Nguyễn Thế Huy

DATABASE INDEXING

& NHỮNG ĐIỀU DEVELOPER CẦN BIẾT

https://huynt.dev
huynt57@gmail.com

Lời mở đầu

Chào bạn, và cám ơn bạn đã đọc ebook này của mình.
Mình là Huy, một Developer với trên 10 năm kinh nghiệm qua nhiều doanh nghiệp như ViettelPost, Giaohangtietkiem, CBTW … Ebook này hy vọng có thể chia sẻ tới các bạn những kinh nghiệm xương máu của mình trong hành trình làm việc với CSDL quan hệ trong các hệ thống từ vừa tới “hơi” lớn một chút, giúp bạn “dễ thở” hơn trong công việc hàng ngày cũng như khi phỏng vấn.

Nếu bạn là developer và đang đọc ebook này, khả năng cao bạn đã từng gặp tình huống thế như sau: query chạy ngon lành trên local với vài trăm record, nhưng lên production với vài triệu record thì... chậm không tưởng. Bạn Google "how to make SQL query faster", thấy mọi người bảo "thêm index đi", bạn thêm index, nhưng vẫn chậm. Hoặc tệ hơn, database không thèm dùng cái index bạn vừa tạo.
Vấn đề không phải bạn kém. Vấn đề là hầu hết tài liệu về indexing hoặc quá sơ sài (chỉ nói "tạo index trên cột WHERE"), hoặc quá hàn lâm (viết cho DBA với hàng trăm trang về B-tree internals). Không có mấy tài liệu nào vừa đủ sâu, vừa đủ thực tế cho developer.

Cuốn ebook này ra đời vì lý do đó. Mục tiêu là giúp bạn hiểu indexing một cách có hệ thống, đồng thời trang bị thêm các kỹ thuật database thực chiến mà developer nào cũng nên biết.

Bạn sẽ học được gì:

- Cách index thực sự hoạt động (không đào quá sâu vào thuật toán B-tree)
- Bốn nguyên tắc cốt lõi để tạo index hiệu quả cho mọi query
- Tại sao database đôi khi "phớt lờ" index của bạn và cách xử lý
- Các tips & tricks thực chiến khi làm việc với database hàng ngày

Hãy đọc từ đầu đến cuối lần đầu tiên, vì các chương sau xây dựng trên kiến thức của chương trước. Sau đó bạn có thể dùng từng chương như một tài liệu tra cứu độc lập.

Nền tảng: Hiểu Index từ gốc rễ

B+ Tree: Không cần hiểu chi tiết, chỉ cần hiểu ý tưởng

Mọi cuốn sách về database đều bắt đầu bằng việc giải thích chi tiết cấu trúc B+ tree: leaf nodes, internal nodes, thuật toán insert, delete, rebalance... Thành thật mà nói, bạn không cần biết tất cả những thứ đó để tạo được index tốt. Số người thực sự cần hiểu chi tiết kỹ thuật B+ tree là rất nhỏ, và bạn không phải một trong số đó. Kiến thức quá chi tiết về internal thậm chí còn gây hại, vì bạn sẽ mắc kẹt trong cảm giác "mình chưa hiểu đủ" và không dám áp dụng.

Thay vào đó, hãy nhớ hai điều cốt lõi về B+ tree:

Index là một danh sách đã được sắp xếp (sorted list) + bảng tóm tắt phân cấp

Hãy tưởng tượng bạn có một cuốn từ điển dày 2000 trang. Bạn muốn tìm từ "performance". Bạn sẽ:

- Nhìn vào phần gáy sách → thấy P nằm khoảng trang 1200
- Lật đến trang 1200, nhìn header → thấy "PER" bắt đầu từ trang 1245
- Lật đến 1245, scan vài trang → tìm thấy "performance"

Bạn chỉ cần 3 bước thay vì đọc 2000 trang. Index trong database hoạt động y hệt:

- Leaf nodes (tầng dưới cùng) = các trang từ điển, chứa danh sách giá trị đã sorted
- Internal nodes (các tầng trên) = phần gáy/header sách, chứa "tóm tắt phạm vi" để nhảy nhanh
- Kể cả bảng có hàng tỷ row, số bước nhảy qua internal nodes cũng chỉ khoảng 3-4 lần (vì mỗi tầng phân chia dữ liệu ra hàng trăm/hàng nghìn nhánh)

Từ đây bạn có thể hình dung đơn giản: index = sorted list + bảng tóm tắt giúp nhảy nhanh. Không cần phức tạp hơn thế.

Database tự quản lý index hoàn toàn

Mỗi khi bạn INSERT, UPDATE hay DELETE một row, database tự động cập nhật tất cả các index liên quan:

- Thêm row → tạo entry mới trong index, đặt đúng vị trí sorted
- Xóa row → xóa entry tương ứng trong index

- Sửa row → xóa entry cũ, thêm entry mới (chỉ khi cột trong index bị thay đổi). Nếu bạn chỉ update một cột không nằm trong bất kỳ index nào, index không bị ảnh hưởng gì.

Điều này dẫn đến một trade-off quan trọng: càng nhiều index → write càng chậm (vì mỗi lần ghi phải cập nhật nhiều index hơn). Để bạn hình dung cụ thể:

```
Bảng users có 5 index:
- INSERT 1 row → database phải thêm entry vào CẢ 5 index
- UPDATE cột email (nằm trong 2 index) → 2 index bị cập nhật
- UPDATE cột bio (không nằm trong index nào) → 0 index bị cập nhật
- DELETE 1 row → xóa entry khỏi CẢ 5 index
```

Nhưng đừng lo lắng quá: trong thực tế, hầu hết ứng dụng đọc nhiều hơn ghi rất nhiều (tỷ lệ read:write thường là 90:10 hoặc cao hơn), và mỗi bảng thường chỉ cần 3-7 index. Chi phí write thêm cho index hầu như không đáng kể so với lợi ích read mà nó mang lại.

Câu hỏi hay gặp: "Vậy tạo bao nhiêu index là quá nhiều?" Không có con số cố định, nhưng nếu bảng có hơn 10 index và bạn thấy write performance giảm đáng kể, đó là lúc nên review lại. Dùng query kiểm tra unused index (sẽ học ở Phần 5) để tìm và xóa index thừa.

Có sắp xếp hay không: Chuyện của Primary Key

Vì index là sorted list, vị trí insert entry mới ảnh hưởng đến performance:

- Thêm vào cuối (giá trị luôn tăng): nhanh, vì vị trí cuối luôn được cache trong memory
- Thêm vào giữa (giá trị random): chậm hơn, vì phải tìm vị trí đúng, có thể phải di chuyển entries xung quanh, và trang chứa vị trí đó có thể không nằm trong memory

Đây là lý do lựa chọn kiểu primary key quan trọng:

Loại PK	Giá trị mẫu	Thứ tự insert	Tốc độ
Auto-increment	1, 2, 3, 4...	Luôn tăng → cuối list	Nhanh nhất
UUIDv4	`a3f8b2c1-...` (random)	Random → giữa list	Chậm nhất
UUIDv7 / ULID	`019abc12-...` (time-based)	Gần như tăng → gần cuối	Nhanh
Snowflake ID	`142506829879...` (time-based)	Luôn tăng → cuối list	Nhanh

Để bạn hình dung mức độ ảnh hưởng: trên một bảng MySQL với 10 triệu row, insert với UUIDv4 có thể chậm hơn 3-5 lần so với auto-increment. Khoảng cách này càng lớn khi bảng càng to, vì index không fit trong memory nữa và mỗi random insert có thể trigger disk I/O.

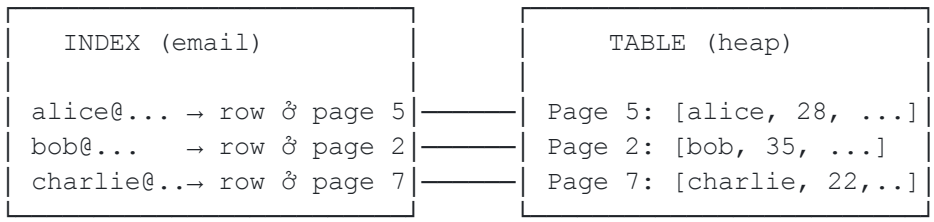
Lưu ý quan trọng: Vấn đề random key ảnh hưởng nghiêm trọng nhất với MySQL (InnoDB) vì nó dùng Clustered Index (PK = table, xem phần tiếp). Với PostgreSQL (Heap Table) thì ảnh hưởng nhẹ hơn vì dữ liệu bảng append vào cuối bất kể PK là gì: chỉ secondary index bị ảnh hưởng. Nhưng nếu bảng có hàng triệu row và insert liên tục, bạn nên dùng sorted key cho cả hai.

Hai cách lưu trữ khác nhau: Heap Table và Clustered Index

Đây là kiến thức nền quan trọng mà nhiều developer bỏ qua, nhưng nó ảnh hưởng trực tiếp đến cách bạn thiết kế schema và chọn primary key. Có hai cách database lưu trữ dữ liệu trên disk:

Heap Table (PostgreSQL mặc định)

Dữ liệu được append vào cuối file, không quan tâm thứ tự. Bạn insert row theo thứ tự nào, nó nằm ở vị trí đó luôn: kể cả PK = 999 được insert trước PK = 1. Tất cả index (kể cả primary key) đều lưu "địa chỉ vật lý" (physical location: gọi là tuple ID hoặc ctid trong PostgreSQL) của row.

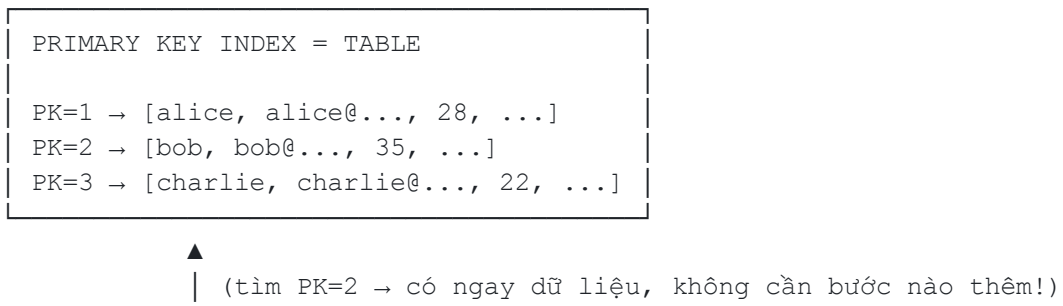


Primary key index cũng hoạt động y hệt: chỉ khác là có UNIQUE constraint.

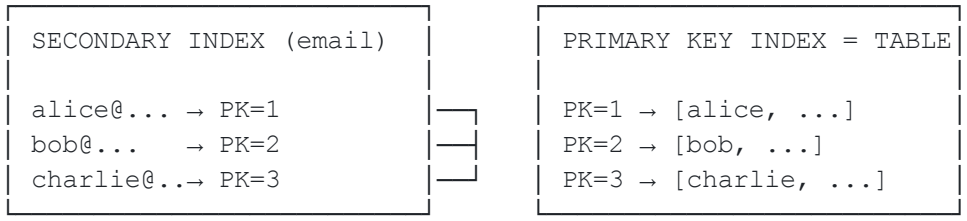
Điểm quan trọng: vì tất cả index đều bình đẳng (đều trỏ đến vị trí vật lý), không có sự khác biệt performance giữa lookup bằng PK hay bằng secondary index: cả hai đều cần 1 bước nhảy từ index vào table.

Clustered Index (MySQL/InnoDB mặc định)

Ở đây mọi thứ khác hẳn. Primary key chính là bảng: dữ liệu row được lưu ngay trong leaf node của primary key index, sắp xếp theo thứ tự PK. Không có "bảng heap" riêng.



Còn secondary index thì không trỏ đến vị trí vật lý (vì không có heap), mà trỏ đến giá trị primary key:



`Secondary index lookup = 2 bước: tìm PK trong secondary index → tìm row trong PK index`

So sánh hai cách tiếp cận:

	Heap Table (PostgreSQL)	**Clustered Index (MySQL)**
PK lookup	2 bước: PK index → heap table	1 bước: PK index = table
Secondary index lookup	2 bước: sec. index → heap table	2 bước: sec. index → PK index
Insert random PK	Nhanh (append vào heap)	Chậm (phải insert đúng vị trí trong sorted PK)
PK size ảnh hưởng?	Ít (chỉ ảnh hưởng PK index)	Nhiều (PK copy vào MỌI secondary index)

Hệ quả thực tế cho MySQL (Clustered Index):

Hệ quả 1: *KHÔNG dùng UUIDv4 làm primary key!*

Vì table = PK index (sorted), mỗi row mới với UUIDv4 random phải insert vào một vị trí random trong tree. Với bảng lớn không fit trong RAM, mỗi insert có thể trigger đọc disk → insert chậm gấp 3-10 lần so với auto-increment.

Hệ quả 2: *Primary key size ảnh hưởng toàn bộ database*

PK value được copy vào mỗi secondary index entry. Tính toán cụ thể:

```
Bảng: 1 triệu row, 5 secondary index
PK = BIGINT (8 bytes):     5 index × 1M × 8 bytes = 40 MB overhead
PK = ULID string (26 bytes): 5 index × 1M × 26 bytes = 130 MB overhead
Chênh lệch: 90 MB: chỉ cho MỘT bảng!

Với 50 bảng tương tự: chênh lệch 4.5 GB
→ Có thể là khác biệt giữa "fit trong RAM" và "phải đọc disk"
```

Thực tiễn: Dùng auto-increment integer hoặc UUIDv7/ULID (binary format, không phải string) làm primary key là lựa chọn an toàn nhất. Nếu dùng ULID/UUIDv7, hãy lưu dạng `BINARY(16)` thay vì `CHAR(36)` để tiết kiệm space.

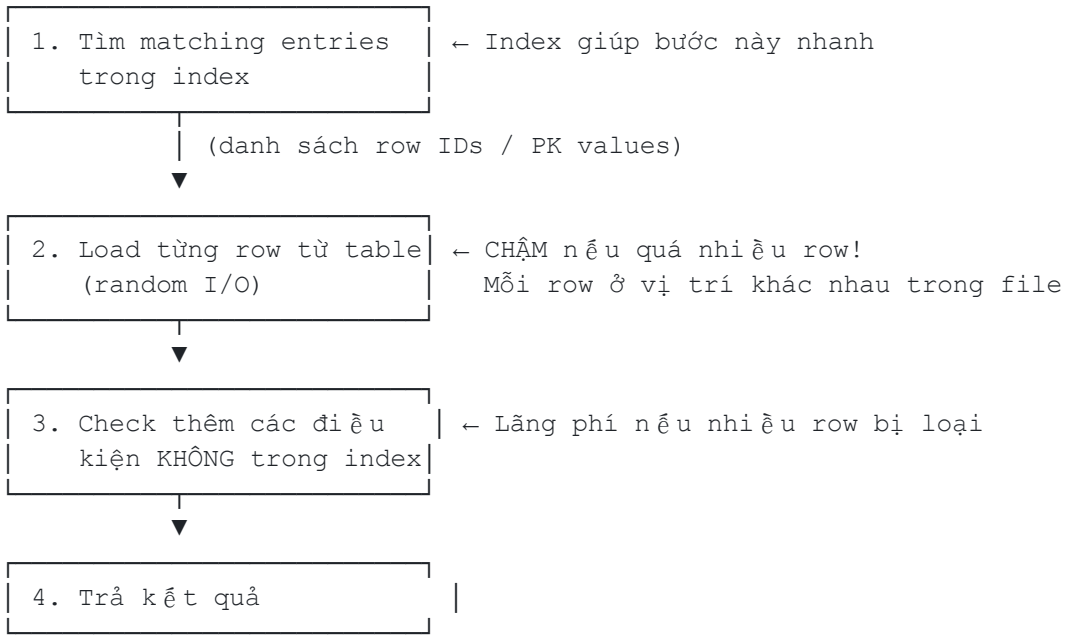
Hệ quả 3: *PK lookup cực nhanh: tận dụng cho CRUD apps*

Nếu app chủ yếu là CRUD (tìm user theo ID, lấy order theo ID...), clustered index cho performance tuyệt vời vì data có sẵn ngay khi tìm thấy PK entry. Đây là lý do MySQL/InnoDB là lựa chọn phổ biến cho web applications.

Có index chưa chắc query nhanh: Hiểu lầm phổ biến nhất

Đây là điều nhiều người không nhận ra: dùng index không đảm bảo query nhanh. Cảm giác "tôi đã tạo index rồi mà vẫn chậm" xuất phát từ việc không hiểu quy trình thực tế.

Quy trình khi database dùng index:



Bước 2 và 3 chính là nơi query có thể chậm. Hãy xem một ví dụ thực tế:

```
-- Bảng orders: 5 triệu rows
-- Index: chỉ có trên (status)

SELECT * FROM orders
WHERE status = 'pending'      -- index lọc: 5M → 200,000 rows
  AND region = 'southeast'    -- KHÔNG trong index
  AND total > 1000000;        -- KHÔNG trong index

-- Chuyện gì xảy ra:
-- 1. Index tìm 200,000 entries có status = 'pending'        ← nhanh
-- 2. Load 200,000 rows từ table (random I/O)                ← CHẬM!
-- 3. Check region = 'southeast' → còn 15,000 rows           ← lãng phí 185,000
lần đọc
-- 4. Check total > 1000000 → còn 500 rows                   ← lãng phí thêm
14,500 lần
-- Kết quả: chỉ cần 500 rows nhưng đã load 200,000 rows!
```

Giải pháp: Tạo index bao phủ nhiều điều kiện hơn:

```
-- Index tốt hơn: (status, region, total)
-- 1. status = 'pending' → fast lookup
-- 2. region = 'southeast' → tiếp tục lọc trong index (không cần load row)
-- 3. total > 1000000 → scan range trên index (vẫn không load row)
-- 4. Chỉ load ~500 rows thực sự cần → nhanh hơn RẤT NHIỀU

-- Hoặc thậm chí index-only nếu chỉ cần vài cột:
-- Index: (status, region, total) INCLUDE (order_id, customer_id)
-- → Không cần load row từ table nào cả!
```

Quy tắc thực hành: Sau khi tạo index, luôn chạy `EXPLAIN` (PostgreSQL) hoặc `EXPLAIN ANALYZE` (MySQL) để xác nhận database thực sự dùng index đó, và kiểm tra số row phải load từ table (rows examined/filtered). Nếu số row examined >> số row trả về, index chưa đủ tốt.

Bốn nguyên tắc vàng khi sử dụng Index

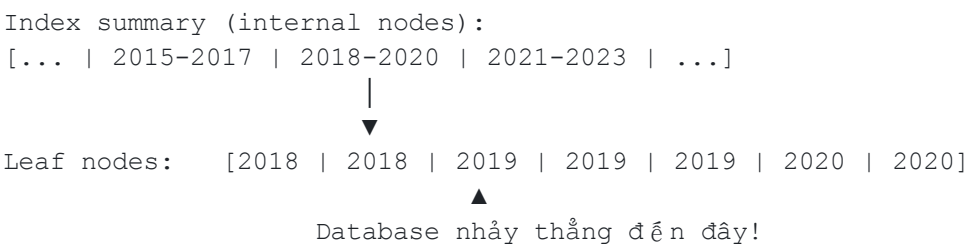
Đây là phần quan trọng nhất của cả cuốn ebook. Bốn nguyên tắc này là tất cả những gì bạn cần để tạo index tốt cho bất kỳ query nào. Khi bạn gặp một query chậm, hãy vẽ ra giấy (hoặc hình dung trong đầu) cách query đó map vào index theo 4 nguyên tắc này: đó là cách tốt nhất để xác định index cần tạo.

Nguyên tắc 1: Tra cứu nhanh: Nhảy thẳng đến vị trí cần tìm

Thao tác cơ bản nhất của index: tìm một giá trị cụ thể gần như tức thì bằng cách nhảy qua các tầng internal node, thay vì scan từ đầu đến cuối.

```
SELECT * FROM movies WHERE release_year = 2019;
```

Hình dung index trên `release_year` là một cuốn sách sorted:



Database không cần đọc qua 2015, 2016, 2017, 2018... Nó nhảy thẳng đến khu vực 2018-2020 trong index summary, rồi tìm chính xác 2019 trong leaf nodes.

Hiểu nhầm phổ biến: **"Index càng lớn thì query càng chậm"**

Chưa chắc! Hãy nhớ index là cấu trúc cây (tree), không phải danh sách phẳng. Mỗi tầng internal node chia dữ liệu thành hàng trăm nhánh. Kết quả:

Số rows	Số bước nhảy (tree depth)
1,000	~2
1,000,000	~3

1,000,000,000	~4

Từ 1 nghìn lên 1 tỷ row, chỉ thêm 2 bước nhảy. Đó là sức mạnh của O(log n). Index đã được tối ưu hóa suốt hàng chục năm cho đúng use case này: đừng lo lắng về kích thước.

Nguyên tắc 2: Quét theo một hướng

Sau khi nhảy đến một vị trí trong index (bằng Fast Lookup), database có thể tiếp tục đọc liên tục theo một hướng (ascending hoặc descending). Vì leaf nodes của B+ tree được liên kết với nhau (linked list), việc di chuyển sang entry tiếp theo là cực kỳ nhanh.

```
SELECT * FROM users WHERE age >= 35 ORDER BY age ASC LIMIT 3;
```

```
Index (age):
[18 | 22 | 25 | 28 | 30 | 35 | 37 | 42 | 48 | 55 | 61]
                          ▲
                Fast Lookup: age >= 35
                          |
                          ├──→ 35 ✅ (lấy)
                          ├──→ 37 ✅ (lấy)
                          ├──→ 42 ✅ (lấy, đủ 3 → DỪNG!)
                          |    48, 55, 61... (không cần đọc)
```

Tương tự với hướng ngược lại:

```
SELECT * FROM users WHERE age <= 35 ORDER BY age DESC LIMIT 3;
-- Fast lookup đến 35, scan ngược: 35 → 30 → 28 → DỪNG!
```

Sức mạnh thực sự khi kết hợp với LIMIT:

Không có index, query `WHERE age >= 35 ORDER BY age ASC LIMIT 3` trên bảng 10 triệu row phải: scan toàn bộ 10M rows → filter → sort → lấy 3. Với index, chỉ cần: nhảy đến 35 → đọc 3 entries → xong. Chênh lệch có thể từ vài giây xuống dưới 1ms.

Nhưng nhớ: Scan chỉ đi một hướng. Không thể vừa scan ascending vừa scan descending cùng lúc trong một index traversal. Nếu query cần sort theo 2 cột với hướng khác nhau (vd: `ORDER BY score DESC, created_at ASC`), bạn cần tạo index với đúng thứ tự sort đó (xem thêm ở Phần 3 - ORDER BY).

Nguyên tắc 3: Từ trái sang phải: Nguyên tắc "Phễu" cho index nhiều cột

Đây là nguyên tắc quan trọng nhất và cũng dễ hiểu sai nhất. Single-column index đơn giản, nhưng multi-column index (composite index) mới là nơi mang lại cải thiện performance lớn nhất: và cũng là nơi dễ sai nhất. Hãy hình dung multi-column index như một cái phễu (funnel) lọc dữ liệu từ trái sang phải.

Cách multi-column index được sắp xếp:

Index trên `(country, lastname, firstname)` sắp xếp dữ liệu theo nguyên tắc: sort by country trước, trong mỗi country sort by lastname, trong mỗi lastname sort by firstname.

```
Index (country, lastname, firstname):

┌─────────┬──────────┬───────────┐
│ country │ lastname │ firstname │
├─────────┼──────────┼───────────┤
│ JP      │ Sato     │ Kenji     │
│ JP      │ Suzuki   │ Yuki      │
│ JP      │ Tanaka   │ Hiroshi   │
│ US      │ Johnson  │ Emily     │
│ US      │ Smith    │ Alice     │
│ US      │ Smith    │ Bob       │
│ VN      │ Le       │ Minh      │
│ VN      │ Nguyen   │ An        │  ← target
│ VN      │ Nguyen   │ Huy       │  ← target
│ VN      │ Tran     │ Duc       │
└─────────┴──────────┴───────────┘
```

Bạn có thể thấy: trong mỗi country, các lastname được sorted. Nhưng nhìn toàn bộ cột lastname, nó KHÔNG sorted (Sato, Suzuki, Tanaka, Johnson, Smith...). Đây chính là lý do tại sao bạn phải đi từ trái sang phải.

Phễu hoạt động thế nào:

```
WHERE country = 'VN' AND lastname = 'Nguyen' AND firstname = 'Huy'

Bước 1: country = 'VN'          → Phễu thu hẹp: 4 entries (VN block)
Bước 2: lastname = 'Nguyen'     → Phễu thu hẹp: 2 entries (An, Huy)
Bước 3: firstname = 'Huy'       → Phễu thu hẹp: 1 entry (chính xác!)
```

Mỗi bước "thắt" phễu lại, giảm số entries phải xét. Rất hiệu quả!

Các query dùng được index này:

```
-- ✅ Dùng 3/3 bước phễu
WHERE country = 'VN' AND lastname = 'Nguyen' AND firstname = 'Huy';

-- ✅ Dùng 2/3 bước phễu (firstname bị bỏ → vẫn OK, chỉ kém tối ưu hơn)
WHERE country = 'VN' AND lastname = 'Nguyen';

-- ✅ Dùng 1/3 bước phễu
WHERE country = 'VN';

-- ❌ KHÔNG dùng được! (bỏ qua country: cột đầu tiên)
WHERE lastname = 'Nguyen';
-- Vì: lastname 'Nguyen' nằm rải rác ở JP, US, VN... không nằm liền nhau

-- ❌ KHÔNG dùng được! (bỏ qua cả country và lastname)
WHERE firstname = 'Huy';
```

Câu thần chú: "Từ trái sang phải, không bỏ qua cột": hãy khắc câu này vào đầu.

Hiểu nhầm phổ biến: "Đặt cột selective nhất lên đầu"

Bạn sẽ thấy nhiều người khuyên rằng: "đặt cột có nhiều distinct values nhất (selective nhất) lên đầu index". Hãy xem tại sao nó chưa chắc đã luôn đúng:

Giả sử bạn đổi thứ tự index thành `(lastname, firstname, country)` vì lastname có nhiều giá trị nhất. Query `WHERE country = 'VN' AND lastname = 'Nguyen' AND firstname = 'Huy'` vẫn dùng được toàn bộ phễu: vì dùng đủ 3 cột. Số bước phễu giống nhau, selectivity ở đây không tạo ra khác biệt.

Nhưng bây giờ, query `WHERE country = 'VN'` (rất phổ biến trong app multi-tenant) hoàn toàn không dùng được index này! Vì country ở vị trí cuối.

Cách đúng: Thứ tự cột nên được quyết định bởi tập hợp các query mà app bạn chạy, nhằm tối đa hóa số query được phục vụ bởi cùng một index:

```
-- App chạy các query này:
-- Q1: WHERE country = 'VN'                                    (rất thường xuyên)
-- Q2: WHERE country = 'VN' AND lastname = 'Nguyen'            (thường xuyên)
-- Q3: WHERE country = 'VN' AND lastname = 'Nguyen' AND firstname = 'Huy' (ít
hơn)

-- Index (country, lastname, firstname) → phục vụ CẢ 3 query
-- Index (lastname, firstname, country) → chỉ phục vụ Q3 tốt, Q1 không dùng
được
-- Index (firstname, lastname, country) → chẳng phục vụ tốt query nào ở trên
```

Bỏ qua cột giữa: Vẫn dùng được, nhưng kém hiệu quả:

```
-- Index: (firstname, lastname, country)
-- Query: WHERE firstname = 'Huy' AND country = 'VN'
-- (bỏ qua lastname ở giữa)
```

Database vẫn dùng index này! Nhưng cách nó hoạt động kém tối ưu hơn:

```
Bước 1: firstname = 'Huy' → Fast Lookup, tìm được block entries cho 'Huy'
Bước 2: lastname bị bỏ qua → KHÔNG thể dùng phễu tiếp
Bước 3: Scan TOÀN BỘ entries có firstname = 'Huy',
         kiểm tra TỪNG entry xem country = 'VN' không

Index entries cho firstname = 'Huy':
[Huy | Le    | JP ] → country != 'VN' ❌ (bỏ)
[Huy | Nguyen| US ] → country != 'VN' ❌ (bỏ)
[Huy | Nguyen| VN ] → country  = 'VN' ✅ (giữ)
[Huy | Tran  | JP ] → country != 'VN' ❌ (bỏ)
[Huy | Tran  | VN ] → country  = 'VN' ✅ (giữ)
→ Scan 5 entries, giữ 2
```

So sánh: index hoàn hảo `(firstname, country)` chỉ cần scan 2 entries (đúng entries cần). Với bảng lớn, "firstname = 'Huy'" có thể match hàng trăm nghìn entries → scan tất cả để filter country là rất lãng phí.

Tuy nhiên, skipping a column vẫn tốt hơn không có index: vì ít nhất database giới hạn được phạm vi scan (chỉ entries có firstname = 'Huy'), và có thể filter trên country ngay trong index mà không cần load row từ table.

Index trùng lặp: Dọn dẹp index thừa:

Mỗi index phải được cập nhật khi write, nên index thừa = tốn tài nguyên vô ích. Quy tắc:

```
✅ Index (country, lastname, firstname) BAO GỒM chức năng của:
   - (country)
   - (country, lastname)
   → Xóa 2 index đơn/đôi này đi nếu tồn tại

❌ Nhưng KHÔNG BAO GỒM:
   - (country, lastname, telephone) → cột cuối khác
   - (lastname, country)            → thứ tự cột khác
   → Đây là các index độc lập, không thể thay thế
```

Ví dụ thực tế dọn dẹp:

```
-- Bảng users hiện có 4 index:
-- idx_1: (tenant_id)
-- idx_2: (tenant_id, email)
-- idx_3: (tenant_id, created_at)
-- idx_4: (email)

-- Phân tích:
-- idx_1 bị bao gồm bởi idx_2 (cùng prefix) → XÓA idx_1
-- idx_2 và idx_3 có prefix giống nhưng cột 2 khác → GIỮ cả hai
-- idx_4 phục vụ query WHERE email = '...' (không có tenant_id) → GIỮ

-- Kết quả: giữ idx_2, idx_3, idx_4. Xóa idx_1.
```

Nguyên tắc 4: Quét khi gặp điều kiện phạm vi: Điều kiện phạm vi "phá vỡ" phễu

Đây là nguyên tắc hay bị bỏ sót nhất, nhưng lại ảnh hưởng performance rất lớn. Khi gặp range condition (`>`, `<`, `>=`, `<=`, `BETWEEN`), database chuyển sang chế độ scan: và từ lúc đó, phễu không thể thu hẹp thêm bằng các cột phía sau.

Tại sao range condition "phá vỡ" phễu?

Hãy nhìn index `(country, age, married)` với query:

```
WHERE country = 'VN' AND age > 28 AND married = 'yes'
```

```
Index (country, age, married):
┌──────────┬─────┬──────────┐
│ country  │ age │ married  │
├──────────┼─────┼──────────┤
│ VN       │ 25  │ no       │
│ VN       │ 27  │ yes      │
│ VN       │ 29  │ no       │  ← age > 28 bắt đầu scan từ đây
│ VN       │ 29  │ yes      │  ← married = 'yes' ✅
│ VN       │ 31  │ no       │  ← married = 'no'  ❌ (vẫn phải đọc!)
│ VN       │ 31  │ yes      │  ← married = 'yes' ✅
│ VN       │ 35  │ no       │  ← married = 'no'  ❌ (vẫn phải đọc!)
│ VN       │ 42  │ yes      │  ← married = 'yes' ✅
└──────────┴─────┴──────────┘
```

Sau khi `age > 28` bắt đầu scan, các entry có `married = 'no'` và `married = 'yes'` xen kẽ nhau. Database không thể "nhảy qua" các entry `married = 'no'`: nó phải đọc từng entry và check. Đọc 6 entries nhưng chỉ giữ 3.

Giờ đổi thứ tự: index `(country, married, age)`:

```
Index (country, married, age):
┌─────────┬─────────┬─────┐
│ country │ married │ age │
├─────────┼─────────┼─────┤
│ VN      │ no      │ 25  │
│ VN      │ no      │ 29  │  ← married = 'no' → bỏ qua toàn bộ block!
│ VN      │ no      │ 31  │
│ VN      │ no      │ 35  │
│ VN      │ yes     │ 27  │
│ VN      │ yes     │ 29  │  ← Fast lookup đến đây: country='VN',
married='yes', age>28
│ VN      │ yes     │ 31  │  ← scan
│ VN      │ yes     │ 42  │  ← scan
└─────────┴─────────┴─────┘
```

Bây giờ: Fast Lookup qua 2 bước phẫu (country → married) → scan từ `age > 28`. Chỉ đọc 3 entries thay vì 6! Và không cần filter thêm gì.

Ảnh hưởng thực tế: Với bảng nhỏ, chênh lệch không đáng kể. Nhưng hãy tưởng tượng bảng 10 triệu users:

- Index sai `(country, age, married)`: scan 500,000 entries cho `age > 28`, filter ra 250,000 → đọc gấp đôi cần thiết
- Index đúng `(country, married, age)`: scan đúng 250,000 entries → không lãng phí

Khi có NHIỀU range condition:

```
WHERE country = 'VN' AND age > 25 AND salary > 20000000
```

Chỉ một cột range có thể hưởng lợi từ index scan. Cột range thứ hai chỉ dùng để filter:

```
-- Index: (country, age, salary)
-- country → phẫu, age > 25 → scan, salary > 20M → filter (KHÔNG giới hạn scan)

-- Index: (country, salary, age)
-- country → phẫu, salary > 20M → scan, age > 25 → filter

-- Chọn cột nào đặt trước? Cột nào filter được NHIỀU row hơn!
-- Nếu 90% users có age > 25 nhưng chỉ 10% có salary > 20M
-- → Đặt salary trước: (country, salary, age)
```

Ngoại lệ: Loose Index Scan / Skip Scan

Một số database có thể "nhảy qua" entries trong trường hợp đặc biệt (thường là GROUP BY min/max):

- MySQL: Loose Index Scan
- Oracle / SQL Server: Skip Scan
- PostgreSQL: chưa hỗ trợ

Nhưng đây là tối ưu đặc biệt, không phải hành vi mặc định. Đừng thiết kế index dựa trên nó.

Quy tắc vàng cần nhớ:

1. ***Equality columns trước, Range columns sau***
2. ***Nếu có nhiều range conditions, đặt cột filter được nhiều nhất trước***
3. ***Sau cột range đầu tiên, các cột tiếp theo chỉ dùng để filter (vẫn hữu ích, nhưng không giới hạn scan range)***

Index hoạt động thế nào với từng thao tác SQL

Bốn nguyên tắc ở Phần 2 là nền tảng. Giờ chúng ta sẽ xem chúng hoạt động thế nào với các SQL operation phức tạp hơn. Một điều quan trọng cần nhớ: thứ tự thực thi SQL không giống thứ tự bạn viết. Database thực thi theo thứ tự:

```
FROM / JOIN → WHERE → GROUP BY → HAVING → SELECT → ORDER BY → LIMIT
```

Điều này ảnh hưởng trực tiếp đến cách xây dựng index: cột WHERE phải nằm trước cột ORDER BY trong index, cột GROUP BY phải nằm trước cột SELECT aggregate, v.v.

Phép so sánh không bằng (!=): Kẻ giết hiệu suất thầm lặng

Điều kiện `!=` là một trong những thứ tệ nhất cho index. Hãy xem tại sao:

```sql
SELECT * FROM payments WHERE status != 'open';
```

Với index trên `(status)`, database phải scan toàn bộ index entries, kiểm tra từng cái xem có `!= 'open'` không: vì kết quả match nằm ở cả hai phía (trước và sau 'open'). Đây không phải range scan (chỉ scan một hướng): mà phải đọc tất cả.

```
Index (status):
[cancelled | cancelled | open | open | open | paid | paid | refunded]
     ✅         ✅        ❌     ❌     ❌     ✅     ✅       ✅
→ Phải đọc 8 entries, bỏ 3, giữ 5
→ Chẳng khác gì full table scan!
```

Tệ hơn: database không thể dự đoán bao nhiêu row match `!= 'open'` (có thể 5% hoặc 95%), nên cost model không đánh giá được → thường fall back về full table scan luôn, hoàn toàn bỏ qua index.

Cách xử lý:

Cách 1: Kết hợp thêm cột equality thu hẹp phạm vi

Đây là cách đơn giản và hiệu quả nhất:

```
SELECT * FROM payments
WHERE shop_id = 42 AND status != 'open';
-- Index: (shop_id, status)

-- Workflow:
-- 1. Fast lookup: shop_id = 42 → thu hẹp xuống vài trăm entries
-- 2. Scan entries của shop_id = 42, filter status != 'open'
-- → Dù inequality vẫn phải scan, phạm vi scan đã nhỏ hơn RẤT nhiều!
```

Quan trọng: database giờ chấp nhận dùng index vì nó biết chắc upper bound (tất cả rows của shop_id = 42) → cost model tính được.

Cách 2: Chuyển inequality thành IN(...) nếu biết trước giá trị

```
-- Nếu status chỉ có: 'open', 'paid', 'cancelled', 'refunded'
-- Thay vì: WHERE status != 'open'
-- Viết: WHERE status IN ('paid', 'cancelled', 'refunded')
-- → Database tách thành 3 equality lookups, hiệu quả hơn nhiều!
```

Cách 3: Chuyển thành boolean column/functional index

```
-- Tạo cột is_open (computed/virtual column)
-- Hoặc functional index:
CREATE INDEX payments_not_open ON payments ((status != 'open'));
-- Sau đó: WHERE (status != 'open') = true → equality check!
```

NULL: Giá trị đặc biệt cần đặc biệt chú ý

`NULL` trong SQL nghĩa là "không biết" (unknown), không phải "rỗng" hay "zero". Và nó có hàng loạt behavior kỳ lạ ảnh hưởng đến index.

Behavior quan trọng:

```
NULL = NULL     → NULL (không phải TRUE!)
NULL != NULL    → NULL (không phải TRUE!)
NULL > 5        → NULL
NULL + 10       → NULL
-- Bất kỳ phép tính nào với NULL đều trả về NULL
-- Trong context WHERE, NULL được coi như FALSE
```

NULL và Index:

SQL standard không quy định NULL đứng trước hay sau trong sort order: mỗi database tự quyết:

- MySQL: NULL đứng trước (nhỏ nhất)
- PostgreSQL: NULL đứng sau (lớn nhất)

Điều này ảnh hưởng đến vị trí NULL trong index. Nhưng quan trọng hơn:

- `IS NULL` hoạt động giống equality check. Database nhảy thẳng đến block NULL entries → mọi nguyên tắc index áp dụng bình thường:

```
-- Tìm nhân viên chưa có supervisor (supervisor_id = NULL) tên Huy
SELECT * FROM employees
WHERE supervisor_id IS NULL AND name = 'Huy';
-- Index: (supervisor_id, name) → Fast lookup NULL → tìm 'Huy' → hiệu quả!
```

- `IS NOT NULL` hoạt động giống inequality: phải scan tất cả entries khác NULL. Có thể match quá nhiều rows → database bỏ qua index.

```
-- Tìm tất cả nhân viên CÓ supervisor
SELECT * FROM employees WHERE supervisor_id IS NOT NULL;
-- Nếu 95% nhân viên có supervisor → match 95% rows → full table scan nhanh hơn
```

NULL trong phép so sánh không bằng: Cái bẫy ngầm:

Đây là bug logic rất phổ biến, không chỉ ảnh hưởng performance mà cả tính đúng đắn của kết quả:

```
-- Bảng users: có 1000 rows, trong đó 50 rows có country = NULL
```

```sql
-- Query: tìm tất cả users KHÔNG ở VN
SELECT * FROM users WHERE country != 'VN';

-- Bạn nghĩ: trả về tất cả user trừ user ở VN → 900 rows?
-- Thực tế: trả về 850 rows! 50 rows NULL bị BỎ SÓT
-- Vì: NULL != 'VN' → NULL → coi như FALSE → không có trong kết quả
```

Cách fix:

```sql
-- Cách dài:
WHERE country != 'VN' OR country IS NULL

-- MySQL (ngắn gọn):
WHERE NOT(country <=> 'VN')    -- <=> là NULL-safe equality operator

-- PostgreSQL (ngắn gọn):
WHERE country IS DISTINCT FROM 'VN'
```

NULL trong ORDER BY: Kiểm soát vị trí NULL:

```sql
-- MySQL: NULL mặc định ở đầu (ascending)
-- Muốn NULL ở cuối:
SELECT * FROM customers ORDER BY country IS NULL, country ASC;

-- PostgreSQL: NULL mặc định ở cuối (ascending)
-- Muốn NULL ở đầu:
SELECT * FROM customers ORDER BY country ASC NULLS FIRST;
-- Muốn NULL ở cuối (mặc định nhưng explicit):
SELECT * FROM customers ORDER BY country ASC NULLS LAST;
```

LIKE: Tìm kiếm mẫu chuỗi và "cái bẫy" ký tự đại diện ở đầu

Khi bạn viết `LIKE 'Nguyễn%'`, database trong nội bộ chuyển thành range condition:

```
-- LIKE 'Nguyễn%' tương đương:
WHERE firstname >= 'Nguyễn' AND firstname < 'Nguyễo'
-- (Nguyễo = ký tự kế tiếp sau 'n' trong bảng mã)
```

Vì là range condition, nó tuân theo Nguyên tắc 4: LIKE column nên đặt sau các equality columns trong index, và các cột phía sau LIKE chỉ dùng để filter.

```
-- Query: Tìm customer có tên bắt đầu bằng 'Nguyễn'
SELECT * FROM contacts
WHERE type = 'customer' AND firstname LIKE 'Nguyễn%';

-- Index analysis:
-- ✅ (type, firstname): type equality → phễu, firstname LIKE → range scan
-- ❌ (firstname, type): firstname LIKE → range scan ngay, type chỉ filter
```

Wildcard ở giữa: `LIKE 'Nguyễn%s'` (bắt đầu bằng Nguyễn, kết thúc bằng s) vẫn dùng được index: database tìm range `Nguyễn..Nguyễo`, rồi filter pattern `%s` trên kết quả.

Ký tự đại diện ở đầu: Không dùng được B-tree index:

```
SELECT * FROM contacts WHERE name LIKE '%Nguyễn%';
-- Database KHÔNG thể dùng B-tree index!
```

Tại sao? Vì `%Nguyễn%` match bất cứ vị trí nào trong string. Database không biết phải bắt đầu scan từ đâu trong sorted list: 'Nguyễn' có thể nằm trong 'Tên tôi là Nguyễn Văn A', 'Anh Nguyễn đẹp trai', 'xNguyễn123'... Chúng rải rác khắp nơi trong index. Database cũng không thể ước lượng số rows match → cost model thất bại → fallback full table scan.

Giải pháp cho leading wildcard:

PostgreSQL: Trigram Index (pg_trgm)

```
CREATE EXTENSION IF NOT EXISTS pg_trgm;
CREATE INDEX trgm_idx ON contacts USING GIN (name gin_trgm_ops);

-- Giờ query này dùng được index:
SELECT * FROM contacts WHERE name LIKE '%Nguyễn%';
```

Cách hoạt động: text được chia thành mọi chuỗi con 3 ký tự (trigram):

```
'Nguyễn Minh' → trigrams: 'ngu', 'guy', 'uyễ', 'yễn', 'ễn ', 'n m', ' mi',
'min', 'inh'

Khi tìm '%Minh%':
1. Tách chuỗi tìm kiếm thành trigram: 'min', 'inh'
2. Tìm các entry chứa TẤT CẢ trigram này → danh sách ứng viên
3. Kiểm tra lại chính xác trên ứng viên (vì trigram có thể khớp sai thứ tự)
```

Ưu điểm: hỗ trợ ký tự đại diện ở bất kỳ vị trí nào. Nhược điểm: index có thể rất lớn (mỗi chuỗi sinh ra nhiều trigram), và chuỗi tìm kiếm phải có ít nhất 3 ký tự liên tục (không có ký tự đại diện) để dùng được.

MySQL: Không có giải pháp built-in. Cân nhắc dùng Full-Text Search (`MATCH AGAINST`) hoặc external search engine (Elasticsearch, MeiliSearch) cho use case search phức tạp.

ORDER BY: Tránh bước sort bổ sung bằng mọi giá

Index không chỉ giúp tìm nhanh: nó còn giúp trả kết quả đã sorted. Nếu bạn thêm cột sort vào cuối index (sau các cột WHERE), database đọc từ index ra đã đúng thứ tự → không cần sort thêm.

```
-- Query: Hiển thị bugs nghiêm trọng nhất, mới nhất
SELECT * FROM issues
WHERE type = 'bug'
ORDER BY severity DESC, created_at DESC;

-- ✅ Index: (type, severity DESC, created_at DESC)
-- Workflow:
-- 1. Fast lookup: type = 'bug'
-- 2. Scan descending → entries đã sorted theo severity, created_at
-- 3. Trả về trực tiếp → KHÔNG cần sort bổ sung!
```

Tại sao tránh sort bổ sung lại quan trọng đến vậy?

Khi database phải sort kết quả mà index không hỗ trợ:

1. Tất cả matching rows phải được load vào memory trước
2. Nếu fit trong memory → sort nhanh (in-memory sort)
3. Nếu KHÔNG fit (vượt `sort_buffer_size` / `work_mem`) → disk-based sort:
    - Ghi data ra file tạm trên disk theo từng chunk
    - Sort từng chunk trong memory
    - Merge các chunk sorted → kết quả cuối
    - Quá trình lặp lại nhiều lần nếu data quá lớn

Disk-based sort có thể biến query 10ms thành 10 giây. Ngay cả khi dùng SSD.

Tip MySQL: `sort_buffer_size` mặc định 256KB: rất nhỏ! Tăng lên 4-8MB cho production. Tip PostgreSQL: `work_mem` mặc định 4MB. Tăng lên 32-64MB nếu có query sort nhiều data. Nhưng cẩn thận: setting này apply per-operation, nên nhiều concurrent queries có thể dùng hết RAM.

Lưu ý quan trọng với LIMIT: Nhiều người nghĩ `ORDER BY ... LIMIT 10` sẽ nhanh vì chỉ lấy 10 rows. Sai! Database vẫn phải sort toàn bộ matching rows trước, rồi mới lấy top 10. Index giúp bạn tránh sort hoàn toàn: scan 10 entries đầu tiên từ index và dừng.

Index giảm dần: Khi thứ tự sắp xếp bị "lẫn lộn":

B-tree index có thể scan cả ascending và descending (Nguyên tắc 2). Nên `ORDER BY col ASC` hay `ORDER BY col DESC` đều dùng cùng 1 index được. Nhưng khi sort nhiều cột với hướng khác nhau, bạn cần tạo index matching:

```
-- Bảng xếp hạng: score cao nhất trước, nếu bằng nhau thì ai tạo trước lên trước
SELECT * FROM highscores ORDER BY score DESC, created_at ASC LIMIT 10;

-- Index mặc định: (score, created_at) → cả hai ASC
-- Database có thể scan backward cho score DESC... nhưng created_at cũng bị DESC!
-- → Phải sort lại kết quả

-- ✅ Index đúng:
CREATE INDEX highscores_correct ON highscores (score DESC, created_at ASC);
-- Scan forward: score giảm dần, created_at tăng dần → đúng order!
```

Ví dụ thực tế: Kết hợp WHERE + ORDER BY:

```
-- E-commerce: hiển thị sản phẩm trong category, giá thấp nhất trước
SELECT * FROM products
WHERE category_id = 5 AND in_stock = true
ORDER BY price ASC
LIMIT 20;

-- Phân tích index:
-- ❌ (category_id, in_stock): filter OK, nhưng phải sort 50,000 products by price
-- ❌ (price): sort OK, nhưng phải filter TOÀN BỘ products cho category + stock
-- ✅ (category_id, in_stock, price): filter bằng phễu → scan price ascending → lấy 20 → DONE!
--    Tổng: chỉ đọc ~20 index entries. Từ nhiều giây xuống <1ms!
```

GROUP BY & DISTINCT: Thách thức lớn nhất

GROUP BY và DISTINCT là hai operation mà nhiều developer "quên" tối ưu index. Đây là sai lầm nghiêm trọng vì chúng thường phải xử lý hàng chục nghìn đến hàng triệu row. Thiếu index → performance thảm họa.

DISTINCT = GROUP BY về mặt execution. Database chuyển DISTINCT thành GROUP BY nội bộ:

```
SELECT DISTINCT country FROM users;
-- Database thực thi giống hệt:
SELECT country FROM users GROUP BY country;
-- Nên mọi quy tắc index cho GROUP BY đều áp dụng cho DISTINCT
```

Cách GROUP BY dùng index: Thuật toán "duyệt và đếm":

Khi có index phù hợp, database dùng thuật toán cực kỳ hiệu quả: vì giá trị đã sorted, nó chỉ cần scan qua index, đếm entries liên tục có cùng giá trị, khi gặp giá trị mới → kết thúc group cũ, bắt đầu group mới.

```
SELECT is_paying, COUNT(*) FROM users GROUP BY is_paying;
-- Index: (is_paying)

-- Database scan index:
-- [no | no | no | yes | yes | yes | yes]
--  └─────────┘     └──────────────────┘
--  count = 3           count = 4
-- Hiệu quả: 1 lần scan, không cần temporary table, không cần sort
```

KHÔNG có index phù hợp → database phải:

1. Scan toàn bộ table
2. Tạo hash table tạm trong memory: key = giá trị group, value = aggregate
3. Nếu hash table quá lớn cho memory → spill ra disk → rất chậm

Quy tắc tạo index cho GROUP BY:

Trường hợp 1: Simple GROUP BY

```
SELECT is_paying, gender, COUNT(*) FROM users GROUP BY is_paying, gender;
-- Index cùng cột, cùng thứ tự: (is_paying, gender)
```

Trường hợp 2: GROUP BY + WHERE

Nhớ SQL execution order: WHERE chạy trước GROUP BY. Nên cột WHERE đặt trước trong index:

```
SELECT is_paying, gender, COUNT(*) FROM users
WHERE onboarding = 'yes' GROUP BY is_paying, gender;
-- Index: (onboarding, is_paying, gender)
-- Workflow:
-- 1. Fast lookup: onboarding = 'yes'
-- 2. Scan: entries đã sorted by (is_paying, gender) → loop-and-count
```

Trường hợp 3: GROUP BY + WHERE range → CẨN THẬN!

Đây là nơi Nguyên tắc 4 (Range breaks funnel) gây ra vấn đề lớn:

```
SELECT is_paying, gender, COUNT(*) FROM users
WHERE age BETWEEN 20 AND 29 GROUP BY is_paying, gender;

Index (age, is_paying, gender):
[20|no |F] [20|yes|M] [21|no |M] [21|yes|F] [22|no |F] ...
       ↑ is_paying và gender bị XEN KẼ giữa các giá trị age!
```

Sau range scan trên `age`, giá trị `is_paying` và `gender` không nằm liền nhau nữa: database không thể dùng thuật toán loop-and-count. Nó phải tạo temporary hash table để group → chậm hơn.

Giải pháp: Chuyển range condition thành equality (xem Phần 5). Ví dụ: tạo cột `age_group = 'twenties'` rồi `WHERE age_group = 'twenties'` → equality → phễu tiếp tục hoạt động cho GROUP BY.

Trường hợp 4: GROUP BY + Aggregate Function

Khi dùng aggregate function (AVG, SUM, MAX...) trên một cột, cột đó nên nằm cuối index để database đọc giá trị từ index mà không cần load row:

```
SELECT is_paying, gender, AVG(projects_cnt) FROM users
GROUP BY is_paying, gender;

-- ❌ Index (is_paying, gender): GROUP BY OK, nhưng projects_cnt phải load từ table
--    → Nếu 500,000 rows match → load 500,000 rows chỉ để đọc 1 cột!

-- ✅ Index (is_paying, gender, projects_cnt):
--    GROUP BY + aggregate đều chạy trên index → index-only operation
```

```
--    → 0 rows cần load từ table!
```

Tip viết GROUP BY ngắn gọn: Khi GROUP BY trên primary key, bạn không cần liệt kê các cột khác của cùng bảng:

```
-- Thay vì viết dài dòng:
SELECT actors.firstname, actors.lastname, COUNT(*)
FROM actors JOIN actors_movies USING(actor_id)
GROUP BY actors.id, actors.firstname, actors.lastname;

-- Viết gọn:
SELECT actors.firstname, actors.lastname, COUNT(*)
FROM actors JOIN actors_movies USING(actor_id)
GROUP BY actors.id;  -- Primary key là đủ! Database tự thêm các cột khác.
```

JOIN: Phân tách và kết hợp

Join trông phức tạp: nhiều bảng, nhiều điều kiện, khó biết cần index gì. Nhưng khi bạn hiểu cách database thực sự thực thi join, mọi thứ trở nên rõ ràng.

Vòng lặp lồng nhau: Cách database thực thi join

Cách cơ bản nhất (và phổ biến nhất) là nested-loop join: hoạt động y hệt vòng lặp for-each:

```
SELECT employee.* FROM employee
JOIN department USING(department_id)
WHERE employee.salary > 100000 AND department.country = 'NR';
```

Database thực thi giống pseudo-code này:

```
# Bước 1: Query bảng "driving" (bảng được chọn chạy trước)
results_employee = SELECT * FROM employee WHERE salary > 100000

# Bước 2: Với MỖI row, query bảng "driven" (bảng trong vòng lặp)
for emp in results_employee:
    result = SELECT * FROM department
             WHERE department_id = emp.department_id
               AND country = 'NR'
    if result:
        output(emp)  # Thêm vào kết quả
```

Bây giờ bạn thấy: join = hai query độc lập, một chạy 1 lần (driving), một chạy N lần trong loop (driven). Mỗi query cần index riêng:

```
-- Index cho driving table (employee):
CREATE INDEX idx_emp_salary ON employee (salary);

-- Index cho driven table (department):
-- department_id đến từ join condition, country từ WHERE
CREATE INDEX idx_dept ON department (department_id, country);
-- Hoặc (country, department_id): cả hai equality nên thứ tự không quan trọng
```

Thứ tự JOIN không cố định: Tại sao điều này lại quan trọng:

SQL là declarative language: bạn nói muốn gì, database tự quyết làm thế nào. Thứ tự bạn viết JOIN trong SQL không phải thứ tự database thực thi.

Xem data thực tế:

```
employee: 10,000 rows tổng. 511 rows có salary > $100k
department: 500 rows tổng. 2 departments ở Nauru (country = 'NR')
```

Approach	Driving table	Loop iterations	Tổng operations
Employee first	employee (salary > 100k)	511 rows → 511 lookups vào department	~512
Department first	department (country = 'NR')	2 rows → 2 lookups vào employee	~3

Cách tiếp cận 2 nhanh hơn 170 lần! Database sẽ tự chọn approach 2 nếu:

1. Statistics cho thấy 'NR' chỉ match 2 departments
2. Index phù hợp tồn tại trên bảng employee cho reverse lookup

Nếu bạn chỉ tạo index cho approach 1 mà không tạo cho approach 2, database buộc phải dùng approach chậm hơn:

```
-- Index cho cả hai hướng:
-- Approach 1 (employee → department):
CREATE INDEX ON employee (salary);
CREATE INDEX ON department (department_id, country);

-- Approach 2 (department → employee):
CREATE INDEX ON department (country);
CREATE INDEX ON employee (department_id, salary);
-- department_id từ join condition, salary từ WHERE
```

Quy tắc: Luôn tạo index hỗ trợ join theo mọi thứ tự có thể. Nếu thiếu index cho một hướng, database mất đi lựa chọn tối ưu.

Số lượng bảng join ảnh hưởng optimizer:

Join 2 bảng → 2 thứ tự có thể. Join 4 bảng → 24 thứ tự × số index = rất nhiều permutations. Database không thể kiểm tra hết → có thể chọn plan không tối ưu. Nên giữ số bảng join dưới 6-8 cho performance dự đoán được.

Lateral Join: Vòng lặp "cho từng dòng" trong SQL:

Đây là tính năng cực kỳ mạnh mà ít developer biết đến. Normal JOIN chỉ link rows dựa trên condition: không thể control bao nhiêu rows từ mỗi nhóm. Lateral Join giải quyết điều này:

```
-- Use case: Dashboard hiển thị mỗi customer + 3 đơn hàng gần nhất
SELECT customers.*, recent_sales.*
FROM customers
LEFT JOIN LATERAL (
    SELECT * FROM sales
    WHERE sales.customer_id = customers.id
    ORDER BY created_at DESC
    LIMIT 3
) AS recent_sales ON true;

-- Lateral subquery chạy cho MỖI customer:
-- Customer 1 → SELECT TOP 3 sales WHERE customer_id = 1
-- Customer 2 → SELECT TOP 3 sales WHERE customer_id = 2
-- ...

-- Index cần: (customer_id, created_at DESC) trên bảng sales
-- → Mỗi lateral subquery chỉ cần đọc 3 index entries
```

So sánh với cách truyền thống (dùng ROW_NUMBER):

```
-- Cách cũ: load TẤT CẢ sales, đánh số thứ tự, filter
WITH ranked AS (
    SELECT *, ROW_NUMBER() OVER(PARTITION BY customer_id ORDER BY created_at
DESC) as rn
    FROM sales
)
SELECT * FROM customers JOIN ranked ON ... WHERE rn <= 3;
-- Phải đọc TOÀN BỘ bảng sales rồi mới filter → chậm hơn nhiều
```

Lateral Join cũng hoạt động cho: "lấy giá trị mới nhất cho mỗi group", "lấy aggregate cho mỗi entity", v.v.

Subquery: Không chậm như bạn nghĩ

"Subquery chậm" là một thiên kiến khá phổ biến. Chúng chậm vì thiếu index, không phải vì bản chất của subquery.

Independent Subquery: chạy 1 lần duy nhất, kết quả thay thế vào query chính:

```
SELECT * FROM products
WHERE remaining > 500 AND category_id = (
    SELECT category_id FROM categories
    WHERE type = 'book' AND name = 'Science fiction'
);
-- Index cho subquery: (type, name) trên bảng categories
-- Index cho outer query: (category_id, remaining) trên bảng products
```

Dependent Subquery: chạy lặp lại cho mỗi row, giống join:

```
SELECT * FROM products
WHERE remaining = 0 AND EXISTS (
    SELECT * FROM sales
    WHERE created_at >= '2023-01-01' AND product_id = products.product_id
);
-- Index cho outer: (remaining) trên products
-- Index cho subquery: (product_id, created_at) trên sales
```

Tip: `EXISTS` tự động dừng sau khi tìm thấy 1 row match (giống có `LIMIT 1` ngầm). Nhưng vẫn nên tạo index theo nguyên tắc range condition để tối ưu.

UPDATE & DELETE: Đừng quên tối ưu cho chúng

UPDATE và DELETE cũng cần tìm row trước khi thao tác. Phần "tìm row" hoàn toàn giống SELECT và cần index y hệt. Bạn có thể viết lại thành SELECT để test performance:

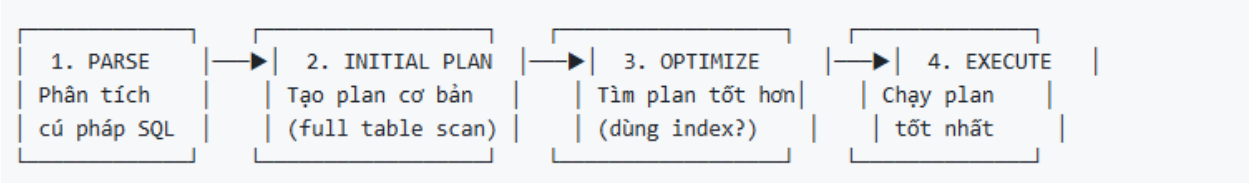
```
-- Thay vì test trực tiếp:
DELETE FROM logs WHERE created_at < '2024-01-01';

-- Viết thành SELECT để kiểm tra:
SELECT * FROM logs WHERE created_at < '2024-01-01';
-- Nếu SELECT chậm → DELETE cũng chậm → cần index trên (created_at)
```

Tại sao Database không dùng Index của tôi

Đây là câu hỏi gây bực bội nhất mà developer hay gặp. Index đã tạo, query rõ ràng match: nhưng database vẫn lờ tịt. Để debug được, bạn cần hiểu quy trình ra quyết định của database.

Quy trình thực thi query: Bên trong "bộ não" của database

Mỗi query đi qua 4 bước:



Điểm quan trọng ở bước 2-3: database luôn bắt đầu với plan "full table scan" (vì nó luôn chạy được, không cần index). Sau đó optimizer kiểm tra xem có plan nào tốt hơn không. Nếu không tìm được plan tốt hơn → giữ full table scan. Đây là lý do index đôi khi bị "bỏ qua": không phải database không thấy index, mà nó đánh giá full scan nhanh hơn.

Công cụ debug: EXPLAIN

Trước khi tìm hiểu từng lý do, hãy biết cách xem database đang làm gì:

```
-- PostgreSQL:
EXPLAIN ANALYZE SELECT * FROM users WHERE email = 'test@example.com';
-- Kết quả cho thấy: plan được chọn, số row ước tính vs thực tế, thời gian

-- MySQL:
EXPLAIN SELECT * FROM users WHERE email = 'test@example.com';
-- Xem cột 'type':
--   ALL = full table scan (tệ)
--   index = full index scan
--   range = range scan trên index (khá)
--   ref = index lookup (tốt)
--   const = tìm đúng 1 row qua unique index (tuyệt vời)
```

Hãy tập thói quen chạy EXPLAIN cho mọi query quan trọng trước khi deploy lên production.

Index không khớp với query: Lý do phổ biến nhất

Biến đổi cột (Column Transformation): Sai lầm #1:

Bất kỳ phép biến đổi nào áp dụng lên cột (không phải giá trị) đều khiến index bị "mù":

```
-- ❌ Hàm trên cột → index trên birthday VÔ DỤNG
SELECT * FROM contacts WHERE YEAR(birthday) = 1988;
-- Database thấy: YEAR(birthday): một biểu thức, không phải cột birthday
-- Index trên birthday lưu '1988-03-15', '1990-07-22'... không phải 1988,
1990...
-- → Không map được!

-- ✅ Viết lại: dùng range condition trên cột gốc
SELECT * FROM contacts
WHERE birthday >= '1988-01-01' AND birthday < '1989-01-01';
-- Giờ database thấy: cột birthday, range condition → dùng index!
```

Tại sao database không tự động rewrite? Vì sẽ phải xử lý mọi tổ hợp hàm có thể: cực kỳ phức tạp và dễ sai. Nên nguyên tắc được giữ đơn giản: biến đổi cột = không dùng index. Không ngoại lệ.

Các trường hợp hay gặp:

```
-- ❌ Phép tính trên cột
WHERE col + 5 < 20       → ✅ WHERE col < 15

-- ❌ Nối chuỗi trên cột
WHERE CONCAT(first, ' ', last) = 'Huy Nguyen'
→ ✅ WHERE first = 'Huy' AND last = 'Nguyen'

-- ❌ Ép kiểu trên cột (MySQL - type juggling)
WHERE varchar_col = 12345   → ✅ WHERE varchar_col = '12345'

-- ❌ Hàm ngày trên cột
WHERE DATE(created_at) = '2024-01-15'
→ ✅ WHERE created_at >= '2024-01-15' AND created_at < '2024-01-16'

-- ❌ Lower/Upper trên cột
WHERE LOWER(email) = 'test@example.com'
→ ✅ Dùng functional index (xem Phần 5.1)
```

Thứ tự cột không khớp:

Index `(firstname, lastname, country)` không dùng được cho `WHERE country = 'VN'` nếu query không có điều kiện trên `firstname` (Nguyên tắc 3).

Index ẩn (Invisible Index): MySQL:

Đôi khi index tồn tại nhưng bị đánh dấu "invisible". Một số tool hiển thị chúng bình thường mà không cho biết chúng đang ẩn. Kiểm tra:

```
-- MySQL: xem index nào bị invisible
SELECT INDEX_NAME, IS_VISIBLE FROM INFORMATION_SCHEMA.STATISTICS
WHERE TABLE_NAME = 'your_table';
```

Full table scan nhanh hơn: Khi database đúng mà bạn sai

Đây là trường hợp hay gặp nhất nhưng ít được hiểu: index hoàn toàn match query, nhưng database vẫn chọn full table scan. Và database đúng!

Hiểu về Cost Model:

Database lưu trữ thống kê (statistics) về mỗi cột: có bao nhiêu giá trị khác nhau (distinct values), phân phối giá trị ra sao (histogram), tổng số row, v.v. Dựa trên thống kê này, nó tính chi phí ước tính cho từng plan:

```
Ví dụ đơn giản: bảng 10,000 rows

Plan A: Full table scan
  - Đọc sequential: 10,000 rows × 0.01 cost/row = 100

Plan B: Dùng index, match 5,000 rows
  - Index lookup: 5,000 entries × 0.005 cost/entry = 25
  - Load rows (random I/O): 5,000 rows × 0.04 cost/row = 200
  - Tổng: 225

Plan A (100) < Plan B (225) → Full table scan THẮNG!
```

Điểm mấu chốt: random I/O đắt hơn sequential I/O rất nhiều. Khi phải load nhiều row từ table (mỗi row ở vị trí khác nhau trên disk), chi phí nhảy qua nhảy lại vượt quá chi phí đọc tuần tự cả bảng.

Ngưỡng chuyển đổi:

Khi query match khoảng 10-30% row trở lên, full table scan thường nhanh hơn. Con số chính xác phụ thuộc vào:

- Loại disk (SSD nhanh hơn HDD ở random I/O)
- Kích thước row (row to = ít row trên mỗi page = nhiều random I/O hơn)
- Bảng có fit trong RAM không

Tip PostgreSQL quan trọng: Config `random_page_cost` mặc định là 4.0: tối ưu cho HDD. Nếu bạn dùng SSD hoặc data fit trong RAM, đổi thành `1.1`:

```
SET random_page_cost = 1.1;
```

Điều này khiến optimizer "tin tưởng" random I/O hơn → dùng index thường xuyên hơn.

Bảng nhỏ (100-200 rows):

Database thường full scan luôn: overhead của việc tra cứu index (đọc internal nodes, nhảy đến leaf, rồi nhảy đến table) lớn hơn lợi ích so với đọc thẳng 100-200 rows. Đây là hành vi bình thường, không cần lo.

Thống kê cũ: Kẻ phá hoại ngầm:

Thống kê được tính dựa trên mẫu (sample) từ bảng, và chỉ được cập nhật khi vượt ngưỡng thay đổi nhất định. Sau bulk operations, thống kê có thể sai lệch nghiêm trọng:

```
Thực tế: 5% rows match query → nên dùng index
Thống kê cũ: ước tính 55% rows match → full table scan
→ Database đưa ra quyết định sai!


-- Giải pháp: luôn chạy ANALYZE sau bulk changes
-- MySQL:
ANALYZE TABLE users;
-- PostgreSQL (SKIP_LOCKED tránh block concurrent queries):
ANALYZE users;
```

Database chọn index khác: Khi có nhiều lựa chọn

Nhiều điều kiện + nhiều index đơn lẻ:

```
-- Query: WHERE firstname = 'Huy' AND lastname = 'Nguyen'
-- Index A: (firstname) → ước tính match 500 rows
-- Index B: (lastname) → ước tính match 200 rows

-- Database chọn Index B vì 200 < 500 → ít row phải load hơn
-- Giải pháp tốt nhất: tạo composite index (firstname, lastname)
-- → chỉ match đúng số row cần thiết
```

Xung đột giữa lọc và sắp xếp:

Đây là tình huống "tiến thoái lưỡng nan" kinh điển:

```
SELECT * FROM issues
WHERE type = 'open' ORDER BY created_at DESC LIMIT 10;
```

Database có 2 lựa chọn:

```
Plan A: Dùng index (type)
  1. Tìm tất cả issues type = 'open' → 50,000 rows
  2. Sort 50,000 rows theo created_at DESC
  3. Lấy top 10
  → Nhanh ở bước 1, chậm ở bước 2 (sort 50k rows)

Plan B: Dùng index (created_at DESC)
  1. Scan từ created_at mới nhất
  2. Với mỗi row, check type = 'open'
  3. Khi đủ 10 rows → dừng
  → Nếu issues open phổ biến: chỉ cần scan ~15-20 rows → CỰC NHANH
  → Nếu issues open hiếm: phải scan hàng nghìn rows → CHẬM

Plan C: Dùng index (type, created_at DESC) ← GIẢI PHÁP TỐT NHẤT
  1. Fast lookup type = 'open'
  2. Scan 10 entries theo created_at DESC
  3. Xong!
  → Luôn nhanh, bất kể phân phối dữ liệu
```

Bài học: Khi query có cả WHERE và ORDER BY, index tốt nhất bao gồm cả hai: cột filter trước, cột sort sau.

Cạm bẫy và mẹo nâng cao về Indexing

Ở các phần trước, chúng ta đã nắm được cách index hoạt động, nguyên tắc sử dụng, và lý do database đôi khi bỏ qua index. Phần này là "kho vũ khí": tập hợp các kỹ thuật nâng cao, những cái bẫy thường gặp, và các mẹo mà developer có kinh nghiệm tích lũy theo thời gian. Mỗi mục đều là một tình huống thực tế mà bạn rất có thể sẽ gặp.

Index trên biểu thức: Khi không thể viết lại query

Ở Phần 4, chúng ta biết rằng khi cột bị biến đổi (ví dụ: `YEAR(birthday)`), index trên cột gốc sẽ vô dụng. Giải pháp đầu tiên là viết lại query để tránh biến đổi cột. Nhưng không phải lúc nào cũng viết lại được.

Ví dụ: Tìm tất cả liên hệ sinh vào tháng 5

```
SELECT * FROM contacts WHERE MONTH(birthday) = 5;
```

Bạn không thể viết lại `MONTH(birthday) = 5` thành dạng `birthday BETWEEN ... AND ...` vì tháng 5 xuất hiện ở mọi năm. Đây chính là lúc cần dùng index trên biểu thức.

Cách tạo:

```
-- Tạo index trên KẾT QUẢ của hàm (lưu ý cặp ngoặc kép bên ngoài)
CREATE INDEX contacts_birthmonth ON contacts ((MONTH(birthday)));
```

Khi query dùng đúng biểu thức đó, database tự động nhận diện và sử dụng index:

```
-- Database thấy MONTH(birthday) khớp với biểu thức trong index → dùng index
SELECT * FROM contacts WHERE MONTH(birthday) = 5;
```

Minh họa quá trình:

```
Bảng contacts (500,000 dòng):
┌─────┬──────────────┬──────────────┐
│ id  │ name         │ birthday     │
├─────┼──────────────┼──────────────┤
│ 1   │ Minh         │ 1990-05-15   │  ← MONTH = 5
│ 2   │ Lan          │ 1988-03-22   │  ← MONTH = 3
│ 3   │ Hùng         │ 1995-05-01   │  ← MONTH = 5
│ 4   │ Thu          │ 1992-12-10   │  ← MONTH = 12
│ ... │ ...          │ ...          │
└─────┴──────────────┴──────────────┘

Không có index trên biểu thức:
```

```
  → Quét toàn bộ 500,000 dòng, tính MONTH() cho từng dòng
  → Chi phí: 500,000 phép tính + 500,000 lần so sánh

Có index ((MONTH(birthday))):
  Index đã tính sẵn:
  ┌─────────┬──────────────┐
  │ month   │ → row_ptr    │
  ├─────────┼──────────────┤
  │   1     │ → [12,89,...]│
  │   2     │ → [5,67,...] │
  │ ...     │ ...          │
  │   5     │ → [1,3,...]  │  ← Nhảy thẳng tới đây
  │ ...     │ ...          │
  └─────────┴──────────────┘
  → Tìm thẳng tháng 5, lấy danh sách con trỏ → nhanh gấp nhiều lần
```

Ứng dụng phổ biến khác: tìm kiếm không phân biệt hoa thường:

```sql
-- Tạo index trên email đã chuyển về chữ thường
CREATE INDEX users_email_lower ON users ((LOWER(email)));

-- Query: tìm user theo email, không phân biệt hoa thường
SELECT * FROM users WHERE LOWER(email) = 'test@example.com';
```

Nếu không có index này, mỗi lần tìm kiếm database phải gọi `LOWER()` cho tất cả email trong bảng rồi so sánh. Với index, giá trị đã được tính sẵn và sắp xếp: chỉ cần tra cứu nhanh.

Lưu ý quan trọng cho MariaDB và SQL Server:

Hai hệ quản trị này không hỗ trợ tạo index trực tiếp trên biểu thức. Thay vào đó, bạn cần tạo cột ảo (virtual/generated column):

```sql
-- MariaDB: tạo cột ảo rồi đánh index lên cột đó
CREATE TABLE contacts (
    id BIGINT PRIMARY KEY AUTO_INCREMENT,
    birthday DATETIME NOT NULL,
    birthday_month INT AS (MONTH(birthday)) VIRTUAL NOT NULL,
    INDEX contacts_birthmonth (birthday_month)
);

-- Query phải dùng tên cột ảo (MariaDB không tự nhận diện biểu thức)
SELECT * FROM contacts WHERE birthday_month = 5;
```

Với SQL Server, bạn cũng tạo cột tính toán (computed column) tương tự. Điểm khác biệt là SQL Server có thể tự nhận diện biểu thức `MONTH(birthday)` trong query và ánh xạ tới cột ảo: nhưng MariaDB thì không, bạn phải dùng đúng tên cột ảo.

Quy tắc nhớ: Biểu thức trong index phải khớp chính xác với biểu thức trong `WHERE`. Nếu index dùng `MONTH(birthday)` mà query dùng `EXTRACT(MONTH FROM birthday)`, hai biểu thức tuy cùng kết quả nhưng database sẽ không nhận diện được.

Cột giá trị ít (boolean, trạng thái): Khi index trở nên vô nghĩa

Đây là một trong những hiểu lầm phổ biến nhất: "Cột nào hay dùng trong `WHERE` thì đánh index." Nghe có vẻ đúng, nhưng với cột boolean hoặc cột có ít giá trị phân biệt, index thường không được dùng.

Tại sao?

Hãy xem một bảng `orders` với 1 triệu dòng và cột `is_processed` (boolean):

```
Phân phối dữ liệu:
┌──────────────┬──────────┬──────────────┐
│ is_processed │ Số dòng  │   Tỷ lệ      │
├──────────────┼──────────┼──────────────┤
│ TRUE         │ 800,000  │    80%       │
│ FALSE        │ 200,000  │    20%       │
└──────────────┴──────────┴──────────────┘
```

Khi bạn query `WHERE is_processed = FALSE`:

```
Phương án 1: Dùng index
  → Index tìm được 200,000 con trỏ
  → Nhảy tới 200,000 vị trí KHÁC NHAU trên đĩa (random I/O)
  → Chi phí: 200,000 × chi_phí_random_IO = RẤT CAO

Phương án 2: Quét toàn bộ bảng
  → Đọc tuần tự 1,000,000 dòng (sequential I/O)
  → Bỏ qua 800,000 dòng không khớp
  → Chi phí: 1,000,000 × chi_phí_sequential_IO = THẤP HƠN
   (vì sequential I/O nhanh hơn random I/O từ 10-100 lần)

→ Database chọn phương án 2: BỎ QUA INDEX
```

Ngay cả khi thêm `LIMIT 5` thì sao? Database tính: với 20% dòng khớp, trung bình chỉ cần đọc khoảng 25 dòng tuần tự là tìm đủ 5 kết quả. 25 lần đọc tuần tự rẻ hơn nhiều so với 5 lần nhảy qua index rồi random I/O.

Vấn đề tương tự với các cột trạng thái:

Không chỉ boolean, mọi cột có phân phối lệch đều gặp vấn đề này:

```
Ví dụ: Bảng issues của hệ thống quản lý lỗi (sau 2 năm hoạt động)
```

```
┌──────────┬──────────┬──────────┐
│ status   │ Số dòng  │ Tỷ lệ    │
├──────────┼──────────┼──────────┤
│ closed   │ 95,000   │   95%    │  ← Index vô nghĩa
│ open     │  3,000   │    3%    │  ← Index CÓ THỂ hữu ích
│ wontfix  │  2,000   │    2%    │  ← Index CÓ THỂ hữu ích
└──────────┴──────────┴──────────┘
```

Tìm `WHERE status = 'closed'` thì index vô dụng (95% dòng khớp). Nhưng tìm `WHERE status = 'open'` thì index có thể hữu ích vì chỉ 3% dòng khớp: dưới ngưỡng ngưỡng chuyển đổi.

Giải pháp: Partial Index (chỉ PostgreSQL)

Thay vì đánh index toàn bộ bảng, chỉ đánh index cho những dòng bạn quan tâm:

```
-- Chỉ index những dòng có is_processed = FALSE
CREATE INDEX orders_unprocessed
ON orders (created_at)
WHERE is_processed = FALSE;
```

Hình dung sự khác biệt:

```
Index thông thường trên (is_processed):
  → Chứa 1,000,000 entry (toàn bộ bảng)
  → Kích thước: ~30 MB
  → Database thường bỏ qua vì quá nhiều kết quả khớp

Partial index WHERE is_processed = FALSE:
  → Chỉ chứa 200,000 entry (phần chưa xử lý)
  → Kích thước: ~6 MB
  → Được sắp xếp theo created_at → lấy theo thời gian cực nhanh
  → Khi dữ liệu được xử lý xong, entry tự biến mất khỏi index


-- Query dùng partial index (PostgreSQL tự nhận diện khi WHERE khớp)
SELECT * FROM orders
WHERE is_processed = FALSE
ORDER BY created_at
LIMIT 100;
-- → Dùng partial index, cực nhanh dù bảng có hàng triệu dòng
```

Quy tắc nhớ: Chỉ đánh index cột boolean/trạng thái khi tìm kiếm giá trị hiếm (dưới vài phần trăm tổng số dòng). Với PostgreSQL, hãy ưu tiên dùng partial index: nhỏ hơn, nhanh hơn, và tự "thu gọn" khi dữ liệu thay đổi.

Biến đổi điều kiện phạm vi: Biến range thành so sánh bằng

Ở Phần 3, bạn đã biết nguyên tắc "Quét khi gặp điều kiện phạm vi": khi index gặp cột có điều kiện range (`>`, `<`, `BETWEEN`), tất cả các cột phía sau trong index không thể thu hẹp phạm vi quét nữa. Đây là hạn chế cơ bản của B+ tree.

Nhưng nếu bạn biến range thành so sánh bằng, hạn chế này biến mất.

Ví dụ: Tìm repo TypeScript phổ biến, sắp xếp theo số nhà tài trợ

```
SELECT * FROM repos
WHERE language = 'TypeScript' AND stars > 1000
ORDER BY sponsors ASC;
```

Thử xem các phương án index:

```
Phương án 1: Index (language, stars, sponsors)
```

language	stars	sponsors	
TypeScript	500	2	← bỏ qua (stars ≤ 1000)
TypeScript	800	5	← bỏ qua
TypeScript	1200	15	← khớp, nhưng...
TypeScript	1200	3	← khớp
TypeScript	2500	1	← khớp
TypeScript	5000	42	← khớp
TypeScript	8000	7	← khớp

```
                ↑ range bắt đầu ở đây
                  → sponsors KHÔNG được sắp xếp liên tục
                  → Phải sắp xếp lại sau khi lấy kết quả (filesort)

Phương án 2: Index (language, sponsors): bỏ qua stars
  → Sắp xếp đúng theo sponsors, nhưng không lọc được stars
  → Phải quét toàn bộ TypeScript rồi lọc stars ở bước sau
```

Cả hai phương án đều không lý tưởng. Nhưng nếu chúng ta biến đổi điều kiện `stars > 1000` thành một giá trị boolean?

```
-- Ý tưởng: thay vì hỏi "bao nhiêu sao?" → hỏi "có phổ biến không?" (0 hoặc 1)
-- Tạo index trên biểu thức boolean
CREATE INDEX repos_search ON repos (
    language,
    (IF(stars > 1000, 1, 0)),    -- boolean: phổ biến hay không
```

```
    sponsors                    -- sắp xếp
);

-- Query dùng đúng biểu thức
SELECT * FROM repos
WHERE language = 'TypeScript'
  AND IF(stars > 1000, 1, 0) = 1
ORDER BY sponsors ASC;
```

Bây giờ index hoạt động hoàn hảo:

```
Index (language, is_popular, sponsors):
┌────────────┬────────────┬────────────┐
│ language   │ is_popular │ sponsors   │
├────────────┼────────────┼────────────┤
│ TypeScript │     0      │     1      │  ← bỏ qua (không phổ biến)
│ TypeScript │     0      │     3      │  ← bỏ qua
│ TypeScript │     0      │     8      │  ← bỏ qua
│ TypeScript │     1      │     1      │  ← khớp ✓ sponsors = 1
│ TypeScript │     1      │     3      │  ← khớp ✓ sponsors = 3
│ TypeScript │     1      │     7      │  ← khớp ✓ sponsors = 7
│ TypeScript │     1      │    15      │  ← khớp ✓ sponsors = 15
│ TypeScript │     1      │    42      │  ← khớp ✓ sponsors = 42
└────────────┴────────────┴────────────┘
          equality ──┘  equality ──┘  sort ──┘

→ Phễu hoạt động hoàn hảo:
  Bước 1: Nhảy tới language = 'TypeScript' (equality)
  Bước 2: Nhảy tới is_popular = 1 (equality)
  Bước 3: Đọc theo thứ tự sponsors tăng dần (đã sắp xếp sẵn!)
  → KHÔNG cần filesort
```

Khi nào dùng kỹ thuật này? Khi query có cả lọc range lẫn sắp xếp, và điều kiện range có thể rút gọn thành "có/không" theo logic nghiệp vụ. Ví dụ: "giá trên 100k" → `is_expensive`, "đăng ký trên 1 năm" → `is_veteran`, "đánh giá trên 4 sao" → `is_highly_rated`.

Kiểu dữ liệu không khớp: Cái bẫy ngầm trong MySQL

Đây là lỗi cực kỳ phổ biến và rất khó phát hiện, đặc biệt trong các dự án đã chạy lâu năm.

Tình huống:

```
-- Schema: cột payment_id kiểu varchar(255)
-- (người tạo schema không chắc ID từ cổng thanh toán là số hay chuỗi)

-- Query tìm đơn hàng theo payment ID:
SELECT * FROM orders WHERE payment_id = 57013925718;
                                        ↑
                                  KHÔNG có dấu nháy → kiểu số
```

Bạn có index trên `payment_id`, nhưng query chạy chậm không tưởng. Lý do?

```
Cột payment_id: VARCHAR(255) ← kiểu chuỗi
Giá trị so sánh: 57013925718  ← kiểu số nguyên

Hai kiểu khác nhau → MySQL phải chuyển đổi một bên.
Bạn nghĩ MySQL sẽ chuyển số → chuỗi? KHÔNG!

MySQL luôn chuyển CHUỖI → SỐ khi so sánh hỗn hợp.

Kết quả: MySQL ngầm viết lại query thành:
SELECT * FROM orders WHERE CAST(payment_id AS UNSIGNED) = 57013925718;
                           ↑ cột bị biến đổi → index vô dụng!
```

Vì cột `payment_id` bị áp hàm `CAST()`, index trên cột gốc không thể sử dụng (đúng quy tắc Phần 4). Database phải quét toàn bộ bảng, chuyển đổi từng giá trị rồi so sánh.

Sửa rất đơn giản: thêm dấu nháy:

```
-- ✅ Đúng kiểu: chuỗi so sánh với chuỗi
SELECT * FROM orders WHERE payment_id = '57013925718';
                                        ↑ có dấu nháy → kiểu chuỗi
-- → Kiểu khớp → index hoạt động bình thường
```

Trường hợp ngược lại thì không sao:

```
-- Cột age kiểu INT, giá trị so sánh kiểu chuỗi
SELECT * FROM users WHERE age = '25';
-- MySQL chuyển '25' → 25 (chuyển đổi trên GIÁ TRỊ, không phải trên CỘT)
-- → Index vẫn hoạt động bình thường
```

Quy tắc: Vấn đề chỉ xảy ra khi cột là chuỗi nhưng giá trị so sánh là số. Chiều ngược lại (cột số, giá trị chuỗi) thì an toàn vì phép chuyển đổi xảy ra trên giá trị, không phải trên cột.

Mẹo phòng tránh: Khi thấy cột `VARCHAR` lưu giá trị trông giống số (mã giao dịch, mã vận đơn, số điện thoại), hãy luôn luôn dùng dấu nháy trong query. Tốt nhất, kiểm tra toàn bộ codebase xem có chỗ nào truyền số nguyên vào cột chuỗi không: đặc biệt là các query được sinh tự động bởi ORM hoặc query builder.

Truy vấn chỉ từ index: Không cần chạm vào bảng dữ liệu

Trong quá trình bình thường, khi database dùng index để tìm kết quả, nó trải qua hai bước: (1) tìm con trỏ trong index, (2) nhảy tới bảng dữ liệu để lấy toàn bộ cột. Bước 2 chính là nơi phát sinh random I/O tốn kém.

Nhưng nếu tất cả các cột cần thiết cho query đều nằm trong index, database có thể bỏ qua bước 2 hoàn toàn. Đây gọi là truy vấn chỉ từ index (index-only query).

Ví dụ đơn giản:

```
-- Bảng sessions(id, token, user_id, created_at, data, ...)
-- Index: (token, user_id)

-- Query xác thực: tìm user_id theo token
SELECT user_id FROM sessions WHERE token = 'abc123';

Luồng thông thường (KHÔNG có index-only):
  Index (token) → tìm được row_ptr
    → Nhảy tới bảng sessions
    → Đọc TOÀN BỘ dòng (id, token, user_id, created_at, data, ...)
    → Trả về chỉ user_id
    → Lãng phí: đọc cả cột data (có thể rất lớn) chỉ để lấy user_id

Luồng index-only (có index (token, user_id)):
  Index (token, user_id) → tìm được token = 'abc123'
    → user_id = 42 nằm NGAY TRONG index
    → Trả về 42, XONG
    → Không cần chạm vào bảng
    → Nhanh hơn đáng kể, đặc biệt khi bảng có nhiều cột lớn
```

Ứng dụng tuyệt vời nhất: bảng quan hệ nhiều-nhiều

Bảng trung gian (join table) trong quan hệ nhiều-nhiều là ứng viên hoàn hảo cho kỹ thuật này:

```
-- Bảng user_roles(user_id, role_id): chỉ có 2 cột

-- Tạo 2 index bao phủ cả 2 chiều truy vấn:
CREATE INDEX idx_user_roles ON user_roles (user_id, role_id);
CREATE INDEX idx_role_users ON user_roles (role_id, user_id);

Query: "Lấy tất cả vai trò của user 42"
```

```
SELECT role_id FROM user_roles WHERE user_id = 42;
→ Index (user_id, role_id) chứa ĐỦ thông tin
→ Index-only query ✓

Query: "Lấy tất cả user có vai trò admin (role_id = 1)"
SELECT user_id FROM user_roles WHERE role_id = 1;
→ Index (role_id, user_id) chứa ĐỦ thông tin
→ Index-only query ✓

Cả hai query đều KHÔNG BAO GIỜ chạm vào bảng dữ liệu.
Với bảng join có hàng triệu dòng, đây là tối ưu cực kỳ đáng kể.
```

Mệnh đề INCLUDE trong PostgreSQL: thêm cột "đi kèm" mà không ảnh hưởng cấu trúc index:

Đôi khi bạn muốn query lấy thêm một vài cột nữa, nhưng không muốn chúng ảnh hưởng đến thứ tự sắp xếp hay ràng buộc duy nhất của index.

```
-- PostgreSQL: cột price chỉ để "đọc thêm", không tham gia lookup/sort
CREATE INDEX ON invoices (customer_id, year) INCLUDE (price);

Index bình thường (customer_id, year, price):
  → Sắp xếp theo customer_id, rồi year, rồi price
  → price ảnh hưởng thứ tự index
  → Nếu là UNIQUE index: thêm price sẽ thay đổi ràng buộc duy nhất!

Index với INCLUDE (customer_id, year) INCLUDE (price):
  → Sắp xếp CHỈ theo customer_id và year
  → price được lưu kèm nhưng KHÔNG ảnh hưởng thứ tự
  → Giữ nguyên ràng buộc UNIQUE nếu có
  → Vẫn cho phép index-only query khi cần đọc price


-- Query này trở thành index-only nhờ INCLUDE:
SELECT customer_id, year, SUM(price)
FROM invoices
WHERE customer_id = 42
GROUP BY customer_id, year;
-- → Không cần chạm bảng invoices
```

MySQL không hỗ trợ `INCLUDE`. Nếu muốn hiệu ứng tương tự, bạn phải thêm cột vào cuối index bình thường: nhưng hãy cẩn thận với ràng buộc `UNIQUE`.

Quy tắc nhớ: Dùng truy vấn chỉ từ index cho những query đơn giản nhưng chạy rất thường xuyên: xác thực phiên đăng nhập, ánh xạ mã sang tên, bảng trung

gian nhiều-nhiều. Đừng lạm dụng: thêm quá nhiều cột vào index làm tăng kích thước và chậm thao tác ghi.

Lọc và sắp xếp khi JOIN: Bài toán không giải được bằng index

Đây là tình huống mà nhiều developer bối rối: đã tạo index hoàn hảo cho từng bảng, nhưng query JOIN vẫn chậm. Vấn đề không nằm ở index: mà ở cách thiết kế bảng.

Ví dụ: Ứng dụng quản lý công việc

```
-- Hiển thị tất cả task đang mở của nhóm,
-- nhưng chỉ từ các project đang hoạt động
SELECT tasks.*
FROM tasks
JOIN projects USING(project_id)
WHERE tasks.team_id = 4
  AND tasks.status = 'open'
  AND projects.status = 'open';
```

Trên dữ liệu test (vài trăm dòng), query chạy tức thì. Nhưng trên production:

```
Dữ liệu thực tế:
  tasks:    500,000 dòng
  projects: 10,000 dòng

Bước 1: Lọc tasks
  WHERE team_id = 4 AND status = 'open'
  → 20,000 dòng khớp (nhóm lớn, nhiều task)

Bước 2: JOIN với projects
  Mỗi dòng trong 20,000 kết quả → join với bảng projects
  → 20,000 lần tra cứu project_id trong bảng projects
  → Lọc tiếp: projects.status = 'open'
  → Kết quả cuối: chỉ 40 dòng (hầu hết project đã đóng)

  20,000 lần JOIN chỉ để lấy 40 dòng = lãng phí cực kỳ!


Phương án ngược: bắt đầu từ projects:
  Bước 1: Lọc projects WHERE status = 'open' → 2,000 project
  Bước 2: Mỗi project → tìm tasks của team 4
           → 2,000 lần JOIN, mỗi lần quét tasks
           → Vẫn chậm!
```

Dù bạn tạo index nào, bản chất vấn đề là thông tin lọc nằm ở 2 bảng khác nhau. Database phải join hàng chục nghìn lần trước khi biết kết quả cuối cùng.

Giải pháp: Thay đổi thiết kế bảng

Có hai cách tiếp cận:

```
-- Cách 1: Copy trạng thái project vào bảng tasks
ALTER TABLE tasks ADD COLUMN project_status VARCHAR(20);
-- Cập nhật khi project thay đổi trạng thái (trigger hoặc logic ứng dụng)

-- Giờ query không cần JOIN:
SELECT * FROM tasks
WHERE team_id = 4 AND status = 'open' AND project_status = 'open';
-- Index (team_id, status, project_status) xử lý hoàn hảo!

-- Cách 2: Đánh dấu tasks là "done" khi project đóng
-- Khi đóng project → UPDATE tasks SET status = 'archived'
-- WHERE project_id = ? AND status = 'open'
-- → Loại bỏ hoàn toàn nhu cầu JOIN
```

Trường hợp tương tự: lọc một bảng, sắp xếp bảng khác:

```
SELECT * FROM invoices
JOIN invoices_metadata USING(invoice_id)
WHERE invoices.tenant_id = 4236
ORDER BY invoices_metadata.due_date
LIMIT 30;
```

Ở đây, lọc trên `invoices` nhưng sắp xếp trên `invoices_metadata`. Database phải: lọc ra hàng nghìn hóa đơn → join tất cả → sắp xếp theo `due_date` → lấy 30 dòng đầu. Giải pháp tương tự: đưa `due_date` vào bảng `invoices` hoặc hợp nhất hai bảng.

Quy tắc nhớ: Khi query cần lọc/sắp xếp trên nhiều bảng khác nhau và phải join hàng chục nghìn lần, đó là tín hiệu cần thiết kế lại schema: không phải thêm index. Đôi khi một chút dư thừa dữ liệu (denormalization) mang lại hiệu suất tốt hơn hẳn so với thiết kế "chuẩn hóa hoàn hảo".

Vượt giới hạn kích thước index

Index phải nhỏ hơn đáng kể so với bảng dữ liệu để hiệu quả: nếu index to bằng bảng thì chẳng khác gì quét toàn bộ bảng. Khi bạn cố tạo index trên nhiều cột text dài, sẽ gặp lỗi kiểu:

```
ERROR: index row size 3480 bytes exceeds maximum 2712 bytes
```

Có ba kỹ thuật để xử lý:

Kỹ thuật 1: Index tiền tố: Chỉ index phần đầu chuỗi

```
-- PostgreSQL: dùng biểu thức cắt chuỗi
CREATE INDEX articles_search
ON articles (type, (SUBSTRING(title, 1, 20)));

-- MySQL: cú pháp tích hợp, gọn hơn
CREATE INDEX articles_search ON articles (type, title(20));
```

Cách hoạt động:

```
Tiêu đề gốc: "Hướng dẫn tối ưu hóa database cho người mới bắt đầu"
Giá trị trong index: "Hướng dẫn tối ưu hóa "  (20 ký tự đầu)

Tiêu đề gốc: "Hướng dẫn tối ưu hóa query với PostgreSQL nâng cao"
Giá trị trong index: "Hướng dẫn tối ưu hóa "  (20 ký tự đầu)
                       ↑ TRÙNG NHAU!
```

Vì tiền tố có thể trùng, query cần thêm bước lọc:

```
-- PostgreSQL: phải lọc 2 lần
SELECT * FROM articles
WHERE SUBSTRING(title, 1, 20) = '..20 ký tự đầu..'  -- dùng index
  AND title = '..tiêu đề đầy đủ..';                   -- lọc chính xác

-- MySQL: tự động xử lý: chỉ cần query bình thường
SELECT * FROM articles WHERE title = '..tiêu đề đầy đủ..';
-- MySQL tự dùng prefix index rồi kiểm tra lại giá trị đầy đủ
```

Cần cân bằng: tiền tố quá ngắn → nhiều kết quả trùng → phải đọc nhiều dòng từ bảng để lọc lại. Tiền tố quá dài → index phình to, mất ý nghĩa.

Kỹ thuật 2: Index giá trị băm (hash): Cho chuỗi rất dài

```
-- Index trên hash của tiêu đề thay vì tiêu đề gốc
CREATE INDEX articles_search ON articles (type, (SHA1(title)));

-- Query phải dùng đúng hàm hash + thêm điều kiện title để tránh trùng hash
SELECT * FROM articles
WHERE SHA1(title) = SHA1('..tiêu đề đầy đủ..')
  AND title = '..tiêu đề đầy đủ..';
```

Ưu điểm so với tiền tố: hash gần như không bao giờ trùng, nên bước lọc lại từ bảng gần như không xảy ra. Có thể kết hợp cả hai kỹ thuật:

```
-- Hash + cắt ngắn: kích thước tối ưu nhất
CREATE INDEX articles_search
ON articles (type, (SUBSTRING(SHA1(title), 1, 20)));
```

Kỹ thuật 3: Hash Index riêng biệt (chỉ PostgreSQL)

PostgreSQL hỗ trợ một loại index đặc biệt dùng bảng băm thay vì B+ tree:

```
CREATE INDEX invoices_uniqid ON invoices USING HASH (uniqid);
```

So sánh:

B-tree Index	Hash Index
Hỗ trợ: =, <, >, ≤, ≥, BETWEEN, ORDER BY	Chỉ hỗ trợ: =
Kích thước: lớn hơn	Kích thước: nhỏ hơn
Tốc độ =: nhanh	Tốc độ =: nhanh hơn

Dùng hash index khi cột chỉ cần tra cứu bằng (lookup by exact value): ví dụ: mã giao dịch, token, UUID.

JSON: Đánh index trong thế giới phi cấu trúc

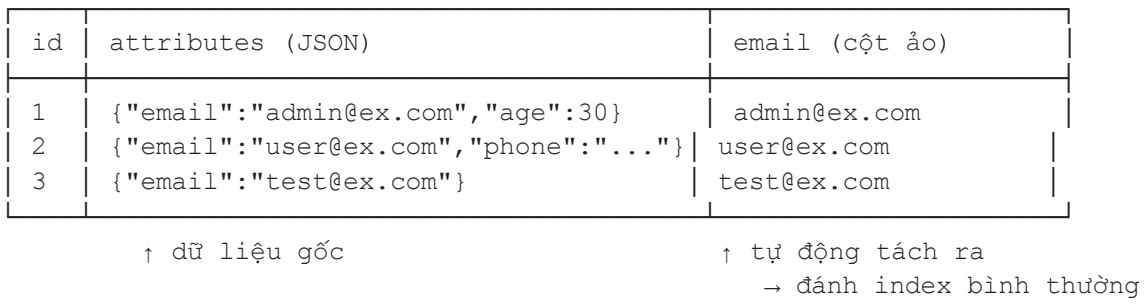
Hầu hết database hiện đại đều hỗ trợ cột JSON, nhưng việc đánh index cho JSON phức tạp hơn nhiều so với cột thông thường. Index trên cả cột JSON chỉ giúp tìm toàn bộ JSON giống hệt: gần như vô dụng. Bạn cần các kỹ thuật chuyên biệt.

Cách 1: Cột ảo (MySQL)

Tách giá trị JSON ra thành cột ảo, rồi đánh index bình thường:

```
CREATE TABLE contacts (
    id BIGINT PRIMARY KEY AUTO_INCREMENT,
    attributes JSON NOT NULL,
    -- Tạo cột ảo: tự động tính từ JSON, không tốn dung lượng lưu trữ
    email VARCHAR(255) AS (attributes->>"$.email") VIRTUAL NOT NULL,
    INDEX contacts_email (email)
);

-- Query dùng cột ảo
SELECT * FROM contacts WHERE email = 'admin@example.com';
-- Hoặc dùng trực tiếp biểu thức JSON (MySQL tự ánh xạ)
SELECT * FROM contacts WHERE attributes->>"$.email" = 'admin@example.com';
```

Hình dung:



Nhược điểm: mỗi trường JSON cần index phải tạo thêm một cột ảo, bảng trở nên rối khi có nhiều trường.

Cách 2: Index trên biểu thức JSON (PostgreSQL)

Gọn gàng hơn: không cần tạo cột ảo:

```
-- Tạo index trực tiếp trên biểu thức trích xuất JSON
CREATE INDEX contacts_email ON contacts ((attributes->>'email'));

-- Query dùng đúng biểu thức
SELECT * FROM contacts WHERE attributes->>'email' = 'admin@example.com';
```

Với MySQL, cách này phức tạp hơn vì phải xử lý kiểu dữ liệu và bộ đối chiếu (collation):

```
-- MySQL: phải ép kiểu và chỉ định collation
CREATE INDEX contacts_email ON contacts ((
    CAST(attributes->>"$.email" AS CHAR(255)) COLLATE utf8mb4_bin
));
-- Query cũng phải dùng đúng toán tử ->>" (hai dấu >)
SELECT * FROM contacts WHERE attributes->>"$.email" = 'admin@example.com';
```

Cách 3: GIN Index: "Index mọi thứ" trong JSON (chỉ PostgreSQL)

Tạo index cho từng trường JSON thì mệt. PostgreSQL có GIN (Generalized Inverted Index): đánh index toàn bộ nội dung JSON trong một lệnh:

```
CREATE INDEX contacts_attrs ON contacts USING GIN (attributes);
```

```
Cách GIN hoạt động (đơn giản hóa):

JSON gốc: {"email": "admin@ex.com", "role": "admin", "age": 30}

GIN index tạo ra các entry ngược (inverted):
```

Giá trị	Xuất hiện ở
key: "email"	dòng 1, 5, 12
key: "role"	dòng 1, 3
key: "age"	dòng 1, 7, 9
val: "admin@ex.com"	dòng 1
val: "admin"	dòng 1, 3
val: 30	dòng 1, 7

```
→ Tìm bất kỳ key hoặc value nào đều nhanh!
```

Nhưng GIN yêu cầu dùng các toán tử đặc biệt thay vì =:

```
-- Tìm dòng có email = 'admin@example.com' (dùng @> : "chứa")
SELECT * FROM contacts
WHERE attributes @> '{"email": "admin@example.com"}';

-- Kiểm tra key "email" có tồn tại không (dùng ?)
SELECT * FROM contacts WHERE attributes ? 'email';

-- Kiểm tra có BẤT KỲ key nào trong danh sách không (dùng ?|)
SELECT * FROM contacts WHERE attributes ?| array['email', 'phone'];
```

```
-- Kiểm tra có TẤT CẢ key trong danh sách không (dùng ?&)
SELECT * FROM contacts WHERE attributes ?& array['email', 'phone'];
```

Index cho mảng JSON:

```
-- PostgreSQL: dùng GIN cho mảng JSON
CREATE INDEX products_cats ON products USING GIN (categories);

-- Tìm sản phẩm thuộc cả 2 danh mục:
SELECT * FROM products
WHERE categories @> '["ebook", "printed"]';

-- MySQL: dùng multi-valued index (chỉ hỗ trợ số nguyên không dấu)
CREATE INDEX products_cats
ON products ((CAST(categories AS UNSIGNED ARRAY)));

-- Query dùng JSON_CONTAINS
SELECT * FROM products
WHERE JSON_CONTAINS(categories, CAST('[17, 23]' AS JSON));
```

Quy tắc nhớ: Nếu chỉ cần index 1-2 trường JSON → dùng cột ảo hoặc index biểu thức. Nếu cần tìm kiếm linh hoạt trên nhiều trường JSON → dùng GIN (PostgreSQL). MySQL hiện tại hạn chế hơn nhiều với JSON indexing.

Ràng buộc duy nhất và giá trị NULL: Lỗi bất ngờ

Đây là một trong những lỗi bẫy kinh điển mà hầu hết developer đều mắc ít nhất một lần.

Tình huống: Hệ thống đặt hàng sản phẩm giới hạn

```
-- Mỗi khách hàng chỉ được 1 đơn hàng chưa giao cho mỗi sản phẩm
-- shipment_id = NULL khi chưa giao, có giá trị khi đã giao

CREATE UNIQUE INDEX one_pending_order
ON orders (customer_id, shipment_id);
```

Bạn kỳ vọng: nếu khách hàng 17 đã có một đơn chưa giao `(17, NULL)`, thì không thể tạo thêm đơn `(17, NULL)` nữa. Nhưng thực tế:

```
Index chứa:

┌─────────────┬─────────────┐
│ customer_id │ shipment_id │
├─────────────┼─────────────┤
│     17      │    NULL     │  ← đơn 1, chưa giao
│     17      │    NULL     │  ← đơn 2, chưa giao: KHÔNG BÁO LỖI!
│     17      │    1001     │  ← đơn 3, đã giao
│     23      │    NULL     │  ← đơn 4, khách khác
└─────────────┴─────────────┘

Lý do: Trong SQL, NULL ≠ NULL (NULL không bằng bất kỳ giá trị nào, kể cả chính nó)
→ (17, NULL) và (17, NULL) được coi là KHÁC NHAU
→ Ràng buộc UNIQUE không bị vi phạm
→ Khách hàng 17 có thể đặt vô hạn đơn chưa giao!
```

Giải pháp cho PostgreSQL: ngắn gọn và hiệu quả:

```
-- PostgreSQL 15+: coi tất cả NULL là giống nhau trong ràng buộc UNIQUE
CREATE UNIQUE INDEX one_pending_order
ON orders (customer_id, shipment_id)
NULLS NOT DISTINCT;

-- Giờ (17, NULL) chỉ xuất hiện được 1 lần → đúng như mong đợi
```

Giải pháp chung cho mọi database:

```
-- Thay NULL bằng giá trị đặc biệt (ví dụ: -1)
CREATE UNIQUE INDEX one_pending_order ON orders (
    customer_id,
    (CASE WHEN shipment_id IS NULL THEN -1 ELSE shipment_id END)
```

```
);
```

Index sau khi sửa:

customer_id	CASE...	
17	-1	← đơn 1, chưa giao (NULL → -1)
17	-1	← LỖI! Vi phạm UNIQUE ✓
17	1001	← đơn 3, đã giao

```
-1 = -1 → đúng → vi phạm ràng buộc → không cho phép → ĐÚNG HÀNH VI MONG MUỐN
```

Lưu ý: Vì cột `shipment_id` đã bị biến đổi trong index, index này không dùng được cho query tìm kiếm bình thường trên `(customer_id, shipment_id)`. Bạn cần tạo thêm một index riêng nếu cần tra cứu.

Tìm và dọn dẹp index không sử dụng

Mỗi index bạn tạo đều có chi phí: mỗi lần `INSERT`, `UPDATE`, `DELETE` phải cập nhật tất cả index liên quan. Index thừa = ghi chậm hơn mà không mang lại lợi ích gì cho đọc.

Hai loại index thừa cần dọn:

```
Loại 1: Index trùng lặp (overlapping)
  Index A: (country, lastname, firstname)
  Index B: (country)
  → Index B hoàn toàn thừa! Index A đã bao phủ mọi query mà B phục vụ
  → Xóa Index B

  Tương tự:
  Index C: (country, lastname)
  Index D: (country, lastname, firstname)
  → Index C thừa vì D đã bao phủ C
  → Xóa Index C

  NHƯNG:
  Index E: (country, lastname, phone)
  Index F: (country, lastname, email)
  → KHÔNG thừa! Hai index phục vụ query khác nhau
     (E không thể thay thế F và ngược lại)


Loại 2: Index hoàn toàn không ai dùng
  → Tạo từ lâu, query đã thay đổi, không ai để ý xóa
  → Mỗi thao tác ghi đều lãng phí tài nguyên cập nhật index này
```

Cách phát hiện index không sử dụng:

```sql
-- MySQL: dùng performance_schema
SELECT
    object_schema AS 'database',
    object_name   AS 'bảng',
    index_name    AS 'index',
    count_star    AS 'số_lần_sử_dụng'
FROM performance_schema.table_io_waits_summary_by_index_usage
WHERE object_schema NOT IN ('mysql', 'performance_schema')
  AND index_name IS NOT NULL
  AND index_name != 'PRIMARY'
ORDER BY count_star ASC;
-- Index có count_star = 0 → ứng viên xóa

-- PostgreSQL: dùng pg_stat_all_indexes
```

```sql
SELECT
    schemaname    AS schema,
    tablename     AS bảng,
    indexrelname  AS index,
    idx_scan      AS số_lần_quét,
    idx_tup_read  AS số_dòng_đọc
FROM pg_stat_all_indexes
WHERE schemaname NOT IN ('pg_catalog', 'information_schema')
ORDER BY idx_scan ASC;
-- Index có idx_scan = 0 → ứng viên xóa
```

Cẩn thận: Thống kê chỉ tính từ lần khởi động lại database gần nhất (hoặc lần reset stats). Một index có thể "không dùng" trong tuần này nhưng quan trọng cho báo cáo cuối tháng. Hãy theo dõi ít nhất 1-2 chu kỳ nghiệp vụ đầy đủ trước khi quyết định xóa.

Xóa an toàn với tính năng ẩn index (MySQL):

```sql
-- Bước 1: Ẩn index thay vì xóa ngay
ALTER TABLE website_visits ALTER INDEX twitter_referrals INVISIBLE;
-- Database sẽ KHÔNG dùng index này cho query nào nữa
-- Nhưng index vẫn TỒN TẠI và được cập nhật khi ghi

-- Bước 2: Theo dõi 1-2 tuần
-- Nếu không có query nào chậm đi → an toàn để xóa
DROP INDEX twitter_referrals ON website_visits;

-- Nếu có vấn đề → phục hồi tức thì (không cần rebuild)
ALTER TABLE website_visits ALTER INDEX twitter_referrals VISIBLE;
```

PostgreSQL không hỗ trợ tính năng ẩn index. Cách an toàn nhất là: tạo index mới trước (nếu cần thay thế), rồi xóa index cũ.

Mẹo thực tế: Đặt lịch nhắc nhở (mỗi quý hoặc nửa năm) kiểm tra danh sách index không sử dụng. Đặc biệt quan trọng với các bảng có tần suất ghi cao: mỗi index thừa bị xóa đều cải thiện tốc độ ghi ngay lập tức.

Điều kiện "ma": Giúp database mà không thay đổi kết quả

Đây là kỹ thuật thông minh khi bạn biết logic nghiệp vụ mà database không thể tự suy ra. Bạn thêm điều kiện "thừa" vào query: điều kiện này không thay đổi kết quả nhưng giúp database tận dụng index tốt hơn.

Ví dụ: Tìm kiện hàng đang mở có bảo hiểm vận chuyển

```
-- Query gốc:
SELECT * FROM shipments
WHERE status = 'open' AND transportinsurance = 1;
```

Index hiện có: `(status, type)`. Cột `transportinsurance` không nằm trong index nào. Database chỉ dùng được `status` từ index, rồi quét và lọc `transportinsurance` từ bảng.

Nhưng bạn biết rằng theo quy tắc nghiệp vụ: chỉ có kiện hàng loại 3, 6, và 11 mới được phép mua bảo hiểm vận chuyển. Nghĩa là nếu `transportinsurance = 1`, thì chắc chắn `type IN (3, 6, 11)`.

```
-- Thêm điều kiện "ma": không thay đổi kết quả:
SELECT * FROM shipments
WHERE status = 'open'
  AND transportinsurance = 1
  AND type IN (3, 6, 11);     -- ← điều kiện "ma"

Không có điều kiện "ma":
  Index (status, type) → chỉ dùng được status = 'open'
  → Quét TẤT CẢ kiện hàng đang mở (có thể hàng chục nghìn)
  → Lọc transportinsurance từ bảng → chậm

Có điều kiện "ma":
  Index (status, type) → dùng ĐƯỢC CẢ HAI cột!
  → status = 'open' AND type IN (3, 6, 11)
  → Thu hẹp phạm vi quét xuống còn vài nghìn dòng
  → Lọc transportinsurance → nhanh hơn nhiều
```

Khi nào dùng kỹ thuật này?

Khi bạn biết mối quan hệ logic giữa các cột (ví dụ: "loại A chỉ có thể xuất hiện khi trạng thái là X") mà database không thể tự suy ra. Những mối quan hệ này thường đến từ quy tắc nghiệp vụ, không được mã hóa trong schema.

Lưu ý: Nếu quy tắc nghiệp vụ thay đổi (ví dụ: sau này tất cả loại kiện hàng đều được mua bảo hiểm), điều kiện "ma" sẽ lọc sai và trả về thiếu kết quả. Hãy đảm bảo ghi chú rõ ràng trong code tại sao điều kiện đó tồn tại.

Tìm kiếm theo vị trí: Khi hai điều kiện phạm vi đụng nhau

Nhiều ứng dụng cần tìm kiếm theo vị trí địa lý: nhà hàng gần đây, tài xế quanh khu vực, cửa hàng trong bán kính X km. Cách phổ biến nhất là dùng "hộp giới hạn" (bounding box) với kinh độ và vĩ độ.

```
-- Tìm nhà hàng trong khu vực Manhattan
SELECT * FROM businesses
WHERE type = 'restaurant'
  AND longitude BETWEEN -74.0083 AND -73.9752
  AND latitude BETWEEN 40.7216 AND 40.7422;
```

Vấn đề: cả `longitude` và `latitude` đều là điều kiện phạm vi. Theo nguyên tắc "quét khi gặp range", index chỉ dùng được một cột range:

```
Index (type, longitude, latitude):
  Bước 1: type = 'restaurant'       → equality, thu hẹp tốt ✓
  Bước 2: longitude BETWEEN ...     → range, quét ✓
  Bước 3: latitude BETWEEN ...      → SAU range → không thu hẹp được ✗
  → Phải quét tất cả nhà hàng có longitude phù hợp,
    rồi lọc latitude từng dòng một

  Với dữ liệu cả nước: longitude khớp = hàng triệu dòng
  → Quét hàng triệu dòng chỉ để lọc latitude → CHẬM
```

Giải pháp: Spatial Index (chỉ mục không gian)

Đây là loại index được thiết kế riêng cho dữ liệu đa chiều, dùng cấu trúc R-tree thay vì B+ tree:

```
-- PostgreSQL: dùng kiểu geometry + GIST index
CREATE TABLE businesses (
    id BIGINT PRIMARY KEY,
    type VARCHAR(255) NOT NULL,
    name VARCHAR(255) NOT NULL,
    location GEOMETRY(Point, 4326) NOT NULL  -- hệ tọa độ WGS 84
);
CREATE INDEX search_idx ON businesses USING GIST (type, location);

-- Query dùng hàm không gian
SELECT * FROM businesses
WHERE type = 'restaurant'
  AND location && ST_MakeEnvelope(
      -74.0083, 40.7216,   -- góc dưới trái (kinh, vĩ)
```

```
      -73.9752, 40.7422,    -- góc trên phải (kinh, vĩ)
      4326                   -- hệ tọa độ
  );

-- MySQL: cú pháp khác một chút
CREATE TABLE businesses (
    id BIGINT PRIMARY KEY,
    type VARCHAR(255) NOT NULL,
    name VARCHAR(255) NOT NULL,
    location POINT SRID 0 NOT NULL
);
-- MySQL: spatial index chỉ chứa được 1 cột
CREATE SPATIAL INDEX search_idx ON businesses (location);

SELECT * FROM businesses
WHERE type = 'restaurant'
  AND ST_CONTAINS(
      ST_MakeEnvelope(
          POINT(-74.0083, 40.7216),
          POINT(-73.9752, 40.7422)
      ),
      location
  );
```

Khác biệt quan trọng giữa MySQL và PostgreSQL:

PostgreSQL:
- ✓ Spatial index hỗ trợ nhiều cột (type + location trong cùng index)
- ✓ Hỗ trợ hệ tọa độ SRID 4326 (tính toán theo độ cong trái đất)
- ✓ Khoảng cách chính xác trên bề mặt cầu

MySQL:
- ✗ Spatial index chỉ chứa 1 cột (phải lọc type riêng)
- ✗ Một số hàm không hỗ trợ SRID 4326
- ✗ Tính khoảng cách trên mặt phẳng (hơi sai với khoảng cách lớn)

Tìm kiếm ký tự đại diện ở đầu: Trường hợp đặc biệt

Ở Phần 3, bạn đã biết rằng `LIKE 'abc%'` có thể dùng index (chuyển thành range scan), nhưng `LIKE '%abc%'` (ký tự đại diện ở đầu) thì không thể dùng index vì database không biết bắt đầu quét từ đâu.

Đây là hạn chế của B+ tree: không có cách nào khắc phục. Nhưng PostgreSQL có một giải pháp riêng: Trigram Index.

```
-- Bật extension pg_trgm (có sẵn, chỉ cần kích hoạt)
CREATE EXTENSION IF NOT EXISTS pg_trgm;

-- Tạo trigram index
CREATE INDEX trgm_idx ON contacts USING GIN (name gin_trgm_ops);
```

Cách hoạt động:

```
Tên gốc: "Nguyễn Văn Minh"

Tách thành trigram (chuỗi 3 ký tự):
  "ngu", "guy", "uyễ", "yễn", "ễn ", "n v", " vă", "văn",
  "ăn ", "n m", " mi", "min", "inh"

Mỗi trigram được đánh index riêng (giống GIN):
  "min" → dòng 1, 45, 289, ...
  "inh" → dòng 1, 45, 67, ...

Query: WHERE name LIKE '%Minh%'
  → Tách "minh" thành trigram: "min", "inh"
  → Tìm giao của 2 tập kết quả
  → Kiểm tra lại pattern chính xác trên các ứng viên
```

Yêu cầu: chuỗi tìm kiếm phải có ít nhất 3 ký tự liên tiếp (không tính ký tự đại diện) để trigram index hoạt động hiệu quả. Index trigram có thể rất lớn vì số lượng trigram tăng nhanh theo độ dài chuỗi.

Lưu ý: Trigram index chỉ có trên PostgreSQL. MySQL, SQL Server và các database khác không có tính năng tương đương tích hợp sẵn. Nếu cần tìm kiếm toàn văn phức tạp, hãy cân nhắc dùng công cụ chuyên biệt (Elasticsearch, Meilisearch, v.v.).

Kỹ thuật thao tác dữ liệu hiệu quả

Tranh chấp khóa: Khi bộ đếm bị "nghẽn cổ chai"

Hãy tưởng tượng một post Facebook viral: hàng nghìn like mỗi giây. Mỗi `UPDATE likes_count = likes_count + 1` phải lock row → các update xếp hàng chờ nhau.

Giải pháp: Bộ đếm phân tán

```
-- Thay vì 1 row, tạo 100 row cho mỗi counter
INSERT INTO post_statistics (post_id, fanout, likes_count)
VALUES (1475870220422107137, FLOOR(RAND() * 100), 1)
ON DUPLICATE KEY UPDATE likes_count = likes_count + VALUES(likes_count);

-- Khi đọc: SUM tất cả fanout rows
SELECT SUM(likes_count) FROM post_statistics WHERE post_id =
1475870220422107137;
```

Throughput tăng gấp 100 lần vì 100 row khác nhau có thể update song song.

Cập nhật dữ liệu từ bảng khác: JOIN trong UPDATE

```
-- MySQL:
UPDATE products
JOIN categories USING(category_id)
SET price = price_base - price_base * categories.discount;

-- PostgreSQL:
UPDATE products
SET price = price_base - price_base * categories.discount
FROM categories
WHERE products.category_id = categories.category_id;
```

Thay vì loop trong application code, một query duy nhất xử lý tất cả.

RETURNING: Lấy dữ liệu ngay sau khi thay đổi (PostgreSQL)

```
DELETE FROM sessions WHERE ip = '127.0.0.1'
RETURNING id, user_agent, last_access;
-- Xóa VÀ trả về data trong cùng 1 query
```

```
-- Hoạt động với DELETE, INSERT, UPDATE
```

Xóa dòng trùng lặp: Dùng CTE thay vì xử lý ở tầng ứng dụng

```
-- PostgreSQL:
WITH duplicates AS (
    SELECT id, ROW_NUMBER() OVER(
        PARTITION BY firstname, lastname, email
        ORDER BY age DESC  -- giữ lại row có age cao nhất
    ) AS rownum
    FROM contacts
)
DELETE FROM contacts
USING duplicates
WHERE contacts.id = duplicates.id AND duplicates.rownum > 1;
```

Thay vì viết logic chunking phức tạp trong code, một CTE query giải quyết gọn gàng.

Viết query như chuyên gia

Phân trang đúng cách: Phân trang theo khóa

Vấn đề #1: Thứ tự sắp xếp không ổn định

```
-- ❌ Sai: nhiều user cùng firstname có thể xuất hiện ở 2 page
SELECT * FROM users ORDER BY firstname LIMIT 20 OFFSET 40;

-- ✅ Đúng: thêm primary key để đảm bảo unique ordering
SELECT * FROM users ORDER BY firstname, lastname, user_id LIMIT 20 OFFSET 60;
```

Vấn đề #2: OFFSET chậm với page lớn + data thay đổi

```
-- ❌ LIMIT OFFSET: page 1000 phải skip 999 * 30 = 29,970 rows
SELECT * FROM users ORDER BY firstname, lastname, id LIMIT 30 OFFSET 29970;

-- ✅ Keyset Pagination: dùng giá trị row cuối trang trước
SELECT * FROM users
WHERE (firstname, lastname, id) > ('Huy', 'Nguyen', 3150)
ORDER BY firstname, lastname, id
LIMIT 30;
-- Nhanh hơn RẤT nhiều cho page lớn, nhưng không hỗ trợ nhảy page
```

FOR UPDATE: Khóa dòng ở tầng database

```
START TRANSACTION;
SELECT balance FROM account WHERE account_id = 7 FOR UPDATE;
-- Row bị lock, không ai khác modify được cho đến khi COMMIT
-- Application xử lý logic...
UPDATE account SET balance = 540 WHERE account_id = 7;
COMMIT;
-- Lock tự động release khi transaction kết thúc hoặc client disconnect
```

Ưu điểm so với application-level locking: không lo quên unlock, không lo crash để lại lock "treo".

Biểu thức bảng tạm (CTE): Xử lý query phức tạp

CTE cho phép chia query phức tạp thành nhiều bước nhỏ, dễ đọc và dễ debug:

```
WITH
-- Bước 1: Top 10 sản phẩm bán chạy nhất
most_popular AS (
    SELECT products.*, COUNT(*) as sales
    FROM products
    JOIN orders_products USING(product_id)
    WHERE created_at BETWEEN '2024-01-01' AND '2024-06-30'
    GROUP BY products.product_id
    ORDER BY COUNT(*) DESC LIMIT 10
),
-- Bước 2: Users đủ điều kiện tham gia raffle
eligible_users AS (
    SELECT DISTINCT users.* FROM users
    JOIN users_raffle USING(user_id)
    WHERE correct_answers > 8
)
-- Bước 3: Kết hợp 2 bước trên
SELECT * FROM eligible_users
JOIN orders_products USING(user_id)
JOIN most_popular USING(product_id);
```

Mỗi CTE step có thể test độc lập → debug dễ hơn hẳn so với nested subquery.

Các Tips Query hữu ích khác

Tránh Division by Zero:

```
SELECT visitors_today / NULLIF(visitors_yesterday, 0) FROM stats;
-- NULLIF trả về NULL nếu visitors_yesterday = 0 → kết quả = NULL thay vì error
```

Gap-filling cho Statistical Queries:

```
-- PostgreSQL: tạo chuỗi ngày liên tục rồi LEFT JOIN với data
SELECT dates.day, COALESCE(SUM(stats.count), 0)
FROM generate_series(
    CURRENT_DATE - INTERVAL '14 days', CURRENT_DATE, '1 day'
) AS dates(day)
LEFT JOIN statistics stats ON stats.day = dates.day
GROUP BY dates.day;
```

Multiple Aggregates trong một query:

```
-- PostgreSQL:
SELECT
    COUNT(*) FILTER (WHERE released_at = 2024) AS released_2024,
    COUNT(*) FILTER (WHERE director = 'Nolan') AS nolan_movies
FROM movies;

-- MySQL:
SELECT
    SUM(released_at = 2024) AS released_2024,
    SUM(director = 'Nolan') AS nolan_movies
FROM movies;
```

Fast Row Count Estimate:

```
-- Không cần COUNT(*) chính xác? Dùng EXPLAIN:
-- PostgreSQL: EXPLAIN SELECT * FROM movies WHERE rating = 'R';
-- Đọc "rows" estimate từ output
```

DISTINCT ON (chỉ PostgreSQL):

```
-- Lấy đơn hàng đắt nhất của mỗi customer trong 2024
SELECT DISTINCT ON (customer_id) *
FROM orders
WHERE EXTRACT(YEAR FROM created_at) = 2024
ORDER BY customer_id ASC, price DESC;
```

Thiết kế Schema: Nền móng vững chắc

UUID vs Auto-increment: Lựa chọn Primary Key

Tiêu chí	Auto-increment	UUIDv4	UUIDv7/ULID
Insert speed	Nhanh nhất	Chậm (random position)	Nhanh (time-sorted)
Size	4-8 bytes	16 bytes	16 bytes
Predictable	Dễ đoán (enumeration attack)	Không đoán được	Không đoán được
Distributed ID gen	Không	Có	Có

Khuyến nghị: Dùng auto-increment cho internal PK. Nếu cần expose ra external (URL, API), thêm cột UUID riêng:

```
-- PostgreSQL:
ALTER TABLE users ADD COLUMN uuid UUID NOT NULL DEFAULT gen_random_uuid();
CREATE UNIQUE INDEX users_uuid ON users (uuid);

-- Dùng auto-increment id cho JOIN, uuid cho API/URL
```

JSON Column: Khi NoSQL gặp SQL

Dùng JSON column khi:

- Dữ liệu ít khi query trực tiếp (metadata, settings)
- Thay thế bảng EAV (Entity-Attribute-Value) phức tạp
- Giảm số lượng JOIN cho data seldom-used

Quy tắc:

- Vẫn dùng relational cho data chính (foreign key, constraints)
- Tránh deeply nested JSON: query/update sẽ rất phức tạp
- Cân nhắc kỹ khi lưu references đến bảng khác trong JSON (mất FK constraint)

JSON Schema Validation (MySQL):

```
ALTER TABLE products ADD CONSTRAINT CHECK(
    JSON_SCHEMA_VALID('{
        "type": "object",
        "properties": {
            "tags": {"type": "array", "items": {"type": "string"}}
        },
        "additionalProperties": false
    }', attributes)
);
```

Constraint: Hàng rào bảo vệ cuối cùng

```
-- Check constraint: checkin phải trước checkout
ALTER TABLE reservations
ADD CONSTRAINT start_before_end CHECK (checkin_at < checkout_at);

-- Business rule: EU customers phải có VAT ID
ALTER TABLE invoices
ADD CONSTRAINT eu_vat CHECK (NOT(is_eu) OR vatid IS NOT NULL);
```

Application validation có thể bị bypass (batch update, manual SQL). Database constraint luôn được enforce.

Ràng buộc loại trừ: Chống chồng chéo (PostgreSQL)

```
CREATE TABLE bookings (
    room_number INT,
    reservation TSTZRANGE,
```

```
    EXCLUDE USING GIST (room_number WITH =, reservation WITH &&)
);
-- Tự động ngăn 2 booking trùng thời gian cho cùng phòng
-- Không cần application-level locking!
```

Đường dẫn vật lý hóa: Lưu trữ cây đơn giản

```
-- PostgreSQL (với extension ltree):
CREATE EXTENSION ltree;
CREATE TABLE categories (path LTREE);
INSERT INTO categories VALUES ('Food'), ('Food.Fruit'), ('Food.Fruit.Cherry');

-- Tìm tất cả con của Food.Fruit:
SELECT * FROM categories WHERE path ~ 'Food.Fruit.*{1,}';
-- Tìm tổ tiên của Food.Fruit.Cherry:
SELECT * FROM categories WHERE path @> subpath('Food.Fruit.Cherry', 0, -1);
```

Partition: Xóa data lớn trong tích tắc

```
-- Thay vì DELETE FROM logs WHERE created_at < '2024-01-01' (rất chậm)
-- Dùng partition theo tháng:
ALTER TABLE logs DROP PARTITION logs_2023_january;
-- Xóa toàn bộ partition trong vài giây, bất kể có bao nhiêu row
```

Bảng sắp xếp trước: Tối ưu cho quét phạm vi

```
-- MySQL: Dùng composite primary key để sắp xếp vật lý
CREATE TABLE product_comments (
    product_id BIGINT,
    comment_id BIGINT AUTO_INCREMENT UNIQUE KEY,
    message TEXT,
    PRIMARY KEY (product_id, comment_id)
);
-- Comments của cùng product nằm liên tiếp trên disk → đọc nhanh hơn

-- PostgreSQL: Dùng CLUSTER
CLUSTER product_comments USING product_comments_pkey;
```

Tính toán trước: Khi index cũng không đủ nhanh

Khi dashboard query phải aggregate hàng trăm nghìn row, không index nào giúp được. Giải pháp: lưu sẵn giá trị đã aggregate.

```
-- Bảng pre-aggregated:
CREATE TABLE articles_stats (
    user_id BIGINT,
    publish_year INT,
    total_likes BIGINT,
    PRIMARY KEY (user_id, publish_year)
);

-- Query siêu nhanh:
SELECT total_likes FROM articles_stats
WHERE user_id = 1 AND publish_year = 2024;
-- Thay vì SUM(likes) trên hàng trăm nghìn rows
```

Lời kết

Bạn đã đọc hết cuốn “Database Indexing & Những Điều Developer Cần Biết” của mình. Từ đáy lòng, mình cám ơn bạn rất nhiều vì đã đọc đến cuối quyển Ebook này. Có thể cách viết mình còn lộn xộn, mình mong bạn hiểu những tâm huyết của mình, và những mong mỏi được chia sẻ kiến thức đến cộng đồng. Làm việc với cơ sở dữ liệu quan hệ như MySQL hay Postgres là điều rất thú vị, nhưng cũng tiềm ẩn rất nhiều thứ cần cẩn trọng. Nếu tận dụng tốt, nó sẽ rất mạnh mẽ, ngược lại chúng ta cũng dễ mắc những lỗi cơ bản. Mình hy vọng ebook này sẽ giúp đỡ bạn phần nào trên con đường tối ưu CSDL của mình.

Cám ơn bạn rất nhiều !

Nếu bạn thích những gì trong cuốn sách này và muốn trao đổi thêm, đừng ngần ngại liên lạc với mình qua:

Email: huynt57@gmail.com

Facebook: Tại đây

Cuốn Ebook này sẽ không tồn tại nếu không có vợ mình Dương Thị Thu Huyền và con trai mình, cố vấn tí hon Nguyễn Dương Hoàng Khôi. Cám ơn gia đình đã luôn bên cạnh và ủng hộ mình.

Happy Coding !