The Ultimate Database Selection Guide for System Design Interviews

> Everything you need to know about choosing the right database — from SQL to NoSQL, with real-world examples from Twitter, Instagram, and Uber

Why This Matters

If you're preparing for system design interviews at top tech companies, you'll inevitably face the question: **"What database would you use for this system?"**

The wrong answer? "I'll use MongoDB for everything."

The right answer? Understanding trade-offs and choosing the optimal database for each component of your system.

This guide will teach you exactly how to approach database selection like engineers at Google, Facebook, and Amazon do.

The Foundation: Understanding Database Types

Before diving into specific databases, let's understand the landscape.

The Two Main Categories

SQL (Relational Databases)

- Think: MySQL, PostgreSQL
- Structured tables with predefined schemas
- ACID transactions (Atomicity, Consistency, Isolation, Durability)
- Best for: Banking, e-commerce, user accounts

NoSQL (Non-Relational Databases)

- Think: MongoDB, Cassandra, Redis
- Flexible schemas, horizontal scaling
- BASE properties (Basically Available, Soft state, Eventually consistent)
- Best for: Social media, IoT, high-scale applications

The CAP Theorem: Choose Your Superpower

In distributed systems, you can only pick TWO of three:

🔷 **Consistency** — All nodes see the same data
🔷 **Availability** — Every request gets a response
🔷 **Partition Tolerance** — System works despite network failures

Since network partitions WILL happen, you must choose:

- **CP (Consistency + Partition Tolerance)**: PostgreSQL, MongoDB
- **AP (Availability + Partition Tolerance)**: Cassandra, DynamoDB

Database #1: PostgreSQL / MySQL (The Reliable Workhorse)

Type: Relational Database (SQL)

When to Use:

✅ You need ACID transactions (payments, banking)
✅ Data has clear relationships (users, orders, products)
✅ Complex queries with JOINs are required
✅ Schema is well-defined and stable

Real-World Examples:

- **Instagram**: User profiles and relationships (12 PostgreSQL instances!)
- **Uber**: Trip data and user accounts
- **Airbnb**: Bookings and payment transactions

How Instagram Scales PostgreSQL:

```
- 12 Quadruple Extra-Large instances
- Master-slave replication for read scaling
- All working set kept in memory (using vmtouch)
- EBS in RAID configuration for I/O
- 12 replicas in different availability zone
```

When NOT to Use:

❌ Write throughput exceeds 10K QPS
❌ Need to store petabytes of data
❌ Schema changes frequently

Interview Tip: Always start with PostgreSQL/MySQL as your default choice. Only move to NoSQL when you can justify why SQL won't work.

Database #2: Redis (The Speed Demon)

Type: In-Memory Key-Value Store

When to Use:

✅ Need caching layer (< 1ms latency)
✅ Session storage with auto-expiry
✅ Real-time leaderboards
✅ Rate limiting
✅ Timeline storage (Twitter, Instagram)

Redis Data Structures (Your Secret Weapons):

Sorted Sets for Twitter/Instagram timelines:

```
ZADD timeline:user:123 1704672000 tweet-abc
ZADD timeline:user:123 1704672100 tweet-def
ZREVRANGE timeline:user:123 0 49  # Get latest 50 tweets
```

Counters for rate limiting:

```
INCR rate_limit:user:123:tweets
EXPIRE rate_limit:user:123:tweets 3600  # 1 hour
```

Strings for caching:

```
SET user:123 '{"name":"Alice","email":"alice@example.com"}'
EXPIRE user:123 3600  # Cache for 1 hour
```

Real-World Examples:

- **Twitter**: Timeline storage, rate limiting (saves 90% of DB queries!)
- **Instagram**: Feed cache, session management
- **GitHub**: Job queues, caching

The Instagram Story:

Instagram uses Redis for their main feed and activity feed. By caching timelines in Redis sorted sets, they reduced PostgreSQL load by 90% and achieved sub-100ms feed load times.

Interview Tip: Always mention Redis as your caching layer. Say "I'd put Redis in front of PostgreSQL to handle 80-90% of reads."

Database #3: Cassandra (The Scale Master)

Type: Distributed Wide-Column Store

When to Use:

✅ Write-heavy workloads (10K+ writes/second)
✅ Time-series data (logs, events, metrics)
✅ Need geographic distribution
✅ Petabyte-scale datasets

Why Cassandra for Twitter?

Twitter posts **500 million tweets per day**. That's:

- **6,000 writes per second** (average)
- **60,000 writes per second** (peak during events)

PostgreSQL would require complex sharding and still struggle. Cassandra? Just add more nodes. Linear scalability.

Cassandra's Secret Sauce:

```
1. Masterless architecture (no single point of failure)
2. Tunable consistency (choose CP or AP per query)
3. Write-optimized (append-only commit log)
4. Linear scalability (2x nodes = 2x capacity)
```

Real-World Examples:

- **Netflix**: 2.5 trillion operations per day!
- **Apple**: iMessage (billions of messages)
- **Instagram**: Photo metadata
- **Twitter**: Tweet storage

Interview Tip: When the interviewer says "500M writes per day," immediately think Cassandra.

Database #4: Elasticsearch (The Search Wizard)

Type: Distributed Search Engine

When to Use:

✅ Full-text search required
✅ Autocomplete/suggestions
✅ Log analytics (ELK stack)
✅ Fuzzy matching (handle typos)

Why Not Use SQL for Search?

SQL's `LIKE '%keyword%'` is slow and doesn't handle:

- Relevance scoring

- Fuzzy matching ("alcie" should match "alice")
- Language-specific analysis (stemming, stop words)
- Typo tolerance

Elasticsearch solves all of this.

Real-World Examples:

- **GitHub**: Code search across millions of repositories
- **LinkedIn**: Job and people search
- **Uber**: Driver and trip search
- **Netflix**: Content discovery

Interview Tip: Mention Elasticsearch whenever "search" is a requirement. Always pair it with another database as the source of truth.

Database #5: Amazon S3 (The Storage Giant)

Type: Object Storage

When to Use:

✅ Storing media files (images, videos)
✅ Backups and archives
✅ Static website assets
✅ Data lakes for analytics

The Numbers That Matter:

- **99.999999999% durability** (11 nines!)
- **Infinite scalability**
- **$0.023 per GB per month** (Standard)
- **$0.00099 per GB per month** (Deep Archive)

How Netflix Uses S3:

Netflix stores all their video content in S3:

- Multiple versions (4K, 1080p, 720p, etc.)
- CloudFront CDN in front for delivery
- Lifecycle policies move old content to Glacier
- Saves millions in storage costs

Real-World Examples:

- **Instagram**: Stores all photos (petabytes!)
- **Netflix**: Video storage and transcoding
- **Spotify**: Music files
- **Airbnb**: Property photos

Interview Tip: Always mention S3 for media storage, followed by "with CloudFront CDN for global delivery."

The Decision Framework: What Would Zuckerberg Choose?

Here's the framework I use in every interview:

Step 1: Start with SQL

Default to PostgreSQL unless you have a SPECIFIC reason not to.

Why? Because:

- It handles 90% of use cases
- ACID transactions out of the box
- Mature tooling and ecosystem
- Well-understood scaling patterns

Step 2: Add NoSQL for Specific Needs

Ask yourself:

- **Need caching?** → Add Redis
- **Need search?** → Add Elasticsearch
- **Write-heavy?** → Use Cassandra
- **Storing files?** → Use S3

Step 3: Justify Every Choice

For each database, explain:

1. **Why this database?** (specific strengths)
2. **What's the trade-off?** (limitations)
3. **How to scale it?** (replication, sharding, clustering)
4. **Alternatives?** (other options you considered)

Real-World Architecture: Designing Twitter

Let's apply this framework to Twitter.

Requirements:

- 400M daily active users
- 500M tweets per day
- Billions of timeline requests
- Real-time search
- Trending topics

Database Architecture:

1. User Profiles → MySQL

- Why: Structured data, ACID for account operations
- Scale: Shard by user_id

2. Tweets → Cassandra

- Why: 500M writes/day, time-series data
- Scale: 100+ node cluster, linear scaling

3. Timeline Cache → Redis

- Why: Fast reads (< 1ms), sorted sets perfect for timelines
- Scale: Redis Cluster with 50+ nodes

4. Social Graph → Cassandra

- Why: Fast follower lookups for feed generation
- Scale: Partition by user_id

5. Search → Elasticsearch

- Why: Full-text search on tweets and users
- Scale: 20-node cluster

6. Media → S3

- Why: Durable, scalable, cheap ($0.023/GB)
- Scale: Infinite, with CloudFront CDN

7. Analytics → Redshift

- Why: Historical analysis, business intelligence
- Scale: Columnar storage, MPP

The Magic: 7 Different Databases Working Together!

This is **polyglot persistence** — using the right database for each job.

The Celebrity Problem: Twitter's Billion-Dollar Challenge

Here's a problem that cost Twitter millions to solve:

The Problem:
When someone with 10 million followers tweets, how do you update 10 million timelines?

Approach 1: Fan-out on Write (Push)

```
User tweets → Write to 10M follower timelines
```

- ✅ Fast reads (pre-computed)
- ❌ Slow writes for celebrities (10M database writes!)

Approach 2: Fan-out on Read (Pull)

```
User requests timeline → Query tweets from all followed users
```

- ✅ Fast writes (one database write)
- ❌ Slow reads (query hundreds of users)

Twitter's Solution: Hybrid Approach

For normal users (< 1M followers):

- Fan-out on write (pre-compute timelines)
- Store in Redis for fast access

For celebrities (> 1M followers):

- Fan-out on read (fetch tweets on-demand)
- Cache heavily

Result: Fast for everyone, scales to billions of users!

Interview Gold: This is the answer interviewers want when they ask about Twitter's architecture.

Common Interview Mistakes (Don't Be That Candidate)

❌ Mistake #1: "MongoDB for Everything"

Why Bad: Shows you don't understand trade-offs

Better: "PostgreSQL for users (ACID), Cassandra for tweets (write-heavy), Redis for cache"

❌ Mistake #2: "NoSQL is Always Faster"

Why Bad: Not true for all workloads

Better: "NoSQL is faster for simple key-value lookups, but SQL with proper indexing is faster for complex JOINs"

❌ Mistake #3: Forgetting the Cache

Why Bad: Cache is critical for scalability

Better: "I'd put Redis in front of PostgreSQL to handle 90% of reads from cache, reducing database load"

❌ Mistake #4: No Justification

Why Bad: SQL is the default, NoSQL needs justification

Better: "I'd start with PostgreSQL, but when writes exceed 10K QPS, I'd migrate to Cassandra"

❌ Mistake #5: Ignoring Consistency

Why Bad: Shows incomplete understanding

Better: "User profiles need strong consistency (SQL), but the feed can use eventual consistency (Cassandra)"

The Interview Answer Template

Memorize this framework:

```
"For [component], I would use [database] because:

1. Data characteristics: [structured/unstructured/time-series]
2. Access pattern: [read-heavy/write-heavy/balanced]
3. Scale requirements: [QPS, storage size]
4. Consistency needs: [strong/eventual]
5. Special requirements: [transactions/search/real-time]

To scale this, I would [specific strategy].

The trade-off is [limitation], but that's acceptable
because [business reason].

Alternatives would be [other options], but I prefer
[chosen database] because [specific advantage]."
```

Example Answer:

"For user authentication, I would use **PostgreSQL** because:

1. User data is structured with clear relationships
2. Access pattern is read-heavy (100:1 ratio)
3. Scale: 10M users, 1K QPS
4. Need strong consistency (ACID) for account operations
5. Require transactions for signup/payment operations

To scale this, I would use master-slave replication with 5 read replicas and shard by user_id when we hit 50M users.

The trade-off is that sharding adds complexity, but that's acceptable because we can delay sharding until we validate product-market fit.

Alternatives would be MongoDB for flexibility, but I prefer PostgreSQL because financial data needs ACID guarantees."

The Quick Reference Cheat Sheet

If You Need...	Use This Database
💰 Transactions (money, inventory)	PostgreSQL, MySQL
⚡ Caching	Redis, Memcached
📝 500M+ writes/day	Cassandra, DynamoDB
🔍 Full-text search	Elasticsearch
👥 Social graph	Neo4j, Cassandra
📊 Analytics	Redshift, BigQuery
🗄️ File storage	Amazon S3
📱 Mobile app backend	MongoDB, Firebase
🎮 Gaming leaderboards	Redis Sorted Sets
📈 Time-series data	Cassandra, InfluxDB

Real-World Case Study: How Uber Chooses Databases

Uber uses **EIGHT different databases**. Here's why:

PostgreSQL: User accounts, payment transactions

- Why: Cannot lose money, need ACID

Cassandra: Trip history, location pings

- Why: Write-heavy, billions of location updates

Redis: Real-time driver locations

- Why: Geo-spatial queries (GEOADD, GEORADIUS)

Elasticsearch: Driver/rider search

- Why: Need geo-search with filters

Kafka: Event streaming

- Why: Connect all systems, event sourcing

S3: Trip receipts, documents

- Why: Durable storage for compliance

Redshift: Business analytics

- Why: Historical trip analysis

Graph Database: Fraud detection

- Why: Pattern recognition in payment networks

Key Lesson: No single database can handle everything at scale!

The Most Important Graph You'll See

```
Data Size vs Database Choice:

< 1 GB     → Any database (don't over-engineer)
< 100 GB   → Single SQL instance
< 1 TB     → SQL with read replicas
< 10 TB    → Sharded SQL or NoSQL
< 100 TB   → Cassandra, DynamoDB
> 100 TB   → Distributed NoSQL + data warehouse
```

Advanced Topic: The Fan-out Problem

This is what separates senior from junior engineers.

The Problem:

When a celebrity with 10M followers tweets, how do you update 10M timelines in real-time?

Bad Solution:

```
# This takes 10 minutes for 10M followers!
for follower in get_followers(celebrity_id):
    add_tweet_to_timeline(follower, tweet)
```

Twitter's Solution (Hybrid Fan-out):

Regular users (< 1M followers):

- Pre-compute timelines (fan-out on write)
- Store in Redis
- Fast reads

Celebrities (> 1M followers):

- Don't pre-compute (fan-out on read)
- Fetch tweets on-demand
- Merge with pre-computed timeline

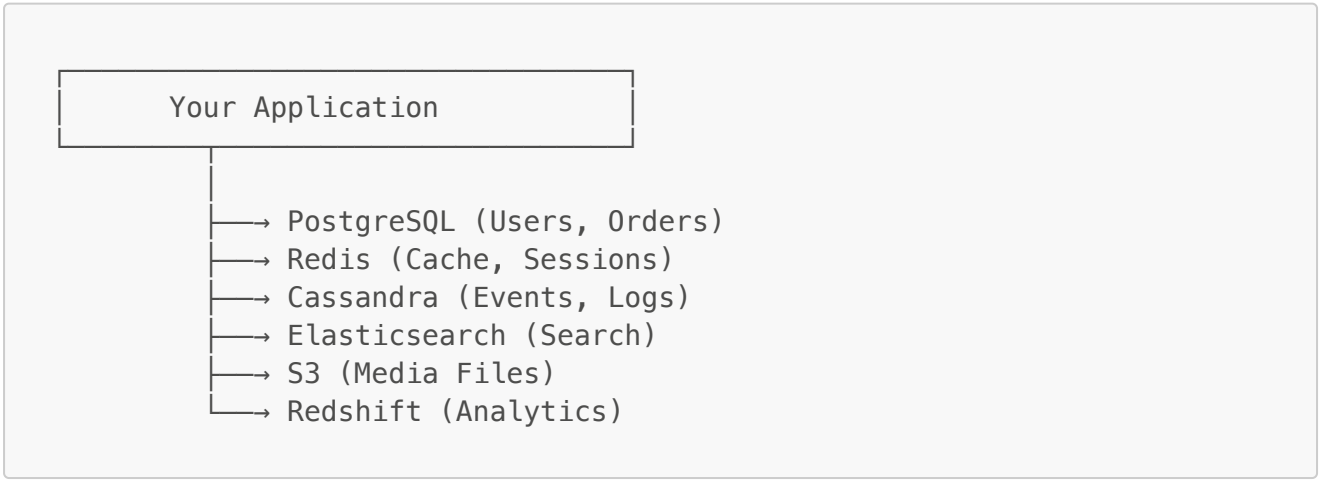
Code:

```
def get_timeline(user_id):
    # Pre-computed tweets from normal users
    normal_tweets = redis.zrevrange(f"timeline:{user_id}", 0, 50)

    # Celebrity tweets fetched on-demand
    celebrities = get_celebrity_following(user_id)
    celebrity_tweets = db.query_recent_tweets(celebrities)

    # Merge and return
    return merge_and_rank(normal_tweets, celebrity_tweets)
```

Interview Impact: Mentioning this solution shows you understand real-world scaling challenges!

The Polyglot Persistence Pattern

Modern applications use **3-5 databases**. Here's the typical stack:



This is **not over-engineering** — this is how Google, Facebook, and Amazon build systems!

Cost Comparison (The Reality Check)

For **1 TB storage + moderate traffic**:

- Redis: $1,500/month (but worth it for speed)
- PostgreSQL: $800/month (RDS managed)
- MongoDB: $1,200/month (Atlas)
- Cassandra: $500/month (self-hosted)

- DynamoDB: $1,500/month (varies widely)
- **S3: $23/month** (incredibly cheap!)
- Elasticsearch: $2,000/month
- Redshift: $3,000/month

Interview Insight: Mention cost trade-offs. "S3 is 50x cheaper than EBS for media storage."

The Final Word: What Top Companies Actually Use

Twitter

- MySQL (users)
- Cassandra (tweets)
- Redis (timelines)
- Manhattan (their custom DB)

Instagram

- PostgreSQL (users, photos metadata)
- Cassandra (activity feed)
- Redis (feed cache)
- S3 (photos)

Uber

- PostgreSQL (payments, users)
- Cassandra (trips, locations)
- Redis (real-time matching)
- Elasticsearch (search)

Netflix

- Cassandra (viewing history)
- Redis (session data)
- S3 (videos)
- Redshift (analytics)

Pattern: Everyone uses multiple databases!

Your Action Plan for Interviews

Before the Interview:

1. ✅ Memorize the decision framework
2. ✅ Know 3 real-world examples for each database
3. ✅ Practice the answer template
4. ✅ Understand trade-offs (pros AND cons)

During the Interview:

1. **Start with SQL** (PostgreSQL/MySQL)
2. **Add Redis** for caching
3. **Justify NoSQL** with specific requirements
4. **Mention scaling strategies** for each database
5. **Discuss consistency requirements**

The Magic Words:

- "I'd start with PostgreSQL because..."
- "The trade-off is..."
- "To scale this, I would..."
- "Alternative options include..."
- "This is similar to how [Company] handles it..."

Conclusion: Everything is a Trade-off

There's no perfect database. The key is understanding:

🔷 **When** to use each database
🔷 **Why** you're choosing it
🔷 **How** to scale it
🔷 **What** trade-offs you're making

Remember: **The interviewer doesn't expect you to know every database.** They want to see you:

- Think systematically
- Justify your choices
- Understand trade-offs
- Know real-world patterns

Master these principles, and you'll ace every database question thrown at you!

Resources to Go Deeper

📚 **Books**:

- "Designing Data-Intensive Applications" by Martin Kleppmann
- "Database Internals" by Alex Petrov

🌐 **Online**:

- System Design Primer (GitHub: donnemartin)
- High Scalability Blog
- Company engineering blogs (Netflix, Uber, Twitter)

💡 **Practice**:

- Design Twitter (use this guide!)
- Design Instagram
- Design Uber
- Design Amazon

Found this helpful? Drop a comment with your favorite database or share your interview experience!

Preparing for system design interviews? Follow me for more deep dives into system architecture, scalability patterns, and interview strategies.

Written for engineers preparing for system design interviews at top tech companies. Based on real architectures from Twitter, Instagram, Netflix, and Uber.

Tags: #SystemDesign #DatabaseDesign #SoftwareEngineering #TechInterviews #PostgreSQL #Redis #Cassandra #MongoDB #Elasticsearch

---

About the Author: System design enthusiast who has studied architectures from 50+ tech companies. Passionate about making complex concepts simple and helping engineers ace their interviews.