Indexes and Performance: The Reality Check

What Indexes Actually Are

An index is a **separate data structure** that stores a sorted copy of one or more columns, plus pointers to the actual rows.

Think of it like a book's index: instead of scanning every page to find "SQL," you look in the index, which tells you it's on pages 42, 87, and 103.

Without an Index

```
SELECT * FROM users WHERE email = 'alice@example.com';
```

Database does:

1. Scan every row in `users` (sequential scan)
2. Check if `email` matches
3. Return matching rows

Cost: O(N) — reads entire table.

With an Index on `email`

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

Database does:

1. Search the index (B-tree lookup)
2. Find the pointer to the matching row
3. Fetch the row

Cost: O(log N) for the lookup + O(1) for the fetch.

Massive difference when N is large.

B-Tree Indexes: The Default

Mental Model

A B-tree is a **self-balancing tree** where:

- **Leaf nodes** contain sorted data + pointers to table rows
- **Internal nodes** contain keys for navigation

ASCII visualization:

```
            [M]
           /   \
      [D G]     [P T]
     / | \      / | \
  [A] [E] [J] [N] [Q] [U]
```

```
    |   |   |   |   |   |
  (rows) (rows) (rows) ...
```

How a lookup works:

Query: `WHERE email = 'john@example.com'`

1. Start at root: "J" is between "G" and "M", go left
2. Middle node: "J" is between "G" and "M", go to [J] leaf
3. Leaf node: Binary search to find "john@example.com"
4. Follow pointer to table row

Cost: O(log N) — tree height is logarithmic.

Why B-Trees Are Fast

- **Logarithmic lookups:** Even with 1 million rows, only ~20 comparisons
- **Range scans:** Because leaf nodes are sorted, range queries are efficient
- **Cache-friendly:** Internal nodes are small, fit in memory

When B-Trees Are Used

- **Equality:** `WHERE id = 123`
- **Range:** `WHERE created_at > '2024-01-01'`
- **Sorting:** `ORDER BY name`
- **Prefix matching:** `WHERE email LIKE 'alice%'` (but not `LIKE '%alice'`)

Index Scans vs Sequential Scans

Sequential Scan (Table Scan)

What it does: Read every row in the table.

When it's used:

- No suitable index
- Query matches most rows (index would be slower)
- Table is small (index overhead isn't worth it)

Cost: O(N) — always.

Index Scan

What it does: Use the index to find rows.

Cost: O(log N + M) where M = number of matching rows.

Index-Only Scan (Covering Index)

What it does: Fetch data directly from the index without hitting the table.

When it's possible: The index includes all columns in SELECT and WHERE.

Example:

```
CREATE INDEX idx_users_email_name ON users(email, name);
```

```sql
SELECT name FROM users WHERE email = 'alice@example.com';
```

Database does:

- Look up `email` in the index
- Fetch `name` from the index (no table access needed)

Fastest possible.

Bitmap Index Scan (Postgres)

What it does: Use multiple indexes, combine results with a bitmap.

Example:

```sql
SELECT * FROM users WHERE age > 30 AND city = 'New York';
```

If indexes exist on `age` and `city`:

1. Scan `idx_age` → bitmap of matching row IDs
2. Scan `idx_city` → bitmap of matching row IDs
3. AND the bitmaps
4. Fetch rows

Use case: Combining filters on different columns.

Composite Indexes: Column Order Matters

Single-Column Index

```sql
CREATE INDEX idx_users_email ON users(email);
```

Helps: `WHERE email = ...`

Doesn't help: `WHERE city = ...`

Multi-Column Index

```sql
CREATE INDEX idx_users_country_city ON users(country, city);
```

Helps:

- `WHERE country = 'USA'`
- `WHERE country = 'USA' AND city = 'New York'`

Doesn't help:

- `WHERE city = 'New York'` (city isn't the leading column)

Why? The index is sorted by `country` first, then `city` within each country. If you filter only by `city`, the database can't use the sorted structure.

The Leftmost Prefix Rule

An index on `(A, B, C)` can be used for:

- `WHERE A = ...`
- `WHERE A = ... AND B = ...`
- `WHERE A = ... AND B = ... AND C = ...`

But NOT:

- `WHERE B = ...`
- `WHERE C = ...`
- `WHERE B = ... AND C = ...`

Column order matters. Put the most selective/frequently filtered columns first.

Choosing Column Order

Rule of thumb:

1. **Equality first:** `WHERE status = 'active'` comes before range filters
2. **Higher cardinality first:** More unique values = more selective
3. **Most common filters first**

Example:

```
-- Query pattern:
SELECT * FROM orders
WHERE status = 'pending'
  AND created_at > NOW() - INTERVAL '7 days';

-- Index:
CREATE INDEX idx_orders_status_created ON orders(status, created_at);
```

Why `status` first?

- `status` is an equality filter (more selective)
- `created_at` is a range filter (use after narrowing by status)

When Indexes Hurt

Indexes Slow Down Writes

Every INSERT/UPDATE/DELETE must update all relevant indexes.

Example: Table with 5 indexes

- **INSERT:** Write 1 row + update 5 indexes (6 writes total)
- **UPDATE:** Modify 1 row + update affected indexes
- **DELETE:** Remove 1 row + update 5 indexes

Trade-off: Fast reads, slow writes.

In high-write workloads (OLTP): Be judicious with indexes.

Indexes Consume Space

Indexes are **separate data structures**. A table with 10 GB of data might have 5 GB of indexes.

Cost: Disk space, memory (caching indexes), backup size.

Too Many Indexes Slow Down the Optimizer

The query planner evaluates possible index combinations. With 20 indexes, that's a lot of possibilities.

Result: Longer planning time (usually negligible, but noticeable for simple queries).

Indexes Don't Always Help

Scenario 1: Low selectivity

```
SELECT * FROM users WHERE active = true;
```

If 95% of users are active, an index scan isn't faster than a sequential scan. The database scans most rows anyway.

The optimizer will choose seq scan.

Scenario 2: Function on indexed column

```
SELECT * FROM users WHERE LOWER(email) = 'alice@example.com';
```

Problem: The function `LOWER()` prevents index usage.

Fix: Create a functional index:

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

Now `WHERE LOWER(email) = ...` uses the index.

Scenario 3: Implicit type conversion

```
-- email is TEXT, but you're comparing to an integer:
SELECT * FROM users WHERE email = 123;
```

Problem: Postgres converts `email::INTEGER`, which prevents index usage.

Fix: Use the correct type:

```
WHERE email = '123'
```

When the Database Ignores Your Index

Even with an index, the optimizer might choose a sequential scan if:

- **Small table:** Index overhead exceeds scan cost
- **Low selectivity:** Matching many rows
- **Statistics are stale:** Optimizer thinks index isn't helpful (run `ANALYZE`)

Covering Indexes: Index-Only Scans

A covering index includes all columns needed by the query.

Example:

```
-- Query:
SELECT name, email FROM users WHERE active = true;

-- Covering index:
CREATE INDEX idx_users_active_name_email ON users(active, name, email);
```

What happens:

- Database looks up `active = true` in the index
- Fetches `name` and `email` directly from the index
- **No table access needed**

Result: Index-only scan (fastest).

INCLUDE Clause (Postgres 11+)

```
CREATE INDEX idx_users_active ON users(active) INCLUDE (name, email);
```

What it does:

- `active` is in the B-tree structure (used for filtering)
- `name, email` are stored in the leaf nodes (for retrieval)

Use case: Add columns to enable index-only scans without bloating the tree structure.

Partial Indexes: Index Only What Matters

Example:

```
CREATE INDEX idx_users_active ON users(email) WHERE active = true;
```

What it does: Only indexes rows where `active = true`.

Benefits:

- Smaller index (only indexes active users)
- Faster writes (no need to update index for inactive users)
- Faster scans (less data to read)

When to use:

- When you mostly query a specific subset
- Example: "Active users", "Pending orders", "Recent records"

Postgres-specific: MySQL doesn't support partial indexes.

Unique Indexes and Constraints

```sql
CREATE UNIQUE INDEX idx_users_email ON users(email);
```

What it does:

- Enforces uniqueness (like a UNIQUE constraint)
- Also creates an index (for fast lookups)

vs UNIQUE constraint:

```sql
ALTER TABLE users ADD CONSTRAINT users_email_unique UNIQUE (email);
```

Postgres: These are equivalent. A UNIQUE constraint is implemented as a unique index.

Performance benefit: Unique indexes can use faster lookup algorithms (no need to scan for duplicates).

Functional Indexes: Index Expressions

Use case: Case-insensitive search.

```sql
-- Slow (can't use index):
SELECT * FROM users WHERE LOWER(email) = 'alice@example.com';

-- Fast (with functional index):
CREATE INDEX idx_users_email_lower ON users(LOWER(email));
SELECT * FROM users WHERE LOWER(email) = 'alice@example.com';
```

What it does: Index stores `LOWER(email)` instead of `email`.

Other use cases:

- `CREATE INDEX idx_created_date ON orders((created_at::DATE));` — index the date part only
- `CREATE INDEX idx_full_name ON users((first_name || ' ' || last_name));` — index concatenated columns

GIN and GiST Indexes (Postgres)

GIN (Generalized Inverted Index)

Use case: JSONB, arrays, full-text search.

```sql
CREATE INDEX idx_users_tags ON users USING GIN(tags);
```

Query:

```sql
SELECT * FROM users WHERE tags @> ARRAY['vip', 'premium'];
```

What it does: Efficiently searches array/JSONB containment.

Cost: Large indexes, slow writes, fast reads.

GiST (Generalized Search Tree)

Use case: Geometric data, ranges, full-text search.

```
CREATE INDEX idx_events_date_range ON events USING GIST(date_range);
```

Query:

```
SELECT * FROM events WHERE date_range && '[2024-01-01, 2024-12-31]';
```

Use case: Overlapping ranges, proximity searches.

When to Use

- **B-tree (default):** Most queries (equality, range, sorting)
- **GIN:** JSONB, arrays, full-text search
- **GiST:** Geometric data, ranges
- **BRIN:** Very large tables with natural clustering (time-series)

BRIN Indexes: For Huge Tables

BRIN (Block Range Index): Stores min/max values for each block of rows.

Use case: Time-series data, append-only tables.

```
CREATE INDEX idx_logs_created_at ON logs USING BRIN(created_at);
```

What it does:

- Doesn't store every value (like B-tree)
- Stores min/max per block (e.g., 128 pages)
- Query checks if block might contain matching rows

Benefits:

- Tiny index (1000x smaller than B-tree)
- Fast to build
- Works well if data is naturally clustered (e.g., sorted by `created_at`)

Trade-off: Less precise (may scan rows that don't match), but worth it for huge tables.

Index Maintenance

VACUUM and Index Bloat

Postgres: UPDATE/DELETE creates "dead tuples" (old row versions). Indexes can accumulate these.

Fix: VACUUM reclaims space.

```
VACUUM ANALYZE users;
```

Alternatively: Enable `autovacuum` (on by default).

REINDEX: Rebuild Indexes

When to use:

- Index bloat (after heavy UPDATEs/DELETEs)
- Corruption (rare)
- After bulk data load

```
REINDEX INDEX idx_users_email;
```

Caution: Locks the table (Postgres 12+ has `REINDEX CONCURRENTLY`).

Statistics: Help the Optimizer

The optimizer uses statistics to estimate row counts and selectivity.

Update statistics:

```
ANALYZE users;
```

When to run:

- After bulk INSERT/UPDATE/DELETE
- After schema changes
- If query plans are bad

Postgres: `autovacuum` runs ANALYZE automatically, but you can force it.

Real-World Indexing Strategies

Strategy 1: Index Foreign Keys

Always.

```
CREATE INDEX idx_orders_user_id ON orders(user_id);
CREATE INDEX idx_orders_product_id ON orders(product_id);
```

Why: JOINs depend on foreign keys. Without indexes, every join is a sequential scan.

Strategy 2: Index WHERE Clause Columns

Query pattern:

```
SELECT * FROM orders WHERE status = 'pending' AND created_at > NOW() - INTERVAL '7
days';
```

Index:

```
CREATE INDEX idx_orders_status_created ON orders(status, created_at);
```

Strategy 3: Index ORDER BY Columns

Query:

```
SELECT * FROM users ORDER BY created_at DESC LIMIT 10;
```

Index:

```
CREATE INDEX idx_users_created_at ON users(created_at DESC);
```

Why: Database can scan index in order, return first 10 rows without sorting.

Strategy 4: Covering Indexes for Hot Queries

Query:

```
SELECT id, email FROM users WHERE active = true;
```

Covering index:

```
CREATE INDEX idx_users_active_id_email ON users(active) INCLUDE (id, email);
```

Result: Index-only scan (no table access).

Strategy 5: Partial Indexes for Common Filters

Query pattern:

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

Partial index:

```
CREATE INDEX idx_orders_pending ON orders(created_at) WHERE status = 'pending';
```

Why: Smaller index, faster scans, no wasted space indexing completed orders.

PostgreSQL vs MySQL Index Differences

Covering Indexes

Postgres: INCLUDE clause (Postgres 11+)

```
CREATE INDEX idx ON users(email) INCLUDE (name);
```

MySQL (InnoDB): Secondary indexes implicitly include the primary key

```
CREATE INDEX idx ON users(email);
-- Implicitly includes PRIMARY KEY columns
```

Partial Indexes

Postgres: Supported

```
CREATE INDEX idx ON users(email) WHERE active = true;
```

MySQL: Not supported. Must index the entire column.

Index Order

Both support: `ASC` / `DESC`

```
CREATE INDEX idx ON users(created_at DESC);
```

Functional Indexes

Postgres: Fully supported

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

MySQL: Supported (MySQL 8.0+)

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

Clustered vs Non-Clustered

MySQL (InnoDB): Primary key is clustered (table is sorted by PK). Secondary indexes store PK as pointer.

Postgres: All indexes are non-clustered. `CLUSTER` command physically reorders table (once, not maintained).

Practical Debugging: Is My Index Being Used?

Check with EXPLAIN

```
EXPLAIN SELECT * FROM users WHERE email = 'alice@example.com';
```

Look for:

- `Index Scan using idx_users_email` — Good!
- `Seq Scan` — Bad! Index not used.

Reasons Index Might Not Be Used

1. **No index exists:** Create one.
2. **Function on column:** Create a functional index.
3. **Type mismatch:** Fix the query.
4. **Low selectivity:** Optimizer chose seq scan (might be correct).
5. **Stale statistics:** Run `ANALYZE`.

When NOT to Add an Index

❌ **High-write tables with many indexes**: Every write updates all indexes.

❌ **Low-selectivity columns**: `WHERE gender = 'M'` (50/50 split) won't benefit much.

❌ **Small tables**: Sequential scan is fast enough.

❌ **One-off queries**: Don't optimize for queries you run once.

✅ **Do add indexes for:**

- Foreign keys (always)
- Common WHERE filters
- ORDER BY columns
- JOIN keys

Key Takeaways

1. **Indexes are sorted data structures** (usually B-trees) that enable O(log N) lookups.
2. **Composite indexes: column order matters.** Leftmost prefix rule applies.
3. **Indexes speed up reads, slow down writes.** Balance accordingly.
4. **Covering indexes enable index-only scans** (fastest).
5. **Partial indexes** reduce size and improve performance for common filters.
6. **Functional indexes** allow indexing expressions (e.g., `LOWER(email)`).
7. **The optimizer might ignore your index** if it thinks a seq scan is faster.
8. **Always index foreign keys.** JOINs depend on them.
9. **Use EXPLAIN to verify index usage.** Don't guess.
10. **Keep statistics up-to-date with ANALYZE.** The optimizer relies on them.

Next up: Reading EXPLAIN ANALYZE output to debug slow queries.