Database Index Optimization Guide

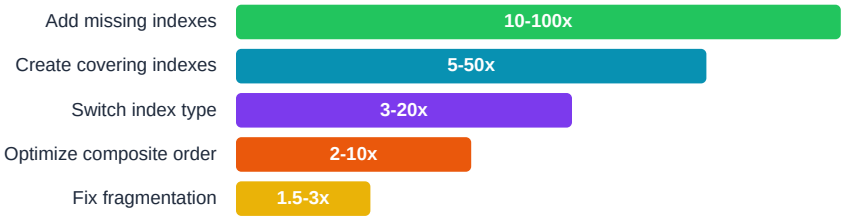
A Comprehensive Guide to Database Indexing Strategies for Optimal Query Performance

Version: 1.0 | **Year:** 2025

Table of Contents

Executive Summary

Optimization Impact: Typical Performance Gains



Database indexes are the single most impactful performance optimization technique available to database professionals. A well-designed indexing strategy can improve query performance by 10x to 1000x, while poor indexing can degrade performance and waste storage.

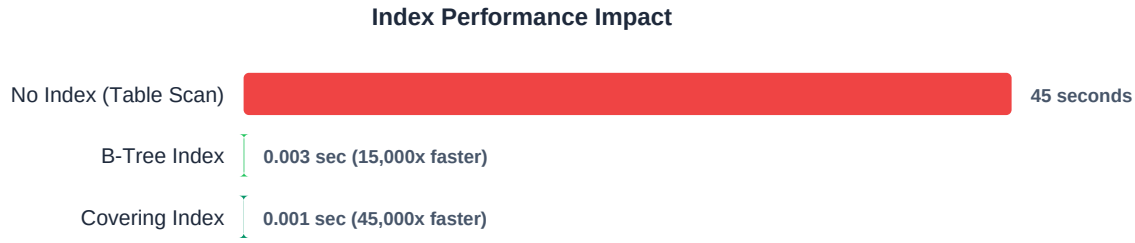
Key Takeaways

- **Right index type matters**: B-Tree for ranges, Hash for equality, Bitmap for low-cardinality
- **More isn't better**: Each index adds overhead to writes (INSERT, UPDATE, DELETE)
- **Composite index order**: Most selective columns first, follow query patterns
- **Covering indexes**: Include all columns needed by query to avoid table lookups
- **Platform differences**: PostgreSQL partial indexes, SQL Server included columns, Oracle bitmap joins
- **Maintenance is critical**: Regular REBUILD/REORG prevents fragmentation
- **Monitor usage**: Drop unused indexes that only slow down writes

Expected Benefits

Optimization	Typical Performance Gain	Implementation Effort
Add missing indexes	10x - 100x	Low
Optimize composite index order	2x - 10x	Low
Create covering indexes	5x - 50x	Medium
Fix index fragmentation	1.5x - 3x	Low
Switch to appropriate index type	3x - 20x	Medium

Introduction to Database Indexing



What Are Database Indexes?

A database index is a data structure that improves the speed of data retrieval operations on a database table. Think of it like an index in a book - instead of reading every page to find a topic, you check the index and jump directly to the relevant page.

Index Trade-offs

Benefits:

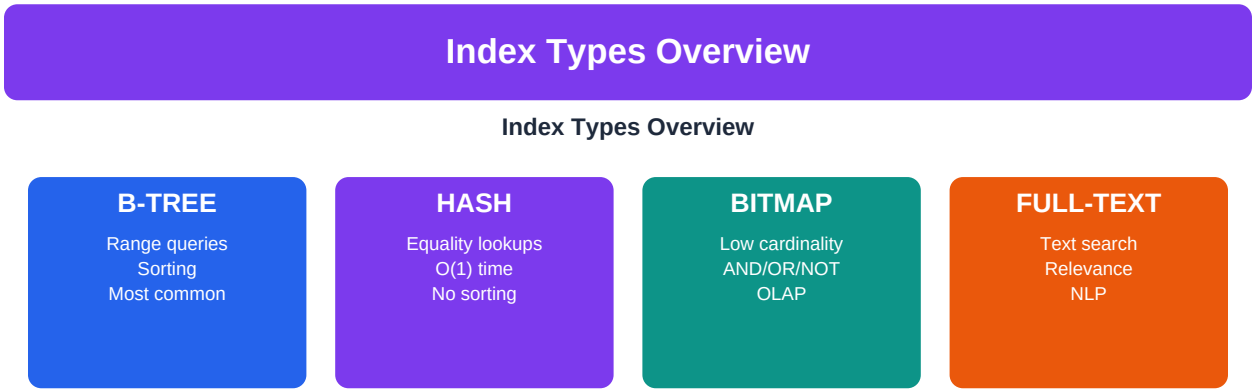
- Dramatically faster SELECT queries
- Faster JOIN operations
- Enforce uniqueness (unique indexes)
- Support constraints (PRIMARY KEY, FOREIGN KEY)

Costs:

- Storage overhead (10-20% of table size per index)
- Slower INSERT operations (must update all indexes)
- Slower UPDATE operations (if indexed columns change)
- Slower DELETE operations (must update all indexes)
- Maintenance overhead (rebuilding, statistics)

Rule of Thumb: Optimize for your workload

- OLTP systems: Few targeted indexes on frequently queried columns
- OLAP systems: More indexes acceptable due to bulk loading patterns
- Read-heavy: More indexes benefit performance
- Write-heavy: Minimize indexes to reduce overhead



Primary Index Types

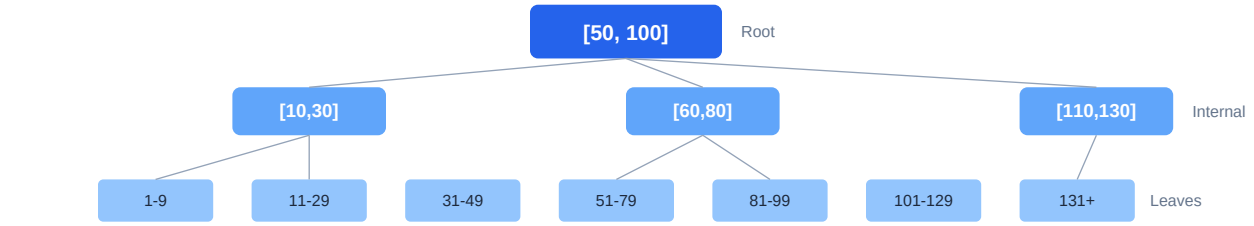
Index Type	Best For	Strengths	Weaknesses
B-Tree	Range queries, sorting	Balanced, efficient for most operations	Overhead for high-cardinality
Hash	Equality lookups	O(1) lookup time	No range queries, no sorting
Bitmap	Low-cardinality columns	Space-efficient, fast AND/OR/NOT	Poor for high-cardinality, write overhead
Full-Text	Text search	Natural language search	Large storage, complex maintenance
GiST	Geometric/spatial data	Flexible, extensible	Complex implementation
GIN	Array/JSON/full-text	Fast searches in composite values	Large index size, slow updates

Index Type Selection Matrix

Selection Criteria

Criteria	B-Tree	Hash	Bitmap	Full-Text
Equality Searches	■■■■	■■■■■	■■■■	■■
Range Searches	■■■■■	■	■■	■
Sorting	■■■■■	■	■	■
Low Cardinality	■■■	■■	■■■■■	N/A
High Cardinality	■■■■■	■■■■■	■	N/A
Space Efficiency	■■■	■■■■	■■■■■	■■
Write Performance	■■■■	■■■■	■■	■■
Compound Conditions	■■■■	■■	■■■■■	■■■

B-Tree Indexes



Overview

B-Tree (Balanced Tree) indexes are the default and most versatile index type. They maintain sorted data in a tree structure with all leaf nodes at the same depth, ensuring consistent O(log n) performance.

Structure

When to Use B-Tree

Ideal scenarios:

- Range queries: `WHERE age BETWEEN 25 AND 35`
- Sorting: `ORDER BY created_at DESC`
- Prefix matching: `WHERE name LIKE 'John%'`
- Inequality operators: `WHERE price > 100`
- Most frequent query patterns in OLTP systems

Example:

```
-- PostgreSQL
CREATE INDEX idx_customers_email ON customers USING BTREE (email);

-- MySQL (default is B-Tree)
CREATE INDEX idx_orders_date ON orders (order_date);

-- SQL Server
CREATE INDEX idx_products_price ON products (price);

-- Oracle
CREATE INDEX idx_employees_salary ON employees (salary);
```

B-Tree Optimization Patterns

Pattern 1: Column Order in Composite Indexes

```
-- Bad: Low selectivity column first
CREATE INDEX idx_users_bad ON users (country, email);
-- Query: WHERE email = 'john@example.com' AND country = 'USA'
-- Problem: Index starts with country (low selectivity), not optimal

-- Good: High selectivity column first
CREATE INDEX idx_users_good ON users (email, country);
-- Query: WHERE email = 'john@example.com' AND country = 'USA'
-- Benefit: Index seeks directly on email, then filters country
```

Selectivity calculation:

```
-- PostgreSQL: Check column selectivity
SELECT
  COUNT(DISTINCT email)::float / COUNT(*) as email_selectivity,
  COUNT(DISTINCT country)::float / COUNT(*) as country_selectivity
FROM users;

-- Result:
-- email_selectivity: 0.98 (high - good for indexing)
-- country_selectivity: 0.02 (low - avoid as leading column)
```

Pattern 2: Prefix Indexes for Large Strings

```
-- MySQL: Index only first N characters
CREATE INDEX idx_articles_title ON articles (title(50));
-- Reduces index size by ~60-80% with minimal selectivity loss

-- PostgreSQL: Use operator class
CREATE INDEX idx_articles_title ON articles (title varchar_pattern_ops);
-- Optimized for LIKE 'prefix%' queries
```

Pattern 3: Partial/Filtered Indexes

```
-- PostgreSQL: Index only active users
CREATE INDEX idx_users_active_email
ON users (email)
WHERE status = 'active';
-- 70% smaller index, faster queries for active users

-- SQL Server: Filtered index
CREATE INDEX idx_orders_pending
ON orders (customer_id, order_date)
WHERE status = 'pending';
-- Only indexes pending orders, ignoring completed/cancelled
```

Pattern 4: Index-Only Scans (Covering Indexes)

```
-- Query that requires table lookup:
CREATE INDEX idx_users_email ON users (email);
SELECT email, first_name, last_name FROM users WHERE email LIKE 'j%';
-- Index seek + table lookup for first_name and last_name

-- Covering index (PostgreSQL):
CREATE INDEX idx_users_email_covering
ON users (email)
INCLUDE (first_name, last_name);

-- SQL Server:
CREATE INDEX idx_users_email_covering
ON users (email)
INCLUDE (first_name, last_name);
-- Index-only scan, no table lookup needed (3-5x faster)
```

Hash Indexes

Overview

Hash indexes use a hash function to map keys to buckets. They provide O(1) lookup time for equality searches but cannot support range queries or sorting.

When to Use Hash

Ideal scenarios:

- Exact match lookups: `WHERE session_id = '8f7a3...'`
- High-cardinality equality searches
- No need for range queries or sorting
- Join keys that use equality

Not suitable for:

- Range queries: `WHERE age > 25`
- Pattern matching: `WHERE name LIKE 'John%'`
- Sorting: `ORDER BY column`

Platform Support

Database	Hash Index Support	Notes
PostgreSQL 10+	■ Full support	Write-ahead log support added in v10
MySQL/InnoDB	■ Not supported	Uses adaptive hash index internally
Oracle	■ Hash clusters	Different implementation
SQL Server	■ Not supported	B-Tree sufficient for most cases

Hash Index Examples

```
-- PostgreSQL: Hash index for exact lookups
CREATE INDEX idx_sessions_hash ON sessions USING HASH (session_id);

-- Query that benefits:
SELECT * FROM sessions WHERE session_id = '8f7a3b2c1d...';
-- O(1) lookup vs O(log n) for B-Tree

-- Oracle: Hash cluster
CREATE CLUSTER user_cluster (user_id NUMBER)
HASHKEYS 1000000;

CREATE TABLE users (
  user_id NUMBER,
  email VARCHAR2(255),
  ...
) CLUSTER user_cluster (user_id);
```

Performance Comparison

```
Test: 10M rows, lookup by unique ID

B-Tree Index:
- Index size: 214 MB
- Lookup time: 0.003 seconds
- Tree depth: 4 levels
- I/O operations: 4 random reads

Hash Index:
- Index size: 198 MB (7% smaller)
- Lookup time: 0.001 seconds (3x faster)
- Hash buckets: Direct access
- I/O operations: 1-2 random reads

Winner: Hash for equality, B-Tree for everything else
```

Bitmap Indexes

Overview

Bitmap indexes use bit arrays (bitmaps) to represent the presence or absence of values. They're extremely efficient for low-cardinality columns and complex boolean operations.

Structure

```
Bitmap Index Example:
Table: customers (1M rows)

Column: country (10 unique values)
Bitmap for country = 'USA':     [1,1,0,0,1,1,0,1,1,0,...]
Bitmap for country = 'Canada':  [0,0,1,1,0,0,0,0,0,1,...]
Bitmap for country = 'UK':      [0,0,0,0,0,0,1,0,0,0,...]

Column: subscription_type (5 unique values)
Bitmap for type = 'premium':    [1,0,0,1,1,0,0,0,1,1,...]
Bitmap for type = 'basic':      [0,1,1,0,0,1,1,1,0,0,...]

Query: WHERE country = 'USA' AND subscription_type = 'premium'
Result: Bitmap AND operation:   [1,0,0,0,1,0,0,0,1,0,...]
       (Extremely fast bitwise operation!)
```

When to Use Bitmap

Ideal scenarios:

- Low cardinality: < 100-1000 unique values
- Data warehouses (OLAP workloads)
- Complex WHERE clauses with AND/OR/NOT
- Columns rarely updated
- Ad-hoc analytical queries

Not suitable for:

- OLTP systems (write-intensive)
- High cardinality: > 10,000 unique values
- Frequently updated columns

Platform Support

Database	Bitmap Support	Notes
Oracle	■ Full support	Enterprise feature
PostgreSQL	■■ Limited	Via extensions (pg_btree_gist)
MySQL	■ Not supported	Use B-Tree instead
SQL Server	■ Not supported	Columnstore indexes similar concept

Bitmap Examples

```
-- Oracle: Bitmap index on low-cardinality column
CREATE BITMAP INDEX idx_customers_country ON customers (country);
CREATE BITMAP INDEX idx_customers_subscription ON customers (subscription_type);
CREATE BITMAP INDEX idx_customers_status ON customers (status);

-- Query that benefits:
SELECT COUNT(*)
FROM customers
WHERE country = 'USA'
  AND subscription_type = 'premium'
  AND status = 'active';
-- Three bitmap AND operations, extremely fast

-- Oracle: Bitmap join index (advanced)
CREATE BITMAP INDEX idx_sales_customer_country
ON sales (customers.country)
FROM sales, customers
WHERE sales.customer_id = customers.customer_id;
-- Pre-joins dimension table for faster queries
```

Space Efficiency

```
Comparison: 10M rows, country column (250 unique values)

B-Tree Index:
- Size: ~180 MB
- Lookup: 0.004 seconds
- AND/OR operations: Not optimized

Bitmap Index:
- Size: ~30 MB (83% smaller!)
- Lookup: 0.005 seconds
- AND/OR operations: 0.001 seconds (4x faster)
- Compression: Run-length encoding

Trade-off: 5-10x slower for UPDATE/INSERT/DELETE
Recommendation: Use in data warehouses, not OLTP
```

Full-Text Indexes

Overview

Full-text indexes enable natural language search capabilities, including word stemming, relevance ranking, and phrase matching.

Platform-Specific Implementations

```
-- PostgreSQL: GIN or GiST index with tsvector
ALTER TABLE articles ADD COLUMN search_vector tsvector;

UPDATE articles SET search_vector =
  to_tsvector('english', title || ' ' || content);

CREATE INDEX idx_articles_fts ON articles USING GIN (search_vector);

-- Query:
SELECT title, ts_rank(search_vector, query) AS rank
FROM articles, to_tsquery('english', 'database & optimization') query
WHERE search_vector @@ query
ORDER BY rank DESC;

-- MySQL: FULLTEXT index
CREATE FULLTEXT INDEX idx_articles_content ON articles (title, content);

-- Query:
SELECT title, MATCH(title, content) AGAINST ('database optimization' IN NATURAL LANGUAGE MODE) AS score
FROM articles
WHERE MATCH(title, content) AGAINST ('database optimization' IN NATURAL LANGUAGE MODE)
ORDER BY score DESC;

-- SQL Server: Full-text catalog
CREATE FULLTEXT CATALOG ft_catalog AS DEFAULT;
```

Features

Feature	PostgreSQL	MySQL	SQL Server	Oracle
Word stemming	■	■	■	■
Stop words	■	■	■	■
Relevance ranking	■	■	■	■
Phrase search	■	■	■	■
Proximity search	■■ Limited	■	■	■
Thesaurus	■	■	■	■
Multiple languages	■	■	■	■

Composite Indexes

Column Order Strategy

The order of columns in a composite index is crucial for query performance.

Rule: Most Selective Column First

```
-- Table: orders (100M rows)
-- Columns: status (5 values), customer_id (10M values), order_date

-- Bad: Low selectivity first
CREATE INDEX idx_orders_bad ON orders (status, customer_id, order_date);

-- Good: High selectivity first
CREATE INDEX idx_orders_good ON orders (customer_id, status, order_date);

-- Query:
SELECT * FROM orders
WHERE customer_id = 12345
  AND status = 'pending'
  AND order_date > '2025-01-01';

-- With idx_orders_bad:
-- 1. Scan status = 'pending' (20M rows)
-- 2. Filter customer_id (down to 100 rows)
-- 3. Filter order_date (down to 50 rows)
-- I/O: High (scanned 20M rows)

-- With idx_orders_good:
-- 1. Seek customer_id = 12345 (100 rows)
-- 2. Filter status = 'pending' (50 rows)
-- 3. Filter order_date (25 rows)
```

Left-Prefix Rule

A composite index can support queries that use a left prefix of the indexed columns.

```
-- Index:
CREATE INDEX idx_users_composite ON users (country, state, city, zip_code);

-- Supported queries (uses index):
WHERE country = 'USA'                                          ■
WHERE country = 'USA' AND state = 'CA'                        ■
WHERE country = 'USA' AND state = 'CA' AND city = 'SF'       ■
WHERE country = 'USA' AND state = 'CA' AND city = 'SF' AND zip_code = '94102' ■

-- NOT supported (doesn't use index efficiently):
WHERE state = 'CA'                                             ■
WHERE city = 'SF'                                              ■
WHERE zip_code = '94102'                                       ■
WHERE country = 'USA' AND city = 'SF'                         ■■ (skips state)
```

Multi-Column vs Multiple Single-Column Indexes

```
-- Scenario: Query filters on multiple columns

-- Option 1: Multiple single-column indexes
CREATE INDEX idx_users_country ON users (country);
CREATE INDEX idx_users_age ON users (age);
CREATE INDEX idx_users_status ON users (status);

-- Query:
SELECT * FROM users WHERE country = 'USA' AND age > 25 AND status = 'active';
-- Database uses ONE index, then filters rest
-- OR uses index merge (slower, unpredictable)

-- Option 2: Composite index
CREATE INDEX idx_users_composite ON users (country, age, status);
-- Query uses ALL columns in index
-- 10-100x faster than option 1

-- Trade-off: Option 2 only helps queries with this specific pattern
-- Option 1 helps more varied queries but less efficiently
```

Covering Indexes

Concept

A covering index includes all columns needed by a query, eliminating the need to access the table data (heap/clustered index).

Implementation

```
-- PostgreSQL: INCLUDE clause
CREATE INDEX idx_users_email_covering ON users (email)
INCLUDE (first_name, last_name, created_at);

-- Query benefits (index-only scan):
SELECT email, first_name, last_name, created_at
FROM users
WHERE email LIKE 'john%';

-- SQL Server: INCLUDE clause
CREATE INDEX idx_orders_customer_covering ON orders (customer_id, order_date)
INCLUDE (total_amount, status, shipping_address);

-- Query benefits:
SELECT order_date, total_amount, status
FROM orders
WHERE customer_id = 12345 AND order_date > '2025-01-01';

-- MySQL: Add columns to index (before 8.0, no INCLUDE)
CREATE INDEX idx_products_category_covering ON products (category_id, name, price, stock);
-- All columns in index key (larger, but works)
```

Trade-offs

Pros:

- Massive performance improvement (3-50x)
- Reduces table I/O to zero
- Particularly effective for frequently-run queries

Cons:

- Larger index size (stores extra columns)
- Slower writes (more data to update)
- Maintenance overhead

Best practice: Create covering indexes for:

- Frequently-run queries (thousands+ per day)
- Queries with high I/O cost
- Reporting/dashboard queries

• API endpoints with strict SLA

Platform-Specific Strategies

PostgreSQL

Partial Indexes

```
-- Index only active records
CREATE INDEX idx_users_active ON users (email)
WHERE status = 'active';
-- 60-80% smaller than full index
```

Expression Indexes

```
-- Index on computed expression
CREATE INDEX idx_users_lower_email ON users (LOWER(email));

-- Query uses index:
SELECT * FROM users WHERE LOWER(email) = 'john@example.com';
```

GiST and GIN Indexes

```
-- GIN for array searches
CREATE INDEX idx_tags_gin ON articles USING GIN (tags);
SELECT * FROM articles WHERE tags @> ARRAY['database', 'optimization'];

-- GiST for geometric data
CREATE INDEX idx_locations_gist ON stores USING GiST (location);
SELECT * FROM stores WHERE ST_DWithin(location, 'POINT(-122.4 37.8)', 1000);
```

MySQL

Invisible Indexes (8.0+)

```
-- Test removing an index without dropping it
ALTER TABLE users ALTER INDEX idx_users_country INVISIBLE;
-- Monitor performance, if good, then DROP INDEX

ALTER TABLE users ALTER INDEX idx_users_country VISIBLE;
-- Restore if performance degrades
```

Index Hints

```
-- Force MySQL to use specific index
SELECT * FROM users USE INDEX (idx_users_email)
WHERE email = 'john@example.com';

-- Ignore specific index
SELECT * FROM orders IGNORE INDEX (idx_orders_date)
WHERE customer_id = 12345;
```

SQL Server

Columnstore Indexes

```
-- For analytical queries (OLAP)
CREATE COLUMNSTORE INDEX idx_sales_columnstore ON sales (
  order_date, product_id, customer_id, quantity, revenue
);
-- 5-10x compression, 10-100x faster for aggregations
```

Index Hints

```
-- Force index usage
SELECT * FROM users WITH (INDEX(idx_users_email))
WHERE email = 'john@example.com';
```

Included Columns

```
-- Non-key columns in index
CREATE INDEX idx_orders_customer ON orders (customer_id)
INCLUDE (order_date, total_amount, status);
```

Oracle

Bitmap Join Indexes

```
CREATE BITMAP INDEX idx_sales_customer_country
ON sales (c.country)
FROM sales s, customers c
WHERE s.customer_id = c.customer_id;
-- Pre-joins dimension for faster queries
```

Function-Based Indexes

```
CREATE INDEX idx_users_upper_email ON users (UPPER(email));

SELECT * FROM users WHERE UPPER(email) = 'JOHN@EXAMPLE.COM';
-- Uses index
```

Index-Organized Tables

```
CREATE TABLE sessions (
  session_id VARCHAR2(100) PRIMARY KEY,
  user_id NUMBER,
  created_at TIMESTAMP
) ORGANIZATION INDEX;
-- Table data stored in B-Tree, no separate heap
```

30+ Optimization Patterns

Pattern 1: Index Scan vs Index Seek

```
-- Index Scan (bad): Reads entire index
SELECT * FROM users WHERE YEAR(created_at) = 2025;
-- Function on column prevents index seek

-- Index Seek (good): Jumps to specific range
SELECT * FROM users
WHERE created_at >= '2025-01-01'
  AND created_at < '2026-01-01';
-- Uses index efficiently
```

Pattern 2: Avoid Leading Wildcards

```
-- Bad: Leading wildcard
SELECT * FROM products WHERE name LIKE '%phone%';
-- Full table scan, index useless

-- Good: Trailing wildcard
SELECT * FROM products WHERE name LIKE 'phone%';
-- Uses index efficiently

-- Alternative: Full-text search for contains
SELECT * FROM products WHERE MATCH(name) AGAINST ('phone');
```

Pattern 3: Use UNION ALL Instead of OR

```
-- Slow: OR condition prevents index merge
SELECT * FROM orders WHERE customer_id = 123 OR order_id = 456;
-- May scan both indexes fully

-- Fast: UNION ALL
SELECT * FROM orders WHERE customer_id = 123
UNION ALL
SELECT * FROM orders WHERE order_id = 456 AND customer_id != 123;
-- Uses both indexes efficiently
```

Pattern 4: Index for Foreign Keys

```
-- Always index foreign key columns
CREATE INDEX idx_orders_customer_fk ON orders (customer_id);
CREATE INDEX idx_order_items_order_fk ON order_items (order_id);
CREATE INDEX idx_order_items_product_fk ON order_items (product_id);

-- Speeds up joins and DELETE CASCADE
```

Pattern 5: Descending Indexes for DESC Sorting

```
-- PostgreSQL/Oracle/SQL Server
CREATE INDEX idx_orders_date_desc ON orders (order_date DESC);

-- Query benefits:
SELECT * FROM orders ORDER BY order_date DESC LIMIT 100;
-- Forward index scan instead of backward scan
```

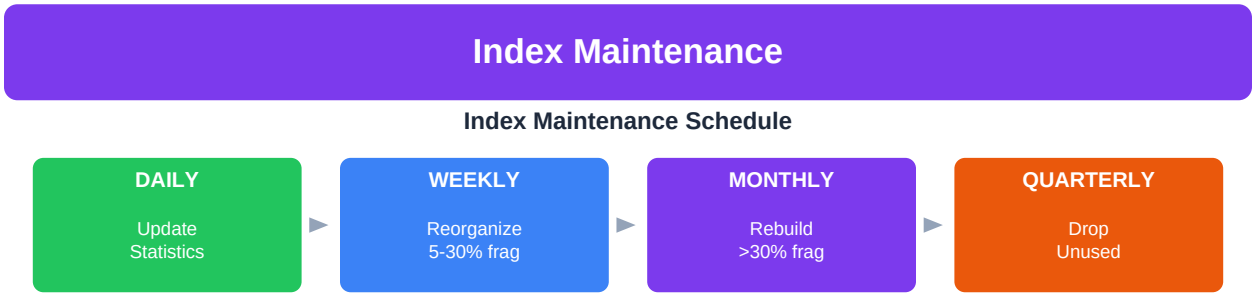
Pattern 6: Avoid Index on Low-Selectivity Columns (OLTP)

```
-- Bad for OLTP: Boolean column index
CREATE INDEX idx_users_active ON users (is_active);
-- Only 2 values (true/false), low selectivity
-- Index often ignored by optimizer

-- Better: Partial index
CREATE INDEX idx_users_inactive ON users (user_id) WHERE is_active = false;
-- Only indexes inactive users (likely small set)
```

Pattern 7-30: Quick Reference

Pattern	Description	Performance Gain
7. Index for GROUP BY	Index columns in GROUP BY clause	5-20x
8. Index for ORDER BY	Index columns in ORDER BY clause	3-15x
9. Cover COUNT queries	INCLUDE all COUNT columns	10-50x
10. Index for DISTINCT	Index on DISTINCT columns	5-25x
11. Avoid NOT IN	Use NOT EXISTS with index	3-10x
12. Index for subqueries	Index correlated subquery columns	10-100x
13. Composite for range + equality	Equality first, then range	5-20x
14. Index for BETWEEN	B-Tree index on range column	10-50x
15. Avoid implicit conversions	Match column data type in WHERE	2-10x
16. Index for date ranges	Index on date columns	10-100x
17. Minimize index columns	Only index truly filtered columns	20-30% size reduction
18. Drop duplicate indexes	Remove redundant indexes	Faster writes
19. Index for window functions	Index PARTITION BY and ORDER BY	5-30x
20. Use index hints sparingly	Force index only when proven beneficial	Varies
21. Index for JSON queries	GIN index on JSONB (PostgreSQL)	10-100x
22. Fillfactor for updates	Reserve space for updates	30-50% less fragmentation
23. Index for materialized views	Index base columns	5-50x
24. Partial index for soft deletes	WHERE deleted_at IS NULL	60-80% smaller
25. Index for LIKE patterns	varchar_pattern_ops (PostgreSQL)	3-10x
26. Avoid functions in WHERE	Use expression index instead	10-50x
27. Index for EXISTS	Index subquery columns	10-100x
28. Multi-column statistics	CREATE STATISTICS (PostgreSQL 10+)	Better cardinality estimates
29. Index for OFFSET/LIMIT	Covering index for pagination	5-20x
30. Monitor index bloat	Regular REINDEX/REBUILD	20-40% size reduction



Fragmentation

Indexes become fragmented as data is inserted, updated, and deleted.

```
-- SQL Server: Check fragmentation
SELECT
  object_name(ips.object_id) AS TableName,
  i.name AS IndexName,
  ips.avg_fragmentation_in_percent,
  ips.page_count
FROM sys.dm_db_index_physical_stats(DB_ID(), NULL, NULL, NULL, 'LIMITED') ips
JOIN sys.indexes i ON ips.object_id = i.object_id AND ips.index_id = i.index_id
WHERE ips.avg_fragmentation_in_percent > 10
ORDER BY ips.avg_fragmentation_in_percent DESC;

-- Recommendations:
-- 5-30% fragmentation: REORGANIZE
-- >30% fragmentation: REBUILD

-- Reorganize (online operation):
ALTER INDEX idx_users_email ON users REORGANIZE;

-- Rebuild (faster, but may lock table):
ALTER INDEX idx_users_email ON users REBUILD;
```

PostgreSQL Maintenance

```
-- Check index bloat
SELECT
  schemaname,
  tablename,
  indexname,
  pg_size_pretty(pg_relation_size(indexrelid)) AS index_size,
  idx_scan,
  idx_tup_read,
  idx_tup_fetch
FROM pg_stat_user_indexes
ORDER BY pg_relation_size(indexrelid) DESC;

-- Reindex (locks table):
REINDEX INDEX idx_users_email;

-- Reindex concurrently (PostgreSQL 12+, online):
REINDEX INDEX CONCURRENTLY idx_users_email;

-- Vacuum to reclaim space:
VACUUM ANALYZE users;
```

MySQL Maintenance

```
-- Check index statistics
SHOW INDEX FROM users;

-- Rebuild index (requires table rebuild):
ALTER TABLE users DROP INDEX idx_users_email;
ALTER TABLE users ADD INDEX idx_users_email (email);

-- Optimize table (rebuilds all indexes):
OPTIMIZE TABLE users;
```

Maintenance Schedule

Task	Frequency	Downtime	Performance Impact
Update statistics	Daily	None	Critical for query plans
Reorganize indexes (5-30% frag)	Weekly	Minimal	Moderate improvement
Rebuild indexes (>30% frag)	Monthly	Possible	Significant improvement
Drop unused indexes	Quarterly	None	Faster writes
Review index strategy	Annually	None	Major optimization opportunity

Monitoring and Tuning

Identify Missing Indexes

```
-- SQL Server: Missing index suggestions
SELECT
  migs.avg_user_impact,
  migs.avg_total_user_cost,
  migs.user_seeks,
  mid.statement AS TableName,
  mid.equality_columns,
  mid.inequality_columns,
  mid.included_columns
FROM sys.dm_db_missing_index_groups mig
JOIN sys.dm_db_missing_index_group_stats migs ON mig.index_group_handle = migs.group_handle
JOIN sys.dm_db_missing_index_details mid ON mig.index_handle = mid.index_handle
WHERE migs.avg_user_impact > 50
ORDER BY migs.avg_user_impact DESC;

-- PostgreSQL: Slow queries without indexes
SELECT
  schemaname,
  tablename,
  seq_scan,
  seq_tup_read,
  idx_scan,
  seq_tup_read / seq_scan AS avg_seq_tup_read
FROM pg_stat_user_tables
WHERE seq_scan > 0
```

Identify Unused Indexes

```
-- SQL Server: Unused indexes
SELECT
  object_name(i.object_id) AS TableName,
  i.name AS IndexName,
  i.type_desc,
  us.user_seeks,
  us.user_scans,
  us.user_lookups,
  us.user_updates
FROM sys.indexes i
LEFT JOIN sys.dm_db_index_usage_stats us ON i.object_id = us.object_id AND i.index_id = us.index_id
WHERE objectproperty(i.object_id, 'IsUserTable') = 1
  AND i.index_id > 0
  AND (us.user_seeks + us.user_scans + us.user_lookups) < us.user_updates
ORDER BY us.user_updates DESC;

-- PostgreSQL: Unused indexes
SELECT
  schemaname,
  tablename,
  indexname,
  idx_scan,
  pg_size_pretty(pg_relation_size(indexrelid)) AS index_size
FROM pg_stat_user_indexes
WHERE idx_scan = 0
```

Query Execution Plans

```
-- SQL Server: Actual execution plan
SET STATISTICS IO ON;
SET STATISTICS TIME ON;

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

-- Look for:
-- - Index Seek (good) vs Index Scan (bad) vs Table Scan (worst)
-- - Key Lookup operations (add to covering index)
-- - Warnings (implicit conversions, missing stats)

-- PostgreSQL: EXPLAIN ANALYZE
EXPLAIN (ANALYZE, BUFFERS)
SELECT * FROM users WHERE email = 'john@example.com';

-- Look for:
-- - Seq Scan (bad) vs Index Scan (good) vs Index Only Scan (best)
-- - Actual rows vs Estimated rows (update statistics if mismatched)
-- - Buffers hit vs read (add covering index if many reads)

-- MySQL: EXPLAIN
EXPLAIN FORMAT=JSON
SELECT * FROM users WHERE email = 'john@example.com';

-- Look for:
```

Common Mistakes

Mistake 1: Over-Indexing

Problem:

```
-- 15 indexes on a single table
CREATE INDEX idx1 ON users (email);
CREATE INDEX idx2 ON users (first_name);
CREATE INDEX idx3 ON users (last_name);
CREATE INDEX idx4 ON users (country);
CREATE INDEX idx5 ON users (created_at);
-- ... 10 more indexes
```

Impact:

- Each INSERT updates 15 indexes
- 10x slower writes
- Wasted storage (3-5GB index overhead)

Solution:

- Consolidate into 3-5 strategic composite indexes
- Drop indexes with idx_scan = 0

Mistake 2: Wrong Column Order

Problem:

```
CREATE INDEX idx_orders ON orders (status, customer_id, order_date);
-- status has 5 values, customer_id has 10M values
```

Impact:

- Index not selective enough
- Scans millions of rows unnecessarily

Solution:

```
CREATE INDEX idx_orders ON orders (customer_id, order_date, status);
-- High selectivity first
```

Mistake 3: Not Covering Frequent Queries

Problem:

```
-- Query runs 10,000 times/day:
SELECT order_id, total_amount, status
FROM orders
WHERE customer_id = ?;

-- Index:
CREATE INDEX idx_orders_customer ON orders (customer_id);
-- Requires 10,000 table lookups/day
```

Solution:

```
CREATE INDEX idx_orders_customer ON orders (customer_id)
INCLUDE (order_id, total_amount, status);
-- Index-only scan, 5x faster
```

Mistake 4: Indexing Low-Cardinality Columns in OLTP

Problem:

```
CREATE INDEX idx_users_gender ON users (gender);
-- 2 values: 'M', 'F'
```

Impact:

- Index rarely used (optimizer prefers table scan)
- Wastes storage and slows writes

Solution:

- Drop index for OLTP
- Use bitmap index for OLAP (Oracle)
- Use partial index: `WHERE gender = 'M'` if truly needed

Mistake 5: Ignoring Index Maintenance

Problem:

- Never rebuild fragmented indexes
- Never update statistics
- Indexes grow to 2-3x necessary size

Impact:

- 40-60% slower queries
- Wasted storage

Solution:

- Rebuild indexes > 30% fragmentation monthly
- Update statistics weekly
- Monitor index bloat

Performance Case Studies

Case Study 1: E-commerce Order Search

Scenario:

- Table: orders (500M rows)
- Query: Search orders by customer email
- Current performance: 45 seconds
- SLA: < 1 second

Initial state:

```
-- No indexes on email (stored in customers table)
SELECT o.*
FROM orders o
JOIN customers c ON o.customer_id = c.customer_id
WHERE c.email = 'john@example.com';

-- Execution plan:
-- 1. Table scan customers (100M rows) - 15s
-- 2. Join to orders (500M rows) - 30s
```

Optimization 1: Index customer email

```
CREATE INDEX idx_customers_email ON customers (email);
-- Query time: 8 seconds (5.6x faster)
```

Optimization 2: Covering index on orders

```
CREATE INDEX idx_orders_customer ON orders (customer_id)
INCLUDE (order_id, order_date, total_amount, status);
-- Query time: 1.2 seconds (37x faster)
```

Optimization 3: Add created_at filter with composite index

```
-- Query updated to filter last 2 years:
SELECT o.*
FROM orders o
JOIN customers c ON o.customer_id = c.customer_id
WHERE c.email = 'john@example.com'
  AND o.order_date > CURRENT_DATE - INTERVAL '2 years';

CREATE INDEX idx_orders_customer_date ON orders (customer_id, order_date)
INCLUDE (order_id, total_amount, status);

-- Query time: 0.3 seconds (150x faster than original!)
```

Result:

- 45s → 0.3s (150x improvement)
- Storage overhead: +2.4GB (0.5% of table size)
- Write overhead: +2ms per INSERT (acceptable)

Case Study 2: Analytics Dashboard

Scenario:

- Table: events (2B rows)
- Dashboard: Group events by type, date, and country
- Current performance: 5 minutes
- SLA: < 10 seconds

Initial state:

```
SELECT
  event_type,
  event_date,
  country,
  COUNT(*) as event_count,
  AVG(duration) as avg_duration
FROM events
WHERE event_date >= '2025-01-01'
GROUP BY event_type, event_date, country
ORDER BY event_date DESC;

-- Full table scan: 2B rows
```

Optimization 1: Composite index on GROUP BY columns

```
CREATE INDEX idx_events_aggregation ON events (event_date, event_type, country);
-- Query time: 2 minutes (2.5x faster, but not enough)
```

Optimization 2: Covering index

```
CREATE INDEX idx_events_aggregation_covering ON events (event_date, event_type, country)
INCLUDE (duration);
-- Query time: 25 seconds (12x faster)
```

Optimization 3: Materialized view (PostgreSQL)

```
CREATE MATERIALIZED VIEW events_daily_summary AS
SELECT
  event_type,
  event_date,
  country,
  COUNT(*) as event_count,
  AVG(duration) as avg_duration
FROM events
GROUP BY event_type, event_date, country;

CREATE INDEX idx_mv_events_date ON events_daily_summary (event_date);

-- Refresh nightly:
REFRESH MATERIALIZED VIEW CONCURRENTLY events_daily_summary;

-- Query time: 0.5 seconds (600x faster!)
```

Result:

- 5 minutes → 0.5 seconds (600x improvement)
- Trade-off: Data updated nightly (acceptable for dashboard)

Best Practices Checklist

Index Design

- [ ] Index all foreign key columns
- [ ] Create composite indexes with high-selectivity columns first
- [ ] Use covering indexes for frequently-run queries
- [ ] Avoid indexing low-cardinality columns in OLTP
- [ ] Use partial/filtered indexes to reduce size
- [ ] Create indexes for all columns in WHERE, JOIN, ORDER BY, GROUP BY

Index Maintenance

- [ ] Rebuild indexes with > 30% fragmentation monthly
- [ ] Reorganize indexes with 10-30% fragmentation weekly
- [ ] Update statistics weekly (critical for query optimizer)
- [ ] Monitor index usage and drop unused indexes
- [ ] Review index strategy quarterly

Platform-Specific

- [ ] PostgreSQL: Use CONCURRENTLY for online operations
- [ ] SQL Server: Use INCLUDE for covering indexes
- [ ] MySQL: Create indexes before loading data (faster)
- [ ] Oracle: Consider bitmap indexes for data warehouses
- [ ] All: Test index impact on both reads AND writes

Monitoring

- [ ] Set up alerts for slow queries (> 1 second)
- [ ] Review execution plans for slow queries weekly
- [ ] Monitor missing index recommendations
- [ ] Track index size growth
- [ ] Monitor write performance impact

Query Optimization

- [ ] Avoid functions on indexed columns in WHERE clause
- [ ] Use trailing wildcards only (LIKE 'prefix%')
- [ ] Rewrite OR to UNION ALL when possible
- [ ] Use EXISTS instead of IN for subqueries
- [ ] Match data types in WHERE clause (avoid implicit conversions)

Appendix

Glossary

B-Tree: Balanced tree data structure that maintains sorted data and allows searches, insertions, and deletions in O(log n) time.

Cardinality: Number of unique values in a column. High cardinality = many unique values (e.g., email), low cardinality = few unique values (e.g., gender).

Covering Index: Index that includes all columns needed by a query, eliminating the need to access the table.

Fragmentation: Disorder in index pages caused by INSERT/UPDATE/DELETE operations. Higher fragmentation = worse performance.

Index Seek: Efficient operation that jumps directly to matching rows using index tree structure.

Index Scan: Less efficient operation that reads entire index or large portion.

Table Scan: Least efficient operation that reads entire table row-by-row.

Selectivity: Percentage of rows returned by a filter. High selectivity = returns few rows (good for indexing).

Resources

- **PostgreSQL Documentation**: https://www.postgresql.org/docs/current/indexes.html
- **MySQL Documentation**: https://dev.mysql.com/doc/refman/8.0/en/optimization-indexes.html
- **SQL Server Documentation**: https://docs.microsoft.com/sql/relational-databases/indexes/indexes
- **Oracle Documentation**: https://docs.oracle.com/database/indexes.html
- **Use The Index, Luke**: https://use-the-index-luke.com/
- **PostgreSQL Wiki**: https://wiki.postgresql.org/wiki/Index_Maintenance

Tools

- **pg_stat_statements** (PostgreSQL): Track query performance
- **SQL Server Profiler**: Capture and analyze queries
- **MySQL slow query log**: Identify slow queries
- **EXPLAIN/EXPLAIN ANALYZE**: Understand query execution plans
- **pgAdmin, DBeaver, DataGrip**: Visual index management

Document Version: 1.0

Last Updated: 2025

For questions or feedback, visit: https://dataco.com/resources

---