MISSION_COMPILE

Created by **Bhavesh Vaswani**

Keeping Database Queries Under 20ms at 200M+ Rows

A Comprehensive Guide to High-Performance Database Optimization

Topics Covered:

Indexing | Read Replicas | Sharding | Denormalization
Caching | Column Optimization | Background Jobs

Table of Contents

Chapter 1: The Problem Statement

The Challenge

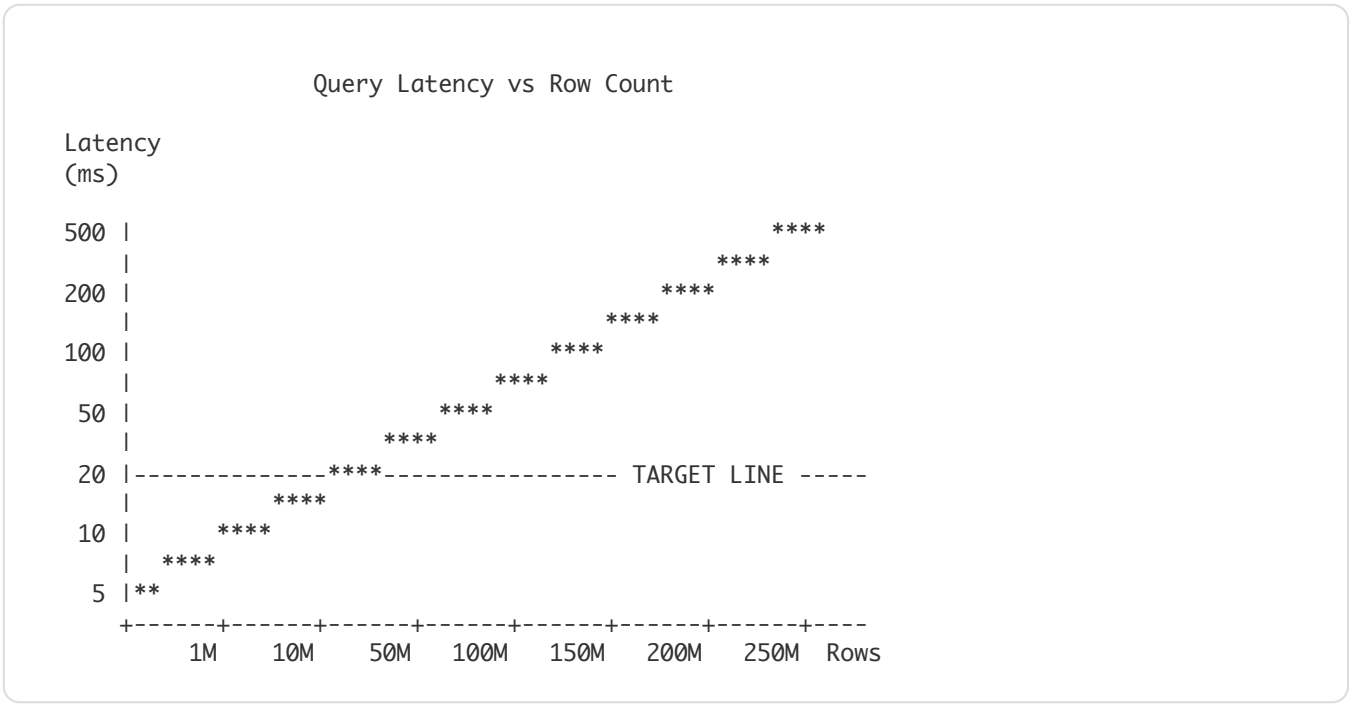
When your database table crosses **200 million rows**, queries that once took 5ms suddenly take 2-5 seconds. This happens because:

- Index scans become slower with more data
- Disk I/O increases significantly
- Memory pressure affects query planning
- Lock contention increases with concurrent users

The Goal

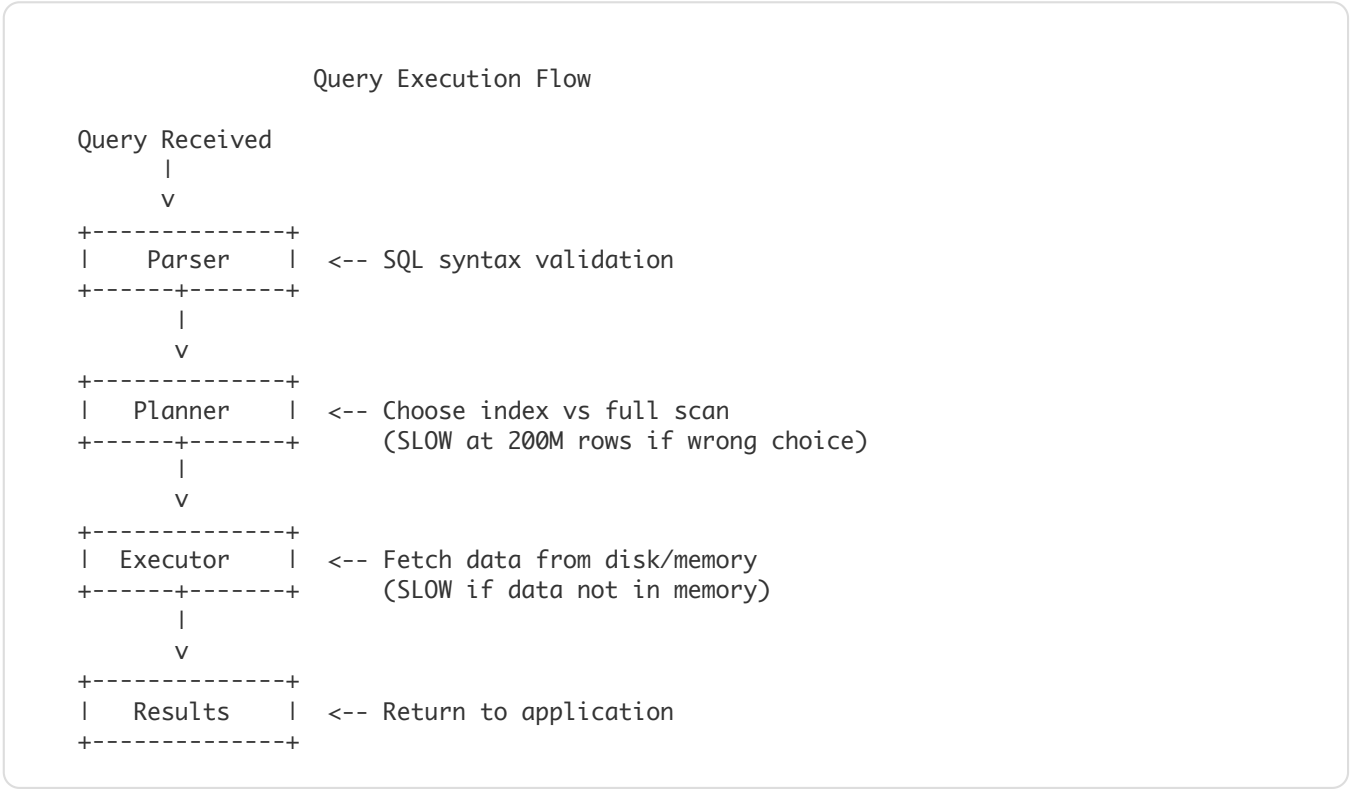
Keep **95th percentile query latency under 20ms** while handling:

- 200M+ rows in primary tables
- 10,000+ queries per second
- Mixed read/write workloads



Chapter 2: Understanding Query Performance

Why Queries Slow Down



The Three Bottlenecks

Bottleneck	Cause	Impact
CPU	Complex JOINs, sorting, aggregations	Query planning takes longer
Memory	Working set exceeds RAM	Disk swapping occurs
I/O	Full table scans, random disk reads	100x slower than memory

Chapter 3: Proper Indexing Strategy

Definition: An *index* is a data structure that improves the speed of data retrieval operations by providing quick lookup paths to rows based on column values.

Real-Life Analogy: Think of a **book's index at the back**. Instead of reading all 500 pages to find "Database", you check the index: "Database... page 127". The database index works the same way — it tells the system exactly where to find your data without scanning everything.

Why It Matters at 200M Rows

- **Without index:** Scan 200M rows = minutes
- **With proper index:** Lookup specific rows = milliseconds

Index Types and When to Use

Query Pattern	Recommended Index
`WHERE user_id = ?`	B-Tree Index (default)
`WHERE created_at > ? ORDER BY created_at`	B-Tree Index
`WHERE user_id = ? AND status = ?`	Composite Index (user_id, status)
`WHERE tags @> '{tech}'`	GIN Index (PostgreSQL)
`WHERE location nearby`	Spatial Index (R-Tree)

Composite Index: Column Order Matters

Rule: Most selective column FIRST

```
Table: orders (200M rows)
- user_id: 1M unique values (high selectivity)
- status: 5 unique values (low selectivity)
```

```
Query: WHERE user_id = 123 AND status = 'completed'

GOOD Index: (user_id, status)
+-----------------------------------------+
| user_id | Narrows 200M -> 200 rows first |
| status  | Then filters to ~40 rows       |
+-----------------------------------------+

BAD Index: (status, user_id)
+-----------------------------------------+
| status  | Narrows 200M -> 40M rows first | <-- Too broad!
| user_id | Then filters to ~40 rows       |
+-----------------------------------------+
```

Covering Index (Index-Only Scans)

Include all needed columns in the index to avoid table lookup:

```sql
-- Query
SELECT user_id, status, amount
FROM orders
WHERE user_id = 123;

-- Covering Index
CREATE INDEX idx_orders_covering
ON orders(user_id) INCLUDE (status, amount);
```

Pros

- Dramatic read speedup
- No application changes
- Works with existing schema

Cons

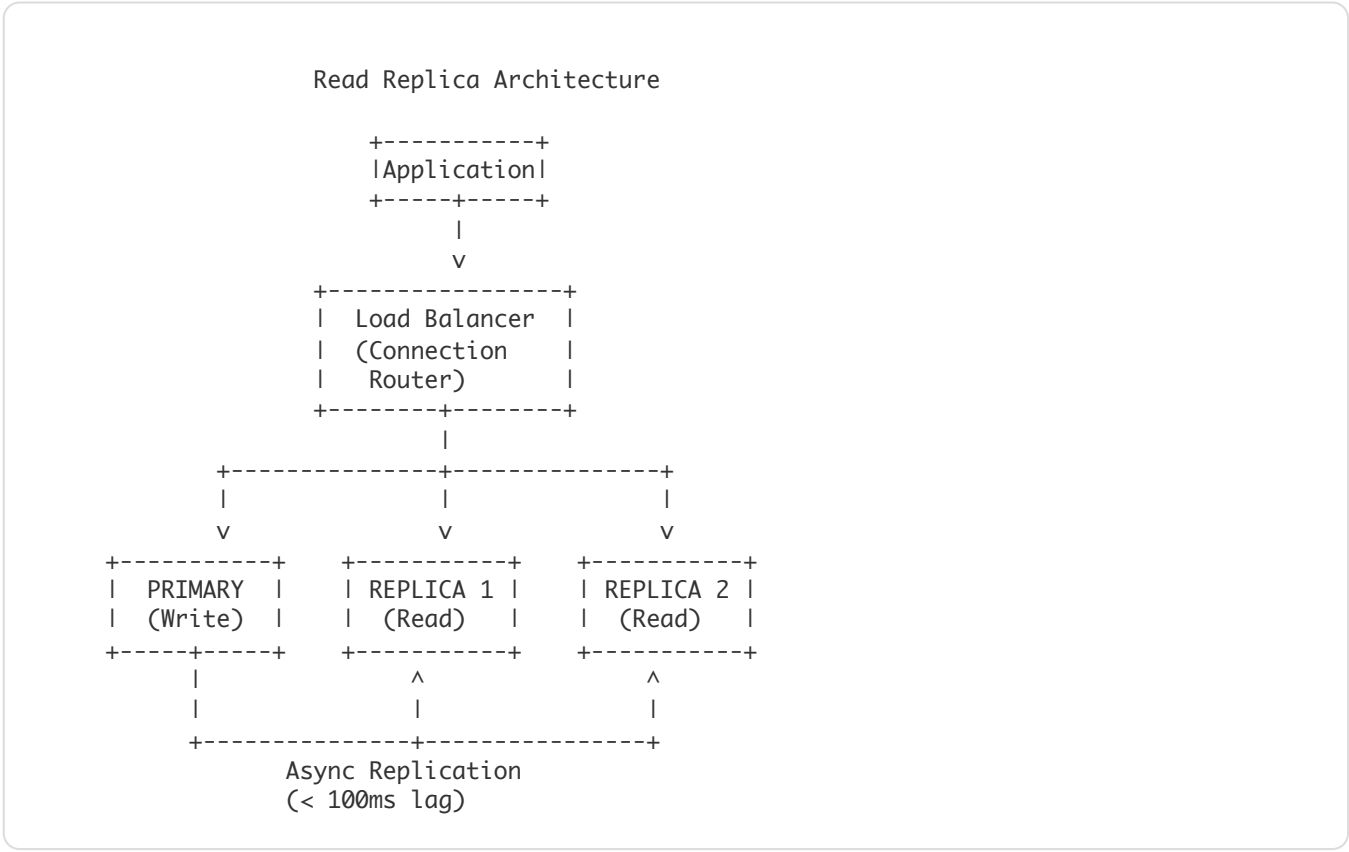
- Slower writes (index updates)
- Storage overhead
- Maintenance required

Chapter 4: Read Replicas

Definition: *Read replicas* are copies of your primary database that handle read queries, distributing the load and reducing latency.

Real-Life Analogy: Imagine a **popular restaurant with one chef**. When too many orders come in, the chef gets overwhelmed. Solution? Hire more chefs (replicas) who can prepare the same dishes. One head chef (primary) creates new recipes, others just copy and serve. More chefs = more customers served simultaneously.

Architecture

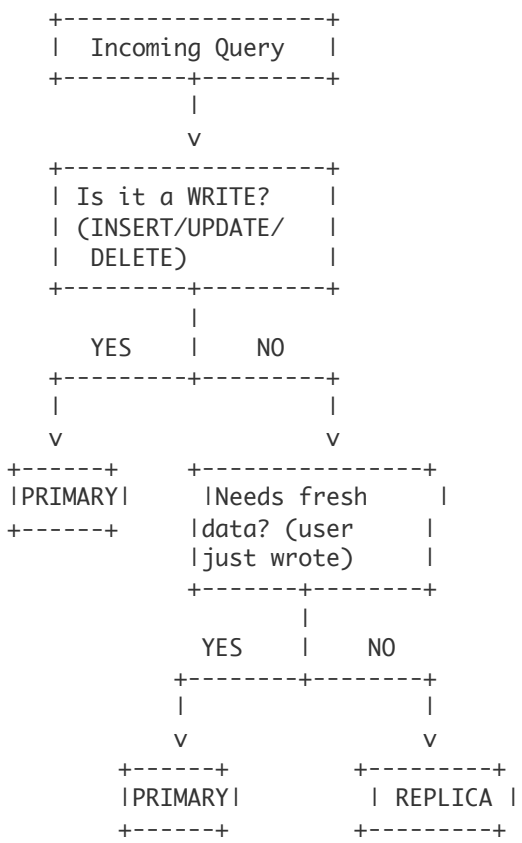


How Replication Works

1. Write hits Primary
2. Primary writes to Transaction Log (WAL)
3. WAL shipped to Replicas (async or sync)
4. Replicas apply WAL entries

5 Replicas now have updated data

Query Routing Strategy



Pros

- Linear read scalability
- High availability
- Geographic distribution
- No application rewrite

Cons

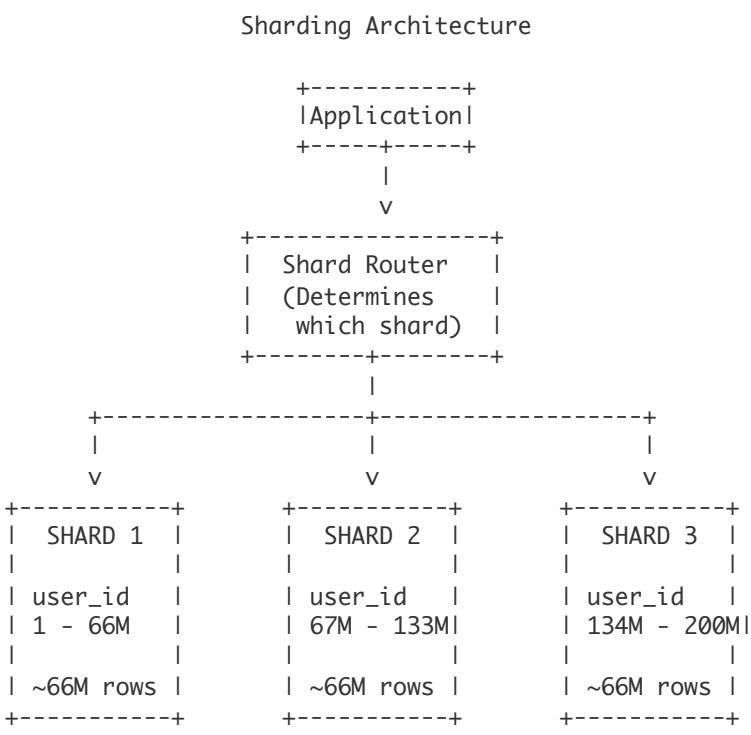
- Replication lag (eventual consistency)
- More infrastructure to manage
- Doesn't help write performance
- Data might be slightly stale

Chapter 5: Database Sharding

Definition: *Sharding* is horizontal partitioning where data is split across multiple database instances based on a shard key.

Real-Life Analogy: Think of a **library that got too big**. Instead of one massive building with 10 million books, you split it into 10 smaller branches by category: Science Library, History Library, Fiction Library, etc. When someone wants a Science book, they go directly to the Science branch — no need to search through 10 million books. That's sharding!

Sharding Architecture



Sharding Strategies

Strategy	How It Works	Pros/Cons
Range-Based	Shard by ID ranges `user_id 1-1M -> Shard 1`	Easy to understand, but risk of hot shards

Hash-Based	`shard_id = hash(user_id) % num_shards`	Even distribution, but harder to add shards
Directory-Based	Lookup table maps keys to shards	Most flexible, but lookup can be bottleneck

Choosing a Shard Key

> **Key Question:** "What column is in 80%+ of my WHERE clauses?"

GOOD Shard Keys	BAD Shard Keys
`user_id` - Most queries are user-specific	`created_at` - All new data hits one shard (hot spot)
`tenant_id` - Multi-tenant SaaS apps	`status` - Only 5 values = uneven distribution
`region` - Geographic queries	`email` - Can't do range queries

Pros

- Horizontal scalability
- Each shard is smaller/faster
- Geographic distribution
- Isolated failures

Cons

- Complex to implement
- Cross-shard queries are slow
- Resharding is painful
- JOINs across shards are hard

Chapter 6: Denormalization

Definition: *Denormalization* is the intentional duplication of data to avoid expensive JOIN operations and reduce query complexity.

Real-Life Analogy: Imagine you're a **delivery driver**. Normally, you have a customer address list and a separate map. For each delivery, you look up the address, then find it on the map (2 lookups = slow). With denormalization, you print the map directions directly on each delivery slip. More paper used, but much faster deliveries!

Normalized vs Denormalized

```
NORMALIZED (3NF) Design:

   orders                    users
   +----------------+       +----------------+
   | id             |       | id             |
   | user_id (FK)---+------>| name           |
   | amount         |       | email          |
   | status         |       | created_at     |
   +----------------+       +----------------+

   Query to get order with user name:
   SELECT o.*, u.name
   FROM orders o
   JOIN users u ON o.user_id = u.id     <-- JOIN = SLOW at 200M
   WHERE o.id = 12345;

-------------------------------------------------------------

DENORMALIZED Design:

   orders
   +----------------+
   | id             |
   | user_id        |
   | user_name      |  <-- Duplicated from users table
   | user_email     |  <-- Duplicated from users table
   | amount         |
   | status         |
   +----------------+

   Query to get order with user name:
   SELECT * FROM orders
   WHERE id = 12345;            <-- No JOIN = FAST!
```

Common Denormalization Patterns

Pattern	Description	Example

Embed Related Data	Copy frequently-accessed fields	`user_name` in orders table
Pre-computed Aggregates	Store COUNT, SUM, AVG	`user.total_orders`, `product.review_count`
Materialized Views	Database-maintained denormalized tables	Auto-refreshed on schedule/trigger
Summary Tables	Daily/hourly rollups	`daily_sales_by_product`

Pros

- Eliminates expensive JOINs
- Faster read queries
- Simpler query patterns
- Better for read-heavy workloads

Cons

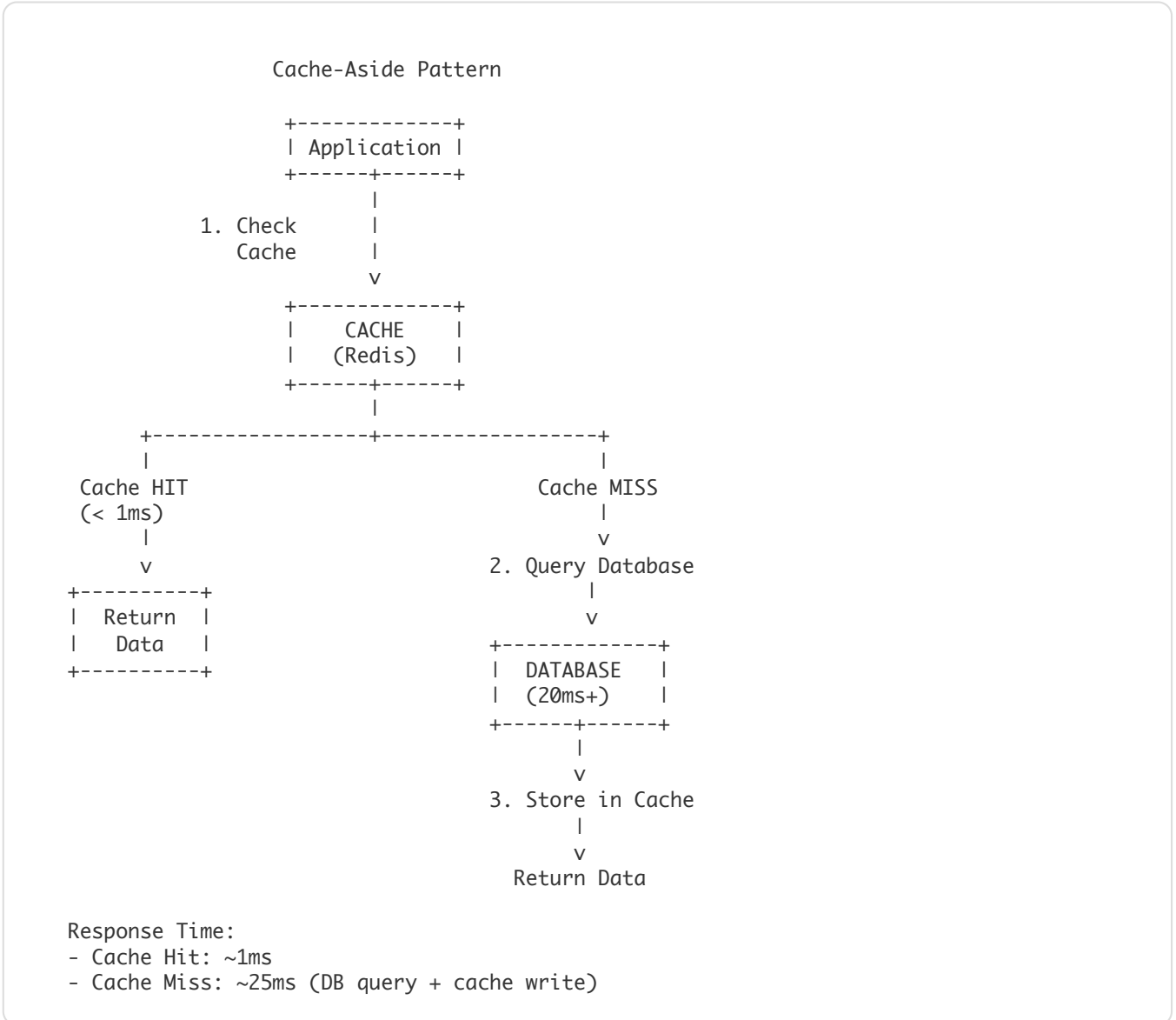
- Data duplication (storage cost)
- Consistency complexity
- Update anomalies possible
- Harder to maintain

Chapter 7: Caching Layer

Definition: *Caching* stores frequently accessed data in fast memory (RAM) to avoid repeated database queries.

Real-Life Analogy: Think of your **phone's recent calls list**. Instead of scrolling through your entire contacts (database) every time, your phone remembers who you called recently (cache). Calling mom again? She's right there at the top — no searching needed. That's exactly how caching works!

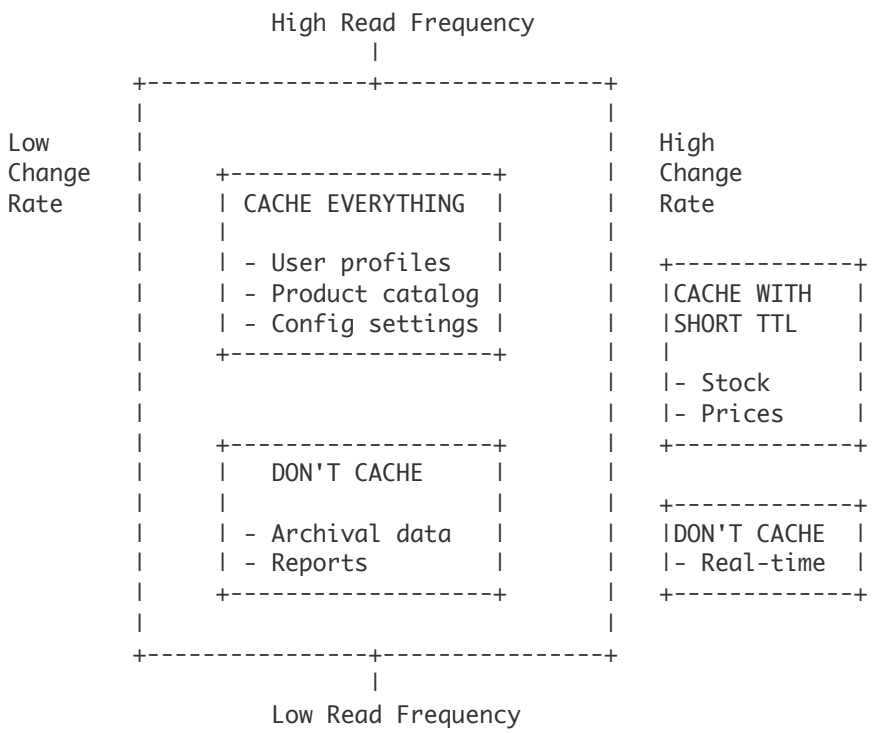
Cache-Aside Pattern



Cache Invalidation Strategies

Strategy	How It Works	Best For
TTL (Time-To-Live)	Cache expires after fixed time	Simple cases, tolerate some staleness
Write-Through	Update cache when database updates	Always-consistent data
Write-Behind	Update cache immediately, DB async	Fast writes (risk of data loss)
Event-Based	Publish event, cache subscribes	Microservices architecture

What to Cache



Pros

- Sub-millisecond reads
- Reduces database load
- Handles traffic spikes
- Works with any database

Cons

- Cache invalidation complexity
- Additional infrastructure
- Memory cost
- Stale data risk

Chapter 8: Column Optimization

Definition: *Column optimization* reduces the amount of data read from disk by selecting only necessary columns and using efficient data types.

Real-Life Analogy: Imagine ordering at **McDonald's**. You want just a burger, but the waiter brings you the entire menu, the kitchen equipment, and the store inventory — just in case you need them. That's what `SELECT *` does! Instead, just ask for what you need: "One burger, please" = `SELECT burger FROM menu`.

The Problem with SELECT *

```
Table: orders (200M rows)
+------+--------+-------+------+---------+--------+
| id   | user_id| amount|status| metadata|created |
| 4B   |   8B   |   8B  |  2B  |   2KB   |   8B   |
+------+--------+-------+------+---------+--------+

Row size: ~2KB

-----------------------------------------------------------

BAD: SELECT * FROM orders WHERE user_id = 123
- Reads: 200 rows x 2KB = 400KB
- Includes metadata column you don't need

GOOD: SELECT id, amount, status FROM orders WHERE user_id = 123
- Reads: 200 rows x 22B = 4.4KB
- 99% less data transferred!

-----------------------------------------------------------

Impact at Scale:
- 10,000 queries/second x 400KB = 4GB/s network
- 10,000 queries/second x 4KB = 40MB/s network
```

Data Type Optimization

Type	Size	Use Case
TINYINT	1 byte	Status codes (0-255)
SMALLINT	2 bytes	Small counters
INT	4 bytes	Standard IDs

`BIGINT`	8 bytes	Large IDs, timestamps
`TIMESTAMP`	4 bytes	Unix timestamp (use instead of DATETIME's 8 bytes)

Pros

- Reduced I/O and memory
- Faster queries
- Lower storage costs
- Better cache utilization

Cons

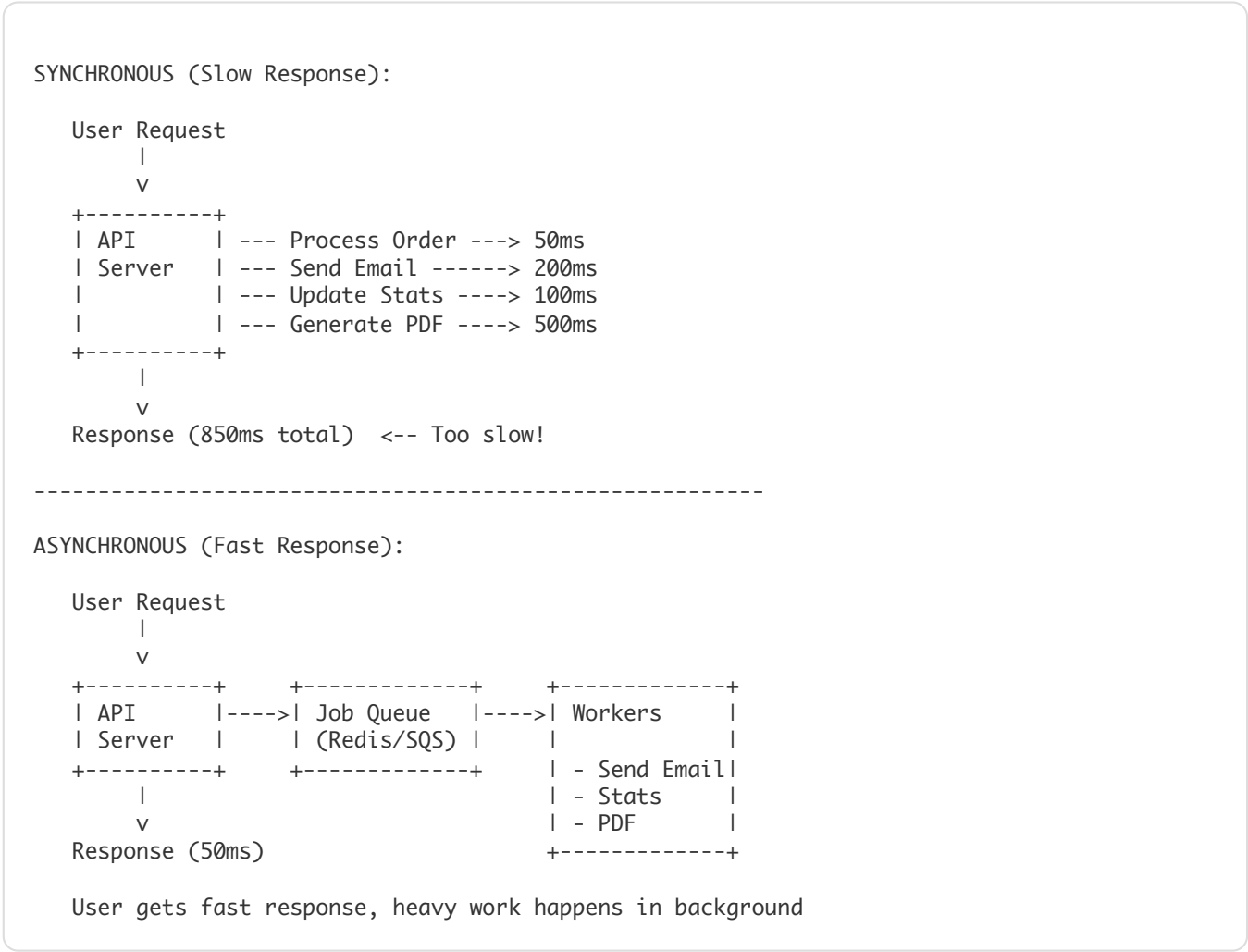
- Requires query refactoring
- Must know access patterns
- May need application changes
- Initial analysis effort

Chapter 9: Background Job Processing

Definition: *Background job processing* moves expensive, non-urgent operations out of the request path to improve response times.

Real-Life Analogy: Think of **ordering coffee at Starbucks**. Synchronous: You wait at the counter while they make your coffee, blocking the line. Asynchronous: You order, get a receipt number, and sit down. They call your number when ready. You're not blocked, the line moves, everyone's happy. That's background processing!

Synchronous vs Asynchronous Processing



Database-Related Background Jobs

Job Type	Purpose

Aggregate Calculations	Background job updates `user.order_count` every hour instead of `SELECT COUNT(*)`
Data Archival	Move old records (> 1 year) to archive table, keeps main table small
Cache Warming	Pre-populate cache with frequently accessed data during low-traffic hours
Index Maintenance	REINDEX operations during maintenance windows
Denormalization Sync	Batch update denormalized fields, reconcile inconsistencies
Report Generation	Pre-compute daily/weekly reports, store results for instant access

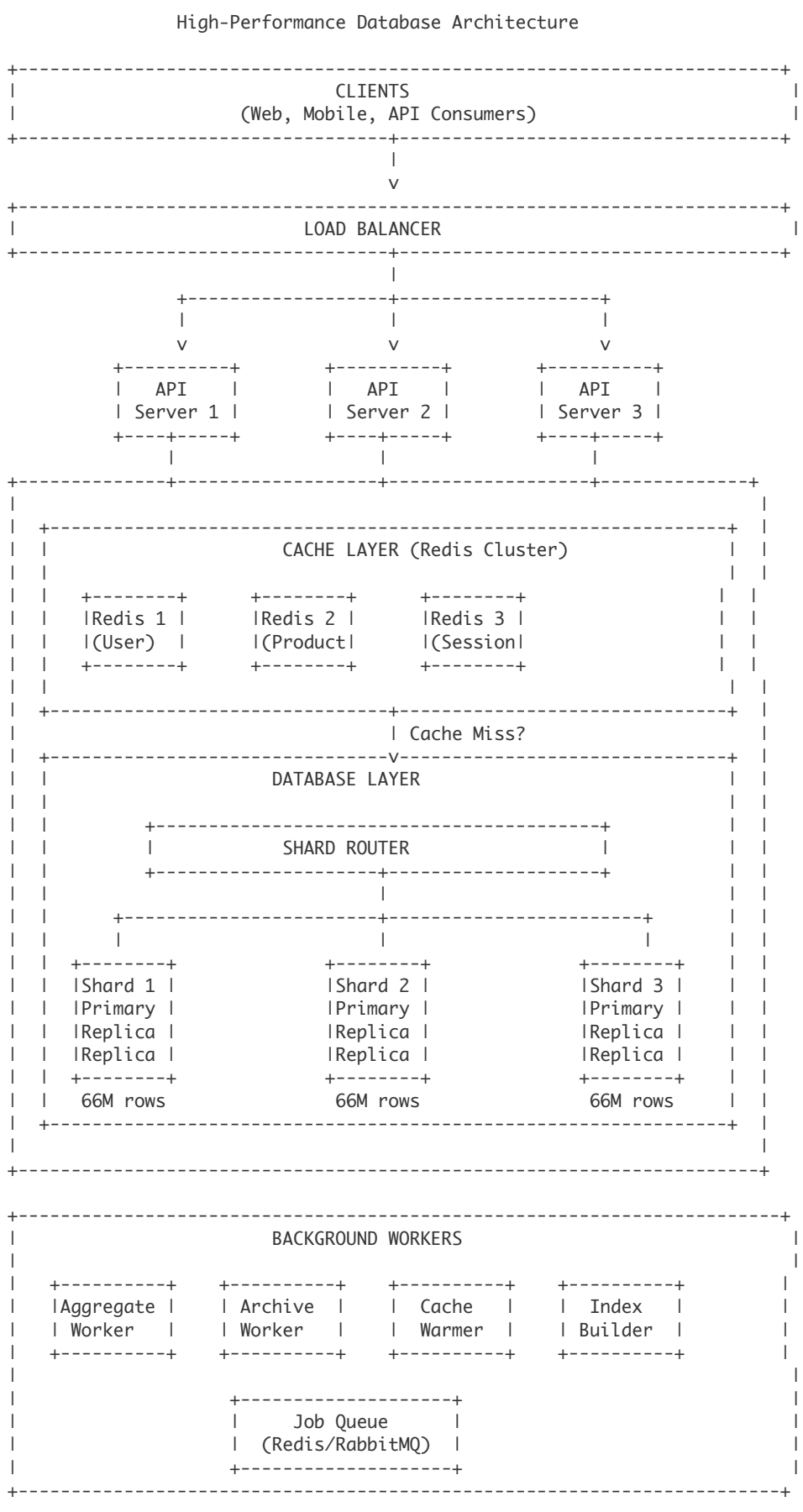
Pros

- Fast API responses
- Better resource utilization
- Scalable processing
- Batch efficiency

Cons

- Eventual consistency
- Additional infrastructure
- Job failure handling needed
- Monitoring complexity

Chapter 10: Complete Architecture



Chapter 11: Implementation Checklist

Phase 1: Quick Wins (Week 1)

- □ Audit slow queries (EXPLAIN ANALYZE)
- □ Add missing indexes for frequent WHERE clauses
- □ Replace SELECT * with specific columns
- □ Implement query result caching (Redis)
- □ Set up query monitoring and alerting

Expected Impact: 50-70% query time reduction

Phase 2: Infrastructure (Week 2-3)

- □ Set up read replicas (2-3 replicas)
- □ Implement query routing (write to primary, read to replica)
- □ Configure connection pooling
- □ Set up cache cluster (Redis Cluster)
- □ Implement cache invalidation strategy

Expected Impact: 3-5x read throughput increase

Phase 3: Data Architecture (Week 4-6)

- □ Identify denormalization opportunities
- □ Create summary/aggregate tables
- □ Implement data archival strategy
- □ Set up background job processing
- □ Optimize data types and table structure

Expected Impact: 80% reduction in complex query times

Phase 4: Sharding (If Needed) (Week 7-12)

- □ Analyze query patterns for shard key selection
- □ Design sharding strategy (hash vs range)
- □ Implement shard routing layer
- □ Plan and execute data migration
- □ Test cross-shard query handling

  Expected Impact: Linear scalability (add shards = add capacity)

Chapter 12: Quick Reference

Solution Summary

Solution	Effort	Impact	When to Use
Indexing	Low	High	Always - first step
Caching	Low-Medium	High	Read-heavy workloads
Read Replicas	Medium	High	High read traffic
Column Optimization	Low	Medium	Wide tables
Denormalization	Medium	High	JOIN-heavy queries
Background Jobs	Medium	Medium	Aggregations, reports
Sharding	High	Very High	100M+ rows, need scale

Decision Flowchart

Remember With Analogies

Indexing	= Book's index at the back
Read Replicas	= Multiple chefs cooking same dishes
Sharding	= Library split into branches by category
Denormalization	= Printing directions on delivery slip
Caching	= Phone's recent calls list
Column Optimization	= Ordering just a burger, not entire menu
Background Jobs	= Starbucks order number system

One-Page Summary

1. **INDEX PROPERLY** - Composite indexes with selective columns first, covering indexes
2. **CACHE AGGRESSIVELY** - Redis for frequently accessed data, TTL + event invalidation
3. **SCALE READS WITH REPLICAS** - Route writes to primary, reads to replicas
4. **DENORMALIZE FOR SPEED** - Duplicate data to avoid JOINs, pre-compute aggregates
5. **OPTIMIZE COLUMNS** - Never SELECT *, use appropriate data types
6. **PROCESS IN BACKGROUND** - Move heavy operations off request path
7. **SHARD WHEN NECESSARY** - Split data across databases based on query patterns

Remember: Start with indexing and caching. Only shard when other optimizations aren't enough.

Document Version: 1.0 | **Last Updated:** 2025

Target Audience: Backend Engineers, Database Administrators, System Architects