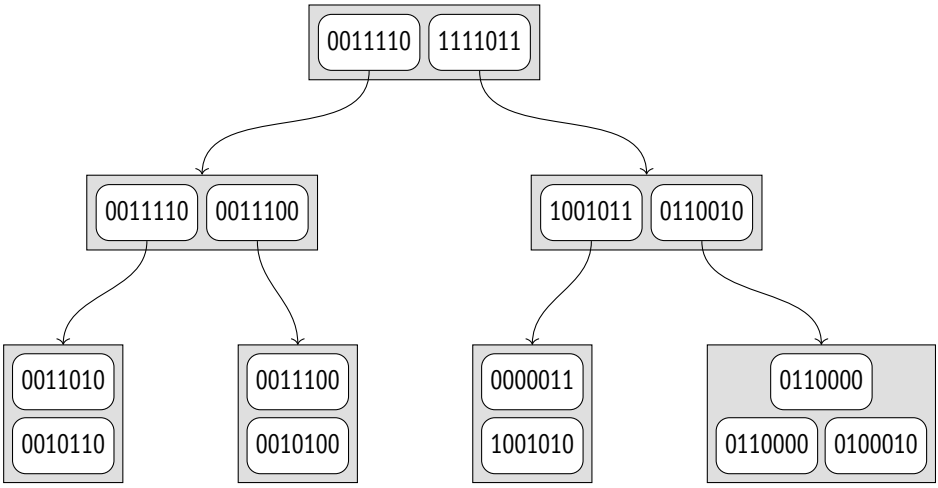
Допустим, мы присвоили нашим лексемам такие сигнатуры:

бел	`1000000`
берез	`0001000`
залома	`0000010`
заломат	`0010000`
кудряв	`0000100`
люл	`0100000`
нек	`0000100`
погуля	`0000001`
пойд	`0000010`
пол	`0000010`
стоя	`0010000`

Тогда сигнатуры документов будут следующими:

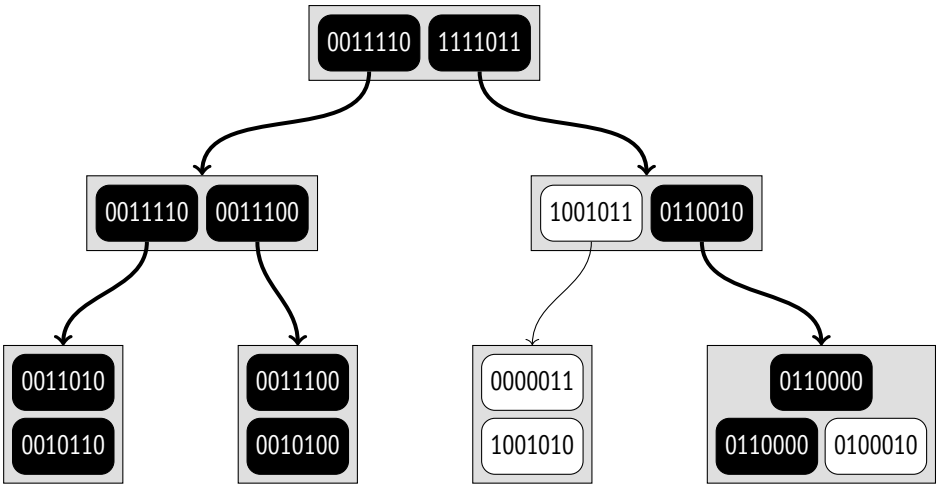
Во поле береза стояла	`0011010`
Во поле кудрявая стояла	`0010110`
Люли, люли, стояла	`0110000`
Некому березу заломати	`0011100`
Некому кудряву заломати	`0010100`
Люли, люли, заломати	`0110000`
Я пойду погуляю	`0000011`
Белую березу заломаю	`1001010`
Люли, люли, заломаю	`0100010`

Дерево индекса в этом случае можно представить так:



Преимущества такого подхода налицо: индексные записи имеют одинаковый и небольшой размер, индекс получается компактным. Но есть и существенные недостатки. Во-первых, сканирование только индекса оказывается невозможным, поскольку индекс больше не хранит значения ключей индексирования и каждый возвращаемый идентификатор должен перепроверяться по таблице. Во-вторых, теряется точность: индекс может возвращать много ложных результатов, которые затем отсеиваются при перепроверке.

Рассмотрим все то же условие `doc_tsv @@ to_tsquery('стояла')`. Сигнатура поискового запроса вычисляется так же, как и для документа, и в нашем случае равна `0010000`. Функция согласованности[1] должна определить все дочерние узлы, сигнатура которых содержит установленные биты сигнатуры запроса:



По сравнению с предыдущим примером здесь перебирается больше узлов из-за ложноположительных срабатываний. Причина в том, что при большом количестве лексем некоторые из них неизбежно будут получать совпадающие сигнатуры из-за их ограниченной разрядности. В этом примере такими лексемами являются «стоя» и «заломат». Это приводит к тому, что одна и та же сигнатура соответствует разным документам; в данном случае «люли, люли, стояла» и «люли, люли, заломати» имеют сигнатуру `0110000`.

[1] backend/utils/adt/tsgistidx.c, функция gtsvector_consistent.

Ложноположительные срабатывания снижают эффективность индекса, но не ставят под угрозу корректность, поскольку ложноотрицательных срабатываний в такой схеме быть не может и нужное значение гарантированно не будет пропущено.

Конечно, в реальности размер сигнатуры выбирается больше. По умолчанию используется 124 байта (992 бита), так что вероятность коллизий существенно ниже, чем в примере. При необходимости можно установить другой размер сигнатуры вплоть до примерно 2000 байт, используя *параметр класса операторов*:

v. 13

```
CREATE INDEX ... USING gist(столбец tsvector_ops(siglen = размер));
```

Кроме того, если значения достаточно малы[1] (чуть меньше $\frac{1}{16}$ страницы, что составляет около 500 байт для стандартной страницы), класс операторов `tsvector_ops` хранит в листовых страницах индекса не сигнатуры, а сами значения `tsvector`.

Чтобы посмотреть, как индексирование работает на реальных данных, возьмем архив рассылки pgsql-hackers[2]. В нем содержится 356 125 писем с датой отправления, темой, именем автора и текстом.

Добавим столбец типа `tsvector` и построим индекс. Здесь я объединяю в один вектор три значения (тему, автора и текст письма), чтобы показать, что документ может формироваться динамически и не обязан храниться в каком-то одном столбце.

```
=> ALTER TABLE mail_messages ADD COLUMN tsv tsvector
GENERATED ALWAYS AS ( to_tsvector(
  'pg_catalog.english', subject||' '||author||' '||body_plain
) ) STORED;
NOTICE:  word is too long to be indexed
DETAIL:  Words longer than 2047 characters are ignored.
 ...
NOTICE:  word is too long to be indexed
DETAIL:  Words longer than 2047 characters are ignored.
ALTER TABLE
```

[1] backend/utils/adt/tsgistidx.c, функция gtsvector_compress.

[2] edu.postgrespro.ru/mail_messages.sql.gz.

```
=> CREATE INDEX mail_gist_idx ON mail_messages USING gist(tsv);
=> SELECT pg_size_pretty(pg_relation_size('mail_gist_idx'));
 pg_size_pretty
----------------
 127 MB
(1 row)
```

При заполнении столбца некоторое количество слов было отброшено из-за слишком большого размера, но в итоге индекс построен и готов поддерживать поисковые запросы:

```
=> EXPLAIN (analyze, costs off, timing off, summary off, buffers off)
SELECT *
FROM mail_messages
WHERE tsv @@ to_tsquery('magic & value');
                          QUERY PLAN
---------------------------------------------------------------
 Index Scan using mail_gist_idx on mail_messages
   (actual rows=898.00 loops=1)
   Index Cond: (tsv @@ to_tsquery('magic & value'::text))
   Rows Removed by Index Recheck: 7852
   Index Searches: 1
(5 rows)
```

Вместе с 898 строками, подходящими под условие, метод доступа вернул еще 7852 строки, которые были отсеяны перепроверкой по таблице. Увеличив размер сигнатуры, можно повысить точность (и, следовательно, эффективность), жертвуя компактностью индекса:

```
=> DROP INDEX mail_messages_tsv_idx;
=> CREATE INDEX ON mail_messages
USING gist(tsv tsvector_ops(siglen=248));
=> SELECT pg_size_pretty(pg_relation_size('mail_messages_tsv_idx'));
 pg_size_pretty
----------------
 139 MB
(1 row)
=> EXPLAIN (analyze, costs off, timing off, summary off, buffers off)
SELECT *
FROM mail_messages
WHERE tsv @@ to_tsquery('magic & value');
```

```
                          QUERY PLAN
-------------------------------------------------------------
 Index Scan using mail_messages_tsv_idx on mail_messages
   (actual rows=898.00 loops=1)
   Index Cond: (tsv @@ to_tsquery('magic & value'::text))
   Rows Removed by Index Recheck: 2057
   Index Searches: 1
(5 rows)
```

Свойства

с. 574 Свойства метода доступа я уже показывал, и по большей части они совпадают для всех классов операторов. Внимания заслуживают два свойства уровня столбца:

```
=> SELECT p.name,
  pg_index_column_has_property('mail_messages_tsv_idx', 1, p.name)
FROM unnest(array[
   'returnable', 'distance_orderable'
]) p(name);
        name        | pg_index_column_has_property
--------------------+------------------------------
 returnable         | f
 distance_orderable | f
(2 rows)
```

Индекс потерял возможность участвовать в сканировании только индекса, поскольку по сигнатуре невозможно восстановить исходное значение. В данном случае в этом нет ничего страшного: значение `tsvector` используется только для поиска, а интересует нас сам документ.

Упорядочивающий оператор для класса `tsvector_ops` тоже не определен.

26.4. Другие типы данных

Я рассмотрел только два наиболее показательных примера. Они демонстрируют, что метод GiST, имея в основе сбалансированное дерево, может поразному применяться для разных типов данных за счет разной реализации

опорных функций класса операторов. Говоря о GiST-индексе, всегда необходимо уточнять класс операторов, поскольку он существенно влияет на свойства и характеристики индекса.

Вот еще несколько типов данных, которые в настоящее время поддерживает метод доступа GiST.

Геометрические типы. Кроме точек, GiST-индекс может использоваться и для других геометрических типов: прямоугольников, окружностей, полигонов. Любой из этих типов представляется для целей индексирования своим ограничивающим прямоугольником.

Расширение `cube` добавляет одноименный тип данных для представления многомерных кубов, которые индексируются с помощью R-дерева. Вместо ограничивающих прямоугольников в индексе используются ограничивающие параллелепипеды соответствующей размерности.

Диапазонные типы. К стандартным типам PostgreSQL относятся числовые и временны́е диапазоны, такие как `int4range` или `tstzrange`[1]. Можно определять и собственные диапазонные типы командой `CREATE TYPE AS RANGE`.

Любые диапазонные типы, как стандартные, так и пользовательские, поддерживаются GiST-индексом с помощью одного и того же класса операторов `range_ops`[2]. Для индексирования применяется одномерное R-дерево: в данном случае ограничивающие прямоугольники превращаются в ограничивающие отрезки.

Мультидиапазонные типы также поддерживаются; для них использ- v. 14
уется класс операторов `multirange_ops`. Ограничивающий диапазон в этом случае охватывает все диапазоны, входящие в мультидиапазонное значение.

Расширение `seg` предоставляет одноименный тип данных для интервалов с границами, заданными с определенной точностью. Такой тип не относится к диапазонным, но по сути является им и индексируется точно так же.

[1] postgrespro.ru/docs/postgresql/18/rangetypes.
[2] backend/utils/adt/rangetypes_gist.c.

Порядковые типы. Напомню еще раз про расширение `btree_gist`, предоставляющее классы операторов метода GiST для большого количества порядковых типов данных, которые обычно индексируются с помощью B-дерева. Такие классы операторов можно использовать для создания составного индекса, когда тип одного из столбцов не поддерживается B-деревом.

Сетевые адреса. Тип данных `inet` имеет встроенную GiST-поддержку, реализованную классом операторов `inet_ops`[1].

Целочисленные массивы. Расширение `intarray` делает целочисленные массивы более функциональными и добавляет для них GiST-поддержку. Имеются два класса операторов. Для небольших массивов можно использовать `gist__int_ops`, реализующий RD-дерево с полным представлением ключей в индексных записях. Для больших массивов подойдет более компактное, но менее точное сигнатурное RD-дерево, реализуемое `gist__bigint_ops`.

> «Лишнее» подчеркивание в именах классов операторов появляется, потому что с подчеркивания начинаются названия массивов базовых типов. Например, для целочисленного массива наряду с более привычным обозначением `int4[]` можно использовать тип `_int4`. Правда, типов `_int` и `_bigint` не существует.

Дерево меток. Расширение `ltree` добавляет одноименный тип данных для древовидных структур с метками. GiST-поддержка реализована с помощью сигнатурных RD-деревьев и представлена классами операторов `gist_ltree_ops` для значений типа `ltree` и `gist__ltree_ops` для массивов значений такого типа.

Хранилище «ключ–значение». Расширение `hstore` предоставляет тип данных `hstore` для хранения пар «ключ–значение». Класс операторов `gist_hstore_ops` реализует индексную поддержку на основе сигнатурного RD-дерева.

с. 630 **Триграммы.** Расширение `pg_trgm` добавляет класс `gist_trgm_ops`, реализующий индексную поддержку сравнения схожести текстовых строк и поиска по шаблону.

[1] backend/utils/adt/network_gist.c.

27

Индекс SP-GiST

27.1. Общий принцип

Буквы «SP» в названии SP-GiST расшифровываются как Space Partitioning, то есть разбиение пространства. Под пространством здесь подразумевается произвольная область значений, в которой производится поиск, а не обязательно пространство в обыденном понимании (например, двумерная плоскость). Ну а «GiST» намекает на определенную схожесть с этим методом: и GiST, и SP-GiST являются обобщенными деревьями поиска и служат каркасами для индексации различных типов данных.

Идея метода SP-GiST[1] состоит в разбиении пространства поиска на непересекающиеся области, каждая из которых, в свою очередь, также может быть рекурсивно разбита на подобласти. Такое разбиение порождает *несбалансированные* деревья (в отличие от B-деревьев и метода GiST) и может использоваться для реализации таких известных структур, как деревья квадрантов (quadtree), k-мерные деревья (k-D tree) или префиксные деревья (trie).

Обычно такие деревья слабо ветвятся и, следовательно, имеют большую глубину. Например, узел дерева квадрантов имеет не больше четырех дочерних узлов, а узел k-мерного дерева — только два. Это не проблема, когда дерево находится в оперативной памяти, но для дискового представления приходится решать непростую задачу эффективной упаковки узлов в страницы, чтобы минимизировать операции ввода-вывода. Для B-деревьев и GiST это не требуется, поскольку каждый узел дерева занимает страницу полностью.

[1] postgrespro.ru/docs/postgresql/18/spgist; backend/access/spgist/README.

Внутренний узел дерева SP-GiST содержит значение, соответствующее условию, которое выполняется для всех дочерних узлов. Такое значение часто называют *префиксом*; оно играет ту же роль, что *предикат* в индексе GiST. Ссылки на дочерние узлы в SP-GiST могут иметь *метки*.

Элементы листовых узлов содержат индексируемое значение (или его часть) и идентификатор соответствующей версии строки.

Как и GiST, метод доступа SP-GiST реализует только основные алгоритмы и берет на себя заботу о таких низкоуровневых задачах, как организация конкурентного доступа, блокирование и журналирование. Для добавления новых типов данных и алгоритмов разбиения пространства на области этот метод предоставляет интерфейс класса операторов. Класс операторов содержит значительную часть логики и определяет многие аспекты работы индекса.

Поиск[1] в дереве SP-GiST начинается с корневого узла и производится в глубину. Узлы, в которые имеет смысл заходить, выбираются с помощью опорной функции *согласованности*, похожей на аналогичную функцию в GiST. Для внутреннего узла дерева эта функция возвращает набор дочерних узлов, значения которых «согласуются» с поисковым предикатом. Функция выбирает узлы, не спускаясь в них, а ориентируясь на значения префикса и меток. Для листового узла функция определяет, удовлетворяет ли индексированное значение в этом узле поисковому предикату.

Время поиска значений в несбалансированном дереве может отличаться из-за разной глубины ветвей.

Для вставки значения в дерево SP-GiST используются две опорные функции. В узлах дерева, начиная с корня, *функция выбора* принимает одно из возможных решений: отправить новое значение в существующий дочерний узел, создать для него новый дочерний узел или (если значение не согласуется с префиксом узла) расщепить текущий узел. В выбранной в итоге листовой странице индекса может не оказаться достаточно места, и тогда *функция расщепления* решает, какие узлы перенести в новую страницу.

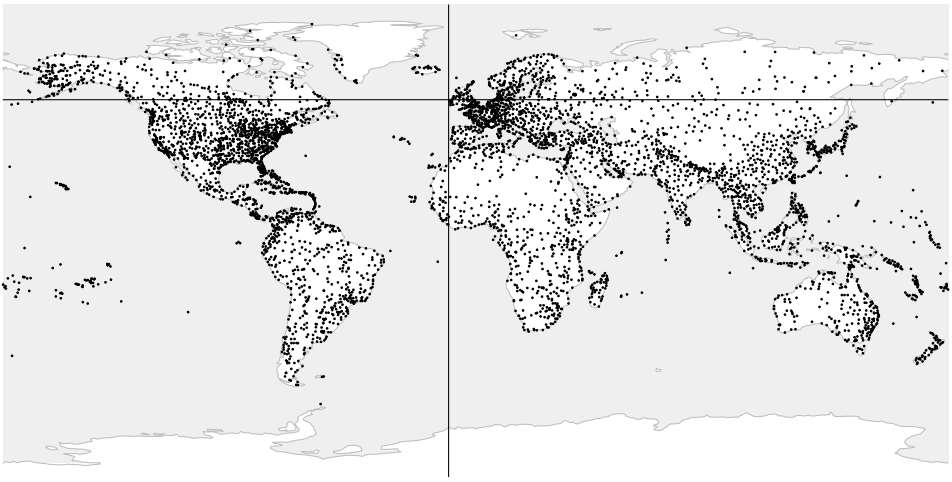
Дальше я рассмотрю эти алгоритмы на конкретных примерах.

[1] backend/access/spgist/spgscan.c, функция spgWalk.

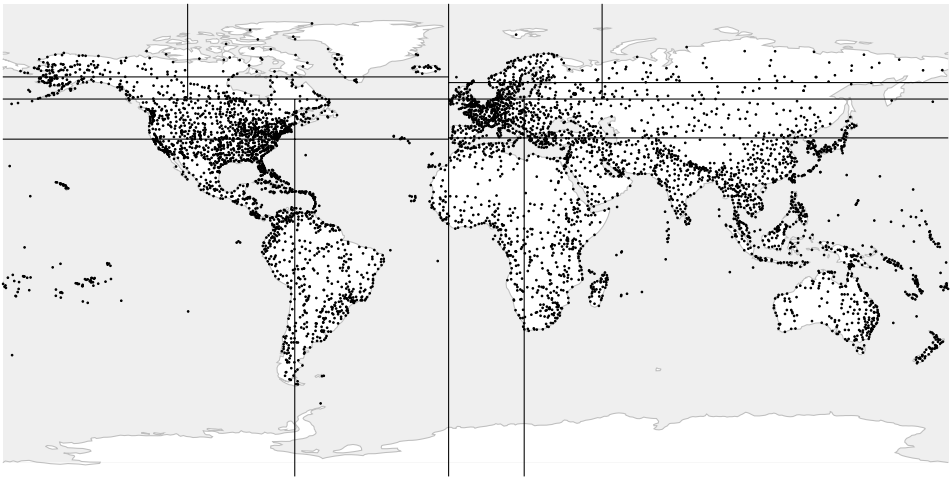
27.2. Дерево квадрантов для точек

Дерево квадрантов (quadtree) используется для индексирования точек на плоскости. Область рекурсивно разбивается на четыре части (квадранта) по отношению к выбранной точке. Эта точка называется *центроидом* и служит префиксом узла, то есть условием, определяющим положение дочерних значений.

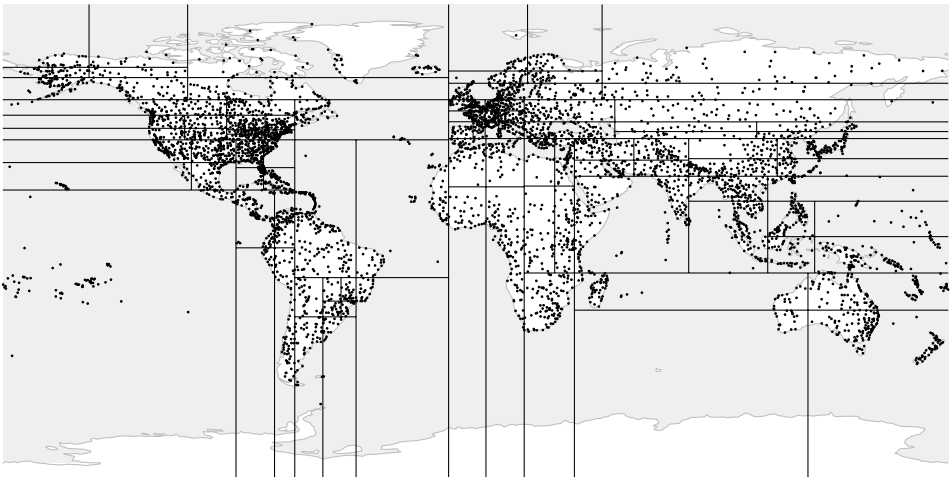
Корневой узел делит плоскость на четыре квадранта.



Затем каждый из них делится на собственные квадранты.



Процедура продолжается, пока не будет получено итоговое разбиение.



В этом примере использован индекс, построенный на расширенной табли-
с. 558 це аэропортов. На рисунках видно, что глубина ветвей дерева отличается в зависимости от плотности точек в соответствующих квадрантах. Для на-
80 глядности я взял небольшое значение параметра хранения *fillfactor*, чтобы дерево имело больше уровней:

```
=> CREATE INDEX airports_quad_idx ON airports_big
USING spgist(coordinates) WITH (fillfactor = 10);
```

По умолчанию для точек используется класс операторов `quad_point_ops`.

Класс операторов

Я уже называл основные опорные функции SP-GiST: функцию согласованности для поиска и функции выбора и расщепления для вставки.

Ниже приведен список опорных функций[1], которые реализует класс операторов `quad_point_ops`[2]. Все они являются обязательными.

[1] postgrespro.ru/docs/postgresql/18/spgist-extensibility.
[2] backend/access/spgist/spgquadtreeproc.c.

```
=> SELECT amprocnum, amproc::regproc                                    \dAp
FROM pg_am am
  JOIN pg_opclass opc ON opcmethod = am.oid
  JOIN pg_amproc amop ON amprocfamily = opcfamily
WHERE amname = 'spgist'
AND opcname = 'quad_point_ops'
ORDER BY amprocnum;
 amprocnum |          amproc
-----------+--------------------------
         1 | spg_quad_config
         2 | spg_quad_choose
         3 | spg_quad_picksplit
         4 | spg_quad_inner_consistent
         5 | spg_quad_leaf_consistent
(5 rows)
```

Функции имеют следующий смысл:

1) настроечная функция, которая сообщает методу доступа характеристики класса операторов;
2) функция выбора узла для вставки значения;
3) функция распределения узлов при расщеплении;
4) функция внутренней согласованности, которая определяет, согласуется ли с поисковым предикатом значение во *внутреннем* узле;
5) функция листовой согласованности, которая определяет, согласуется ли с поисковым предикатом значение в *листовом* узле.

Кроме перечисленных, есть и несколько необязательных функций.

Класс операторов `quad_point_ops` поддерживает стратегии, общие с GiST[1]: *с. 562*

```
=> SELECT amopopr::regoperator, oprcode::regproc, amopstrategy          \dAo
FROM pg_am am
  JOIN pg_opclass opc ON opcmethod = am.oid
  JOIN pg_amop amop ON amopfamily = opcfamily
  JOIN pg_operator opr ON opr.oid = amopopr
WHERE amname = 'spgist'
AND opcname = 'quad_point_ops'
ORDER BY amopstrategy;
```

[1] include/access/stratnum.h.

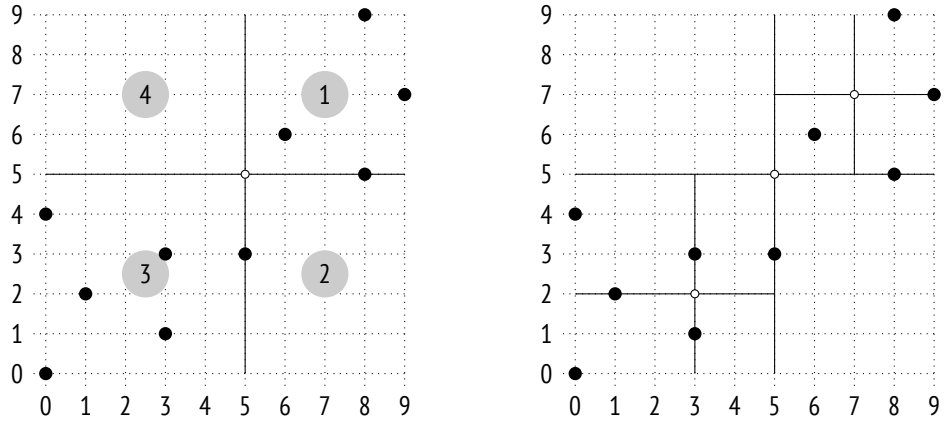
```
     amopopr      |    oprcode     | amopstrategy
------------------+----------------+--------------
 <<(point,point)  | point_left     |            1
 >>(point,point)  | point_right    |            5
 ~=(point,point)  | point_eq       |            6
 <@(point,box)    | on_pb          |            8
 <<|(point,point) | point_below    |           10
 |>>(point,point) | point_above    |           11
 <->(point,point) | point_distance |           15
 <^(point,point)  | point_below    |           29
 >^(point,point)  | point_above    |           30
(9 rows)
```

Например, с помощью оператора «выше» >^ можно найти аэропорты, расположенные севернее Диксона:

```
=> SELECT airport_code, airport_name->>'en'
FROM airports_big
WHERE coordinates >^ '(80.3817,73.5167)'::point;
 airport_code |          ?column?
--------------+---------------------------
 THU          | Thule Air Base
 YEU          | Eureka Airport
 YLT          | Alert Airport
 YRB          | Resolute Bay Airport
 LYR          | Svalbard Airport, Longyear
 NAQ          | Qaanaaq Airport
 YGZ          | Grise Fiord Airport
 DKS          | Dikson Airport
(8 rows)
=> EXPLAIN (costs off) SELECT airport_code
FROM airports_big
WHERE coordinates >^ '(80.3817,73.5167)'::point;
                          QUERY PLAN
---------------------------------------------------------------
 Bitmap Heap Scan on airports_big
   Recheck Cond: (coordinates >^ '(80.3817,73.5167)'::point)
   -> Bitmap Index Scan on airports_quad_idx
       Index Cond: (coordinates >^ '(80.3817,73.5167)'::point)
(4 rows)
```

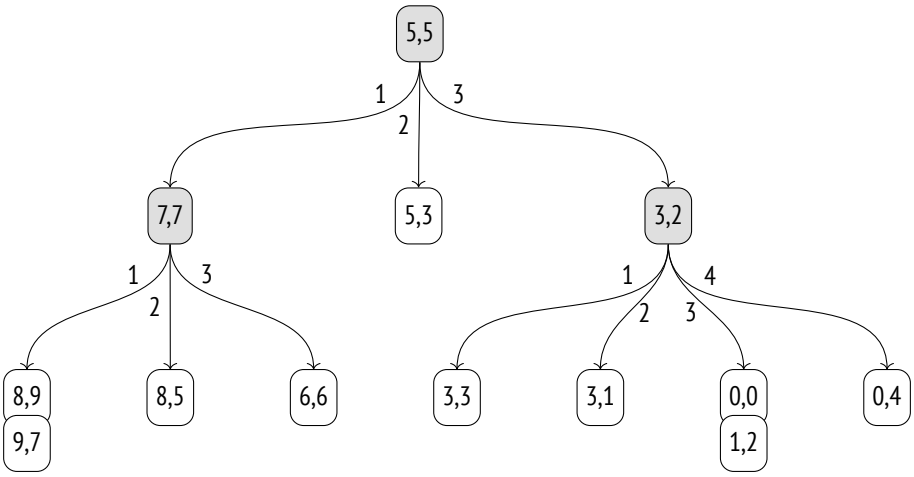
Рассмотрим теперь подробнее устройство и функционирование дерева квадрантов на простом примере с несколькими точками, который нам уже
с. 563 встречался в главе про GiST.

Вот как может выглядеть разбиение плоскости в этом случае:



Левый рисунок показывает нумерацию квадрантов на примере одного уровня дерева; на следующих иллюстрациях я для определенности буду располагать дочерние узлы слева направо именно в такой последовательности. Точки, лежащие на границах, относятся к квадранту с меньшим номером. На правом рисунке показано окончательное разбиение.

Ниже приведена возможная структура индекса для этого примера. Каждый внутренний узел ссылается максимум на четыре дочерних узла, и каждая ссылка помечена номером квадранта:



Страничная организация

Между узлами дерева SP-GiST и страницами нет однозначного соответствия, которое характерно для B-деревьев и GiST-индексов. Из-за того, что внутренние узлы обычно имеют не очень много дочерних, в одну страницу приходится упаковывать несколько узлов. Внутренние узлы при этом хранятся на одних (*внутренних*) страницах, а листовые — на других (*листовых*).

Индексные записи внутренних страниц содержат значение, используемое как префикс, и набор ссылок на дочерние узлы, каждая из которых может сопровождаться меткой.

Записи листовых страниц состоят из значения и идентификатора табличной версии строки.

Все листовые узлы, относящиеся к одному и тому же внутреннему, хранятся в одной странице и связаны в список. Если в странице не хватает места, список может быть перемещен на другую страницу[1] или может произойти расщепление, но в любом случае список никогда не разрывается между несколькими страницами.

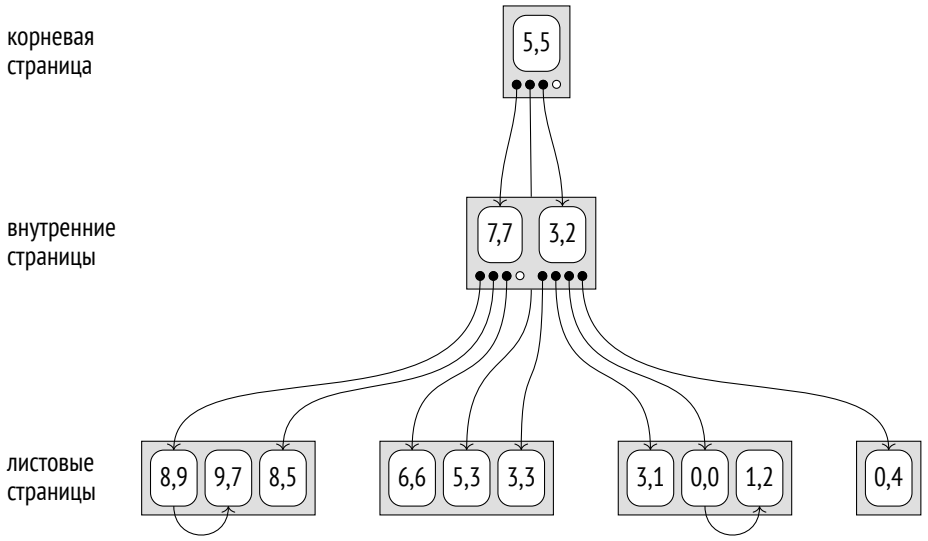
Чтобы экономно расходовать место, алгоритм старается располагать новые узлы в одних и тех же страницах, пока те не заполнятся. Номера последних использованных страниц кешируются обслуживающими процессами, а также периодически сохраняются в *метастранице* с нулевым номером. Метастраница не ссылается на корневой узел, как это происходит в случае B-дерева; корень SP-GiST всегда находится в первой странице.

> К сожалению, в `pageinspect` нет функций для исследования индекса SP-GiST, но вместо него можно воспользоваться сторонним расширением `gevel`[2]. Попытка перенести его функциональность в `pageinspect` предпринималась, но не увенчалась успехом[3].

Вариант распределения узлов дерева по страницам для нашего примера показан на рисунке ниже. На самом деле класс операторов `quad_point_ops` не

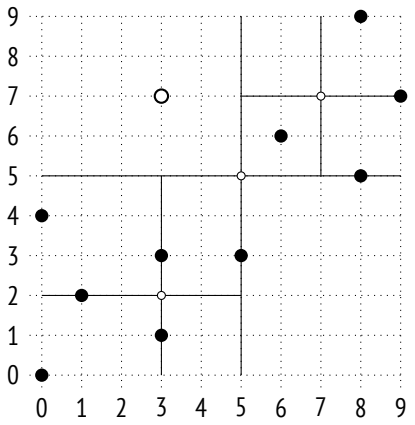
[1] backend/access/spgist/spgdoinsert.c, функция moveLeafs.
[2] sigaev.ru/git/gitweb.cgi?p=gevel.git.
[3] commitfest.postgresql.org/15/1207.

использует метки. Поскольку дочерних узлов не может быть больше четырех, хранится массив фиксированного размера из четырех ссылок, некоторые из которых могут быть пустыми.



Поиск

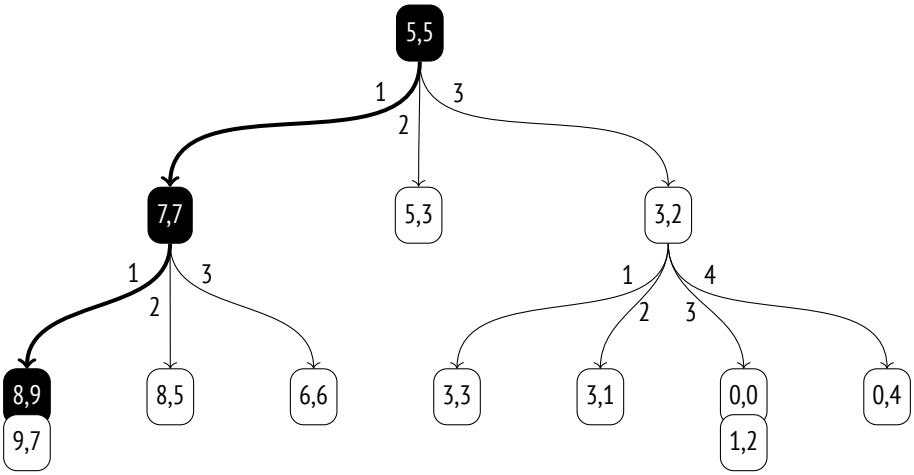
Рассмотрим на этом же примере алгоритм поиска точек, расположенных выше точки (3,7).



Поиск начинается с корневого узла. Функция внутренней согласованности[1] показывает, в какие дочерние узлы надо спускаться. Точка (3,7) сравнивается с центроидом корневого узла (5,5), и выбираются квадранты, в которых могут находиться искомые точки — в данном случае первый и четвертый.

В узле с центроидом (7,7) снова выбираются дочерние узлы для спуска. Подходят первый и четвертый квадранты, но четвертый пуст, так что необходимо проверить один листовой дочерний узел. Его точки сравниваются с точкой (3,7) из запроса с помощью функции листовой согласованности[2]. Из них только (8,9) удовлетворяет условию «выше».

Остается вернуться на уровень выше и проверить узел, соответствующий четвертому квадранту корневого узла. Он пуст, и на этом поиск завершен.

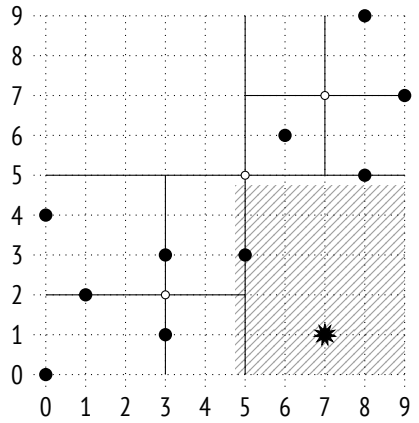


Вставка

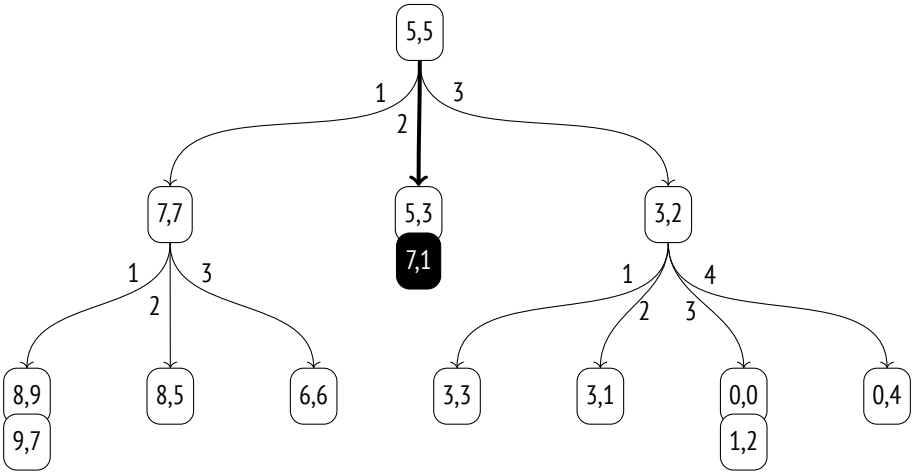
При вставке значения в дерево SP-GiST[3] каждое следующее действие определяется функцией выбора[4]. В данном случае она просто направляет новую точку в тот из существующих квадрантов, к которому она принадлежит.

[1] backend/access/spgist/spgquadtreeproc.c, функция spg_quad_inner_consistent.
[2] backend/access/spgist/spgquadtreeproc.c, функция spg_quad_leaf_consistent.
[3] backend/access/spgist/spgdoinsert.c, функция spgdoinsert.
[4] backend/access/spgist/spgquadtreeproc.c, функция spg_quad_choose.

Например, добавим значение (7,1):



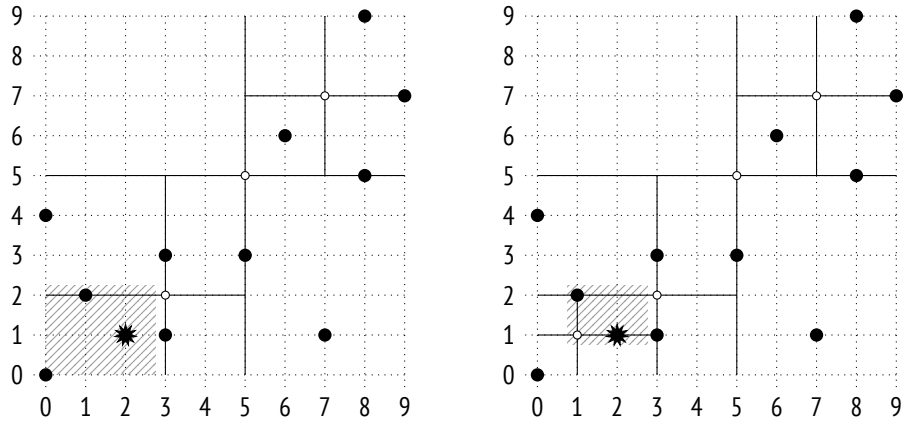
Значение принадлежит второму квадранту и будет добавлено к соответствующему узлу дерева:



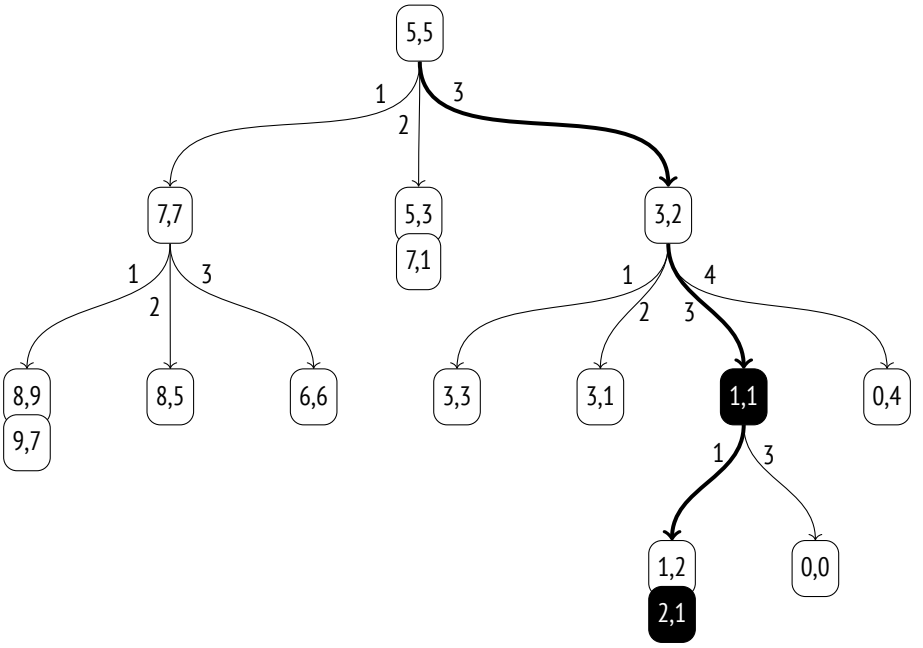
Если значение попадает в квадрант, список листовых узлов которого оказывается слишком велик (он должен помещаться в одной странице), происходит расщепление. Функция расщепления[1] определяет новый центроид, вычисляя среднее значение координат всех точек, и таким образом распределяет дочерние узлы по новым квадрантам более или менее равномерно.

[1] backend/access/spgist/spgquadtreeproc.c, функция spg_quad_picksplit.

В примере на рисунке ниже добавление точки (2,1) приводит к переполнению узла:



В дерево добавляется новый внутренний узел с центроидом (1,1), а точки (0,0), (1,2) и (2,1) распределяются между новыми квадрантами:



Свойства

Свойства уровня метода доступа

```
=> SELECT a.amname, p.name, pg_indexam_has_property(a.oid, p.name)
FROM pg_am a, unnest(array[
  'can_order', 'can_unique', 'can_multi_col',
  'can_exclude', 'can_include'
]) p(name)
WHERE a.amname = 'spgist';
 amname |     name      | pg_indexam_has_property
--------+---------------+-------------------------
 spgist | can_order     | f
 spgist | can_unique    | f
 spgist | can_multi_col | f
 spgist | can_exclude   | t
 spgist | can_include   | t
(5 rows)
```

Поддержка сортировки значений и уникальности отсутствует. Не поддерживаются и многоколоночные индексы.

Ограничения исключения поддерживаются, как и в случае GiST.

Индекс SP-GiST можно создавать с дополнительными include-столбцами. v. 14

Свойства уровня индекса

Свойства индекса SP-GiST отличаются от свойств GiST тем, что не допускается кластеризация:

```
=>  SELECT p.name, pg_index_has_property('airports_quad_idx', p.name)
FROM unnest(array[
  'clusterable', 'index_scan', 'bitmap_scan', 'backward_scan'
]) p(name);
     name      | pg_index_has_property
---------------+-----------------------
 clusterable   | f
 index_scan    | t
 bitmap_scan   | t
 backward_scan | f
(4 rows)
```

Оба способа получения идентификаторов версий (по одному и битовой картой) поддерживаются. Обратное сканирование SP-GiST не имеет смысла.

Свойства уровня столбца

Основная часть свойств столбцов неизменна:

```
=> SELECT p.name,
  pg_index_column_has_property('airports_quad_idx', 1, p.name)
FROM unnest(array[
  'orderable', 'search_array', 'search_nulls'
]) p(name);
     name     | pg_index_column_has_property
--------------+------------------------------
 orderable    | f
 search_array | f
 search_nulls | t
(3 rows)
```

Сортировка не поддерживается, и все свойства, связанные с ней, не имеют смысла и отключены.

До сих пор я ничего не говорил про неопределенные значения, но, как видно из свойств индекса, они поддерживаются. В отличие от GiST, индекс SP-GiST не хранит неопределенные значения в основном дереве. Для них создается отдельное дерево с корнем на второй странице индекса. Таким образом, первые три страницы всегда имеют фиксированный смысл: метастраница, корень основного дерева и корень дерева неопределенных значений.

Часть свойств уровня столбца может меняться в зависимости от класса операторов:

```
=> SELECT p.name,
  pg_index_column_has_property('airports_quad_idx', 1, p.name)
FROM unnest(array[
  'returnable', 'distance_orderable'
]) p(name);
        name        | pg_index_column_has_property
--------------------+------------------------------
 returnable         | t
 distance_orderable | t
(2 rows)
```

В данном случае (как и во всех остальных примерах этой главы) индекс может использоваться для сканирования только индекса.

Но в общем случае класс операторов может не хранить полное значение в листовых страницах, выполняя вместо этого перепроверку по таблице.

Это, например, позволяет использовать индексы SP-GiST в PostGIS для потенциально больших значений типа `geometry`.

Поиск ближайших соседей поддерживается; в классе операторов мы видели упорядочивающий оператор `<->`. v. 12

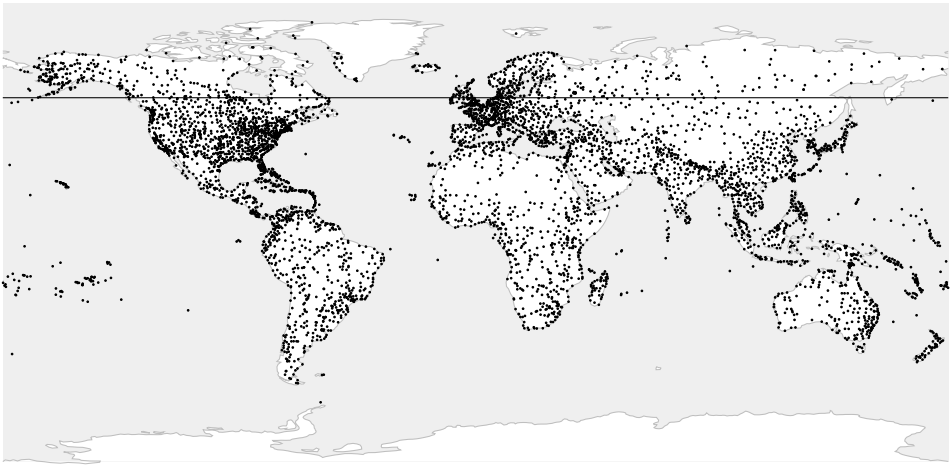
27.3. К-мерные деревья для точек

Для точек на плоскости можно предложить и другой способ разбиения пространства: делить плоскость не на четыре части, а на две. Такое разбиение использует класс операторов `kd_point_ops`[1]:

```
=> CREATE INDEX airports_kd_idx ON airports_big
USING spgist(coordinates kd_point_ops);
```

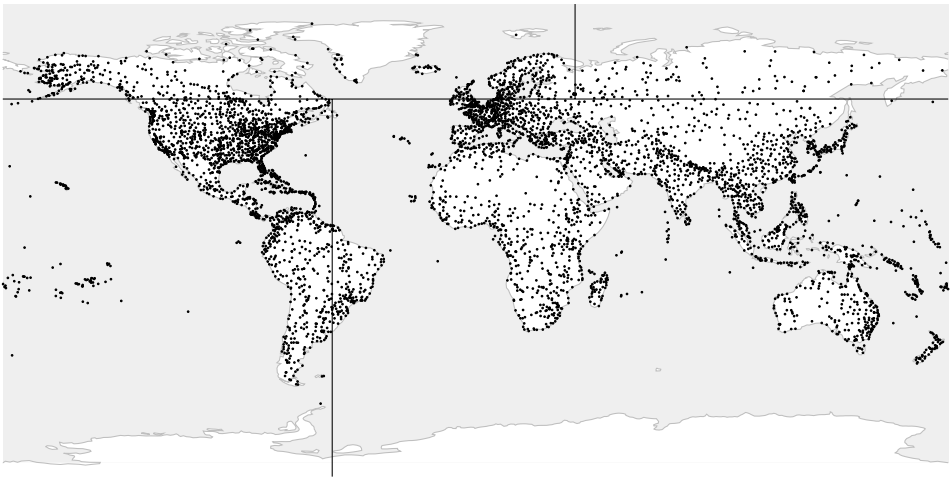
Обратите внимание, что индексируемые значения, префиксы и метки могут иметь различные типы данных. Для этого класса операторов значения являются точками, префиксы — вещественными числами, а метки отсутствуют (как и в `quad_point_ops`).

Выберем какую-нибудь координату на оси ординат (в примере с аэропортами — широту). Она поделит плоскость на две части, верхнюю и нижнюю:

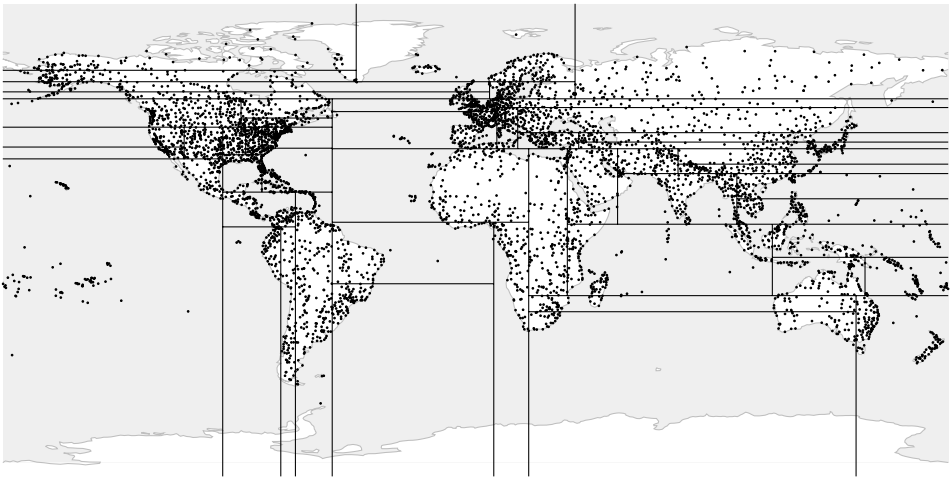


[1] backend/access/spgist/spgkdtreeproc.c.

Для каждой из двух областей выберем координаты на оси абсцисс (долготу), которые поделят области на две части, левую и правую:



Будем продолжать делить каждую из частей попеременно то по горизонтали, то по вертикали, пока в каждой из них не останется столько точек, сколько помещается на одну страницу индекса:



У всех внутренних узлов дерева, построенного таким образом, будет всего два дочерних узла. Метод легко обобщается на случай пространства произвольной размерности, поэтому и деревья называются в литературе *k-мерными* (k-D tree).

27.4. Префиксное дерево для строк

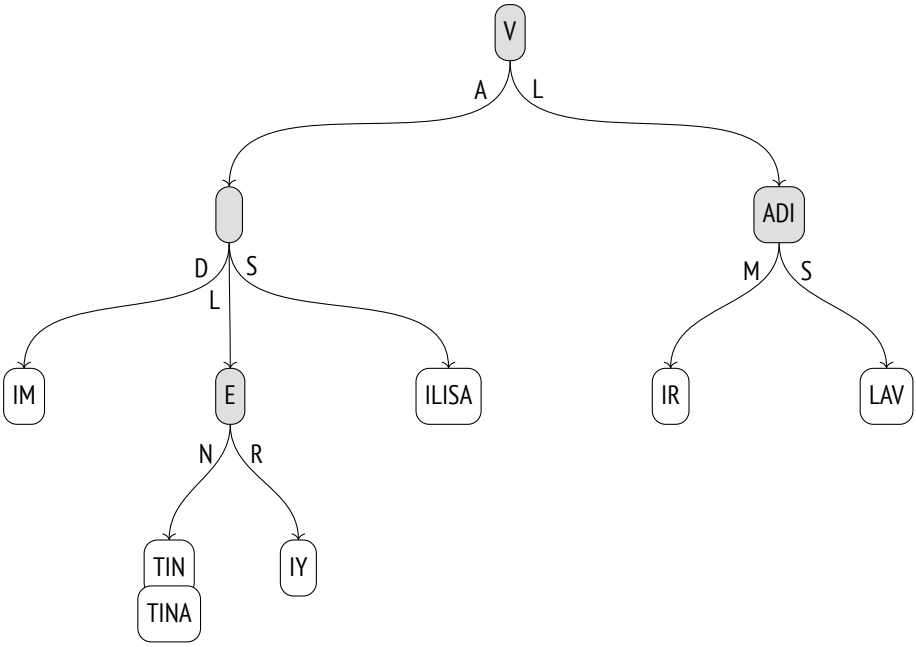
Класс операторов `text_ops` [1] для SP-GiST реализует префиксное дерево (radix tree) для строк. Здесь *префикс* внутреннего узла действительно является префиксом, общим для всех строк в дочерних узлах.

Ссылки на дочерние узлы дерева помечены первым байтом значений, следующих за префиксом.

> На рисунке я для ясности показываю префикс как один символ, но это верно только для восьмибитных кодировок. В общем случае класс операторов работает со строкой как с последовательностью байтов. Кроме того, префикс может принимать еще несколько значений, имеющих специальный смысл, так что фактически под префикс выделено два байта.

В дочерних узлах хранятся те части значений, которые следуют за префиксом и меткой. На долю листовых узлов остаются только окончания.

Вот пример префиксного дерева, построенного по нескольким именам:



[1] backend/access/spgist/spgtextproc.c.

Полное значение ключа индексирования в листовом узле можно восстановить конкатенацией всех префиксов и меток, следуя от корня дерева к этому узлу. За счет того, что значения не хранятся целиком, а реконструируются по мере необходимости, представление в виде префиксного дерева может иногда оказаться существенно компактнее B-дерева.

Класс операторов

Класс операторов `text_ops` поддерживает операторы сравнения, традиционные для порядкового типа, которым является текстовая строка:

```
\dAo  => SELECT oprname, oprcode::regproc, amopstrategy
      FROM pg_am am
        JOIN pg_opclass opc ON opcmethod = am.oid
        JOIN pg_amop amop ON amopfamily = opcfamily
        JOIN pg_operator opr ON opr.oid = amopopr
      WHERE amname = 'spgist'
      AND opcname = 'text_ops'
      ORDER BY amopstrategy;
 oprname |     oprcode     | amopstrategy
---------+-----------------+--------------
 ~<~     | text_pattern_lt |            1
 ~<=~    | text_pattern_le |            2
 =       | texteq          |            3
 ~>=~    | text_pattern_ge |            4
 ~>~     | text_pattern_gt |            5
 <       | text_lt         |           11
 <=      | text_le         |           12
 >=      | text_ge         |           14
 >       | text_gt         |           15
 ^@      | starts_with     |           28
(10 rows)
```

Операторы с тильдами отличаются от обычных тем, что работают не с сим-
с. 399 волами, а с байтами (аналогично операторам класса `text_pattern_ops` для B-дерева), и поэтому органично поддерживаются префиксным деревом, которое тоже размечено байтами. Обычные же операторы `>=` и `<` учитывают правила сортировки и вынуждены анализировать символы. Они поддерживаются индексом только при использовании правила сортировки «C».

Для ускорения поиска по условию `LIKE` класс операторов поддерживает оператор *префиксного поиска* `^@`:

```
=> CREATE INDEX tickets_spgist_idx ON tickets
  USING spgist(passenger_name);
=> EXPLAIN (costs off) SELECT *
FROM tickets
WHERE passenger_name LIKE 'IVAN%';
                     QUERY PLAN
-----------------------------------------------------
 Bitmap Heap Scan on tickets
   Filter: (passenger_name ~~ 'IVAN%'::text)
   -> Bitmap Index Scan on tickets_spgist_idx
       Index Cond: (passenger_name ^@ 'IVAN'::text)
(4 rows)
```

Поиск

Рассмотрим на примере с именами выполнение следующего запроса:

```
SELECT * FROM names
WHERE name ~>=~ 'VALERIY' AND name ~<~ 'VLADISLAV';
```

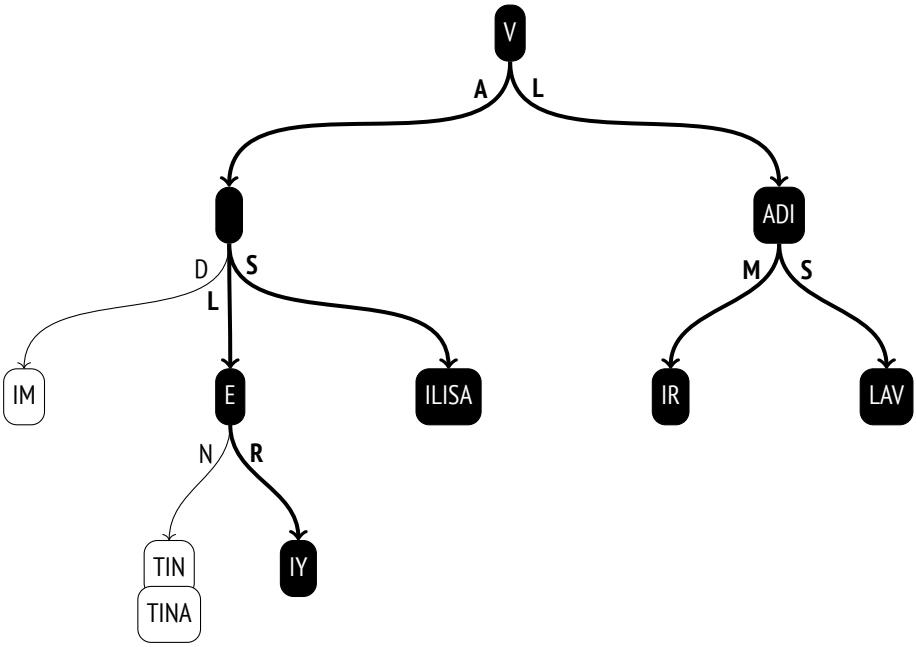
Сначала функция внутренней согласованности[1] вызывается в корне дерева, и она должна определить, в какие дочерние узлы спускаться. Функция конкатенирует префикс `V` и метки `A` и `L`. Значение *VA* подставляется в условие запроса, в котором строковые константы обрезаются так, чтобы их длина не превышала длины проверяемого значения: *VA* `~>=~ 'VA' AND` *VA* `~<~ 'VL'`. Условие выполняется, так что дочерний узел с меткой `A` должен быть проверен. Аналогично проверяется и значение *VL*. Оно тоже подходит, поэтому и узел с меткой `L` требует проверки.

Возьмем теперь узел, соответствующий значению `VA`. Его префикс пуст, и для трех дочерних узлов функция внутренней согласованности восстанавливает значения *VAD*, *VAL* и *VAS*, конкатенируя полученное на предыдущем шаге `VA` с меткой. Условие *VAD* `~>=~ 'VAL' AND` *VAD* `~<~ 'VER'` не выполняется, но два других значения подходят.

Обходя таким образом дерево вглубь и отбрасывая неподходящие ветви, алгоритм добирается до листовых узлов. Функция листовой согласованности[2]

[1] backend/access/spgist/spgtextproc.c, функция spg_text_inner_consistent.
[2] backend/access/spgist/spgtextproc.c, функция spg_text_leaf_consistent.

проверяет, соответствует ли восстановленное по мере спуска значение исходному условию. Подходящие значения возвращаются в качестве результата индексного сканирования.



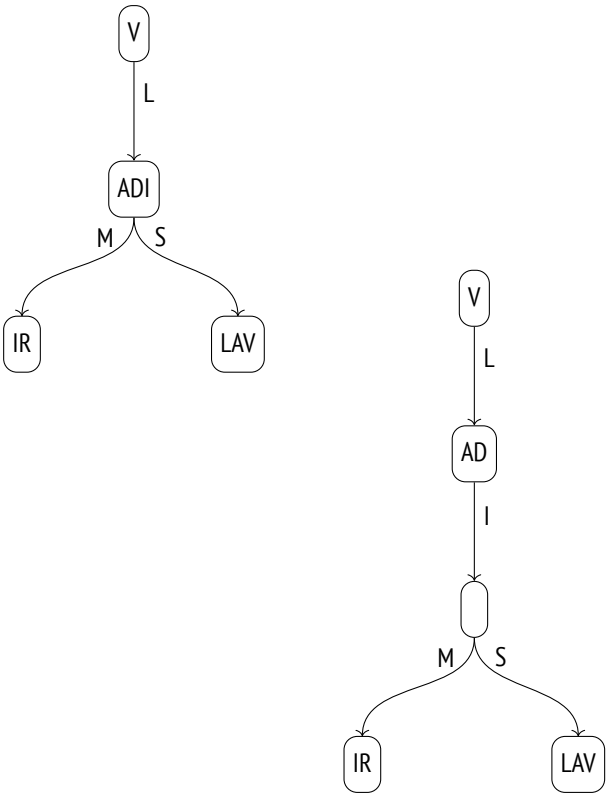
Обратите внимание, что хотя в запросе используются операторы «больше» и «меньше», обычные для B-деревьев, поиск диапазона по дереву SP-GiST гораздо менее эффективен. В случае B-дерева достаточно спуститься только к одному крайнему значению диапазона и затем прочитать цепочку листовых страниц.

Вставка

Функция выбора классов операторов для точек всегда может направить новое значение в одну из существующих подобластей (квадрант или половину). В префиксном дереве это не так: новое значение может не соответствовать имеющемуся префиксу, и тогда внутренний узел приходится разделять.

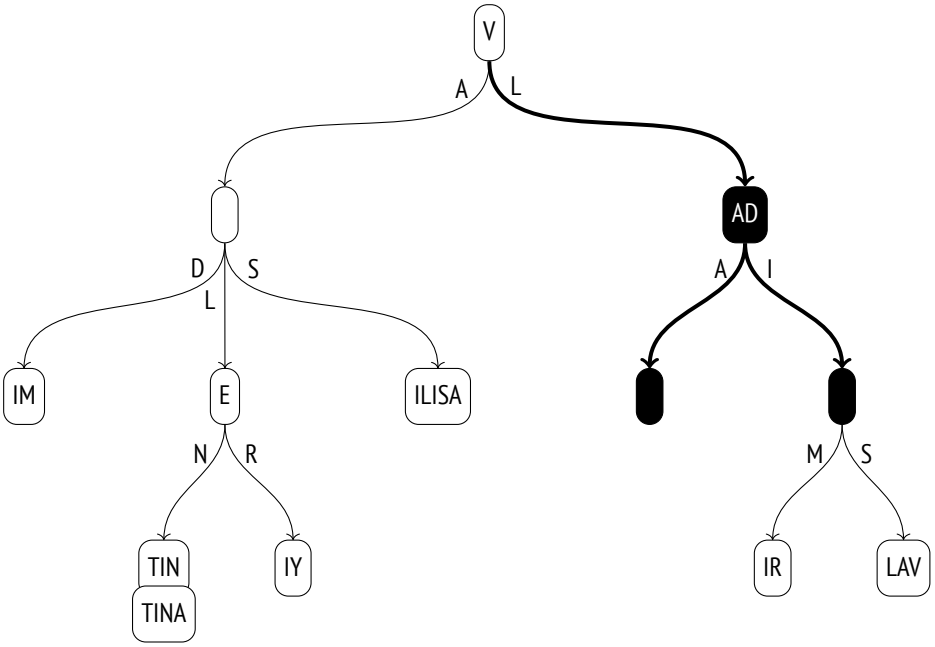
В качестве примера рассмотрим добавление имени `VLADA` к уже имеющемуся дереву.

От корня дерева с помощью функции выбора [1] можно успешно спуститься на один узел (V + L), но остаток значения ADA не согласуется с префиксом ADI. Узел необходимо разделить на два: в одном оставить наибольший общий префикс, то есть AD, а остаток префикса перенести на уровень ниже:



После этого функция выбора вызывается в том же узле еще раз. Префикс теперь соответствует значению, но нет дочернего узла с подходящей меткой (A), и функция принимает решение создать такой узел. Конечный результат показан на рисунке ниже; на нем выделены узлы, добавленные или измененные в ходе вставки.

[1] backend/access/spgist/spgtextproc.c, функция spg_text_choose.



Свойства

Свойства уровня метода доступа и индекса я уже показывал, они не меняются от класса к классу. Большинство свойств уровня столбца также постоянны.

```
=> SELECT p.name,
  pg_index_column_has_property('tickets_spgist_idx', 1, p.name)
FROM unnest(array[
  'returnable', 'distance_orderable'
]) p(name);
        name         | pg_index_column_has_property
---------------------+------------------------------
 returnable          | t
 distance_orderable  | f
(2 rows)
```

Хотя индексируемые значения и не хранятся в явном виде в дереве, сканирование только индекса работает, поскольку значения восстанавливаются по мере спуска от корня до листовых узлов.

А вот оператор расстояния для строк не определен, и поиск ближайших соседей не поддерживается этим классом операторов.

Это не значит, что для строк нельзя ввести понятие расстояния. Например, расширение pg_trgm добавляет оператор для расстояния на основе триграмм: считается, что чем меньше доля общих триграмм у двух строк, тем дальше они находятся друг от друга. Другой вариант — расстояние Левенштейна, которое определяется как минимальное количество односимвольных операций, преобразующих одну строку в другую. Функция для расчета такого расстояния есть в расширении fuzzystrmatch. Но ни одно из расширений не предоставляет класс операторов с поддержкой SP-GiST.

27.5. Другие типы данных

Помимо рассмотренных примеров индексирования точек и текстовых строк, реализованы и другие классы операторов для SP-GiST.

Геометрические типы. Класс операторов box_ops[1] реализует дерево квадрантов для прямоугольников. Прямоугольники представляются точками в четырехмерном пространстве, так что область разбивается на шестнадцать частей.

Класс операторов poly_ops позволяет индексировать полигоны. Это неточный класс операторов: фактически вместо полигонов используются ограничивающие прямоугольники, как в box_ops, с перепроверкой по таблице.

Какой из методов доступа предпочесть — GiST или SP-GiST, — существенно зависит от вида индексируемых данных. Например, документация PostGIS рекомендует SP-GiST для сильно пересекающихся объектов («лапши»)[2].

Диапазонные типы. Дерево квадрантов для диапазонов предоставляет класс операторов range_ops[3]. Интервал представляется двумерной точкой: нижняя граница становится абсциссой, а верхняя — ординатой.

Сетевые адреса. Для типа данных inet класс операторов inet_ops[4] реализует префиксное дерево.

[1] backend/utils/adt/geo_spgist.c.
[2] postgis.net/docs/using_postgis_dbmanagement.html#spgist_indexes.
[3] backend/utils/adt/rangetypes_spgist.c.
[4] backend/utils/adt/network_spgist.c.

28

Индекс GIN

28.1. Общий принцип

По задумке авторов, GIN — не американский спиртной напиток, а могущественный и неустрашимый дух[1]. Но есть и формальная расшифровка: Generalized Inverted Index, обобщенный обратный (инвертированный) индекс.

Этот метод доступа работает с типами данных, *значения* которых состоят из *элементов*, а не являются атомарными (так, например, в контексте полнотекстового поиска документы состоят из лексем). При этом индексируются не сами значения, как мы видели в случае GiST, а только их элементы; от каждого элемента по индексу можно перейти ко всем значениям, в которых этот элемент встречается.

Хорошая аналогия для этого метода — предметный указатель в конце книги. В указателе собраны все важные термины, и для каждого приведен список страниц, на которых этот термин упоминается. Чтобы указателем было удобно пользоваться, он составляется по алфавиту, иначе в нем было бы невозможно быстро ориентироваться. Так и GIN полагается на то, что элементы составных значений можно упорядочить, и в качестве основной
с. 523 структуры данных использует B-дерево.

Реализация дерева элементов GIN во многом более проста, чем «настоящее» B-дерево, но изначально рассчитана на то, что разные значения будут состоять из относительно небольшого набора многократно повторяющихся элементов.

[1] postgrespro.ru/docs/postgresql/18/gin;
backend/access/gin/README.

Из этого предположения следуют два важных вывода:

- Элементы в индексе следует хранить в единственном экземпляре.

  С каждым элементом связан список идентификаторов версий. Если такой список (posting list) не слишком велик, то он хранится вместе с элементом, но при увеличении может быть вынесен в отдельное B-дерево (posting tree). Упорядочены не только деревья, но и списки идентификаторов; это не имеет значения для пользователя, но позволяет ускорить доступ и уменьшить объем.

- Нет смысла удалять элементы из дерева.

  Если набор идентификаторов элемента оказался пустым, с большой вероятностью тот же элемент появится снова в составе другого значения.

Таким образом, индекс состоит из дерева элементов, к листовым записям которого привязаны либо плоские списки, либо деревья идентификаторов.

Как и рассмотренные ранее методы доступа GiST и SP-GiST, GIN позволяет индексировать самые разные типы данных, предоставляя упрощенный интерфейс класса операторов. Обычно операторы этого класса проверяют, удовлетворяет ли проиндексированное составное значение некоторому набору элементов (как, например, в случае полнотекстового поиска оператор `@@` проверяет соответствие документа запросу).

Для работы с конкретными типами данных метод GIN должен уметь разбивать составные значения на лексемы, сортировать лексемы и проверять, соответствует ли найденное значение запросу. Эти операции реализуются опорными функциями класса операторов.

28.2. Индекс для полнотекстового поиска

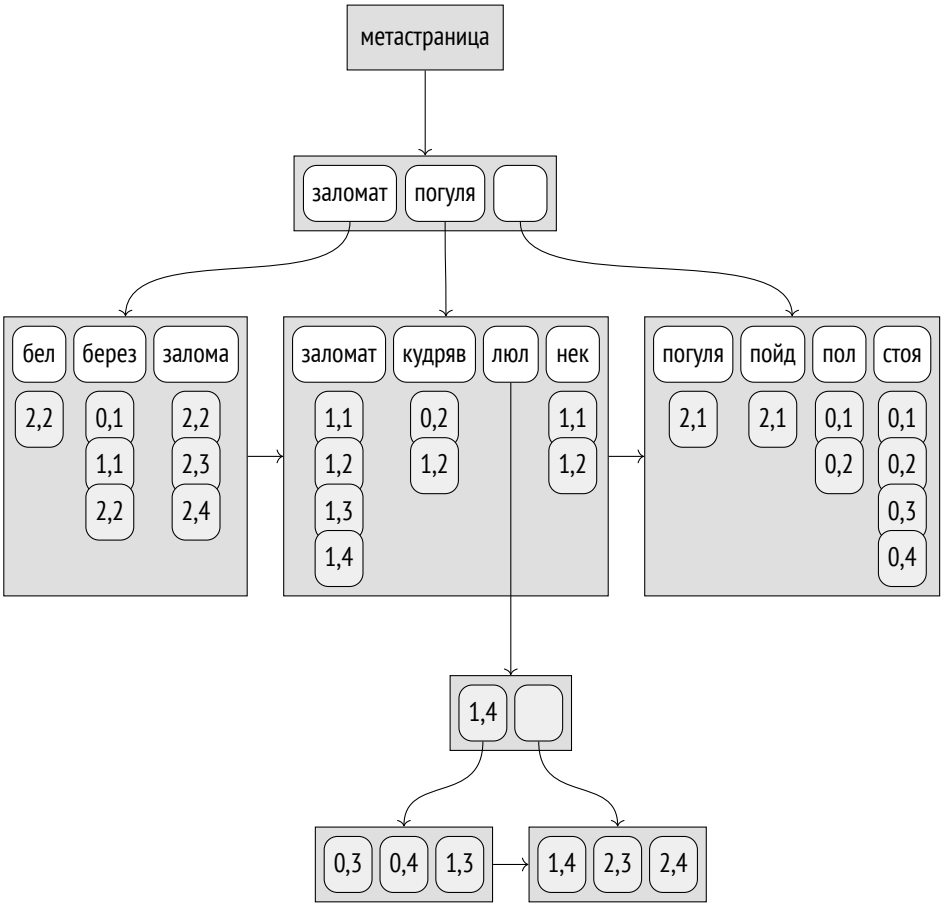
Основная область применения GIN — ускорение полнотекстового поиска, поэтому я продолжу пример, начатый в главе про GiST-индекс. Понятно, что составными значениями в этом случае являются *документы*, а элементами этих значений — *лексемы*.

с. 578

с. 579 Построим GIN-индекс на таблице с «Во поле береза...»:

```
=> CREATE INDEX ts_gin_idx ON ts USING gin(doc_tsv);
```

Возможная структура такого индекса показана на рисунке:



В отличие от предыдущих иллюстраций, здесь показаны (и выделены цветом) конкретные значения идентификаторов, поскольку они будут важны для понимания алгоритмов. Значения предполагают, что строки таблицы имеют следующие идентификаторы:

```
=> SELECT ctid, * FROM ts;
  ctid |           doc            |            doc_tsv
-------+--------------------------+-------------------------------
 (0,1) | Во поле береза стояла    | 'берез':3 'пол':2 'стоя':4
 (0,2) | Во поле кудрявая стояла  | 'кудряв':3 'пол':2 'стоя':4
 (0,3) | Люли, люли, стояла       | 'люл':1,2 'стоя':3
 (0,4) | Люли, люли, стояла       | 'люл':1,2 'стоя':3
 (1,1) | Некому березу заломати   | 'берез':2 'заломат':3 'нек':1
 (1,2) | Некому кудряву заломати  | 'заломат':3 'кудряв':2 'нек':1
 (1,3) | Люли, люли, заломати     | 'заломат':3 'люл':1,2
 (1,4) | Люли, люли, заломати     | 'заломат':3 'люл':1,2
 (2,1) | Я пойду погуляю          | 'погуля':3 'пойд':2
 (2,2) | Белую березу заломаю     | 'бел':1 'берез':2 'залома':3
 (2,3) | Люли, люли, заломаю      | 'залома':3 'люл':1,2
 (2,4) | Люли, люли, заломаю      | 'залома':3 'люл':1,2
(12 rows)
```

Обратите внимание на некоторые отличия от обычного индекса на основе B-дерева. Крайний левый ключ во внутренних узлах B-дерева пуст из-за его избыточности; в GIN-индексе он не хранится вовсе. Из-за этого оказываются сдвинутыми и ссылки на дочерние узлы. Верхний ключ используется в обоих индексах, но в GIN он располагается на своем законном месте в крайней правой позиции. Узлы одного уровня B-дерева соединены двунаправленным списком; в GIN-индексе — однонаправленным, поскольку обход дерева выполняется всегда только в одну сторону. *с. 529*

В нашем умозрительном примере все списки идентификаторов поместились в обычные страницы, за исключением списка для лексемы «люл». Эта лексема встретилась в целых шести документах, и ее идентификаторы были перенесены в отдельное дерево.

Страничная организация

С точки зрения страничной организации GIN очень похож на B-дерево. Заглянуть внутрь индекса можно с помощью расширения `pageinspect`. Создадим GIN-индекс на таблице почтовой рассылки pgsql-hackers: *с. 584*

```
=> CREATE INDEX mail_gin_idx ON mail_messages USING gin(tsv);
```

Нулевая страница (метастраница) содержит общую статистику, такую как количество элементов и страниц других типов:

```
=> SELECT *
FROM gin_metapage_info(get_raw_page('mail_gin_idx',0)) \gx
-[ RECORD 1 ]----+-----------
pending_head     | 4294967295
pending_tail     | 4294967295
tail_free_size   | 0
n_pending_pages  | 0
n_pending_tuples | 0
n_total_pages    | 27108
n_entry_pages    | 17699
n_data_pages     | 9408
n_entries        | 999191
version          | 2
```

с. 76 GIN использует специальную область индексных страниц, например чтобы сохранить в ней биты, определяющие тип страницы:

```
=> SELECT flags, count(*)
FROM generate_series(0,27107) AS p, -- n_total_pages
  gin_page_opaque_info(get_raw_page('mail_gin_idx',p))
GROUP BY flags
ORDER BY 2;
         flags          | count
------------------------+-------
 {meta}                 |     1
 {}                     |   263
 {data}                 |  1517
 {data,leaf,compressed} |  7891
 {leaf}                 | 17436
(5 rows)
```

Страница с признаком meta — это, конечно, метастраница. Страницы с признаком data относятся к деревьям идентификаторов, а страницы без этого признака — к деревьям элементов. Внутри деревьев выделяются листовые страницы, несущие признак leaf.

Еще одна функция расширения pageinspect выдает информацию об идентификаторах, хранящихся в листовых страницах деревьев. Фактически каждая запись такого дерева содержит не один идентификатор, а небольшой список:

```
=> SELECT left(tids::text,67)||'...' tids
FROM gin_leafpage_items(get_raw_page('mail_gin_idx',2));
                                  tids
------------------------------------------------------------------------
 {"(23,1)","(31,1)","(56,1)","(59,5)","(306,5)","(469,5)","(538,5)",...
 {"(5257,3)","(5373,4)","(5686,8)","(5808,3)","(5864,4)","(5883,1)",...
 {"(12832,8)","(12833,1)","(13454,5)","(13590,3)","(13591,1)","(1392...
 ...
 {"(54024,5)","(54036,3)","(54072,6)","(54139,4)","(54140,6)","(5415...
 {"(57117,7)","(57187,1)","(57230,8)","(57231,1)","(57231,3)","(5723...
 {"(61503,4)","(61513,3)","(61527,7)","(61561,4)","(61562,1)","(6156...
(18 rows)
```

Идентификаторы в списках упорядочены, что позволяет сжимать данные (отсюда признак compressed). Вместо шестибайтного значения tid хранится разность с предыдущим значением, которая кодируется переменным числом байтов [1]: чем меньше эта разность, тем меньше места занимают данные.

Класс операторов

Вот список опорных функций для классов операторов GIN [2]:

```
=> SELECT amprocnum, amproc::regproc                                    \dAp
FROM pg_am am
  JOIN pg_opclass opc ON opcmethod = am.oid
  JOIN pg_amproc amop ON amprocfamily = opcfamily
WHERE amname = 'gin'
AND opcname = 'tsvector_ops'
ORDER BY amprocnum;
 amprocnum |              amproc
-----------+----------------------------------
         1 | gin_cmp_tslexeme
         2 | pg_catalog.gin_extract_tsvector
         3 | pg_catalog.gin_extract_tsquery
         4 | pg_catalog.gin_tsquery_consistent
         5 | gin_cmp_prefix
         6 | gin_tsquery_triconsistent
(6 rows)
```

[1] backend/access/gin/ginpostinglist.c.

[2] postgrespro.ru/docs/postgresql/18/gin-extensibility; backend/utils/adt/tsginidx.c.

Первая опорная функция сравнивает два элемента, в данном случае — лексемы. Если бы лексемы были представлены обычным типом SQL, для которого определен класс операторов B-дерева, то GIN автоматически использовал
с. 546 бы операторы сравнения из этого класса.

Пятая (необязательная) функция используется при *частичном* поиске и проверяет частичное соответствие элемента индекса ключу запроса. В нашем случае частичный поиск — это поиск лексем по префиксу. Например, запрос «п:*» соответствует всем лексемам на букву «п».

Вторая функция выделяет лексемы из документа, а третья — из поискового запроса. Разные функции нужны хотя бы потому, что документ и запрос представлены разными типами данных, `tsvector` и `tsquery`. К тому же функция для поискового запроса определяет, как будет выполняться поиск. Если запрос требует наличия в документе какой-либо лексемы, при поиске будут рассматриваться документы, содержащие хотя бы одну лексему из запроса. Если же такого условия нет (например, нужны документы, *не* содержащие определенную лексему), будут рассматриваться все документы вообще — что, конечно, значительно дороже.

v. 13 > При наличии в запросе других ключей поиска индекс сначала сканируется по ним, а затем полученные результаты перепроверяются. Это позволяет избежать бесполезного просмотра индекса.

Четвертая и шестая — функции согласованности, которые определяют соответствие найденного документа поисковому запросу. Четвертая функция получает на вход точные данные о том, какая лексема из запроса содержится в документе, а какая — нет. Шестая функция работает в условиях неопределенности и может вызываться, когда для части лексем еще неизвестно, присутствуют ли они в документе. Класс операторов не обязан реализовывать обе функции: достаточно предоставить одну, но в таком случае поиск может проиграть в эффективности.

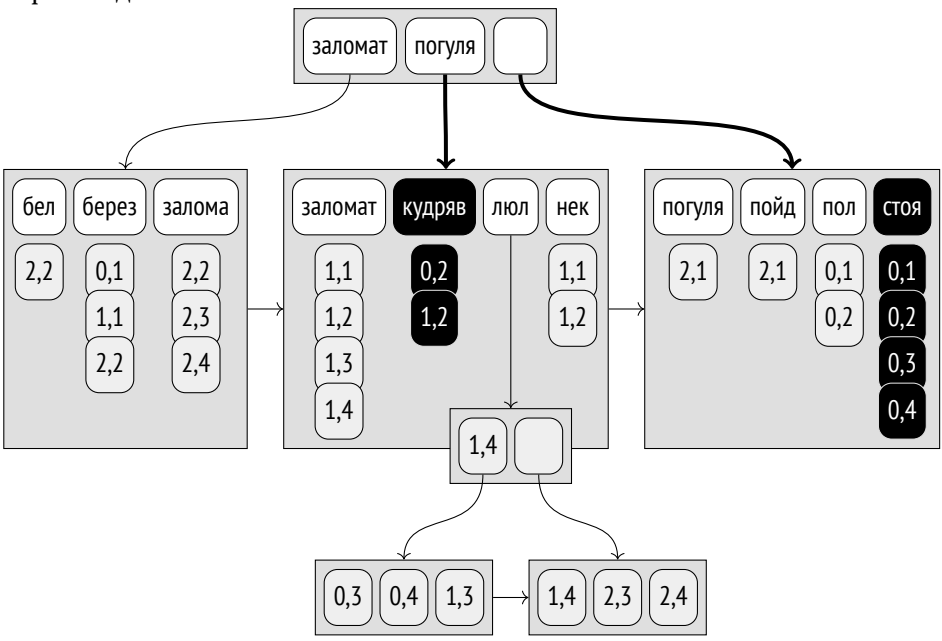
Класс операторов `tsvector_ops` поддерживает только один оператор сопоставления документа с поисковым запросом[1] `@@` — ровно тот же, который включает и класс операторов GiST.

[1] backend/utils/adt/tsvector_op.c, функция ts_match_vq.

Поиск

Рассмотрим процесс выполнения поиска по запросу «`стояла | кудрявая`», в котором две лексемы соединены условием «или». Сначала с помощью опорной функции [1] из поискового запроса типа `tsquery` выделяются отдельные лексемы (*ключи поиска*): «стоя» и «кудряв».

Поскольку запрос требует наличия определенных лексем, формируется список идентификаторов документов, содержащих *хотя бы один* ключ из запроса. Для этого в дереве лексем отыскиваются идентификаторы, соответствующие каждому ключу поиска, и из них составляется один общий список. В индексе все идентификаторы хранятся упорядоченно, что дает возможность использовать слияние нескольких отсортированных потоков идентификаторов в один[2]. *с. 483*



Заметьте, что пока не имеет значения, были ли ключи объединены условием «и», «или» либо каким-то еще, — движок поиска по дереву работает со списком ключей и ничего не знает о смысле поискового запроса.

[1] backend/utils/adt/tsginidx.c, функция gin_extract_tsquery.
[2] backend/access/gin/ginget.c, функция keyGetItem.

Каждый найденный идентификатор, соответствующий документу, проверяется опорной функцией согласованности[1]. Эта функция как раз таки знает, как интерпретировать поисковый запрос, и оставляет только те идентификаторы, которые ему соответствуют (или по крайней мере могут соответствовать и нуждаются в перепроверке по таблице).

В данном случае функция согласованности оставит все идентификаторы:

tid	«стоя»	«кудряв»	функция согласованности
(0,1)	✓	–	✓
(0,2)	✓	✓	✓
(0,3)	✓	–	✓
(0,4)	✓	–	✓
(1,2)	–	✓	✓

Вместо обычной лексемы в поисковом запросе может быть указан префикс. Это полезно в случаях, когда пользователь приложения может ввести в поле поиска первые буквы слова, ожидая получить по ним результаты. Например, для запроса «`залом:*`» будут найдены документы с лексемами, начинающимися на «залом»: «залома» (которая получилась из слова «заломаю») и «заломат» (из слова «заломати»).

При таком *частичном* поиске проиндексированные лексемы сравниваются с ключом поиска с помощью отдельной опорной функции[2], которая в общем случае может реализовывать не только проверку совпадения префикса, но и другую логику частичного соответствия.

Частые и редкие лексемы

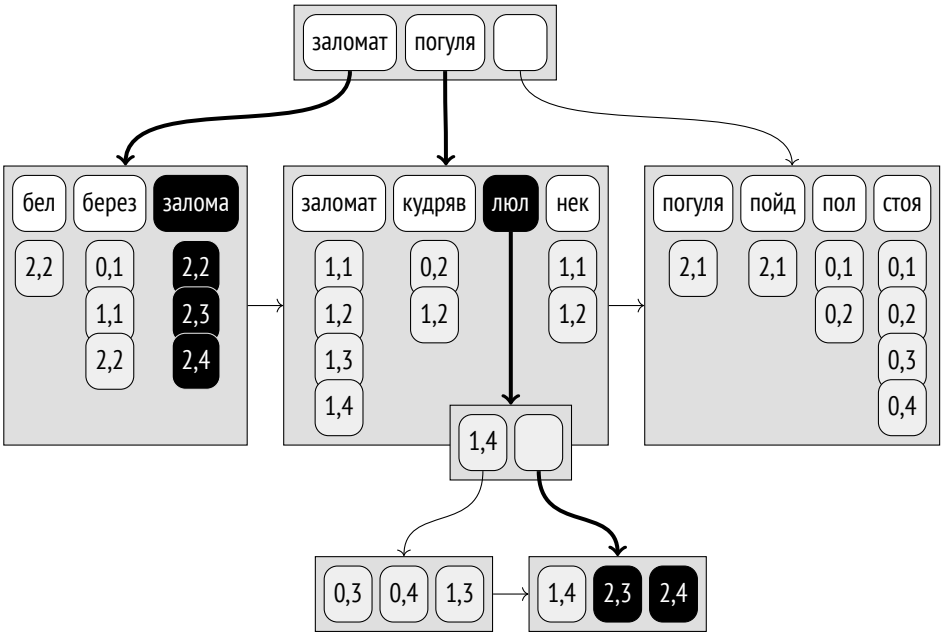
Если лексемы, использованные в поисковом запросе, встречаются часто, сформированный список идентификаторов версий окажется длинным, что, конечно, накладно. К счастью, этого зачастую можно избежать, если в запросе присутствуют и редкие лексемы.

[1] backend/utils/adt/tsginidx.c, функция gin_tsquery_triconsistent.
[2] backend/utils/adt/tsginidx.c, функция gin_cmp_prefix.

Рассмотрим выполнение запроса «люли & заломаю». Лексема «залома» встречается три раза, а «люл» — шесть. Вместо того чтобы считать обе лексемы равноправными и собирать полный перечень идентификаторов по обеим, редкая лексема «залома» считается обязательной, а более частая «люл» — дополнительной, поскольку (с учетом семантики запроса) понятно, что документ с лексемой «люл» будет соответствовать запросу, только если в нем содержится и лексема «залома».

Итак, сначала по индексу определяется первый документ, содержащий «залома»; он имеет идентификатор (2,2). Дальше требуется определить, не содержит ли этот же документ лексему «люл», но все документы с идентификаторами, меньшими (2,2), можно сразу пропустить. Поскольку частым лексемам соответствует много идентификаторов, они, скорее всего, хранятся в отдельном дереве, что позволяет пропустить часть страниц. В данном случае поиск в дереве лексем «люл» начинается с (2,2), и первый найденный документ имеет идентификатор (2,3).

Процедура повторяется для следующих значений обязательной лексемы.



Длина списка идентификаторов будет равна трем, по числу вхождений редкой лексемы:

tid	«залома»	«люл»	функция согласованности
(2,2)	✓	–	–
(2,3)	✓	✓	✓
(2,4)	✓	✓	✓

с. 360 Таким образом, знание частот позволяет организовать слияние наиболее эффективным способом, начиная с редких лексем и пропуская заведомо ненужные диапазоны страниц в деревьях частых лексем. Экономия происходит и на количестве вызовов функции согласованности.

Такая оптимизация работает, конечно, не только для двух лексем, но и в более сложных случаях. Обязательные лексемы, по которым происходит поиск в индексе, должны быть выбраны таким образом, чтобы не пропустить ни одного документа, соответствующего запросу. Алгоритм упорядочивает лексемы от редких к частым и затем по одной добавляет их к списку обязательных до тех пор, пока это условие не выполнится. Оставшиеся лексемы считаются дополнительными[1].

Например, возьмем запрос «`заломати & ( некому | белую )`». Самая редкая лексема — «бел» — сразу добавляется в список обязательных. Возможна ли ситуация, когда документ не содержит ни одной обязательной лексемы, но при этом соответствует запросу? Чтобы проверить это, в функцию согласованности передаются ложные значения для обязательных лексем и истинные — для остальных. В данном случае функция возвращает `true AND ( true OR false ) = true`, то есть часть документов может быть пропущена и поэтому набор обязательных лексем необходимо расширить как минимум одной из оставшихся.

В список обязательных добавляется следующая по частоте лексема — «нек», и теперь функция согласованности возвращает `true AND (false OR false) = false`. Таким образом, лексемы «бел» и «нек» становятся обязательными, а «заломат» — дополнительной.

[1] backend/access/gin/ginget.c, функция startScanKey.

Чтобы убедиться в эффективности этой оптимизации, используем архив рассылки pgsql-hackers. Нам понадобятся две лексемы, частая и редкая: с. 584

```
=> SELECT word, ndoc
FROM ts_stat('SELECT tsv FROM mail_messages')
WHERE word IN ('wrote', 'tattoo');
  word  |  ndoc
--------+--------
 wrote  | 231173
 tattoo |      2
(2 rows)
```

Оказывается, есть один документ, в котором встречаются оба этих слова:

```
=> \timing on
=> SELECT count(*) FROM mail_messages
WHERE tsv @@ to_tsquery('wrote & tattoo');
 count
-------
     1
(1 row)
Time: 0.219 ms
```

Запрос выполняется почти так же быстро, как и поиск одного слова «tattoo»:

```
=> SELECT count(*) FROM mail_messages
WHERE tsv @@ to_tsquery('tattoo');
 count
-------
     2
(1 row)
Time: 0.453 ms
```

А вот поиск одного слова «wrote» выполняется существенно дольше:

```
=> SELECT count(*) FROM mail_messages
WHERE tsv @@ to_tsquery('wrote');
 count
--------
 231173
(1 row)
Time: 82.090 ms
=> \timing off
```

Вставка

При добавлении элемента к GIN-индексу[1] существующие элементы не дублируются; вместо этого у имеющегося элемента расширяется набор идентификаторов (список или дерево). Список идентификаторов является частью индексной строки, которая не может занимать слишком много места на странице, поэтому при переполнении список преобразуется в дерево[2].

При вставке в дерево нового элемента (или нового идентификатора) может возникнуть переполнение страницы; в этом случае страница расщепляется на две, и элементы перераспределяются между ними[3].

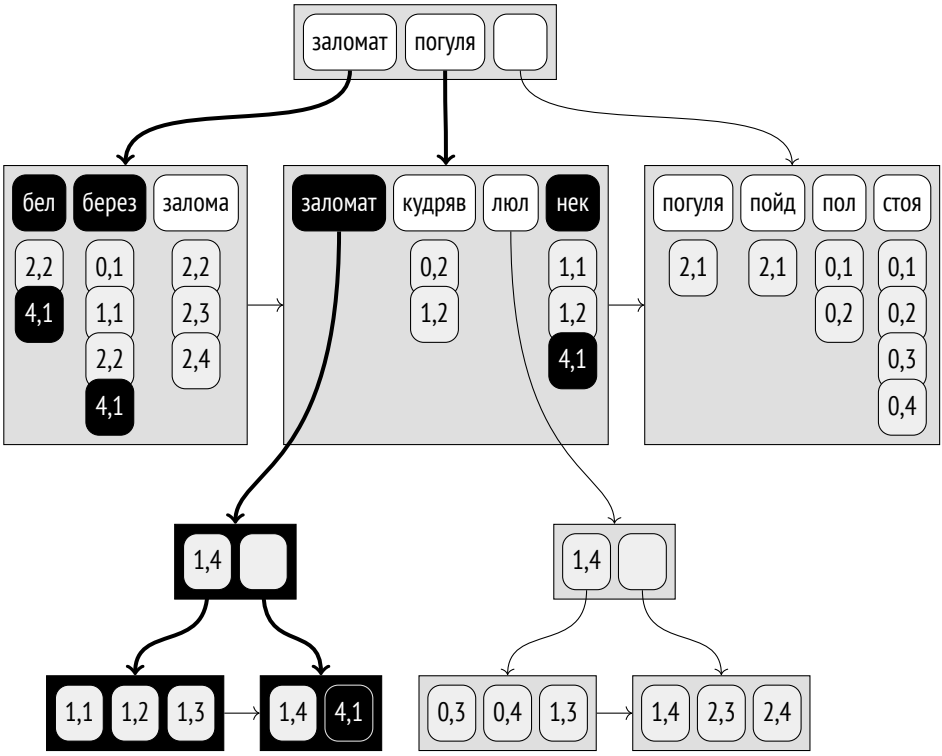
Но каждый документ обычно содержит много лексем, подлежащих индексированию. Поэтому при создании или изменении одного-единственного документа приходится вносить сразу множество изменений в дерево индекса. Из-за этого GIN-индекс обновляется относительно медленно.

На рисунке показано состояние дерева после добавления в таблицу строки «Некому заломати белую березу» с идентификатором (4,1). Список идентификаторов лексем «бел», «берез» и «нек» увеличился, а список лексемы «заломат» превысил максимальный размер и был выделен в отдельное дерево.

С другой стороны, если учитывать в индексе изменения сразу нескольких документов, то общий объем работы, вероятно, уменьшится по сравнению с последовательным изменением, поскольку часть лексем у документов, скорее всего, окажется общей.

Такой *отложенный* режим обновления включается параметром хранения
on *fastupdate*. Изменения в этом режиме накапливаются в отдельном неупоря-
с. 296 доченном списке, который физически хранится вне дерева элементов в отдельных страницах (list pages). Когда этот список становится достаточно большим, все накопленные изменения одномоментно вносятся в индекс, а список очищается[4]. Максимальный размер списка определяется парамет-
4MB ром *gin_pending_list_limit* или одноименным параметром хранения индекса.

[1] backend/access/gin/gininsert.c, функция ginEntryInsert.
[2] backend/access/gin/gininsert.c, функция addItemPointersToLeafTuple.
[3] backend/access/gin/ginbtree.c, функция ginInsertValue.
[4] backend/access/gin/ginfast.c, функция ginInsertCleanup.



По умолчанию отложенный режим включен, но следует иметь в виду, что он замедляет поиск: кроме дерева приходится просматривать и весь неупорядоченный список. К тому же время вставки становится менее предсказуемым, поскольку при очередном изменении список может переполниться, и потребуется выполнить дорогостоящую процедуру слияния. Последний момент отчасти нивелируется тем, что слияние выполняется в том числе и при очистке индекса, то есть асинхронно.

При создании нового индекса[1] элементы тоже добавляются не по одному, что было бы слишком медленно, а отложенно. Вместо неупорядоченного списка на диске изменения накапливаются в области оперативной памяти размером *maintenance_work_mem* и сбрасываются в индекс при переполнении. Чем больше памяти будет выделено под операцию, тем быстрее будет построен индекс.

64MB

[1] backend/access/gin/gininsert.c, функция ginbuild.

Приведенные в этом разделе примеры позволяют убедиться в превосходстве
с. 578 GIN по точности поиска над сигнатурным деревом GiST. Поэтому в большинстве случаев именно GIN используется для поддержки полнотекстового поиска. Однако проблема медленного обновления GIN может повлиять на выбор индекса в пользу GiST, если данные активно изменяются.

Ограничение выборки

Метод доступа GIN всегда возвращает результат в виде битовой карты; получать табличные идентификаторы по одному невозможно. Иными словами,
с. 409 поддерживается свойство BITMAP SCAN, но не INDEX SCAN.

Причина этого — в наличии неупорядоченного списка отложенных изменений. При индексном доступе сначала просматривается список и формируется битовая карта, а затем битовая карта обновляется данными из дерева. Если в процессе поиска неупорядоченный список будет объединен с деревом (в результате обновления индекса или в процессе очистки), одно и то же значение может быть получено дважды, что недопустимо. Но в случае битовой карты это не представляет проблемы: один и тот же бит просто будет установлен два раза.

Как следствие, ограничение выборки по GIN-индексу с помощью предложения LIMIT не вполне эффективно, поскольку битовая карта в любом случае строится полностью, и ее построение может составлять существенную часть общей стоимости:

```
=> EXPLAIN SELECT * FROM mail_messages
WHERE tsv @@ to_tsquery('hacker')
LIMIT 1000;
                          QUERY PLAN
---------------------------------------------------------------
 Limit  (cost=343.76..1869.59 rows=1000 width=1256)
   -> Bitmap Heap Scan on mail_messages
       (cost=343.76..73593.12 rows=48006 width=1256)
       Recheck Cond: (tsv @@ to_tsquery('hacker'::text))
       -> Bitmap Index Scan on mail_gin_idx
           (cost=0.00..331.76 rows=48006 width=0)
           Index Cond: (tsv @@ to_tsquery('hacker'::text))
(7 rows)
```

Поэтому метод GIN имеет специальную возможность ограничить количество результатов, получаемых из индекса. Ограничение задается параметром *gin_fuzzy_search_limit* и по умолчанию не действует. Если его установить, индексный метод будет случайным образом пропускать некоторые значения, чтобы получить *примерно* указанное количество строк [1] (потому и fuzzy):

```
=> SET gin_fuzzy_search_limit = 1000;

=> SELECT count(*)
FROM mail_messages
WHERE tsv @@ to_tsquery('hacker');
 count
-------
   766
(1 row)

=> SELECT count(*)
FROM mail_messages
WHERE tsv @@ to_tsquery('hacker');
 count
-------
   767
(1 row)

=> RESET gin_fuzzy_search_limit;
```

Обратите внимание на отсутствие предложения LIMIT в запросах. Это единственный «легализованный» способ получить разные данные при индексном и при табличном доступе. Планировщик ничего не знает об этой особенности GIN-индексов и никак не учитывает значение параметра при расчете стоимости.

Свойства

Все свойства метода доступа gin на всех уровнях одинаковые и не зависят от конкретного класса операторов.

[1] backend/access/gin/ginget.c, макрос dropItem.

Свойства уровня метода доступа

```
=> SELECT a.amname, p.name, pg_indexam_has_property(a.oid, p.name)
FROM pg_am a, unnest(array[
  'can_order', 'can_unique', 'can_multi_col',
  'can_exclude', 'can_include'
]) p(name)
WHERE a.amname = 'gin';
 amname |     name      | pg_indexam_has_property
--------+---------------+-------------------------
 gin    | can_order     | f
 gin    | can_unique    | f
 gin    | can_multi_col | t
 gin    | can_exclude   | f
 gin    | can_include   | f
(5 rows)
```

GIN не обеспечивает ни сортировку, ни уникальность значений.

Создание составных индексов поддерживается. Интересно, что для GIN порядок столбцов в индексе не имеет никакого значения. В отличие от обычного B-дерева, многоколоночный индекс GIN хранит не составные ключи, а отдельные элементы, дополненные указанием номера столбца.

Ограничение исключения не поддерживается, так как нет свойства INDEX SCAN.

Дополнительные include-столбцы для GIN не реализованы. В них и нет большого смысла, поскольку GIN-индекс вряд ли получится использовать как покрывающий: индексируемое значение хранится в таблице, а индекс содержит только элементы этого значения.

Свойства уровня индекса

```
=>  SELECT p.name, pg_index_has_property('mail_gin_idx', p.name)
FROM unnest(array[
  'clusterable', 'index_scan', 'bitmap_scan', 'backward_scan'
]) p(name);
     name      | pg_index_has_property
---------------+-----------------------
 clusterable   | f
 index_scan    | f
 bitmap_scan   | t
 backward_scan | f
(4 rows)
```

Выдача результатов по одному не поддерживается, индексный доступ всегда возвращает битовую карту результатов.

Поэтому и нет смысла в упорядочивании таблицы по GIN-индексу: битовая карта по определению соответствует физическому расположению данных в таблице, какое бы оно ни было.

Сканирование GIN-индекса в обратную сторону не поддерживается: эта возможность актуальна для индексного сканирования, но не для сканирования по битовой карте.

Свойства уровня столбца

```
=> SELECT p.name,
  pg_index_column_has_property('mail_gin_idx', 1, p.name)
FROM unnest(array[
  'orderable', 'search_array', 'search_nulls',
  'returnable', 'distance_orderable'
]) p(name);
        name        | pg_index_column_has_property
--------------------+------------------------------
 orderable          | f
 search_array       | f
 search_nulls       | f
 returnable         | f
 distance_orderable | f
(5 rows)
```

Ни одно из свойств уровня столбца не доступно: ни сортировка (что понятно), ни использование индекса в качестве покрывающего (поскольку сам документ не хранится в индексе), ни работа с неопределенными значениями (не имеет смысла для элементов неатомарных типов).

Ограничения GIN и RUM-индекс

При всем своем могуществе GIN не решает всех проблем полнотекстового поиска. В типе данных `tsvector` присутствуют позиции лексем, но в индекс они не попадают. Это не позволяет эффективно использовать GIN для ускорения *фразового поиска*, при котором учитывается близость лексем. Кроме

того, поисковые системы обычно возвращают результаты *в порядке релевантности* (что бы ни означал этот термин). Поскольку GIN не поддерживает упорядочивающие операторы, единственный выход — вычислять функцию ранжирования для каждой строки результата, что, конечно, медленно.

Эти недостатки устранены в индексном методе RUM, название которого заставляет усомниться в искренности разработчиков относительно смысла, вкладываемого в аббревиатуру GIN. Это сторонний метод доступа; расширение доступно и как пакет в репозитории PGDG[1], и в исходных кодах[2].

RUM создан на основе GIN и отличается от него двумя принципиальными свойствами. Во-первых, в нем отказались от отложенного режима обновления; за счет этого появилась поддержка не только сканирования по битовой карте, но и обычного индексного сканирования, а также были реализованы упорядочивающие операторы. Во-вторых, к ключам индекса можно добавлять вспомогательную информацию. Это отчасти похоже на дополнительные include-столбцы, но информация привязывается к конкретному ключу. В контексте полнотекстового поиска класс операторов RUM связывает вхождения лексем с их позициями в документе, что позволяет ускорять фразовый поиск и выдачу результатов в порядке ранжирования.

Недостатком такого подхода является медленное обновление и увеличенный объем индекса. Кроме того, являясь расширением, индексный метод `rum` пользуется механизмом унифицированных журнальных записей[3], который работает медленнее встроенного в ядро журналирования и генерирует больший объем WAL.

28.3. Индекс для триграмм

Расширение `pg_trgm`[4] позволяет определять схожесть слов, сравнивая количество совпадающих последовательностей из трех букв (*триграмм*). Его

[1] postgresql.org/download.
[2] github.com/postgrespro/rum.
[3] postgrespro.ru/docs/postgresql/18/generic-wal.
[4] postgrespro.ru/docs/postgresql/18/pgtrgm.

можно использовать вместе с полнотекстовым поиском, чтобы находить варианты даже при вводе слов с опечатками.

Класс операторов `gin_trgm_ops` реализует индексирование текстовых строк, но в качестве элементов текстовых значений выделяются не слова или лексемы, а всевозможные трехсимвольные подстроки (используются только буквы и цифры, остальные символы игнорируются). Внутри индекса триграммы представляются целыми числами: для русских букв, занимающих два байта в кодировке UTF-8, такое кодирование не позволяет расшифровать исходные символы. Поэтому я покажу, как формируются триграммы, в базе данных с восьмибитной кодировкой:

```
=> CREATE DATABASE eightbit
WITH ENCODING 'KOI8R'
LC_CTYPE='ru_RU.koi8r'
LC_COLLATE='ru_RU.koi8r'
TEMPLATE=template0; -- кодировки базы и кластера не совпадают
```

```
=> \c eightbit
=> CREATE EXTENSION pg_trgm;
=> SET client_encoding = 'UTF8';
=> SELECT unnest(show_trgm('заломаю')),
          unnest(show_trgm('заломати'));
 unnest | unnest
--------+--------
 аю     | ало
 ало    | ати
 лом    | лом
 маю    | мат
 ома    | ома
 зал    | ти
  за    | зал
   з    |  за
        |   з
(9 rows)
```

Класс операторов поддерживает операторы точного и нечеткого сравнения строк и слов.

```
=> CREATE EXTENSION pg_trgm;
```

```
\dAo   => SELECT amopopr::regoperator, oprcode::regproc
       FROM pg_am am
         JOIN pg_opclass opc ON opcmethod = am.oid
         JOIN pg_amop amop ON amopfamily = opcfamily
         JOIN pg_operator opr ON opr.oid = amopopr
       WHERE amname = 'gin'
       AND opcname = 'gin_trgm_ops'
       ORDER BY amopstrategy;
           amopopr     |               oprcode
-----------------------+-------------------------------------
 %(text,text)          | similarity_op
 ~~(text,text)         | textlike          } LIKE и ILIKE
 ~~*(text,text)        | texticlike        }
 ~(text,text)          | textregexeq       } регулярные выражения
 ~*(text,text)         | texticregexeq     }
 %>(text,text)         | word_similarity_commutator_op
 %>>(text,text)        | strict_word_similarity_commutator_op
 =(text,text)          | texteq
(8 rows)
```

Для нечеткого сравнения можно определить расстояние между строками как отношение числа совпадающих триграмм к их общему количеству в строке-запросе. Но, как я уже показывал, индекс GIN не поддерживает упорядочивающие операторы, так что все операторы в классе должны быть булевыми. Поэтому для операторов `%`, `%>` и `%>>`, реализующих стратегии нечеткого сравнения, функция согласованности вычисляет расстояние и дает положительный вердикт, если оно не превышает установленный порог.

Для операторов `=` и `LIKE` функция согласованности требует, чтобы в значении присутствовали все триграммы из строки-запроса. Соответствие регулярному выражению проверяется существенно более сложно.

В любом случае поиск по триграммам является неточным и всегда перепроверяется по таблице.

28.4. Индекс для массивов

Другой пример подходящего типа данных — массив. GIN-индекс по элементам массива позволяет быстрее находить пересечения массивов и вхождения одного массива в другой:

```
=> SELECT amopopr::regoperator, oprcode::regproc, amopstrategy          \dAo
FROM pg_am am
  JOIN pg_opclass opc ON opcmethod = am.oid
  JOIN pg_amop amop ON amopfamily = opcfamily
  JOIN pg_operator opr ON opr.oid = amopopr
WHERE amname = 'gin'
AND opcname = 'array_ops'
ORDER BY amopstrategy;
        amopopr        |    oprcode     | amopstrategy
-----------------------+----------------+--------------
 &&(anyarray,anyarray) | arrayoverlap   |            1
 @>(anyarray,anyarray) | arraycontains  |            2
 <@(anyarray,anyarray) | arraycontained |            3
 =(anyarray,anyarray)  | array_eq       |            4
(4 rows)
```

В качестве примера возьмем представление routes демобазы с информацией о рейсах. Столбец days_of_week хранит массив номеров дней недели, по которым выполняются рейсы. Конечно, чтобы построить индекс, представление придется «материализовать»:

```
=> CREATE TABLE routes_tbl
  AS SELECT * FROM routes;
SELECT 710
=> CREATE INDEX ON routes_tbl USING gin(days_of_week);
```

С помощью созданного индекса можно, например, отобрать рейсы, отправляющиеся по вторникам, четвергам и воскресеньям. Я отключаю последовательное сканирование, поскольку иначе планировщик не будет использовать индекс для такой небольшой таблицы:

```
=> SET enable_seqscan = off;
=> EXPLAIN (costs off) SELECT *
FROM routes_tbl
WHERE days_of_week = ARRAY[2,4,7];
                        QUERY PLAN
-----------------------------------------------------------
 Bitmap Heap Scan on routes_tbl
   Recheck Cond: (days_of_week = '{2,4,7}'::integer[])
   -> Bitmap Index Scan on routes_tbl_days_of_week_idx
        Index Cond: (days_of_week = '{2,4,7}'::integer[])
(4 rows)
```

Оказывается, есть одиннадцать таких рейсов:

```
=> SELECT flight_no, departure_airport_name AS departure,
  arrival_airport_name AS arrival, days_of_week
FROM routes_tbl
WHERE days_of_week = ARRAY[2,4,7];
 flight_no |  departure  |     arrival     | days_of_week
-----------+-------------+-----------------+--------------
 PG0023    | Орск        | Курган          | {2,4,7}
 PG0123    | Бегишево    | Ростов-на-Дону  | {2,4,7}
 ...
 PG0482    | Домодедово  | Кемерово        | {2,4,7}
 PG0651    | Усть-Илимск | Хабаровск-Новый | {2,4,7}
(11 rows)
```

Построенный индекс содержит всего семь элементов: целые числа от 1 до 7, представляющие дни недели.

Запрос выполняется примерно так же, как я показывал выше для полнотекстового поиска. Поисковый запрос в данном случае представлен не специальным типом данных, а обычным массивом; такой запрос подразумевает, что в индексированном массиве должны встретиться все перечисленные элементы. Важное отличие состоит в том, что условие *равенства* требует также, чтобы индексированный массив не содержал никаких других элементов. Функция согласованности[1] знает об этой особенности благодаря номеру стратегии, но не может удостовериться в отсутствии лишних элементов и поэтому просит механизм индексирования перепроверить результаты:

```
=> EXPLAIN (analyze, costs off, timing off, summary off, buffers off)
SELECT * FROM routes_tbl
WHERE days_of_week = ARRAY[2,4,7];
                            QUERY PLAN
------------------------------------------------------------------
 Bitmap Heap Scan on routes_tbl (actual rows=11.00 loops=1)
   Recheck Cond: (days_of_week = '{2,4,7}'::integer[])
   Rows Removed by Index Recheck: 482
   Heap Blocks: exact=17
   -> Bitmap Index Scan on routes_tbl_days_of_week_idx (actual ro...
       Index Cond: (days_of_week = '{2,4,7}'::integer[])
       Index Searches: 1
(7 rows)
```

[1] backend/access/gin/ginarrayproc.c, функция ginarrayconsistent.

Может оказаться полезным расширить GIN-индекс, добавив в него другие столбцы. Например, для поиска рейсов, отправляющихся по вторникам, четвергам и воскресеньям *из Москвы*, в индексе не хватает столбца departure_city. Однако для обычных скалярных типов данных классы операторов не предусмотрены:

```
=> CREATE INDEX ON routes_tbl USING gin(days_of_week, departure_city);
ERROR:  data type text has no default operator class for access method
"gin"
HINT:  You must specify an operator class for the index or define a
default operator class for the data type.
```

В таких случаях помогает расширение btree_gin. Оно добавляет классы операторов GIN, имитирующие работу обычного B-дерева, представляя скалярное значение как составное, но содержащее один элемент. *с. 573*

```
=> CREATE EXTENSION btree_gin;
=> CREATE INDEX ON routes_tbl USING gin(days_of_week,departure_city);
=> EXPLAIN (costs off)
SELECT * FROM routes_tbl
WHERE days_of_week = ARRAY[2,4,7]
AND departure_city = 'Москва';
                            QUERY PLAN
------------------------------------------------------------------
 Bitmap Heap Scan on routes_tbl
   Recheck Cond: ((days_of_week = '{2,4,7}'::integer[]) AND
   (departure_city = 'Москва'::text))
   -> Bitmap Index Scan on routes_tbl_days_of_week_departure_city...
       Index Cond: ((days_of_week = '{2,4,7}'::integer[]) AND
       (departure_city = 'Москва'::text))
(6 rows)
=> RESET enable_seqscan;
```

По-прежнему остается справедливым замечание, которое я уже делал в отношении расширения btree_gist: B-дерево гораздо эффективнее справляется с поддержкой операций сравнения, так что использовать расширение имеет смысл лишь в тех случаях, когда GIN-индекс действительно необходим. Например, поиск по условиям «меньше» или «меньше или равно» выполняется в B-дереве сканированием листовых страниц в обратную сторону, а в GIN такая возможность отсутствует.

28.5. Индекс для JSON

Еще один неатомарный тип данных, для которого есть встроенная GIN-поддержка, — `jsonb`[1]. Для работы с JSON предусмотрен целый ряд операторов, часть из которых может быть ускорена с помощью GIN-индекса.

Существует два класса операторов, которые выделяют разные наборы элементов из документа JSON:

```
=> SELECT opcname
FROM pg_am am
  JOIN pg_opclass opc ON opcmethod = am.oid
WHERE amname = 'gin'
AND opcintype = 'jsonb'::regtype;
    opcname
----------------
 jsonb_ops
 jsonb_path_ops
(2 rows)
```

Класс операторов jsonb_ops

Первый класс операторов, `jsonb_ops`, используется по умолчанию. В индекс в качестве элементов попадают все ключи, все значения и все элементы массивов исходного документа JSON[2]. Это позволяет ускорять запросы с условиями включения JSON-значения (`@>`), существования ключей (`?`, `?|` и `?&`) и сопоставления путей JSON (`@?` и `@@`):

```
\dAo   => SELECT amopopr::regoperator, oprcode::regproc, amopstrategy
       FROM pg_am am
         JOIN pg_opclass opc ON opcmethod = am.oid
         JOIN pg_amop amop ON amopfamily = opcfamily
         JOIN pg_operator opr ON opr.oid = amopopr
       WHERE amname = 'gin'
       AND opcname = 'jsonb_ops'
       ORDER BY amopstrategy;
```

[1] postgrespro.ru/docs/postgresql/18/datatype-json.

[2] backend/utils/adt/jsonb_gin.c, функция gin_extract_jsonb.

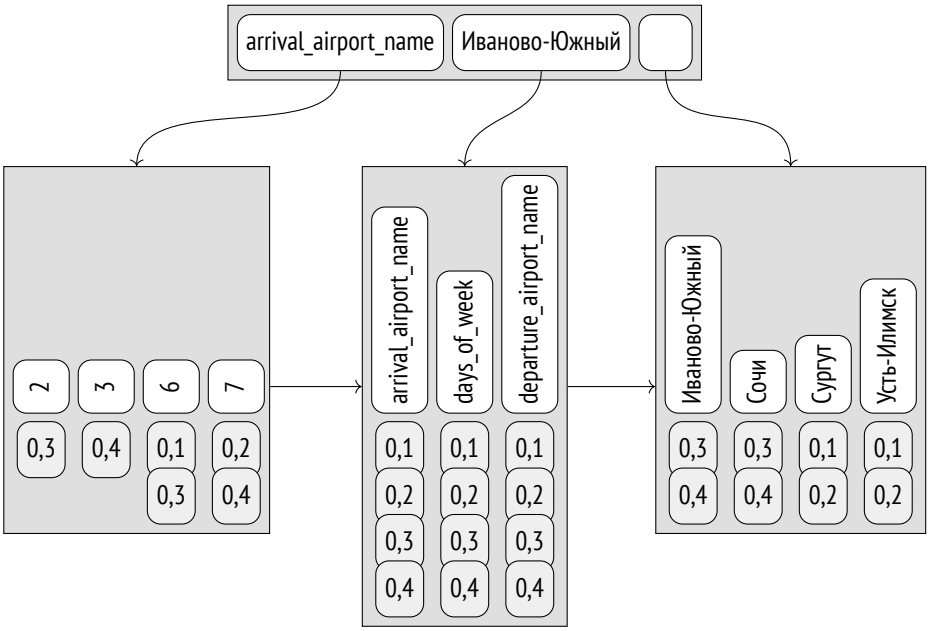
```
      amopopr       |        oprcode        | amopstrategy
--------------------+-----------------------+--------------
 @>(jsonb,jsonb)    | jsonb_contains        |            7
 ?(jsonb,text)      | jsonb_exists          |            9
 ?|(jsonb,text[])   | jsonb_exists_any      |           10
 ?&(jsonb,text[])   | jsonb_exists_all      |           11
 @?(jsonb,jsonpath) | jsonb_path_exists_opr |           15
 @@(jsonb,jsonpath) | jsonb_path_match_opr  |           16
(6 rows)
```

Для примера представим несколько строк из routes в виде JSON:

```
=> CREATE TABLE routes_jsonb AS
SELECT to_jsonb(t) route
FROM (
  SELECT departure_airport_name, arrival_airport_name, days_of_week
  FROM routes
  ORDER BY flight_no
  LIMIT 4
) t;
=> CREATE INDEX ON routes_jsonb USING gin(route);
=> SELECT ctid, jsonb_pretty(route) FROM routes_jsonb;
 ctid  |                jsonb_pretty
-------+---------------------------------------------
 (0,1) | {                                           +
       |     "days_of_week": [                       +
       |         6                                   +
       |     ],                                      +
       |     "arrival_airport_name": "Сургут",       +
       |     "departure_airport_name": "Усть-Илимск" +
       | }
 (0,2) | {                                           +
       |     "days_of_week": [                       +
       |         7                                   +
       |     ],                                      +
       |     "arrival_airport_name": "Усть-Илимск",  +
       |     "departure_airport_name": "Сургут"      +
       | }
 (0,3) | {                                           +
       |     "days_of_week": [                       +
       |         2,                                  +
       |         6                                   +
       |     ],                                      +
       |     "arrival_airport_name": "Сочи",         +
       |     "departure_airport_name": "Иваново-Южный"+
       | }
```

```
 (0,4) | {                                                +
       |     "days_of_week": [                            +
       |         3,                                       +
       |         7                                        +
       |     ],                                           +
       |     "arrival_airport_name": "Иваново-Южный",     +
       |     "departure_airport_name": "Сочи"             +
       | }
(4 rows)
```

Созданный индекс можно представить следующим образом:



Рассмотрим запрос по условию `route @> '{"days_of_week": [6]}'`, которое отбирает документы JSON, содержащие указанный путь (то есть рейсы, совершаемые по субботам).

Из поискового запроса, представленного значением JSON, опорная функция[1] выделяет ключи поиска: «days_of_week» и «6». Эти ключи ищутся в дереве элементов, и документы, содержащие по крайней мере один ключ, проверяются функцией согласованности[2]. Для стратегии «содержит» функция

[1] backend/utils/adt/jsonb_gin.c, функция gin_extract_jsonb_query.
[2] backend/utils/adt/jsonb_gin.c, функция gin_consistent_jsonb.

требует наличия *всех* ключей из поискового запроса, но результаты все равно должны перепроверяться по таблице, поскольку с точки зрения индекса указанный путь соответствует, например, документу `{"days_of_week": [2], "foo": [6]}`.

Класс операторов jsonb_path_ops

Второй класс `jsonb_path_ops` содержит меньше операторов:

```
=> SELECT amopopr::regoperator, oprcode::regproc, amopstrategy            \dAo
FROM pg_am am
  JOIN pg_opclass opc ON opcmethod = am.oid
  JOIN pg_amop amop ON amopfamily = opcfamily
  JOIN pg_operator opr ON opr.oid = amopopr
WHERE amname = 'gin'
AND opcname = 'jsonb_path_ops'
ORDER BY amopstrategy;
      amopopr       |        oprcode        | amopstrategy
--------------------+-----------------------+--------------
 @>(jsonb,jsonb)    | jsonb_contains        |            7
 @?(jsonb,jsonpath) | jsonb_path_exists_opr |           15
 @@(jsonb,jsonpath) | jsonb_path_match_opr  |           16
(3 rows)
```

В этом случае в индекс попадают не разрозненные фрагменты JSON, а пути от корня документа до всех значений и всех элементов массивов[1]. Это делает поиск гораздо более точным и эффективным, хотя и не позволяет ускорять операции, аргументами которых являются не пути, а отдельные ключи.

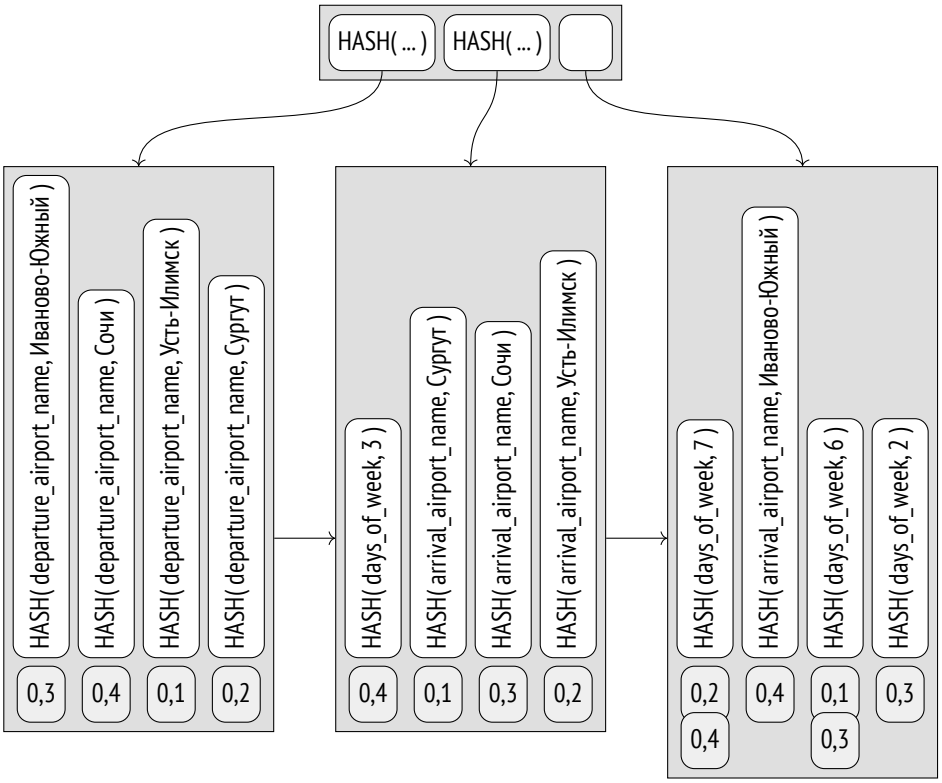
Поскольку путь может быть достаточно длинным, фактически индексируются не сами пути, а значения хеш-функций от них.

Создадим индекс с этим классом операторов для той же таблицы:

```
=> CREATE INDEX ON routes_jsonb USING gin(route jsonb_path_ops);
```

[1] backend/utils/adt/jsonb_gin.c, функция gin_extract_jsonb_path.

Вот как можно представить дерево созданного индекса:



При выполнении запроса с тем же условием `route @> '{"days_of_week": [6]}'` опорная функция[1] выделит из поискового запроса не отдельные фрагменты, а весь путь «days_of_week, 6». В дереве элементов сразу же будут найдены идентификаторы двух подходящих документов.

Конечно, и в этом случае найденные записи будут проверены функцией согласованности[2] и затем перепроверены по таблице механизмом индексирования (как минимум из-за возможных хеш-коллизий). Но поиск по дереву выполняется намного эффективнее, так что стоит выбирать класс `jsonb_path_ops` всегда, когда входящих в него операторов достаточно для индексной поддержки запросов.

[1] backend/utils/adt/jsonb_gin.c, функция gin_extract_jsonb_query_path.
[2] backend/utils/adt/jsonb_gin.c, функция gin_consistent_jsonb_path.

28.6. Другие типы данных

Вот еще несколько расширений, добавляющих поддержку GIN для некоторых типов данных.

Целочисленные массивы. Расширение `intarray` добавляет класс операторов `gin__int_ops` для целочисленных массивов. Он практически идентичен стандартному классу операторов `array_ops`, но поддерживает оператор сопоставления с поисковым запросом `@@`.

Хранилище «ключ–значение». Расширение `hstore` реализует хранилище пар «ключ–значение» и предлагает класс операторов `gin_hstore_ops`. Индексируются и ключи, и значения.

Язык запросов JSON. Стороннее расширение `jsquery` предоставляет собственный язык запросов и поддержку GIN-индексов для JSON.

С принятием стандарта SQL:2016 и реализацией в PostgreSQL языка путей SQL/JSON стандартные возможности выглядят более предпочтительными. v. 12

29

Индекс BRIN

29.1. Общий принцип

В отличие от остальных индексов, BRIN[1] предназначен не для быстрого поиска нужных строк, а для того, чтобы избежать просмотра заведомо ненужных. Этот метод доступа создавался в расчете на таблицы размером в единицы и десятки терабайт, так что меньшему объему отдается предпочтение перед точностью поиска.

Для ускорения поиска вся таблица разбивается на *зоны* (range) размером в несколько страниц. Отсюда и название: Block Range Index, BRIN. Индекс не хранит идентификаторы версий, а ограничивается лишь сводной информацией о данных в каждой зоне. Для порядковых типов данных в простом случае это минимальное и максимальное значения, но разные классы операторов могут собирать разную информацию о значениях в зоне.

Количество страниц в зоне устанавливается при создании индекса параметром хранения *pages_per_range*.

128

При выполнении запроса, содержащего условие на проиндексированный столбец, можно целиком пропустить все зоны, значения в которых гарантированно не попадают под условие запроса. Все страницы остальных зон возвращаются индексом в виде *неточной битовой карты*; все строки на этих страницах проверяются на выполнение условия.

с. 428

Таким образом, BRIN хорошо работает для столбцов, значения в которых локализованы, то есть когда значения, хранящиеся рядом, имеют схожие

[1] postgrespro.ru/docs/postgresql/18/brin;
backend/access/brin/README.

(в смысле сводной информации) свойства. Для порядковых типов данных это означает, что значения должны быть физически расположены в таблице в порядке возрастания или убывания, то есть иметь высокую *корреляцию* *с. 361* между физическим расположением и логическим порядком, определяемым операциями «больше» и «меньше». Для других типов сводной информации требования к «схожим свойствам» могут отличаться.

Не будет ошибкой рассматривать BRIN не как индекс в обычном понимании, а как ускоритель последовательного сканирования таблицы. Можно посмотреть на него и как на аналог секционирования, если каждую зону считать отдельной «виртуальной» секцией.

29.2. Пример

В демонстрационной базе данных нет достаточно больших для BRIN таблиц, но можно представить, что для нужд аналитической отчетности требуется денормализованная таблица с информацией о вылетевших из аэропорта и приземлившихся в аэропорту рейсах с точностью до места в салоне. Данные по каждому аэропорту добавляются в таблицу раз в сутки, как только в соответствующем часовом поясе наступает полночь. После добавления данные не изменяются и не удаляются.

Таблица имеет следующий вид:

```
CREATE TABLE flights_bi(
  airport_code char(3),         -- код аэропорта
  airport_coord point,          -- координаты аэропорта
  airport_utc_offset interval, -- часовой пояс
  flight_no char(6),            -- номер рейса
  flight_type text,             -- тип рейса: вылет или прилет
  scheduled_time timestamptz,  -- вылет/прилет по расписанию
  actual_time timestamptz,      -- реальное время
  aircraft_code char(3),        -- код воздушного судна
  seat_no varchar(4),           -- номер места
  fare_conditions varchar(10), -- класс обслуживания
  passenger_id varchar(20),     -- номер документа пассажира
  passenger_name text           -- имя пассажира
);
```

Процедуру загрузки данных можно имитировать вложенными циклами[1]: внешний — по дням (в демобазе хранятся данные за год), внутренний — по часовым поясам. В результате такого заполнения данные в таблице окажутся более или менее упорядочены как минимум по времени и по географии аэропортов, несмотря на отсутствие сортировки в запросе внутри цикла.

Я сразу загружу готовую копию таблицы, занимающую примерно 4 Гбайта и содержащую около 30 млн строк[2]:

```
postgres$ pg_restore -d demo -c flights_bi.dump
```

```
=> ANALYZE flights_bi;
=> SELECT count(*) FROM flights_bi;
  count
----------
 30517076
(1 row)
=> SELECT pg_size_pretty(pg_total_relation_size('flights_bi'));
 pg_size_pretty
----------------
 4130 MB
(1 row)
```

Такой объем данных сложно назвать большим, но его будет достаточно, чтобы изучить работу метода BRIN. Один индекс я создам заранее:

```
=> CREATE INDEX ON flights_bi USING brin(scheduled_time);
=> SELECT pg_size_pretty(pg_total_relation_size(
  'flights_bi_scheduled_time_idx'
));
 pg_size_pretty
----------------
 184 kB
(1 row)
```

С параметрами по умолчанию он занимает совсем мало места.

v. 13 Индекс на основе B-дерева, даже с учетом компактного хранения дубликатов, оказывается в тысячу раз больше. Конечно, и эффективность такого

[1] edu.postgrespro.ru/internals-18/flights_bi.sql.
[2] edu.postgrespro.ru/internals-18/flights_bi.dump.

индекса существенно выше, но для действительно больших таблиц дополнительный объем может оказаться непозволительной роскошью.

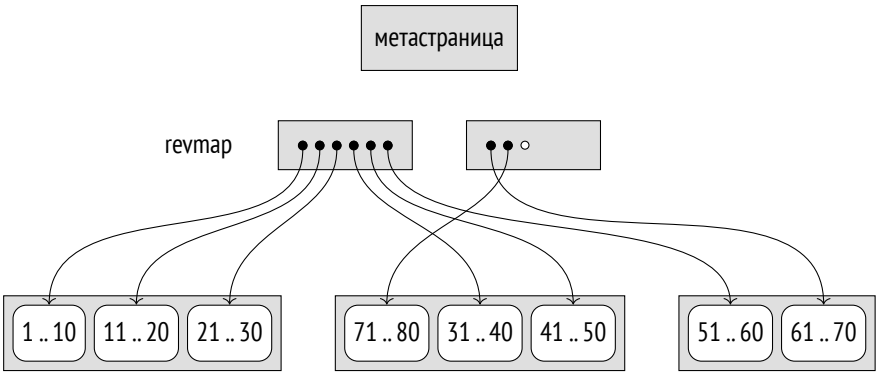
```
=> CREATE INDEX flights_bi_btree_idx ON flights_bi(scheduled_time);
=> SELECT pg_size_pretty(pg_total_relation_size(
  'flights_bi_btree_idx'
));
 pg_size_pretty
----------------
 210 MB
(1 row)
=> DROP INDEX flights_bi_btree_idx;
```

29.3. Страничная организация

Нулевая страница BRIN-индекса — *метастраница* с информацией о структуре индекса.

С некоторым отступом от метаданных находятся страницы со *сводной информацией*. Каждая индексная строка на такой странице содержит сводку по какой-то одной зоне.

Между метастраницей и сводными данными располагается *карта зон* (range map), которая иногда называется *обратной* картой (reverse range map, отсюда принятое сокращение revmap). По сути, это массив указателей на соответствующие индексные строки; номер позиции в массиве равен номеру зоны.



При расширении таблицы размер карты зон увеличивается. Если карта перестает помещаться в отведенные ей страницы, она захватывает следующую, а все индексные строки, которые там были, переносятся на другие страницы. Поскольку на странице помещается много указателей, такие перестановки случаются редко.

Страницы BRIN-индекса можно изучать, как обычно, с помощью расширения `pageinspect`. Метаинформация показывает размер зоны и количество страниц, отведенных под карту зон:

```
=> SELECT pagesperrange, lastrevmappage
FROM brin_metapage_info(get_raw_page(
  'flights_bi_scheduled_time_idx', 0
));
 pagesperrange | lastrevmappage
---------------+----------------
           128 |              4
(1 row)
```

Здесь карта зон занимает четыре страницы, с первой по четвертую. Можно получить ссылки на индексные записи о сводных данных:

```
=> SELECT *
FROM brin_revmap_data(get_raw_page(
  'flights_bi_scheduled_time_idx', 1
));
  pages
----------
 (6,197)
 (6,198)
 (6,199)
 (6,200)
 (6,201)
 ...
 (6,193)
 (6,194)
 (6,195)
 (6,196)
(1360 rows)
```

Если зона еще не обобщена (то есть для нее нет сводной информации), указатель в карте зон будет пустым.

Вот и сами сводные данные для нескольких первых зон:

```
=> SELECT itemoffset, blknum, value
FROM brin_page_items(
  get_raw_page('flights_bi_scheduled_time_idx', 6),
  'flights_bi_scheduled_time_idx'
)
ORDER BY blknum
LIMIT 3 \gx
-[ RECORD 1 ]----------------------------------------------------
itemoffset | 197
blknum     | 0
value      | {2016-08-15 02:45:00+03 .. 2016-08-15 16:20:00+03}
-[ RECORD 2 ]----------------------------------------------------
itemoffset | 198
blknum     | 128
value      | {2016-08-15 05:50:00+03 .. 2016-08-15 18:55:00+03}
-[ RECORD 3 ]----------------------------------------------------
itemoffset | 199
blknum     | 256
value      | {2016-08-15 07:15:00+03 .. 2016-08-15 18:50:00+03}
```

29.4. Поиск

При поиске по условию, которое поддерживается индексом BRIN[1], просматривается карта зон и сводная информация по каждой зоне. Если данные в зоне могут соответствовать ключу поиска, все страницы зоны добавляются к битовой карте. Поскольку индекс не хранит идентификаторы отдельных строк, битовая карта изначально строится с точностью до страниц.

Соответствие данных ключу определяется с помощью опорной *функции согласованности*, которая понимает смысл сводной информации. Конечно, зоны без сводной информации всегда добавляются к битовой карте.

Получившаяся в итоге битовая карта используется для сканирования таблицы обычным образом. Важно, что табличные страницы читаются последовательными фрагментами и используется предвыборка. *с. 427*

[1] backend/access/brin/brin.c, функция bringetbitmap.

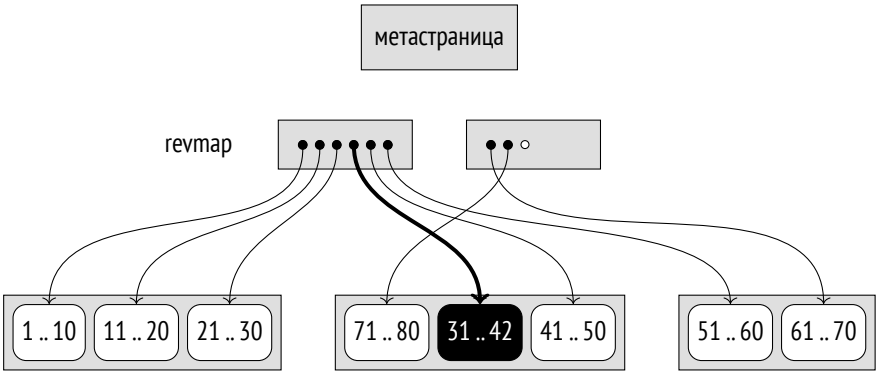
29.5. Обновление сводной информации

Вставка значений

При добавлении новой версии строки в табличную страницу обновляется сводная информация в соответствующей зоне индекса[1]. Номер зоны вычисляется по номеру страницы простыми арифметическими действиями, и с помощью карты зон читается сводная информация.

Нужно ли расширять имеющуюся сводную информацию, решает опорная *функция добавления значения*. Если расширение необходимо, оно происходит по месту (без создания новой индексной строки) при наличии на странице достаточного пространства.

Пусть, например, на странице 13 появилась версия строки со значением 42. Номер зоны (начиная с нуля) вычисляется целочисленным делением номера страницы на размер зоны. Считая, что размер зоны равен четырем страницам, получаем зону № 3, и в карте зон берем четвертый по счету указатель. Минимальное значение для этой зоны — 31, максимальное — 40. Новое значение выходит за эти пределы, и максимальное значение обновляется:



Если обновление по месту невозможно, создается новая строка, и тогда карта зон изменяется.

[1] backend/access/brin/brin.c, функция brininsert.

Обобщение зоны

Все сказанное касается случая, когда новая версия строки появляется в зоне, для которой уже есть сводная информация. При построении индекса обобщаются все существующие зоны, но при дальнейшем росте таблицы могут появиться новые страницы, выходящие за этот диапазон.

Если создать индекс с параметром хранения *autosummarize*, то новая зона будет обобщаться немедленно. Но поскольку обычно в хранилищах данных строки добавляются не по одной, а большими пакетами, такой режим может сильно замедлять вставку. off

По умолчанию немедленного обобщения новых зон не происходит. Это не нарушает корректность работы индекса, поскольку зоны без сводной информации просматриваются полностью. Обобщение выполняется асинхронно при очистке таблицы, либо его можно инициировать вручную, вызвав функцию `brin_summarize_new_values` или `brin_summarize_range` (для отдельной зоны). с. 142

Обобщение зоны [1] не блокирует изменение данных в таблице. Сначала в индекс вставляется строка-пустышка (placeholder). Если во время сканирования зоны данные поменяются, пустышка обновится и будет содержать сводную информацию по случившимся изменениям. Затем данные из этой строки объединяются с вычисленной сводной информацией по зоне с помощью опорной *функции объединения*.

При удалении строк из таблицы в некоторых случаях сводная информация могла бы сжаться. Но если в GiST-индексе пересчет происходит хотя бы при разделении страницы, в индексах BRIN сводная информация только расширяется и никогда не сжимается. Обычно этого и не требуется, поскольку хранилища данных используются, как правило, только для добавления данных. Конечно, можно вручную удалить сводную информацию, вызвав функцию `brin_desummarize_range`, и заново обобщить зону. Однако нет никакой подсказки, для каких зон такой пересчет может быть полезен. с. 557

[1] backend/access/brin/brin.c, функция summarize_range.

Таким образом, BRIN в первую очередь рассчитан на таблицы очень большого размера, которые либо не обновляются, либо обновляются очень незначительно, и, соответственно, новые строки в них добавляются преимущественно в конец файла. Основная область его применения — хранилища данных и аналитическая отчетность.

29.6. Диапазоны значений (minmax)

Для типов данных, значения которых можно сравнивать между собой, сводная информация в самом простом случае состоит из минимального и максимального значений. Соответствующие классы операторов содержат в названии слово `minmax`[1]:

```
\dAc   => SELECT opcname
       FROM pg_am am
         JOIN pg_opclass opc ON opcmethod = am.oid
       WHERE amname = 'brin'
       AND opcname LIKE '%minmax_ops'
       ORDER BY opcname;
              opcname
       ------------------------
        bit_minmax_ops
        bpchar_minmax_ops
        bytea_minmax_ops
        ...
        timetz_minmax_ops
        uuid_minmax_ops
        varbit_minmax_ops
       (26 rows)
```

Опорные функции этих классов операторов:

```
\dAp   => SELECT amprocnum, amproc::regproc
       FROM pg_am am
         JOIN pg_opclass opc ON opcmethod = am.oid
         JOIN pg_amproc amop ON amprocfamily = opcfamily
       WHERE amname = 'brin'
       AND opcname = 'numeric_minmax_ops'
       ORDER BY amprocnum;
```

[1] backend/access/brin/brin_minmax.c.

```
 amprocnum |         amproc
-----------+------------------------
         1 | brin_minmax_opcinfo
         2 | brin_minmax_add_value
         3 | brin_minmax_consistent
         4 | brin_minmax_union
(4 rows)
```

Первая функция возвращает служебную информацию о классе операторов, а остальные я уже описывал: это функции добавления значения, согласованности и объединения.

В классе minmax используются те же самые операторы сравнения, что и в B-дереве: *с. 546*

```
=> SELECT amopopr::regoperator, oprcode::regproc, amopstrategy          \dAo
FROM pg_am am
  JOIN pg_opclass opc ON opcmethod = am.oid
  JOIN pg_amop amop ON amopfamily = opcfamily
  JOIN pg_operator opr ON opr.oid = amopopr
WHERE amname = 'brin'
AND opcname = 'numeric_minmax_ops'
ORDER BY amopstrategy;
       amopopr       |  oprcode   | amopstrategy
---------------------+------------+--------------
 <(numeric,numeric)  | numeric_lt |            1
 <=(numeric,numeric) | numeric_le |            2
 =(numeric,numeric)  | numeric_eq |            3
 >=(numeric,numeric) | numeric_ge |            4
 >(numeric,numeric)  | numeric_gt |            5
(5 rows)
```

Выбор столбцов для индексирования

По каким столбцам имеет смысл строить BRIN-индекс с таким классом операторов? Я уже говорил, что индекс будет хорошо работать, если физическое расположение строк коррелирует с логическим порядком значений.

Проверим это на подготовленном выше примере.

```
=> SELECT attname, correlation, n_distinct
FROM pg_stats
WHERE tablename = 'flights_bi'
ORDER BY correlation DESC NULLS LAST;
      attname       |  correlation  |  n_distinct
--------------------+---------------+--------------
 scheduled_time     |    0.99999493 |        25930
 actual_time        |     0.9999949 |        34564
 fare_conditions    |     0.8010756 |            3
 flight_type        |     0.5012919 |            2
 airport_utc_offset |     0.4440494 |           11
 aircraft_code      |    0.17335044 |            8
 airport_code       |    0.04215337 |          104
 passenger_name     |   0.009984755 |         8298
 seat_no            | 0.00017101977 |          461
 flight_no          | -0.0043201274 |          707
 passenger_id       | -0.0066375434 | 2.505045e+06
 airport_coord      |               |            0
(12 rows)
```

Данные упорядочены по времени (как по плановому, так и по реальному — разница если и есть, то несущественная), поскольку добавляются в хронологическом порядке, а в отсутствие удалений и обновлений строки укладываются в файл основного слоя таблицы последовательно, одна за другой.

с. 29

Столбцы `fare_conditions`, `flight_type` и `airport_utc_offset` показывают относительно высокую корреляцию, но в них слишком мало уникальных значений.

Корреляция остальных столбцов слишком мала, чтобы их индексирование с помощью диапазонного класса операторов представляло интерес.

Размер зоны и эффективность поиска

Подсказкой для определения подходящего размера зоны может послужить количество страниц, на которых встречаются значения.

Рассмотрим время по расписанию `scheduled_time` и предположим, что запросы выбирают данные по рейсам за сутки. Посчитаем, сколько страниц занимают в таблице одни сутки.

Для этого мы воспользуемся тем, что идентификатор версии строки состоит из номера страницы и смещения. К сожалению, нет штатного способа разложить идентификатор на две его составляющие, поэтому придется написать некрасивую функцию, приводящую типы через текстовое представление:

```
=> CREATE FUNCTION tid2page(t tid) RETURNS integer
LANGUAGE sql
RETURN (t::text::point)[0]::integer;
```

Теперь можно посмотреть распределение суток по таблице:

```
=> SELECT min(numblk), round(avg(numblk)) avg, max(numblk)
FROM (
  SELECT count(distinct tid2page(ctid)) numblk
  FROM flights_bi
  GROUP BY scheduled_time::date
) t;
 min  | avg  | max
------+------+------
 1195 | 1456 | 1554
(1 row)
```

Видно, что распределение не совсем равномерное. При стандартном размере зоны в 128 страниц каждые сутки будут занимать от 9 до 12 зон. Получая информацию за одни сутки, сканирование захватит как нужные строки, так и часть строк, относящихся к другим суткам, но попавших в те же зоны. Чем больше размер зоны, тем больше будет прочитано лишних пограничных значений; уменьшая или увеличивая размер зоны, можно увеличить или уменьшить их количество.

Попробуем выполнить запрос, взяв какой-нибудь один день (индекс с параметрами по умолчанию я уже создал выше). Для упрощения картины я отключу параллельное выполнение:

```
=> SET max_parallel_workers_per_gather = 0;
=> \set d '2016-08-15 02:45:00+03'
=> EXPLAIN (analyze, costs off, timing off, summary off)
SELECT *
FROM flights_bi
WHERE scheduled_time >= :'d'::timestamptz
  AND scheduled_time < :'d'::timestamptz + interval '1 day';
```

```
                                QUERY PLAN
-------------------------------------------------------------------------
 Bitmap Heap Scan on flights_bi (actual rows=81964.00 loops=1)
   Recheck Cond: ((scheduled_time >= '2016-08-15 02:45:00+03'::ti...
   Rows Removed by Index Recheck: 11606
   Heap Blocks: lossy=1536
   Buffers: shared hit=1561
   -> Bitmap Index Scan on flights_bi_scheduled_time_idx
       (actual rows=15360.00 loops=1)
       Index Cond: ((scheduled_time >= '2016-08-15 02:45:00+03'::...
       Index Searches: 1
       Buffers: shared hit=25
 Planning:
   Buffers: shared hit=1
(12 rows)
```

Можно определить *коэффициент полезного действия* BRIN-индекса для некоторого запроса как отношение количества страниц, которые удалось пропустить при индексном сканировании, к общему количеству страниц в таблице. При нулевом КПД индексный доступ вырождается в последова-
с. 436 тельное сканирование (если не учитывать накладные расходы). Чем выше КПД, тем меньше страниц приходится читать. Но часть страниц содержат искомые данные, которые никак нельзя пропустить, поэтому КПД всегда меньше единицы.

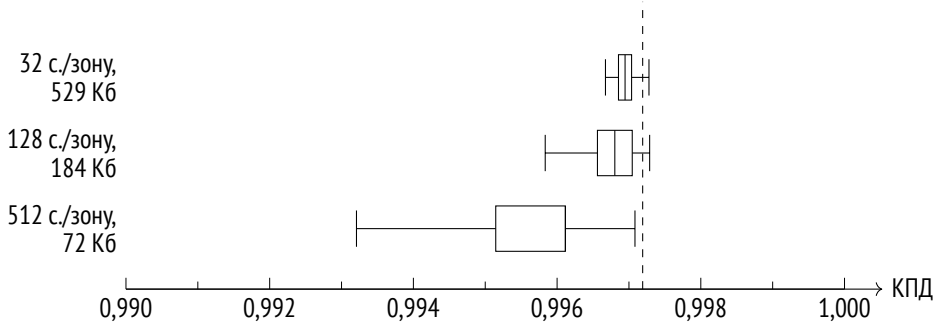
В данном случае КПД составляет $\frac{528\,417 - 1\,561}{528\,417} \approx 0{,}997$, где 528 417 — количество страниц в таблице.

Однако бессмысленно делать какие-то выводы по одному значению. Даже при равномерных исходных данных и идеальной корреляции эффективность будет отличаться как минимум из-за того, что границы зон будут расположены по-разному. Полную картину можно получить, лишь рассматривая КПД как случайную величину и изучая ее распределение.

Для нашего примера можно взять все различные сутки за год, проверить план выполнения для каждого значения и рассчитать статистические показатели по получившейся выборке. Процесс нетрудно автоматизировать, воспользовавшись тем, что команда `EXPLAIN` может выдавать результат в формате JSON, удобном для обработки. Я не буду приводить полный код, но следующий небольшой фрагмент содержит все существенные детали.

```
=> DO $$
DECLARE
  plan jsonb;
BEGIN
  EXECUTE
    'EXPLAIN (analyze, timing off, costs off, format json)
     SELECT * FROM flights_bi
     WHERE scheduled_time >= $1
       AND scheduled_time < $1 + interval ''1 day'''
  USING '2016-08-15 02:45:00+03'::timestamptz
  INTO plan;
  RAISE NOTICE 'shared hit=%, read=%',
    plan -> 0 -> 'Plan' ->> 'Shared Hit Blocks',
    plan -> 0 -> 'Plan' ->> 'Shared Read Blocks';
END;
$$;
NOTICE:  shared hit=1561, read=0
DO
```

Результаты можно представить наглядно с помощью диаграммы размаха, именуемой также «ящиком с усами». Усы обозначают первый и четвертый квартиль (то есть в правый ус попадают 25 % наибольших значений, а в левый — 25 % наименьших). Ящик составляют оставшиеся 50 % значений; в нем также отмечена медиана. Что важно, такое компактное изображение позволяет наглядно сравнивать между собой результаты. На рисунке показано распределение КПД как для размера зоны по умолчанию, так и для двух других значений, отличающихся от него в четыре раза:



Как и следовало ожидать, точность и эффективность поиска высоки даже для довольно большой зоны.

На рисунке пунктиром отмечено среднее значение максимально возможного КПД для этого запроса исходя из того, что одни сутки занимают примерно $\frac{1}{365}$ часть таблицы.

Обратите внимание, что эффективность растет не бесплатно: вместе с ней увеличивается и размер индекса. BRIN довольно гибко позволяет находить компромисс между одним и другим.

Свойства

Свойства BRIN постоянны и не зависят от конкретного класса операторов.

Свойства уровня метода доступа

```
=> SELECT a.amname, p.name, pg_indexam_has_property(a.oid, p.name)
FROM pg_am a, unnest(array[
  'can_order', 'can_unique', 'can_multi_col',
  'can_exclude', 'can_include'
]) p(name)
WHERE a.amname = 'brin';
 amname |     name      | pg_indexam_has_property
--------+---------------+-------------------------
 brin   | can_order     | f
 brin   | can_unique    | f
 brin   | can_multi_col | t
 brin   | can_exclude   | f
 brin   | can_include   | f
(5 rows)
```

Сортировка и поддержка уникальности, разумеется, отсутствуют. Из-за того, что индекс BRIN всегда возвращает битовую карту, нет поддержки и ограничений исключения. Дополнительные include-столбцы тоже лишены смысла, поскольку в индексе не хранятся даже сами ключи индексирования.

А вот создать составной BRIN-индекс можно. В этом случае сводная информация по каждому столбцу собирается и сохраняется в отдельной индексной строке, но привязка к зонам остается общей. Такой индекс имеет смысл, если для всех столбцов подходит один и тот же размер зоны.

В качестве альтернативы составному индексу можно создать BRIN-индексы
с. 429 по отдельным столбцам и пользоваться тем, что битовые карты могут комбинироваться.

Например:

```
=> CREATE INDEX ON flights_bi USING brin(airport_utc_offset);
=> EXPLAIN (analyze, costs off, timing off, summary off, buffers off)
SELECT * FROM flights_bi
WHERE scheduled_time >= :'d'::timestamptz
  AND scheduled_time < :'d'::timestamptz + interval '1 day'
  AND airport_utc_offset = '08:00:00';
                              QUERY PLAN
-----------------------------------------------------------------------
 Bitmap Heap Scan on flights_bi (actual rows=1658.00 loops=1)
   Recheck Cond: ((scheduled_time >= '2016-08-15 02:45:00+03'::ti...
   Rows Removed by Index Recheck: 14077
   Heap Blocks: lossy=256
   -> BitmapAnd (actual rows=0.00 loops=1)
       -> Bitmap Index Scan on flights_bi_scheduled_time_idx (act...
           Index Cond: ((scheduled_time >= '2016-08-15 02:45:00+0...
           Index Searches: 1
       -> Bitmap Index Scan on flights_bi_airport_utc_offset_idx ...
           Index Cond: (airport_utc_offset = '08:00:00'::interval)
           Index Searches: 1
(11 rows)
```

BRIN-индексы для больших таблиц могут создаваться параллельно. Таблица сканируется несколькими процессами, которые рассчитывают сводную информацию по своей части данных. Эта информация сортируется по зонам (выделяется память размером *maintenance_work_mem*), затем ведущий процесс агрегирует ее и формирует индекс[1].

v. 17

с. 552

64MB

Свойства уровня индекса

```
=> SELECT p.name, pg_index_has_property(
  'flights_bi_scheduled_time_idx', p.name
)
FROM unnest(array[
  'clusterable', 'index_scan', 'bitmap_scan', 'backward_scan'
]) p(name);
     name      | pg_index_has_property
---------------+-----------------------
 clusterable   | f
 index_scan    | f
 bitmap_scan   | t
 backward_scan | f
(4 rows)
```

[1] backend/access/brin/brin.c, функция brinbuild.

Очевидно, что поддерживается только сканирование по битовой карте.

Отсутствие кластеризации может вызвать некоторое недоумение. Казалось бы, раз BRIN-индекс чувствителен к физическому порядку строк, то логично было бы уметь перестраивать порядок, максимизируя эффективность индекса. Но в любом случае кластеризация больших таблиц — слишком дорогое удовольствие, учитывая объем необходимой работы и расход места на диске на время перестроения. К тому же, как показывает пример с таблицей `flights_bi`, некоторая упорядоченность в хранилищах данных может возникать естественным образом.

Свойства уровня столбца

```
=> SELECT p.name, pg_index_column_has_property(
  'flights_bi_scheduled_time_idx', 1, p.name
)
FROM unnest(array[
  'orderable', 'distance_orderable', 'returnable',
  'search_array', 'search_nulls'
]) p(name);
```

```
        name        | pg_index_column_has_property
--------------------+------------------------------
 orderable          | f
 distance_orderable | f
 returnable         | f
 search_array       | f
 search_nulls       | t
(5 rows)
```

Из свойств уровня столбца не доступно ничего, кроме возможности работы с неопределенными значениями. Наличие `NULL` в зоне учитывается отдельным признаком в сводной информации:

```
=> SELECT hasnulls, allnulls, value
FROM brin_page_items(
  get_raw_page('flights_bi_airport_utc_offset_idx', 6),
  'flights_bi_airport_utc_offset_idx'
)
WHERE itemoffset= 1;
```

```
 hasnulls | allnulls |         value
----------+----------+-----------------------
 f        | f        | {03:00:00 .. 03:00:00}
(1 row)
```

29.7. Мультидиапазоны значений (minmax-multi) v. 14

Сложившуюся корреляцию легко нарушить, изменяя данные. Дело здесь не в изменении какого-то конкретного значения, а в устройстве многоверсионности: старая версия строки удаляется на одной странице, а новая может быть вставлена на другую. Из-за этого при обновлениях версии строк перемешиваются. *с. 80*

Отчасти с этим явлением можно бороться, уменьшая значение параметра хранения *fillfactor*, чтобы оставить запас места на странице под будущие обновления. Вот только захочется ли увеличивать объем и без того огромной таблицы? К тому же остаются удаления, которые освобождают место внутри существующих страниц и таким образом готовят «ловушки» для новых строк, которые иначе попали бы в конец файла.

Такую ситуацию несложно смоделировать. Сначала удалим 0,1 % случайных строк и выполним очистку, чтобы освободить место для новых версий:

```
=> WITH t AS (
  SELECT ctid
  FROM flights_bi TABLESAMPLE BERNOULLI(0.1) REPEATABLE(0)
)
DELETE FROM flights_bi
WHERE ctid IN (SELECT ctid FROM t);
DELETE 30122

=> VACUUM flights_bi;
```

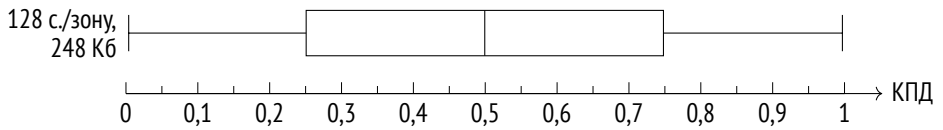
Затем добавим в таблицу новый день для одного из часовых поясов. Я просто скопирую данные предыдущих суток, увеличив день на единицу:

```
=> INSERT INTO flights_bi
SELECT airport_code, airport_coord, airport_utc_offset,
  flight_no, flight_type, scheduled_time + interval '1 day',
  actual_time + interval '1 day', aircraft_code, seat_no,
 fare_conditions, passenger_id, passenger_name
FROM flights_bi
WHERE date_trunc('day', scheduled_time) = '2017-08-15'
  AND airport_utc_offset = '03:00:00';
INSERT 0 40531
```

Количества удаленных строк достаточно, чтобы в каждой или почти каждой зоне оказались свободные места. Новые строки, попадая во внутренние страницы, автоматически расширили диапазоны значений. Вот сводная информация по первой зоне, и если раньше она охватывала неполные сутки, то теперь — целый год:

```
=> SELECT value
FROM brin_page_items(
  get_raw_page('flights_bi_scheduled_time_idx', 6),
  'flights_bi_scheduled_time_idx'
)
WHERE blknum = 0;
                        value
-----------------------------------------------------
 {2016-08-15 02:45:00+03 .. 2017-08-16 16:15:00+03}
(1 row)
```

Чем меньше дата, используемая в запросе, тем больше зон придется перебрать. Диаграмма показывает масштаб бедствия:



Для решения этой проблемы необходимо усложнить сводную информацию: вместо одного непрерывного диапазона хранить несколько, суммарно покрывающих все значения. Тогда один диапазон сможет охватить основной набор данных, а другие — отдельные выбросы.

Именно такую возможность и реализуют классы операторов, содержащие в своем названии `minmax_multi`[1]:

```
\dAc  => SELECT opcname
      FROM pg_am am
        JOIN pg_opclass opc ON opcmethod = am.oid
      WHERE amname = 'brin'
      AND opcname LIKE '%minmax_multi_ops'
      ORDER BY opcname;
```

[1] backend/access/brin/brin_minmax_multi.c.

```
           opcname
-----------------------------
 date_minmax_multi_ops
 float4_minmax_multi_ops
 float8_minmax_multi_ops
 ...
 timestamp_minmax_multi_ops
 timestamptz_minmax_multi_ops
 timetz_minmax_multi_ops
 uuid_minmax_multi_ops
(19 rows)
```

По сравнению с классами операторов minmax, к опорным функциям добавляется еще одна для вычисления расстояния между двумя значениями — она необходима для определения длины диапазона, которую класс операторов старается минимизировать:

```
=> SELECT amprocnum, amproc::regproc                                    \dAp
FROM pg_am am
  JOIN pg_opclass opc ON opcmethod = am.oid
  JOIN pg_amproc amop ON amprocfamily = opcfamily
WHERE amname = 'brin'
AND opcname = 'numeric_minmax_multi_ops'
ORDER BY amprocnum;
 amprocnum |               amproc
-----------+-----------------------------------
         1 | brin_minmax_multi_opcinfo
         2 | brin_minmax_multi_add_value
         3 | brin_minmax_multi_consistent
         4 | brin_minmax_multi_union
         5 | brin_minmax_multi_options
        11 | brin_minmax_multi_distance_numeric
(6 rows)
```

Сами операторы, составляющие такие классы, ровно те же, что и для классов minmax.

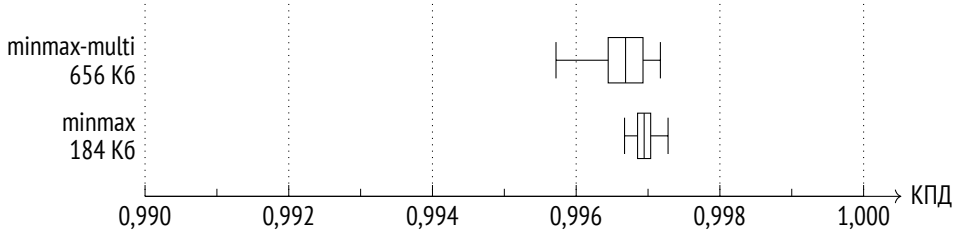
Мультидиапазонные классы могут принимать параметр *values_per_range*, определяющий максимальное количество сводных значений для одной зоны. Для представления диапазона требуется два значения, а для отдельной точки — одно. При нехватке значений часть интервалов «схлопывается»[1]. 32

[1] backend/access/brin/brin_minmax_multi.c, функция reduce_expanded_ranges.

Построим мультидиапазонный индекс вместо существующего. Для примера ограничимся 16 сводными значениями:

```
=> DROP INDEX flights_bi_scheduled_time_idx;
=> CREATE INDEX ON flights_bi USING brin(
  scheduled_time timestamptz_minmax_multi_ops(
    values_per_range = 16
  )
);
```

Диаграмма показывает, что эффективность поиска с новым индексом вернулась к старым значениям — конечно, ценой увеличения размера индекса:



29.8. Охватывающие значения (inclusion)

Разница между диапазонными и охватывающими классами операторов примерно такая же, как между B-деревьями и GiST: последние предназначены для типов данных, к которым неприменимы операции сравнения, но для которых имеет смысл взаимное расположение. Сводная информация охватывающих классов операторов представлена областью, ограничивающей все входящие в зону значения.

Вот эти классы операторов, их немного:

```
\dAc  => SELECT opcname
      FROM pg_am am
        JOIN pg_opclass opc ON opcmethod = am.oid
      WHERE amname = 'brin'
      AND opcname LIKE '%inclusion_ops'
      ORDER BY opcname;
```

```
       opcname
--------------------
 box_inclusion_ops
 inet_inclusion_ops
 range_inclusion_ops
(3 rows)
```

К опорным функциям добавляется функция слияния двух значений и некоторое количество необязательных функций:

```
=> SELECT amprocnum, amproc::regproc                                    \dAp
FROM pg_am am
  JOIN pg_opclass opc ON opcmethod = am.oid
  JOIN pg_amproc amop ON amprocfamily = opcfamily
WHERE amname = 'brin'
AND opcname = 'box_inclusion_ops'
ORDER BY amprocnum;
 amprocnum |          amproc
-----------+--------------------------
         1 | brin_inclusion_opcinfo
         2 | brin_inclusion_add_value
         3 | brin_inclusion_consistent
         4 | brin_inclusion_union
        11 | bound_box
        13 | box_contain
(6 rows)
```

Если про значения, совместимые с операторами сравнения, мы могли делать выводы на основе статистики по корреляции, то для других типов данных такая статистика не собирается[1]. Поэтому сложно заранее спрогнозировать, насколько хорошо будет работать охватывающий BRIN-индекс.

Хуже того, от корреляции существенно зависит оценка стоимости индексного сканирования. Если такая статистика отсутствует, она считается равной нулю[2]. Поэтому планировщик, увы, никак не различает точные и неточные inclusion-индексы и, как правило, отказывается их использовать.

PostGIS собирает статистику по корреляции для пространственных данных. v. 3.1.1

[1] backend/commands/analyze.c, функция compute_scalar_stats.
[2] backend/utils/adt/selfuncs.c, функция brincostestimate.

В нашем примере мы можем предположить, что индекс по координатам аэропортов имеет смысл, поскольку долгота должна коррелировать с часовым поясом.

В отличие от предикатов GiST, сводные значения BRIN имеют тот же тип, что и индексируемые данные, поэтому просто так не удастся построить индекс для точек. Но мы можем создать индекс по выражению, преобразовав точки в вырожденные прямоугольники:

```
=> CREATE INDEX ON flights_bi USING brin(box(airport_coord))
WITH (pages_per_range = 8);
=> SELECT pg_size_pretty(pg_total_relation_size(
  'flights_bi_box_idx'
));
 pg_size_pretty
----------------
 3816 kB
(1 row)
```

Индекс по часовым поясам с зоной такого же размера занимает примерно тот же объем — 3288 Кбайт.

Операторы, входящие в класс, соответствуют операторам GiST-классов. Например, с помощью индекса можно ускорять поиск точек в заданной области:

```
=> SELECT airport_code, airport_name
FROM airports
WHERE box(coordinates) <@ box '135,45,140,50';
 airport_code |  airport_name
--------------+----------------
 KHV          | Хабаровск-Новый
(1 row)
```

Но, как я и говорил, планировщик отказывается использовать индекс без отключения последовательного сканирования:

```
=> EXPLAIN (costs off)
SELECT *
FROM flights_bi
WHERE box(airport_coord) <@ box '135,45,140,50';
```

```
                        QUERY PLAN
-----------------------------------------------------------
 Seq Scan on flights_bi
   Filter: (box(airport_coord) <@ '(140,50),(135,45)'::box)
(2 rows)

=> SET enable_seqscan = off;

=> EXPLAIN (analyze, costs off, timing off, summary off, buffers off)
SELECT *
FROM flights_bi
WHERE box(airport_coord) <@ box '135,45,140,50';
                             QUERY PLAN
--------------------------------------------------------------------
 Bitmap Heap Scan on flights_bi (actual rows=511404.00 loops=1)
   Recheck Cond: (box(airport_coord) <@ '(140,50),(135,45)'::box)
   Rows Removed by Index Recheck: 721565
   Heap Blocks: lossy=21216
   -> Bitmap Index Scan on flights_bi_box_idx (actual rows=212160...
       Index Cond: (box(airport_coord) <@ '(140,50),(135,45)'::box)
       Index Searches: 1
(7 rows)

=> RESET enable_seqscan;
```

29.9. Фильтры Блума (bloom) v. 14

Классы операторов, построенные на фильтрах Блума, позволяют использовать BRIN с любыми типами данных, для которых определена операция «равно» и имеется хеш-функция. Пригодятся они и для обычных порядковых типов в ситуациях, когда значения локализованы в отдельных зонах, но при этом расположение не коррелирует с логическим порядком.

Названия таких классов операторов содержат слово `bloom`[1]:

```
=> SELECT opcname                                                \dAc
FROM pg_am am
  JOIN pg_opclass opc ON opcmethod = am.oid
WHERE amname = 'brin'
AND opcname LIKE '%bloom_ops'
ORDER BY opcname;
```

[1] backend/access/brin/brin_bloom.c.

```
        opcname
----------------------
 bpchar_bloom_ops
 bytea_bloom_ops
 char_bloom_ops
 ...
 timestamptz_bloom_ops
 timetz_bloom_ops
 uuid_bloom_ops
(24 rows)
```

Классический фильтр Блума — структура данных, позволяющая быстро проверить принадлежность элемента множеству. Фильтр очень компактен, но допускает ложноположительные срабатывания (false positive), то есть может приписать множеству лишние элементы. Важно, что ложноотрицательных срабатываний (false negative) быть не может: фильтр не имеет права решить, что элемента нет в множестве, если на самом деле он там присутствует.

Фильтр представляет собой битовый массив (называемый также *сигнатурой*) длиной m бит, изначально заполненный нулями. Выбираются k различных хеш-функций, которые отображают любой элемент множества в k бит сигнатуры. Добавление элемента к множеству сводится к установке в сигнатуре каждого из этих битов в единицу. Следовательно, если все соответствующие элементу биты установлены в единицу, элемент *может* присутствовать в множестве; если хотя бы один из этих битов равен нулю — элемент точно отсутствует.

В случае BRIN-индекса фильтр работает с множеством значений индексируемого столбца, содержащихся в определенной зоне; сводная информация этой зоны представляется построенным фильтром Блума.

> Расширение `bloom`[1] предоставляет самостоятельный индексный метод доступа, основанный на фильтрах Блума. В нем фильтр строится *для каждой строки* таблицы и работает с множеством значений *столбцов* этой строки. Такой индекс рассчитан на индексацию сразу нескольких столбцов и полезен для adhoc-запросов, когда неизвестно, на какие именно столбцы будут накладываться условия. BRIN-индекс тоже можно создать по нескольким столбцам, но в этом случае сводная информация будет содержать несколько независимых фильтров Блума для каждого из столбцов.

[1] postgrespro.ru/docs/postgresql/18/bloom.

Точность фильтра Блума зависит от длины сигнатуры. Теория говорит, что оптимальное количество битов сигнатуры можно оценить как $m = \frac{-n \log_2 p}{\ln 2}$, где n — число элементов множества, а p — вероятность ложного срабатывания фильтра.

Эти два значения и вынесены в параметры класса операторов:

- *n_distinct_per_range* — число элементов множества; в нашем случае это количество уникальных значений индексируемого столбца в одной зоне. Значения интерпретируются так же, как статистика уникальных значений: отрицательные числа показывают долю от числа строк в зоне, а не абсолютное количество. -0.1 *с. 351*
- *false_positive_rate* — вероятность ложноположительных срабатываний. 0.01

  Вероятность, близкая к нулю, говорит о том, что индексное сканирование не будет заглядывать в зоны, где нет искомых значений. Но она не гарантирует точность поиска, поскольку в просматриваемых зонах, помимо нужных значений, будут находиться и «лишние» строки. Это определяется не свойствами фильтра, а шириной зоны и физическим расположением данных.

К опорным функциям класса операторов добавляется функция вычисления хеш-кода:

```
=> SELECT amprocnum, amproc::regproc                                    \dAp
FROM pg_am am
  JOIN pg_opclass opc ON opcmethod = am.oid
  JOIN pg_amproc amop ON amprocfamily = opcfamily
WHERE amname = 'brin'
AND opcname = 'numeric_bloom_ops'
ORDER BY amprocnum;
 amprocnum |        amproc
-----------+----------------------
         1 | brin_bloom_opcinfo
         2 | brin_bloom_add_value
         3 | brin_bloom_consistent
         4 | brin_bloom_union
         5 | brin_bloom_options
        11 | hash_numeric
(6 rows)
```

Поскольку фильтр Блума основан на хешировании, единственный поддерживаемый оператор — оператор равенства:

```
\dAo  => SELECT amopopr::regoperator, oprcode::regproc, amopstrategy
      FROM pg_am am
        JOIN pg_opclass opc ON opcmethod = am.oid
        JOIN pg_amop amop ON amopfamily = opcfamily
        JOIN pg_operator opr ON opr.oid = amopopr
      WHERE amname = 'brin'
      AND opcname = 'numeric_bloom_ops'
      ORDER BY amopstrategy;
            amopopr       |  oprcode   | amopstrategy
--------------------------+------------+--------------
 =(numeric,numeric) | numeric_eq |            1
(1 row)
```

В качестве примера возьмем столбец с номером рейса `flight_no`, имеющий околонулевую корреляцию и поэтому безнадежный для обычного диапазонного класса операторов. Настройку вероятности ложного срабатывания менять не будем, а количество уникальных значений в зоне можно вычислить. Например, для восьмистраничной зоны получим:

```
=> SELECT max(nd)
FROM (
  SELECT count(distinct flight_no) nd
  FROM flights_bi
  GROUP BY tid2page(ctid) / 8
) t;
 max
-----
  26
(1 row)
```

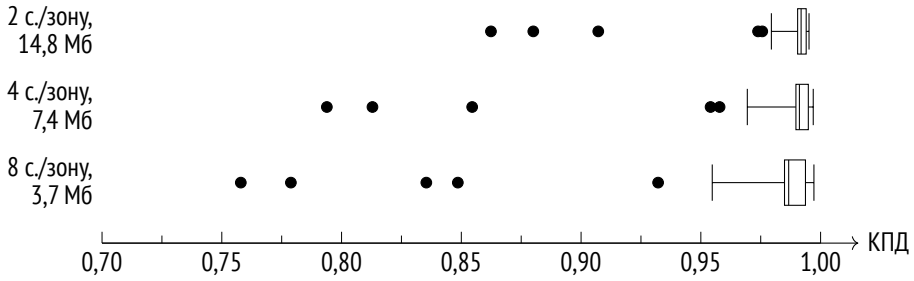
Для зон меньшего размера это количество будет еще меньше (в любом случае класс операторов не принимает значения, меньшие 16).

Остается создать индекс и проверить план выполнения запроса:

```
=> CREATE INDEX ON flights_bi USING brin(
  flight_no bpchar_bloom_ops(
    n_distinct_per_range = 22)
)
WITH (pages_per_range = 8);
```

```
=> EXPLAIN (analyze, costs off, timing off, summary off, buffers off)
SELECT *
FROM flights_bi
WHERE flight_no = 'PG0001';
                              QUERY PLAN
-----------------------------------------------------------------------
 Bitmap Heap Scan on flights_bi (actual rows=5197.00 loops=1)
   Recheck Cond: (flight_no = 'PG0001'::bpchar)
   Rows Removed by Index Recheck: 127510
   Heap Blocks: lossy=2240
   -> Bitmap Index Scan on flights_bi_flight_no_idx (actual rows=...
       Index Cond: (flight_no = 'PG0001'::bpchar)
       Index Searches: 1
(7 rows)
=> RESET max_parallel_workers_per_gather;
```

Из диаграммы видно, что для некоторых номеров рейсов (показанных отдельными точками вне усов) индекс работает не очень хорошо, но в целом его эффективность довольно высока:



Заключение

Что ж, пришло время подвести черту. Надеюсь, эта книга оказалась полезной или хотя бы интересной. Возможно, вы узнали из нее что-то новое (во всяком случае я, пока писал, узнал многое).

Хотя бóльшая часть информации, скорее всего, будет актуальна еще довольно долго, отдельные детали устаревают с неумолимой быстротой. Поэтому наиболее ценным из того, что можно вынести из этой книги, я считаю не набор конкретных фактов, а подход к изучению системы. Не надо верить на слово ни мне, ни даже документации. Обдумывайте, экспериментируйте, проверяйте все сведения самостоятельно. В PostgreSQL для этого есть все инструменты, и я старался показать, как ими пользоваться. В большинстве случаев это не сильно сложнее, чем задать вопрос в чате или спросить у искусственного интеллекта, зато намного надежнее и полезнее.

С этой же целью я хотел побудить вас заглядывать в код. Открытый исходный код — огромное благо, не пренебрегайте этой возможностью. Не переживайте, что ничего не поймете, просто попробуйте.

Буду рад, если вы поделитесь своими впечатлениями и замечаниями. Пишите на edu@postgrespro.ru. Я планирую регулярно обновлять книгу, так что ваши комментарии могут помочь улучшить ее. Самая свежая электронная версия книги находится в свободном доступе на postgrespro.ru/education/books/internals.

Удачи!

Предметный указатель

A

Aggregate 377–378
Append 392, 480
autoprewarm leader 207–208
autoprewarm master 207
autoprewarm worker 209
autosummarize 649
autovacuum 138
autovacuum launcher 138–140
autovacuum worker 139
autovacuum_analyze_scale_factor 143
autovacuum_analyze_threshold 143
autovacuum_enabled 126, 140
autovacuum_freeze_max_age 150, 156–157
autovacuum_freeze_min_age 157
autovacuum_freeze_table_age 157
autovacuum_max_workers 139–140, 168
autovacuum_multixact_freeze_max_age 274
autovacuum_naptime 139–140
autovacuum_vacuum_cost_delay 157, 167–168
autovacuum_vacuum_cost_limit 167–168
autovacuum_vacuum_insert_scale_factor 142
autovacuum_vacuum_insert_threshold 142
autovacuum_vacuum_max_threshold 141
autovacuum_vacuum_scale_factor 141
autovacuum_vacuum_threshold 141
autovacuum_worker_slots 139
autovacuum_work_mem 140

B

bgwriter 226, 250
bgwriter_delay 230
bgwriter_lru_maxpages 230, 232
bgwriter_lru_multiplier. 230
Bison 322
Bitmap Heap Scan 366, 428, 431, 434
Bitmap Index Scan 366, 427, 431, 434, 436
BitmapAnd 430
bloom 666
BRIN 642
  класс операторов 647, 650, 660, 663–664, 667
  КПД 654
  свойства 656
  страницы 645
btree_gin 635
btree_gist 573, 588
B-дерево 523, 573, 612, 615, 635
  класс операторов 546, 618, 651
  свойства 552
  страницы 529

C
checkpoint_completion_target 227–228
checkpointer 219
checkpoint_timeout 227–228, 232
checkpoint_warning 230
client_encoding 631
Clog 85, 157, 212, 216, 219
cmin и cmax 104
commit_delay 235
commit_siblings 235
cpu_index_tuple_cost 417
cpu_operator_cost 377, 417, 461–462, 486, 492, 501
cpu_tuple_cost 376, 378, 418, 450, 461–462, 486, 492
CTE Scan 338, 390
ctid 78, 116
cube 587
cursor_tuple_fraction 335, 345

D
data_checksums 241
deadlock_timeout 287, 295, 310
debug_io_direct 185
debug_parallel_query 389
debug_print_parse 323
debug_print_plan 326
debug_print_rewritten 324
deduplicate_items 534
Deduplication *см.* исключение дубликатов
default_statistics_target 347, 355–356, 359, 370
default_table_access_method 371
default_text_search_config 579
default_toast_compression 36
default_transaction_isolation 73

E
effective_cache_size 420–421
effective_io_concurrency 187–188, 199
enable_bitmapscan 419
enable_hashjoin 487
enable_memoize 450
enable_parallel_hash 470, 473
enable_seqscan 290, 419, 633, 664

F
false_positive_rate 667
fastupdate 296, 624
fdatasync 240
fillfactor 111–112, 119, 123, 148, 151, 171, 300, 514, 558, 592, 659
Finalize Aggregate 383
Finalize GroupAggregate 504
Flex 322
from_collapse_limit 330–333
fsync 240
full_page_writes 243, 246
fuzzystrmatch 611

G
Gather 379, 381–384, 389, 500–503
Gather Merge 500–502
geqo 332
geqo_threshold 332
gevel 596
GIN 612
  класс операторов 613, 617, 632, 636
  отложенное обновление 296, 624

свойства 627
страницы 615
gin_fuzzy_search_limit 627
gin_pending_list_limit 624
GiST 555, 662
класс операторов 556, 561, 664
свойства 574, 586
страницы 560
GroupAggregate 503

H
Hash 457, 461, 466, 511
класс операторов 519
свойства 520
страницы 512
Hash Join 457, 460–461, 466
HashAggregate 479–480, 503
hash_mem_multiplier 449, 458, 472, 480
HOT-обновление 116, 533
hstore 588, 641

I
idle_in_transaction_session_timeout 178
ignore_checksum_failure 242
Incremental Sort 498
Index Only Scan 423
Index Scan 414–416, 419, 444, 447
InitPlan 353, 390, 392
intarray 588, 641
io_combine_limit 187, 199
io_max_combine_limit 187
io_max_concurrency 187
io_method 185
io_workers 185

J
join_collapse_limit 330–333
JSON 636, 641
jsquery 641

K
k-мерное дерево 603

L
lock_timeout 285
log_autovacuum_min_duration 168
log_checkpoints 231
logical 247, 251
log_lock_waits 310
log_temp_files 466, 497
ltree 588

M
maintenance_io_concurrency 187–188
maintenance_work_mem 131, 166, 552, 625, 657
maintenance_work_mem 140
Materialize 441–443, 448, 450, 456
max_connections 258, 302
max_locks_per_transaction 258
max_parallel_maintenance_workers 552
max_parallel_processes 207
max_parallel_workers 384
max_parallel_workers_per_gather 384–386
max_pred_locks_per_page 302
max_pred_locks_per_relation 303
max_pred_locks_per_transaction 302–303
max_wal_senders 247

max_wal_size 228, 232
max_worker_processes 139, 384
Memoize 448–450, 506
Merge Join 483
minimal 239, 247, 251
min_parallel_index_scan_size 132
min_parallel_table_scan_size 385
min_wal_size 229
MixedAggregate 504
multixact_member_buffers 272
multixact_offset_buffers 272

N

n_distinct_per_range 667
Nested Loop 326, 439, 441, 444, 449
Nested Loop Anti Join 452
Nested Loop Left Join 439, 451
Nested Loop Semi Join 454
NULL *см.* неопределенное значение

O

OID 25

P

pageinspect 75, 79, 83, 90, 149, 215, 270, 513, 560, 596, 615, 646
pages_per_range 642
Parallel Append 391–392
Parallel Bitmap Heap Scan 436
Parallel Hash 472
Parallel Hash Join 472
Parallel Index Only Scan 435
Parallel Index Scan 435
Parallel Seq Scan 380–381, 471
parallel_leader_participation 379, 381
parallel_setup_cost 382, 501
parallel_tuple_cost 382, 501
parallel_workers 385
Partial Aggregate 382–383
Partial GroupAggregate 504
pgbench 237, 243, 313
pg_buffercache 189, 203, 206
pg_checksums 240
pg_controldata 223
PGDATA 23
pg_dump 109
pg_prewarm 206
pg_prewarm.autoprewarm 207
pg_prewarm.autoprewarm_interval 207
pg_rewind 213
pgrowlocks 273, 291
pgstattuple 171–172
pg_test_fsync 240
pg_trgm 588, 611, 630
pg_visibility 127, 150
pg_wait_sampling 312
pg_wait_sampling.profile_period 314
pg_waldump 217, 226, 244
pg_walinspect 217, 244, 248
plan_cache_mode 344
postgres 39
postmaster 39–41, 139, 223, 225, 379
ProcArray 85, 99, 314
psql 17, 21, 25, 96, 311, 319, 341, 343

Q

Quadtree 591

R
random_page_cost 376, 419, 432
RD-дерево 580
Read Committed 51–56, 59–60, 63, 65, 73–74, 97, 105–106, 110, 128, 279
Read Uncommitted 51–54
Repeatable Read 52–54, 63–65, 67–68, 70, 72–74, 97, 106, 110, 162, 278, 299
replica 247, 250–251
RTE 323
RUM 630
R-дерево 557

S
search_path 26
seg 587
Seq Scan 326, 375, 377–378
seq_page_cost 376, 419, 432, 469
Serializable 52–53, 70, 72–74, 97, 106, 110, 278, 299, 304, 388, 409
shared_buffers 202
shared_preload_libraries 206, 313
Skip Scan 539
Sort 488–490, 492, 501, 503–504
SP-GiST 589
  класс операторов 590, 592, 606
  свойства 601, 610
  страницы 596
startup 223–224
statement_timeout 285
SubPlan 391
subtransaction_buffers 92
Suffix truncation *см.* исключение атрибутов
synchronous_commit 235–237

T
temp_buffers 209
temp_file_limit 209, 339, 464
Tid Scan 415
TOAST 25, 33, 90, 200
track_commit_timestamp 99
track_cost_delay_timing 166
track_counts 139
track_io_timing 206
transaction_buffers 85
transaction_timeout 178

U
Unique 503

V
vacuum_buffer_usage_limit 200
vacuum_cost_delay 157, 163–164
vacuum_cost_limit 163–164, 167
vacuum_cost_page_dirty 164
vacuum_cost_page_hit 164
vacuum_cost_page_miss 164
vacuum_failsafe_age 150, 157
vacuum_freeze_min_age 150–152, 155
vacuum_freeze_table_age 150, 153–155
vacuum_index_cleanup 158
vacuum_max_eager_freeze_failure_rate 158–159
vacuum_multixact_failsafe_age 274
vacuum_multixact_freeze_min_age 274

vacuum_multixact_freeze_table_age 274
vacuum_truncate 134
values_per_range 661

W

WAL *см.* журнал предзаписи
wal_buffers 214
wal_compression 243
wal_init_zero 233
wal_keep_size 230
wal_level 247
wal_log_hints 243
wal_recycle 229
wal_segment_size 216
walsender 234, 247
wal_skip_threshold 247–248
wal_sync_method 240
walwriter 236
wal_writer_delay 236
wal_writer_flush_after 236
WindowAgg 490
work_mem 20, 338–339, 428–430, 432, 449, 458, 466, 470, 472–473, 480, 491

X

xmin и xmax 78, 80, 84, 87, 98, 134, 268, 274

А

Агрегация 377, 382
  сортировкой 501
  хешированием 479, 501
Анализ 137, 347, 513
Аномалия
  грязного чтения 48, 50, 55
  неповторяющегося чтения 51, 56, 63
  несогласованного чтения 58, 61, 65
  несогласованной записи 67, 70, 299
  потерянного обновления 50, 61–62
  только читающей транзакции 68, 71, 299
  фантомного чтения 51, 63, 298
Атомарность 49, 94

Б

База данных 23
Биты-подсказки *см.* информационные биты
Блок *см.* страница
Блокировка 53, 255, 395
  без ожидания 179, 284
  в памяти 189
  версии строки 275
  легкая 306
  неотношения 293
  номера транзакции 259
  отношения 133, 171, 176, 249, 261
  очередь 264, 274, 281
  повышение уровня 268, 302
  предикатная 298
  расширения отношения 295
  рекомендательная 296
  спин 305
  страницы 296
  строки 179, 268

транш 307
тяжелая 258, 269
Буферное кольцо 198, 374
Буферный кеш 40, 188, 212, 219, 306, 374, 395, 420
вытеснение 196
локальный 209, 392
настройка 202

В
Версия строки 77
только добавление 132, 142
Верхний ключ 530, 532, 615
Ветвь времени 217
Взаимоблокировка 258, 286, 295–296
Видимость 98, 104, 374, 395, 414, 423
Виртуальная транзакция 91, 106, 260
Вложенная транзакция 92, 219
Внедрение SQL-кода 342
Внешний ключ 270–271, 445
Восстановление 223
Выравнивание 79
Вытеснение 196, 215, 227

Г
Гистограмма 356
Горизонт 105–106, 111, 128, 178, 425
Гранулярность 256
Граф ожиданий 286
Группировка 478, 503
Грязное чтение 48, 50, 55

Д
Декартово произведение 438, 440
Демобаза 319, 558, 643
Дерево
несбалансированное 589
плана 326
префиксное 131, 605
разбора 322
сбалансированное 524, 528, 555, 612
сигнатурное 581
Диаграмма размаха 655
Долговечность 49

Ж
Журнал предзаписи 41, 211, 308, 372, 395
уровни 246

З
Заголовок
версии 78, 268
страницы 75, 127
Закрепление буфера 189, 191, 308
Замóк *см.* блокировка
Заморозка 135, 175, 193, 274
вручную 136, 151, 158
нетерпеливое сканирование 158
при загрузке 161
«Звездочка», причины не использовать 38, 459, 497

И
Идентификатор версии 77, 394, 614
Изменчивость функций 60, 402, 412

Изоляция 49
  на основе снимков 54, 70, 97, 268
Индекс 394, 400
  версионность 89
  обеспечение целостности 406, 408, 521, 552, 571, 574, 601
  очистка 123, 518, 529, 534, 625
  по выражению 364, 401, 664
  покрывающий 408, 422, 425, 552, 574, 601
  составной 407, 536, 545, 552, 574, 628, 656
  статистика 364
  уникальный 270, 406, 408, 526, 533, 552
  упорядоченность 405, 410, 523, 536, 545, 547, 552
  частичный 412
Индексирования механизм 395, 404
Информационные биты 78, 83, 86, 98, 243, 268
Исключение
  атрибутов 535
  дубликатов 533, 613
Исполнение запроса 337, 342

К

Кардинальность 335, 346, 420
  соединения 444
Карта
  битовая 131, 427, 512, 642, 647, 656
  видимости 32, 112, 125, 150–151, 175, 423
  заморозки 32, 150, 153, 175
  неопределенных значений 78
  свободного пространства 31, 112, 125
Класс операторов 397, 455, 613
  опорная функция 402
  параметры 584, 661, 667
Кластер баз данных 23
Комбо-номер 104
Контрольная точка 219, 239
  мониторинг 230
  настройка 227
Коррелированные предикаты 336, 366
Корреляция 361, 415, 427, 643, 651, 663
Курсор 104, 193, 334, 345, 388

М

Массив 588, 632, 641
Материализация 338, 390, 441, 448
Метод доступа
  индексный 394, 455
  свойства 404
  табличный 371
Многоверсионность 54, 77, 123, 533
Мультитранзакция 272
  переполнение счетчика 274

Н

Неопределенное значение 78, 350, 412, 545, 554, 575, 602, 658

Неповторяющееся чтение 51, 56, 63
Неравномерное распределение 353, 466, 514
Несогласованная запись 67, 70, 299
Несогласованное чтение 58, 61, 65

О

Обрыв транзакции 88, 92, 94, 278, 299
Ограничение целостности 47
Ожидание 310
  неучтенное время 311, 314
  семплирование 312
Опорная функция 402
Оптимизация *см.* планирование
Отношение 28
Очистка 106, 193, 347, 395, 425, 518, 649
  автоматическая 138, 287
  агрессивный режим 153
  мониторинг 163, 173
  обычная 125
  полная 171
  самоочистка 111, 119, 123, 529, 534
  этапы 131

П

Пакетная обработка 179, 285
Параллельное выполнение 379, 384, 434, 455, 470, 487, 500, 503
  ограничения 388
Переписывание *см.* трансформация
Переполнение счетчика транзакций 134, 155
Перепроверка 395, 415, 429, 578, 585, 602, 634
План выполнения 326
  общий и частный 343
Планирование 326, 342
Подготовка запроса 340
Поиск ближайших соседей 410, 565, 576, 603
Полнотекстовый поиск 577
  индексирование 578, 613
  ранжирование 630
  фразовый 629
  частичный 620
Получение результата 344
Портал 337
Потерянное обновление 50, 61–62
Правило
  перезаписи 325
  сортировки 399, 606
Привязка параметров 341
Протокол запросов 42
  простой 322
  расширенный 340
Процесс 39
  обслуживающий 41
Путь поиска 26

Р

Разбор запроса 322
Раздувание 108, 123, 178, 373, 376, 518, 528, 533, 557

Расщепление
корзины 511, 515
страницы 526, 528, 570, 599, 624
Ресурсное голодание 275, 281

С

Сегмент 28, 216
Селективность 335, 375
соединения 444
Семейство операторов 402
Семплирование ожиданий 312
Сервер 23
Сигнатура 581, 666
Синхронизация 235, 239
Системный каталог 24, 249, 323
Сканирование
индексное 301, 409, 414, 522, 553, 575, 601
оценка стоимости 375, 380, 415, 424, 430, 663
параллельное индексное 434
параллельное последовательное 380
по битовой карте 409, 427, 522, 553, 575, 601, 626, 658
последовательное 300, 374
с пропусками 539
сравнение методов 436
только индекса 348, 411, 422, 576, 583, 602, 610
Слияние 483, 494, 500
Слой 28
видимости 32, 112, 125, 150–151, 175, 423
инициализации 30
основной 29, 77, 652
свободного пространства 31, 112, 125
Снимок 97, 100, 249
для каталога 108
экспорт 109
Согласованность 47, 49
Соединение
анти- и полу- 439, 452
вложенным циклом 440
внешнее 438, 451, 485
внутреннее 438
оценка стоимости 441, 447, 449, 460, 469, 485, 491, 493, 497, 500–501
параллельным хешированием 471, 473
параметризованное 444
порядок 326, 328, 459, 485
слиянием 483, 551
сравнение способов 505
хешированием 457, 463
Сортировка 410, 483–484, 489, 523, 536, 545, 547
быстрая 491
внешняя 494
инкрементальная 498
параллельная 500
пирамидальная 492
Специальная область 76
Статистика 137, 335
базовая 346, 423
гистограмма 356, 486
для нескалярных типов 360
доля NULL 350
корреляция 361, 416, 651, 663
многовариантная 365

по выражению 362, 370
размер поля 361
расширенная 363
уникальные значения 351, 367, 652
частые значения 353, 369, 446, 466
Стоимость 328, 334, 337
Страница 33, 512
грязная 189
заголовок 162, 175
полный образ 223
расщепление 123, 526, 528, 570, 599, 624
фрагментация 78, 113
Схема 25

Т

Табличное пространство 26
Только читающей транзакции аномалия 68, 71, 299
Точка сохранения 92
Транзакция 48, 81, 97
блокировка номера 259
виртуальная 91, 106, 260
вложенная 92, 219
возраст 134
обрыв 88, 92, 94, 299
переполнение счетчика 134, 155
статус 99, 216
фиксация 85, 215, 278
Трансформация 324
Триграммы 630
Тупик *см.* взаимоблокировка

У

Указатель на версию строки 77, 113, 127
Усечение 133

Ф

Фантомное чтение 51, 63, 298
Фиксация транзакции 85, 215, 278
асинхронная 236
синхронная 235
Фильтр Блума 581, 665
Фоновая запись 226
настройка 230
Фоновый рабочий процесс 132, 139, 384

Х

Хеш-индекс *см.* Hash
Хеш-таблица 190, 306, 308, 449, 457, 511

Э

Эквисоединение 438, 476, 485, 511
Экземпляр сервера 23

Книги издательства «ДМК Пресс»
можно купить оптом и в розницу на складе издательства
по адресу: г. Москва, ул. Электродная, д. 2, стр. 12, офис 7,
тел. +7 (499) 322-19-38,
а также заказать на сайте www.dmkpress.com
с доставкой в любой регион РФ.

Рогов Егор Валерьевич

PostgreSQL 18 изнутри

При поддержке Postgres Professional
https://postgrespro.ru

Главный редактор *Яценков В. С.*
editor@dmkpress.com
Литературный редактор *Мантрова Л. В.*
Корректор *Синяева Г. И.*

Формат 70×100 1/16. Печать цифровая.
Гарнитуры ПТ (Паратайп), Iosevka (Renzhi Li) и Две Круглых (Юрий Гордон).
Усл. печ. л. 55,4. Тираж 200 экз.

Веб-сайт издательства: www.dmkpress.com