ПРАКТИЧЕСКАЯ МИНИ-КНИГА ДЛЯ ТЕХ, КТО ВИДЕЛ SLOW QUERY LOG

Индексы без мистики

Как перестать гадать, читать планы запросов и заставить базу делать меньше работы

Константин Потапов

25 июня 2026

Карта книги

Часть I. Сначала реальность

1. Для кого эта книга

Эта книга для backend-разработчиков, которые уже писали SQL, видели ORM-запросы в логах и хотя бы раз произносили опасную фразу: «Да там просто индекс добавить».

Она для тех, кто работает с PostgreSQL, MySQL, MariaDB, SQLite или любой другой нормальной реляционной базой и хочет понимать не заклинания, а механику. Почему один индекс спасает вечер, другой ничего не меняет, а третий тихо душит запись.

Особенно книга полезна, если ты:

- пишешь API, админки, отчёты, ленты, личные кабинеты и фоновые задачи;
- используешь Django ORM, SQLAlchemy, Eloquent, Hibernate или другой слой, который приятно прячет SQL до первого пожара;
- видел slow query log и делал вид, что это не про тебя;
- добавлял `db_index=True` с надеждой, а не с планом;
- хочешь разговаривать с базой без мистики и самоуговора.

Это не академический трактат про B-tree. И не справочник по синтаксису. Это короткая инженерная книга о дисциплине: сначала понять работу запроса, потом менять схему. В таком порядке. Иначе база быстро объяснит, кто здесь взрослый.

ПЛОХАЯ МЫСЛЬ
«Я примерно понимаю, почему тормозит».

ЗРЕЛАЯ МЫСЛЬ
«Я покажу план запроса и количество прочитанных страниц».

2. Зачем индекс вообще нужен

Индекс в базе данных нужен не для красоты схемы. Он нужен, чтобы база читала меньше страниц с диска и из памяти. Всё остальное — следствие.

Если совсем грубо: индекс — это не турбина. Это оглавление. Без оглавления библиотекарь тоже найдёт книгу, просто к вечеру будет ненавидеть и тебя, и литературу.

Когда приложение тормозит на запросе, первая слабая мысль звучит так: «Надо добавить индекс». Иногда это правда. Часто это способ не смотреть на запрос, данные и реальную нагрузку. Индекс не лечит плохую модель данных, не отменяет лишние джойны и не превращает бесконечную выборку в дешёвую операцию. Он просто даёт базе короткую дорогу к нужным строкам.

Представь таблицу пользователей на десять миллионов строк. Если ты ищешь пользователя по `email`, база без индекса вынуждена пройти по таблице и проверить каждую строку. Это последовательное чтение. Иногда оно быстрее, чем кажется, потому что чтение идёт крупными блоками. Но если запрос точечный и повторяется тысячи раз в минуту, полный проход по таблице — роскошь.

Индекс превращает задачу из «посмотри всё» в «найди область, где ответ может лежать». Это не магия, а структура данных.

```
CREATE INDEX users_email_idx ON users (email);
```

После такого индекса база может найти строку по `email` без полного сканирования таблицы. Но сразу появляется цена: индекс надо хранить, обновлять при вставках и изменениях, учитывать при планировании. Чем больше индексов, тем тяжелее запись. Индекс — это инструмент с весом, а не бесплатная кнопка «ускорить».

SQL выглядит невинно, пока не понимаешь: база выполняет ровно то, что ты попросил. Не то, что ты имел в виду.

Нормальный разговор об индексах начинается с трёх вопросов:

- какой запрос мы ускоряем;
- сколько строк он реально отбирает;
- как часто он выполняется по сравнению с записью.

Если этих ответов нет, индекс ставится наугад. Иногда попадёшь. Но чаще просто нарастишь технический жир.

3. Как база ищет строки

База данных хранит таблицу страницами. Страница — это блок данных фиксированного размера. У PostgreSQL по умолчанию 8 КБ. У InnoDB в MySQL — обычно 16 КБ. Внутри страниц лежат строки, служебная информация, ссылки и версии данных.

Запрос не «читает строку». Он читает страницы. Если нужная строка лежит на странице, база поднимает всю страницу. Поэтому важен размер результата, физическое расположение строк и количество страниц, которые приходится трогать.

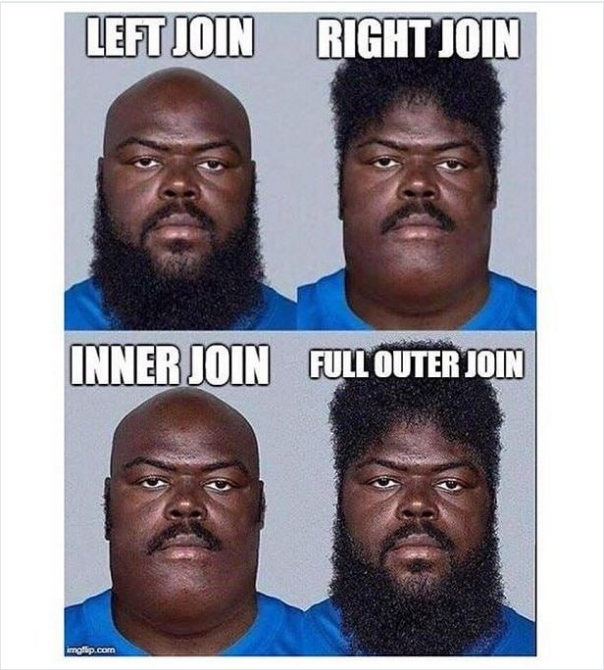
У базы есть несколько базовых способов доступа к данным.

Последовательное сканирование. База идёт по таблице целиком. Это звучит грубо, но на больших диапазонах может быть самым честным и быстрым вариантом. Если нужно вернуть половину таблицы, индекс часто только мешает: база будет прыгать по страницам, собирать строки кусками, а потом всё равно прочитает много.

Индексное сканирование. База идёт по индексу, находит ссылки на строки и затем достаёт строки из таблицы. Хорошо для точечных запросов и узких диапазонов. Плохо, если индекс возвращает слишком много строк: прыжки по таблице становятся дороже, чем прямой проход.

Bitmap scan. Часто встречается в PostgreSQL. База сначала собирает карту страниц, где есть подходящие строки, а потом читает эти страницы пачками. Это компромисс между хаотичным индексным доступом и полным проходом.

Index only scan. База отвечает из индекса, почти не трогая таблицу. Это возможно, когда все нужные колонки есть в индексе и система видимости строк позволяет не проверять таблицу слишком часто.



JOIN — отдельный зоопарк. Индекс помогает найти вход, но не превращает плохой JOIN в хорошую архитектуру.

Планировщик выбирает не «самый умный» путь, а самый дешёвый по своей модели. Модель опирается на статистику. Если статистика устарела, план может быть странным. Если запрос написан так, что база не может использовать индекс, план тоже будет честно плохим.

В этом месте важно не обижаться на базу. Она не обязана угадывать намерение разработчика. Ей нужен запрос, статистика и структура данных, по которым можно принять нормальное решение.

4. Сквозной кейс: лента событий, которая решила умереть

Возьмём обычный SaaS. Не игрушечный, но и не космический. Есть арендаторы, пользователи, события и лента активности. Сначала всё летало. Потом таблица `events` выросла до 80 миллионов строк, и лента стала открываться так, будто на сервере маленький человек вручную листает Excel.

Типичный запрос:

```
SELECT id, tenant_id, actor_id, event_type, created_at, payload
FROM events
WHERE tenant_id = 42
ORDER BY created_at DESC
LIMIT 50;
```

Первая реакция почти всегда одинаковая: добавить индекс по `created_at`. Потому что сортировка же по дате. Звучит логично. И часто мимо.

```
CREATE INDEX events_created_idx ON events (created_at DESC);
```

Что может пойти не так? База получит последние события по всей системе, а потом будет отбрасывать чужие `tenant_id`, пока не наберёт 50 строк нужного арендатора. Если арендатор маленький, база может перелопатить много лишнего. Индекс есть. Польза есть где-то рядом. Запрос всё равно грустный.

Более честный кандидат:

```
CREATE INDEX events_tenant_created_idx
ON events (tenant_id, created_at DESC);
```

Теперь база сначала попадает в область нужного арендатора, потом идёт по событиям в правильном порядке и останавливается на `LIMIT 50`. Уже похоже на работу, а не на надежду.

Но история не закончилась. Через месяц продакт просит фильтр по типу события:

```
WHERE tenant_id = 42
  AND event_type = 'payment_failed'
ORDER BY created_at DESC
LIMIT 50
```

Теперь надо заново смотреть распределение. Если `payment_failed` редкий, индекс `(tenant_id, event_type, created_at DESC)` может быть сильнее. Если фильтр по типу используют редко, отдельный индекс может быть лишним.

Этот кейс будет возвращаться дальше. Он нужен не ради таблицы `events`, а ради привычки: не «какую колонку индексировать», а «какую работу база сейчас делает зря».

ПЛОХАЯ МЫСЛЬ

«Сортировка по дате — значит индекс по дате».

ЗРЕЛАЯ МЫСЛЬ

«Сначала сужаем область, потом используем порядок».

Часть II. Основные конструкции

5. B-tree: рабочая лошадь

Большинство привычных индексов в реляционных базах — B-tree или близкая структура. Если не указано другое, `CREATE INDEX` обычно создаёт именно такой индекс.

B-tree хорошо работает для сравнений и сортировки:

```
WHERE id = 42
WHERE created_at >= '2026-01-01'
WHERE price BETWEEN 100 AND 500
ORDER BY created_at DESC
```

Смысл B-tree простой: значения лежат в упорядоченной структуре. База может быстро спуститься к нужному диапазону, а потом пройти по листовым страницам индекса последовательно. Это хорошо ложится на равенство, диапазоны и сортировку.

Но B-tree не умеет всё. Если ты пишешь:

```
WHERE LOWER(email) = 'user@example.com'
```

обычный индекс по `email` может не помочь, потому что в запросе используется выражение. База видит не колонку `email`, а результат функции `LOWER(email)`. Нужен функциональный индекс:

```
CREATE INDEX users_lower_email_idx ON users (LOWER(email));
```

Та же логика у поиска по окончанию строки:

```
WHERE email LIKE '%@example.com'
```

B-tree не может быстро найти значения, если начало строки неизвестно. Индекс упорядочен слева направо. Когда ты отрезаешь начало шаблоном `%`, база теряет точку входа.

Зато такой запрос может использовать B-tree:

```
WHERE email LIKE 'admin%'
```

Потому что известен префикс.

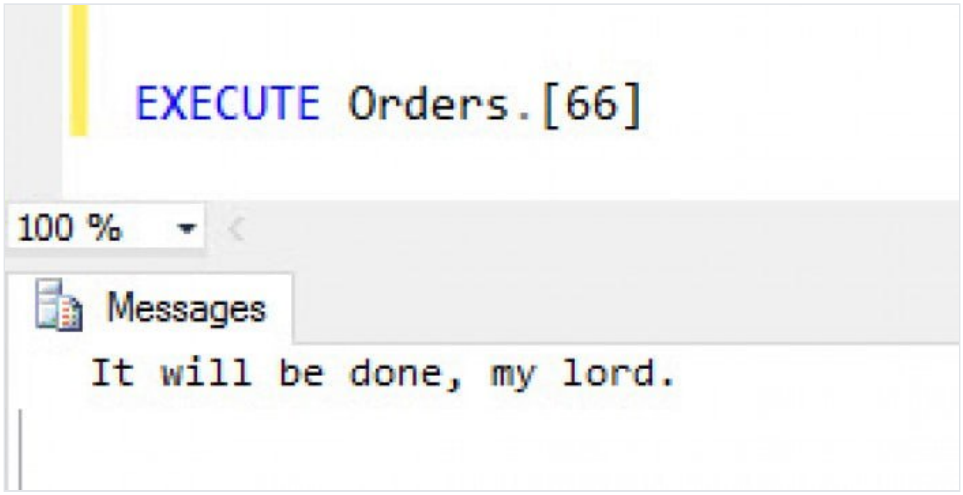
Главное правило: B-tree любит левый край значения и порядок. Он как строгий архивариус: если ты пришёл с началом фамилии — поможет; если сказал «там где-то в середине было `example`» — посмотрит так, что ты сам начнёшь читать документацию. Если запрос сохраняет этот порядок, индекс полезен. Если запрос ломает его функцией, неявным приведением типа или шаблоном с `%` в начале, пользы может не быть.

6. Составные индексы и порядок колонок

Составной индекс — это индекс по нескольким колонкам.

```
CREATE INDEX orders_user_status_created_idx
ON orders (user_id, status, created_at DESC);
```

Он не равен трём отдельным индексам. Это одна упорядоченная структура: сначала `user_id`, внутри него `status`, внутри пары `user_id + status` — `created_at`.



Когда запрос называется `Orders`, база не спорит. Она просто выполняет приказ. Поэтому формулируй приказы аккуратно.

Такой индекс хорошо подходит для запроса:

```
SELECT *
FROM orders
WHERE user_id = 10
  AND status = 'paid'
ORDER BY created_at DESC
LIMIT 20;
```

База быстро находит заказы конкретного пользователя с нужным статусом и уже получает их в нужном порядке. Сортировка может не понадобиться.

Но если запрос такой:

```
SELECT *
FROM orders
WHERE status = 'paid';
```

индекс `(user_id, status, created_at)` может оказаться слабым помощником. Первый столбец индекса — `user_id`. Запрос его не задаёт. База не может сразу прыгнуть в область `status = 'paid'`, потому что `status` упорядочен только внутри каждого `user_id`.

Есть рабочее правило: составной B-tree индекс используется слева направо до первого «размытого» условия.

Хорошая последовательность:

1. Колонки с равенством: `user_id = ?`, `tenant_id = ?`, `status = ?`.
2. Затем колонка диапазона: `created_at >= ?`, `price BETWEEN ? AND ?`.
3. Затем колонки для сортировки, если они продолжают порядок.

Пример:

```
WHERE tenant_id = ?
  AND status = ?
  AND created_at >= ?
ORDER BY created_at DESC
```

Индекс:

```
CREATE INDEX events_tenant_status_created_idx
ON events (tenant_id, status, created_at DESC);
```

Если поменять порядок на `(created_at, tenant_id, status)`, база сможет использовать диапазон по дате, но фильтры по арендатору и статусу станут менее полезными. Иногда это нормально. Например, если главный запрос — «последние события за час по всем арендаторам». Но для типичного SaaS, где почти каждый запрос начинается с `tenant_id`, такой порядок будет промахом.

Составной индекс проектируется под запрос, а не под чувство симметрии.

ПЛОХАЯ МЫСЛЬ
«Сделаю индекс из всех важных колонок».

ЗРЕЛАЯ МЫСЛЬ
«Соберу индекс под форму конкретного запроса».

> Авторская заноза: если индекс выглядит «аккуратно», но не соответствует запросу, это не инженерия. Это фэншуй на продакшене.

7. Селективность: где индекс помогает, а где делает вид

Селективность показывает, какую долю таблицы условие оставляет после фильтра.

`email = 'x'` обычно селективен: одна строка из миллионов.
`gender = 'male'` обычно не селективен: примерно половина таблицы.
`status = 'active'` часто ещё хуже: активных может быть 95%.

Индекс по низкоселективной колонке редко полезен сам по себе.

```
CREATE INDEX users_is_active_idx ON users (is_active);
```

Если почти все пользователи активны, запрос `WHERE is_active = true` вернёт почти всю таблицу. Планировщик часто выберет последовательное сканирование. И будет прав.

Но низкоселективная колонка может быть полезной внутри составного индекса, если перед ней стоит более точный фильтр.

```
CREATE INDEX users_tenant_active_created_idx
ON users (tenant_id, is_active, created_at DESC);
```

Для запроса:

```
WHERE tenant_id = 7
  AND is_active = true
ORDER BY created_at DESC
LIMIT 50
```

такой индекс уже имеет смысл. `tenant_id` резко сужает область, `is_active` уточняет, `created_at` отдаёт порядок.

Есть ещё частый случай: редкое значение в низкоселективной колонке. Например, `deleted_at IS NULL` истинно для 99% строк, а `deleted_at IS NOT NULL` — для 1%. Индекс по `deleted_at` может быть полезен для поиска удалённых строк, но бесполезен для поиска живых. Поэтому нельзя оценивать колонку в целом. Нужно смотреть конкретное условие.

Плохой вопрос: «Нужен ли индекс по `status`?»
Хороший вопрос: «Как распределены значения `status`, какие запросы идут чаще и сколько строк они возвращают?»

Слово `status` в названии колонки ещё не делает её важной. Иногда это просто табличный способ сказать: «у нас тут 95% всего в одном ведре».

8. WHERE, ORDER BY, LIMIT: три разных разговора с планировщиком

Разработчики часто думают об индексе только как о способе ускорить `WHERE`. Это узко.

Индекс может помочь базе:

- найти строки;
- вернуть строки уже отсортированными;

- остановиться после `LIMIT`, не просматривая всё.

Запрос:

```
SELECT id, title, created_at
FROM articles
WHERE published = true
ORDER BY created_at DESC
LIMIT 10;
```

Если есть только индекс по `published`, база найдёт опубликованные статьи, но потом ей придётся сортировать результат по `created_at`. Если опубликованных статей много, это боль.

Лучше:

```
CREATE INDEX articles_published_created_idx
ON articles (published, created_at DESC);
```

Теперь база может идти по области `published = true` сразу в нужном порядке и остановиться после десяти строк. Здесь `LIMIT` превращает индекс в сильное оружие.

Но порядок важен. Индекс `(created_at DESC, published)` может быть хорош, если запрос почти всегда просит самые свежие записи и `published = true` встречается часто. База идёт от новых к старым и отбрасывает неопубликованные. Если неопубликованных мало, это нормально. Если половина черновики — уже спорно.

Сортировка тоже должна совпадать с индексом. Если в запросе:

```
ORDER BY created_at DESC, id DESC
```

то индекс лучше сделать таким же:

```
CREATE INDEX articles_feed_idx
ON articles (published, created_at DESC, id DESC);
```

`id` здесь нужен не ради красоты сортировки. Он даёт стабильный порядок при одинаковом `created_at`. Без него пагинация может плавать: одна и та же запись будет появляться на двух страницах или пропадать между запросами.

Для ленты, истории событий, заказов и логов индекс часто начинается с фильтров, а заканчивается порядком.

9. Покрывающие индексы

Покрывающий индекс содержит все данные, нужные запросу. База может ответить из индекса и не ходить в таблицу за каждой строкой.

Пример:

```
SELECT id, email
FROM users
WHERE tenant_id = 5
  AND created_at >= '2026-01-01';
```

Индекс:

```
CREATE INDEX users_tenant_created_include_idx
ON users (tenant_id, created_at)
INCLUDE (id, email);
```

В PostgreSQL `INCLUDE` добавляет колонки в индекс как полезную нагрузку. Они не участвуют в сортировке и поиске, но доступны для чтения. В MySQL/InnoDB вторичные индексы уже содержат первичный ключ, а остальные поля надо учитывать иначе: если выбранные колонки есть в индексе, запрос может стать covering index scan.

Покрывающий индекс хорош для горячих списков:

- последние заказы клиента;
- краткая карточка пользователя;
- список задач по проекту;
- выдача уведомлений;
- проверка существования.

Но тут легко разжиреть. Если включить в индекс половину таблицы, он станет большим, медленным при записи и менее удобным для кэша. Покрывающий индекс должен покрывать конкретный горячий запрос, а не тревожность разработчика.

Здоровый вопрос: «Какие поля реально нужны на первом экране?» Если список показывает `id`, `title`, `status`, `created_at`, не надо тянуть `description`, `payload`, `metadata` и ещё пятнадцать колонок «на будущее». Индекс не обязан покрывать лень API.

10. Частичные индексы

Частичный индекс строится не по всей таблице, а только по строкам, которые удовлетворяют условию.

```
CREATE INDEX orders_unpaid_created_idx
ON orders (created_at DESC)
WHERE paid_at IS NULL;
```

Такой индекс полезен, если приложение часто работает с небольшой активной частью данных. Например:

- неоплаченные заказы;
- незавершённые задачи;
- непрочитанные уведомления;
- живые записи при soft delete;
- события определённого типа.

Запрос должен совпадать с условием индекса:

```
SELECT *
FROM orders
WHERE paid_at IS NULL
ORDER BY created_at DESC
LIMIT 50;
```

База понимает, что индекс содержит только неоплаченные заказы, и не тащит в индекс весь архив.

Частичный индекс особенно хорош там, где таблица большая, а рабочее множество маленькое. Например, 100 миллионов уведомлений за два года и 200 тысяч непрочитанных. Индекс только по непрочитанным будет компактным и быстрым.

Но частичные индексы требуют дисциплины. Если часть запросов пишет `paid_at IS NULL`, а часть — `status = 'unpaid'`, планировщик может не сопоставить это с одним и тем же смыслом. Для базы смысл — это выражение в SQL, а не бизнес-идея в голове разработчика.

Поэтому условие частичного индекса должно стать частью договорённости в коде. Иначе команда получит индекс, который вроде есть, но половина запросов его не использует.

11. Уникальные индексы и бизнес-правила

Уникальный индекс — не просто ускорение. Это способ зафиксировать бизнес-правило на уровне базы.

```
CREATE UNIQUE INDEX users_email_unique_idx
ON users (email);
```

Если email должен быть уникальным, проверка в приложении недостаточна. Два параллельных запроса могут одновременно увидеть, что email свободен, и оба попытаются вставить строку. Победит тот, кто первым зафиксирует транзакцию. Второй должен получить ошибку от базы. Это нормально.

Для SaaS часто нужна уникальность внутри арендатора:

```
CREATE UNIQUE INDEX users_tenant_email_unique_idx
ON users (tenant_id, email);
```

Теперь один и тот же email может существовать в разных арендаторах, но не дважды внутри одного.

Для soft delete полезен частичный уникальный индекс:

```
CREATE UNIQUE INDEX users_active_email_unique_idx
ON users (email)
WHERE deleted_at IS NULL;
```

Так база запрещает два живых пользователя с одним email, но позволяет хранить удалённые записи в архиве.

Это сильнее, чем «мы в коде проверяем». Код меняется, сервисов становится больше, появляются админки, миграции, импорты. База остаётся последней линией обороны. Если правило важно для денег, доступа или идентичности пользователя, оно должно жить в ограничении базы.

Индекс здесь не оптимизация. Это граница.

Часть III. Не только B-tree

12. Индексы для текста, JSON и географии

Не каждый поиск удобно решается B-tree.

Полнотекстовый поиск

Если нужно искать слова внутри текста, B-tree обычно не подходит. Для PostgreSQL есть `GIN` по `tsvector`:

```
CREATE INDEX articles_search_idx
ON articles
USING GIN (to_tsvector('russian', title || ' ' || body));
```

Запрос:

```
WHERE to_tsvector('russian', title || ' ' || body)
      @@ plainto_tsquery('russian', 'индексы базы данных')
```

Такой индекс хранит не строки целиком, а отображение слов к документам. Это другой тип задачи.

Для простого `ILIKE '%word%'` в PostgreSQL часто используют `pg_trgm`:

```
CREATE EXTENSION IF NOT EXISTS pg_trgm;

CREATE INDEX users_name_trgm_idx
ON users USING GIN (name gin_trgm_ops);
```

Триграммный индекс помогает искать подстроки и похожие строки. Он дороже B-tree, но для поиска по имени, адресу или заголовку может быть правильным выбором.

JSON

JSON-поля удобны, пока не становятся свалкой. Индексировать JSON можно, но сначала надо честно ответить: это действительно гибкая структура или просто отказ от схемы?

PostgreSQL:

```
CREATE INDEX events_payload_gin_idx
ON events USING GIN (payload);
```

Такой индекс помогает запросам по наличию ключей и фрагментов JSON. Но если есть горячее условие по конкретному ключу, часто лучше функциональный индекс:

```
CREATE INDEX events_payload_type_idx
ON events ((payload->>'type'));
```

И запрос:

```
WHERE payload->>'type' = 'signup'
```

Чем конкретнее запрос, тем конкретнее должен быть индекс. GIN по всему JSON может быть удобным, но большим. Индекс по выражению проще, если приложение реально фильтрует по одному ключу.

География

Для координат и геометрии нужны специализированные индексы: GiST, SP-GiST, R-tree-подобные структуры. Обычный B-tree по `latitude` и `longitude` не решает нормальный поиск «объекты рядом со мной».

Если в проекте появляется геопоиск, не надо героически собирать условия руками:

```
WHERE lat BETWEEN ... AND ...
  AND lon BETWEEN ... AND ...
```

Это может быть предварительным фильтром, но не полноценной моделью расстояния. Для серьёзной работы лучше использовать PostGIS и индексы, которые понимают геометрию.

Правило простое: тип индекса должен соответствовать типу вопроса. Равенство и диапазоны — B-tree. Полнотекст — GIN/FTS. Подстроки — триграммы. География — пространственные индексы. JSON — осторожно и по конкретным сценариям.

13. Цена индекса: запись, память, блокировки

Каждый индекс надо обновлять при вставке, изменении и удалении строк. Если таблица имеет десять индексов, вставка одной строки — это не одна операция. База должна вставить запись в таблицу и обновить десять структур.

Индексы замедляют:

- массовые импорты;
- частые `UPDATE` индексируемых колонок;
- удаление больших объёмов;
- миграции;
- репликацию;
- восстановление после сбоя.

Они занимают память и место на диске. Большой индекс хуже помещается в кэш. Если индекс не помещается в памяти, база чаще ходит на диск. Если индексов много, планировщику сложнее выбирать путь. Он справится, но цена планирования растёт.

Есть ещё блокировки. Создание индекса на большой таблице может мешать записи. В PostgreSQL для продакшена часто нужен:

```
CREATE INDEX CONCURRENTLY ...
```

Он работает дольше, но меньше блокирует таблицу. У него свои ограничения: нельзя запускать внутри обычной транзакции, при сбое может остаться невалидный индекс, который надо чистить.

В MySQL/InnoDB многие операции умеют online DDL, но «online» не значит «без цены». Всегда надо смотреть версию, тип операции, размер таблицы и нагрузку.

Новый индекс на проде без плана отката — это не смелость. Это попытка проверить, любит ли тебя Вселенная.

Индекс — это договор: мы платим на записи и хранении, чтобы выиграть на чтении. Если чтения мало, договор плохой. Если запрос критичен для продукта, договор может быть отличным.

ПЛОХАЯ МЫСЛЬ
«Индекс есть — хуже не будет».

ЗРЕЛАЯ МЫСЛЬ
«Каждый индекс берёт налог с записи».

Не надо бояться индексов. Надо считать цену.

Часть IV. Диагностика

14. Как читать EXPLAIN без гадания

`EXPLAIN` показывает план запроса. `EXPLAIN ANALYZE` выполняет запрос и показывает реальные времена и строки. Без него разговор об индексах быстро превращается в гадание по кофейной гуще.

PostgreSQL:

```
EXPLAIN (ANALYZE, BUFFERS)
SELECT *
FROM orders
WHERE user_id = 10
ORDER BY created_at DESC
LIMIT 20;
```

Не застревай на названии узла. Смотри на расхождение между оценкой и реальностью.

Если план ожидал 10 строк, а получил 500 000, статистика или условие не отражают реальность. Планировщик мог выбрать индекс, который на бумаге хорош, а на данных плох.

Важные признаки:

Seq Scan. Не всегда плохо. Плохо, если таблица большая, запрос точечный, а индекс вроде должен быть.

Index Scan. База идёт по индексу и затем читает таблицу. Хорошо для малой выборки. Следи за количеством строк и буферами.

Index Only Scan. Потенциально хорошо. Но если много heap fetches, база всё равно часто ходит в таблицу.

Rows Removed by Filter. База нашла много строк, а потом отфильтровала. Возможно, индекс начинается не с тех колонок или часть условий не попала в индекс.

Sort. Если сортировка дорогая и повторяется часто, индекс может помочь отдать данные уже в нужном порядке.

Buffers. Показывают, сколько страниц читалось из кэша и с диска. Время само по себе плавает. Буферы ближе к физике запроса.

Для MySQL используй:

```
EXPLAIN ANALYZE SELECT ...;
```

и смотри на `type`, `key`, `rows`, `filtered`, фактические итерации и время. Старый `EXPLAIN` без анализа полезен, но показывает только оценку.

Минимальный порядок диагностики:

1. Запусти `EXPLAIN ANALYZE` на реальном запросе.
2. Зафиксируй фактическое количество строк и буферы.
3. Проверь, какие условия попали в индекс, а какие стали фильтром после чтения.
4. Добавь или измени индекс.
5. Повтори `EXPLAIN ANALYZE`.
6. Сравни не настроение, а цифры.

Если после индекса запрос стал быстрее в три раза, но запись просела в два раза, это не всегда победа. Смотри на продуктовую цену.

15. Практический алгоритм проектирования индексов

Индексы лучше проектировать от нагрузки, а не от схемы. Колонки сами по себе ничего не говорят. Говорят запросы.

Алгоритм:

Шаг 1. Найди горячий запрос

Не «кажется медленным», а подтверждённо дорогой:

- часто выполняется;
- долго работает;
- блокирует пользовательский сценарий;
- даёт высокую нагрузку на CPU, диск или буферы.

Логи медленных запросов, APM, `pg_stat_statements`, Performance Schema — нормальная отправная точка.

Шаг 2. Выпиши форму запроса

Не абстрактно, а прямо:

```
SELECT id, total, status, created_at
FROM orders
WHERE tenant_id = ?
  AND customer_id = ?
  AND status IN ('paid', 'shipped')
ORDER BY created_at DESC, id DESC
LIMIT 50;
```

Шаг 3. Разложи запрос

- Фильтры равенства: `tenant_id`, `customer_id`.
- Список значений: `status IN (...)`.
- Порядок: `created_at DESC, id DESC`.
- Лимит: 50.
- Выбранные поля: `id`, `total`, `status`, `created_at`.

Шаг 4. Собери индекс-кандидат

```
CREATE INDEX orders_tenant_customer_status_created_id_idx
ON orders (tenant_id, customer_id, status, created_at DESC, id DESC)
INCLUDE (total);
```

Это не догма. Если `customer_id` уникален внутри арендатора, `status` может быть лишним. Если статусы распределены плохо, порядок можно пересмотреть. Если запрос всегда берёт последние заказы клиента независимо от статуса, статус лучше убрать или поставить позже.

Шаг 5. Проверь на данных

Индекс без `EXPLAIN ANALYZE` — гипотеза без проверки. Без цифр это не оптимизация, а инженерный гороскоп. После создания сравни:

- время выполнения;

- количество прочитанных буферов;
- фактические строки;
- наличие сортировки;
- влияние на запись, если запрос не единственный важный сценарий.

Шаг 6. Удали лишнее

После нескольких итераций часто остаются похожие индексы:

```
(user_id)
(user_id, created_at)
(user_id, created_at, id)
```

Первый может быть покрыт вторым, второй — третьим. Не всегда, но часто. Индексы надо ревизовать. Иначе схема превращается в кладбище осторожности.

Часть V. Ошибки и шпаргалка

16. Когда индекс стал балластом

Индексы создают. И удаляют тоже. Да, звучит как взрослый разговор, поэтому его часто откладывают.

В живой системе индексы накапливаются как старые зарядки в ящике: когда-то были нужны, теперь никто не помнит зачем, но выбросить страшно. Страх понятен. Он не является аргументом.

Балласт обычно выглядит так:

- индекс остался после удалённой фичи;
- есть два почти одинаковых индекса, и один покрывает другой;
- индекс не используется месяцами, но замедляет каждую запись;
- индекс создавался под отчёт, который давно переехал в витрину;
- колонка изменила смысл, а индекс остался как археологический слой;
- команда боится трогать схему, потому что «вдруг где-то нужно».

PostgreSQL даёт хорошую отправную точку:

```
SELECT relname, indexrelname, idx_scan
FROM pg_stat_user_indexes
ORDER BY idx_scan ASC;
```

`idx_scan = 0` не значит «удаляй прямо сейчас». После рестарта статистика сбрасывается, редкие месячные отчёты тоже существуют. Но это повод спросить: кто пользуется этим индексом, когда и зачем?

Ещё полезно искать дубли. Например, есть:

```
(user_id)
(user_id, created_at)
(user_id, created_at, id)
```

Иногда первый индекс уже не нужен, потому что второй или третий закрывает его сценарии. Иногда нужен: например, из-за размера, уникальности или особенностей планов. Проверяем, а не машем топором.

Удаление индекса на продакшене тоже делается аккуратно. В PostgreSQL:

```
DROP INDEX CONCURRENTLY index_name;
```

Идеальный порядок такой:

1. Найти кандидата на удаление.
2. Проверить статистику использования.
3. Найти запросы, которые могут от него зависеть.
4. Проверить планы без этого индекса на стенде или через гипотетические индексы, если есть инструмент.
5. Удалить безопасным способом.
6. Смотреть метрики после релиза.

ПЛОХАЯ МЫСЛЬ
«Не трогаем, вдруг пригодится».

ЗРЕЛАЯ МЫСЛЬ
«Если индекс берёт налог, он должен приносить пользу».

Индекс, который никто не использует, не нейтрален. Он ест диск, кэш, запись и внимание. Мёртвые индексы надо хоронить спокойно. Без героизма, но и без суеверий.

17. ORM не отменяет физику базы

ORM удобен. Он экономит время, держит модель ближе к коду и помогает не писать руками однотипный SQL. Но ORM не отменяет базу данных. Он просто прячет SQL до того момента, когда запрос начинает стоить денег.

В Django можно написать безобидно:

```
Event.objects.filter(tenant_id=tenant_id).order_by('-created_at')[:50]
```

На уровне базы это всё тот же запрос с `WHERE`, `ORDER BY` и `LIMIT`. Ему нужен индекс под форму доступа:

```
CREATE INDEX events_tenant_created_idx
ON events (tenant_id, created_at DESC);
```

`select_related` и `prefetch_related` решают проблему лишних запросов. Индекс решает проблему дорогого доступа к строкам. Это разные звери. N+1 не лечится индексом: если код делает 500 запросов вместо двух, база может выполнять каждый быстро и всё равно суммарно будет больно.

Админки тоже не невинные. Сортировка по колонке в Django Admin, фильтр в списке, поиск по `icontains`, массовая выгрузка CSV — всё это SQL. Если админка открывается только у разработчика на локальной базе, проблем нет. Если ею каждый день пользуется операционный отдел, это уже продуктовый сценарий.

`db_index=True` — не печать производительности. Это просьба создать индекс. Просьба может быть разумной, лишней или вредной. Особенно когда её ставят на поле «потому что по нему иногда фильтруют». Иногда — не нагрузка.

Миграции индексов требуют отдельной осторожности. Создать индекс на маленькой таблице можно почти не думая. Создать индекс на большой таблице в рабочее время — уже разговор с последствиями. Для PostgreSQL часто нужен `CREATE INDEX CONCURRENTLY`; для MySQL надо смотреть online DDL в конкретной версии и типе таблицы.

ПЛОХАЯ МЫСЛЬ
«ORM сам разберётся».

ЗРЕЛАЯ МЫСЛЬ
«ORM пишет SQL от моего имени, а отвечаю за него всё равно я».

Хороший backend-разработчик не обязан ненавидеть ORM. Он обязан иногда смотреть, что ORM отправил в базу. Там, в логах, обычно заканчивается магия и начинается профессия.

18. Антипаттерны

Классика жанра: запрос был правильный. База — нет. Индексы здесь уже не помогут, только холодный пот и бэкап.

Индекс на каждую колонку

Это выглядит заботливо, но обычно говорит о страхе. Отдельные индексы на каждую колонку редко заменяют хороший составной индекс. База иногда умеет комбинировать индексы, но это не повод проектировать вслепую.

Плохой набор:

```
CREATE INDEX ON orders (tenant_id);
CREATE INDEX ON orders (customer_id);
CREATE INDEX ON orders (status);
CREATE INDEX ON orders (created_at);
```

Если главный запрос фильтрует по `tenant_id`, `customer_id`, `status` и сортирует по `created_at`, нужен составной индекс под эту форму.

Индекс без запроса

«Эта колонка важная» — не аргумент. Важна не колонка, а операция над ней. Если по колонке не фильтруют, не сортируют, не соединяют таблицы и не проверяют уникальность, индекс может быть пустой тратой.

Функция поверх колонки

```
WHERE DATE(created_at) = '2026-06-25'
```

Такой запрос может сломать использование индекса по `created_at`. Лучше диапазон:

```
WHERE created_at >= '2026-06-25'
  AND created_at <  '2026-06-26'
```

Неявное приведение типа

Если колонка `uuid`, а приложение передаёт строку странным образом, база может привести тип так, что индекс не используется. То же бывает с числами, датами и текстом. Типы должны совпадать.

Слишком широкий индекс

Индекс на пять-шесть колонок иногда оправдан. Но если ты добавляешь в индекс всё подряд, остановись. Возможно, ты пытаешься компенсировать мутный API.

Индекс для редкого ручного отчёта

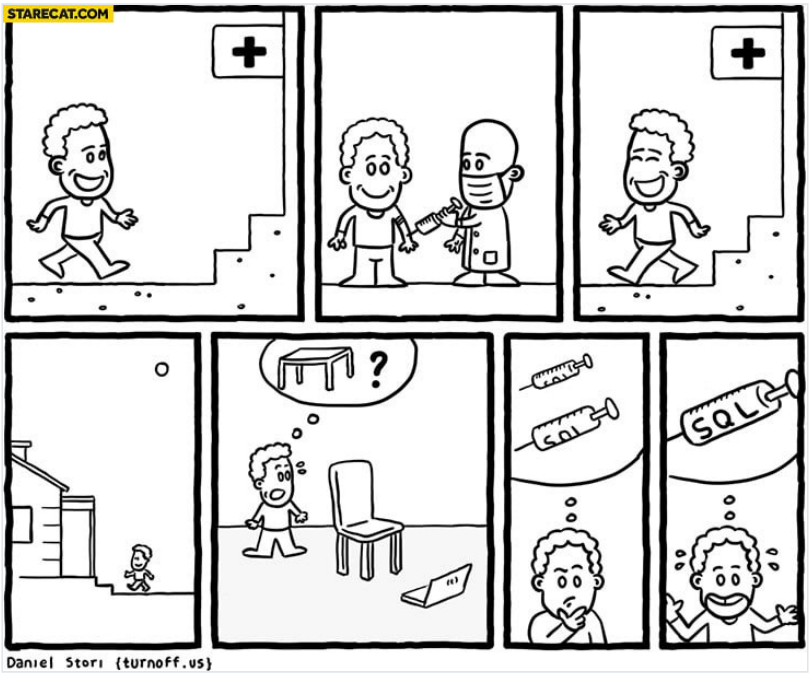
Если отчёт запускается раз в месяц ночью, а индекс замедляет каждую запись днём, решение сомнительное. Иногда лучше отдельная витрина, материализованное представление или фоновая выгрузка.

Игнорирование удаления и обновления

Индексы стареют вместе с данными. В PostgreSQL из-за MVCC появляются мёртвые версии строк, нужны autovacuum и периодическая ревизия bloating. В MySQL тоже есть фрагментация и своя механика обслуживания. Если база живая, индексы не стоят в вакууме.

Бонус: индекс не спасает от SQL-инъекции

Индекс ускоряет доступ к данным. Он не защищает от запроса, который собрали строковой конкатенацией и отправили в базу как письмо бывшей в три часа ночи: вроде слова знакомые, но последствия некрасивые.



Не все инъекции лечат. Некоторые потом лечат вас: аудитом, миграциями и неприятным разговором с безопасником.

Параметризованные запросы, ограничения на уровне базы, минимальные права пользователя и нормальный review SQL — это отдельная дисциплина. Индексы рядом, но не вместо неё.

19. Финальная шпаргалка

Когда индекс почти точно нужен

- Точечный поиск по уникальному или почти уникальному значению.
- Частый фильтр по `tenant_id`, `user_id`, `account_id` в большой таблице.
- Горячий список с `ORDER BY ... LIMIT`.
- Проверка уникальности бизнес-правила.
- Частая выборка малой активной части большой таблицы.
- Соединение таблиц по внешнему ключу на больших объёмах.

Когда индекс под вопросом

- Условие возвращает большую часть таблицы.
- Таблица маленькая.

- Запрос редкий и не влияет на пользователя.
- Колонка часто обновляется.
- Индекс широкий и нужен «на всякий случай».
- Нет `EXPLAIN ANALYZE` до и после.

Порядок колонок в составном индексе

Сначала равенства. Потом диапазон. Потом сортировка. Но это правило надо проверять на реальной форме запроса.

Пример:

```
WHERE tenant_id = ?
  AND status = ?
  AND created_at >= ?
ORDER BY created_at DESC
LIMIT 50
```

Индекс:

```
(tenant_id, status, created_at DESC)
```

Проверочные вопросы перед созданием индекса

- Какой конкретно запрос станет быстрее?
- Сколько строк он возвращает сейчас?
- Какой план у него сейчас?
- Какой план ожидаем после индекса?
- Насколько часто выполняется запрос?
- Насколько часто меняются индексируемые колонки?
- Есть ли похожий индекс, который можно расширить или заменить?
- Как безопасно создать индекс на продакшене?
- Как поймём, что индекс надо удалить?

Минимальный набор команд

PostgreSQL:

```
EXPLAIN (ANALYZE, BUFFERS) SELECT ...;
CREATE INDEX CONCURRENTLY index_name ON table_name (...);
DROP INDEX CONCURRENTLY index_name;
ANALYZE table_name;
```

MySQL:

```
EXPLAIN ANALYZE SELECT ...;
CREATE INDEX index_name ON table_name (...);
DROP INDEX index_name ON table_name;
ANALYZE TABLE table_name;
```

Последнее правило

Индекс — это не украшение схемы. Это ставка: мы платим на записи, чтобы выиграть на чтении. Хороший инженер не верит ставке на слово. Он смотрит план, проверяет цифры и убирает лишнее.

Если запрос стал понятнее после разговора об индексе, ты уже выиграл половину. Потому что база редко тормозит «просто так». Обычно она честно выполняет мутное поручение.

База данных — неприятно честный собеседник. Она не верит в намерения, не уважает оптимизм и не читает мысли. Зато прекрасно исполняет SQL. Иногда именно это и страшно.

Индексная дисциплина

- Не верь ощущению — смотри план.
- Не ускоряй то, что не измерил.
- Не лечи архитектуру индексами.
- Не добавляй индекс, чтобы успокоиться.
- Не бойся удалить лишнее.
- Не спорь с базой на эмоциях: она всё равно выиграет.
- Не путай быстрый запрос на локальной базе с рабочей нагрузкой.
- Не считай ORM оправданием. SQL всё равно существует.

Хороший индекс — не тот, который существует. Хороший индекс — тот, после которого база делает меньше работы, а ты можешь это доказать.

20. Об авторе

Константин Потапов — Senior Python Backend / R&D Engineer. Работает с Python, Django, FastAPI, PostgreSQL, MariaDB, Redis, Docker, Linux, интеграциями и инженерными инструментами.

Главная профессиональная привычка — не верить красивому объяснению, пока оно не прошло через код, данные и проверку реальностью. В backend-разработке это особенно полезно: база данных не впечатляется уверенностью разработчика и не ускоряется от хорошего настроения команды.

Эта книга написана в том же подходе: меньше мистики, меньше позы, больше наблюдаемой работы. Запрос либо читает меньше страниц, либо нет. Индекс либо помогает, либо просто стоит в схеме и берёт свой налог.

potapov.me

Не добавляй индекс, чтобы успокоиться. Добавляй его, когда знаешь, какую работу он заберёт у базы.