Database Selection Guide for System Design Interviews

Table of Contents

Introduction

> **"Everything is a trade-off in system design. There is no perfect database for all use cases."**

Choosing the right database is one of the most critical decisions in system design. This guide provides a comprehensive overview of different database types and when to use them, based on industry best practices and real-world architectures.

Key Concepts

ACID (SQL Databases):

- **Atomicity**: All or nothing transactions
- **Consistency**: Data remains in valid state
- **Isolation**: Concurrent transactions don't interfere
- **Durability**: Committed data is permanent

BASE (NoSQL Databases):

- **Basically Available**: System guarantees availability
- **Soft state**: State may change without input
- **Eventual consistency**: System becomes consistent over time

CAP Theorem: You can only have 2 of 3:

- **Consistency**: All nodes see same data
- **Availability**: Every request gets a response
- **Partition Tolerance**: System works despite network partitions

Since network partitions will happen, you must choose between CP (Consistency) or AP (Availability)

SQL vs NoSQL Decision Framework

Use SQL (Relational) When:

✅ **Data is structured and schema is well-defined**

- Banking transactions, user profiles, e-commerce orders
- Relationships between entities are important
- Example: User has many Orders, Order has many Items

✅ **Need ACID transactions**

- Financial applications (payments, banking)
- Inventory management
- Any operation that must be atomic

✅ **Complex queries with JOINs**

- Reporting and analytics
- Multi-table aggregations
- Example: "Show me all users who bought product X in last 30 days"

✅ **Data integrity is critical**

- Foreign keys enforce relationships
- Constraints prevent invalid data
- Example: Can't delete user if they have active orders

✅ **Mature ecosystem required**

- Lots of tools, ORMs, DBAs available
- Well-understood scaling patterns
- Extensive documentation

Use NoSQL When:

✅ **Massive scale (TB or PB of data)**

- Social media posts (billions of tweets)
- IoT sensor data (millions of devices)
- Logging systems (petabytes of logs)

✅ **High throughput workload**

- Write-heavy: 10K+ writes/second
- Read-heavy: 100K+ reads/second
- Example: Twitter, Instagram feeds

✅ **Flexible/dynamic schema**

- Rapidly changing requirements
- Each document can have different fields
- Example: Product catalog with varying attributes

✅ **Simple access patterns (key-value lookups)**

- No complex JOINs needed
- Access by primary key

- Example: User sessions, cache

✅ **Horizontal scalability required**

- Need to add servers easily
- Linear scalability
- Geographic distribution

Relational Databases (SQL)

1. MySQL / PostgreSQL

Type: Relational (RDBMS)

Best For:

- User data, profiles, accounts
- Financial transactions
- Content management systems
- E-commerce applications
- Any structured data with relationships

Characteristics:

- **ACID compliant**: Full transaction support
- **Schema-based**: Predefined tables and columns
- **Vertical scaling**: Scale up (bigger machines)
- **Read replicas**: Scale reads horizontally
- **Query optimizer**: Automatic query optimization

Scaling Strategies:

```
1. Read Replicas (Master-Slave)
   - 1 Master (writes)
   - Multiple Slaves (reads)
   - Async replication

2. Sharding (Horizontal Partitioning)
   - Split data across multiple databases
   - Shard by user_id, region, etc.
   - Requires app-level routing

3. Federation (Vertical Partitioning)
   - Split by function (users DB, products DB, orders DB)
   - Each DB smaller, fits in memory
   - Reduces replication lag

4. Denormalization
   - Duplicate data to avoid JOINs
   - Trade write performance for read performance
   - Materialized views
```

When to Choose MySQL:

- Proven track record (used by Facebook, Twitter, YouTube)
- Excellent replication support
- Large community
- Good for read-heavy workloads

When to Choose PostgreSQL:

- Need advanced features (JSON, full-text search, arrays)
- Complex queries with CTEs, window functions
- Better write performance than MySQL
- Stronger consistency guarantees

Real Examples:

- **Instagram**: PostgreSQL for user data
- **Uber**: MySQL for trip data, user accounts
- **Airbnb**: PostgreSQL for bookings, payments

Limitations:

- ❌ Difficult to scale writes
- ❌ Schema changes can be slow
- ❌ JOIN performance degrades with scale
- ❌ Not ideal for unstructured data

NoSQL Databases

2. Redis (Key-Value Store)

Type: In-Memory Key-Value Store

Best For:

- Caching layer
- Session storage
- Real-time leaderboards
- Pub/Sub messaging
- Rate limiting

Data Structures:

- **String**: Simple key-value
- **Hash**: Object storage
- **List**: Queue, timeline
- **Set**: Unique items, tags
- **Sorted Set**: Leaderboards, timelines with scores
- **HyperLogLog**: Cardinality estimation

Characteristics:

- **In-memory**: Extremely fast (< 1ms latency)
- **Optional persistence**: Snapshots or AOF
- **Atomic operations**: INCR, DECR, etc.
- **Expiration**: Built-in TTL
- **Pub/Sub**: Message broadcasting

Use Cases:

```
1. Session Store
   Key: session:{user_id}
   Value: JSON serialized session data
   TTL: 30 minutes

2. Cache
   Key: user:{user_id}
   Value: User object
   TTL: 1 hour

3. Leaderboard
   Key: leaderboard:game:123
   Type: Sorted Set
   Score: User score
   Value: User ID

4. Rate Limiting
   Key: rate_limit:{user_id}:tweets
   Value: Counter
   TTL: 1 hour

5. Timeline (Twitter/Instagram)
   Key: timeline:{user_id}
   Type: Sorted Set
   Score: Timestamp
   Value: Tweet ID
```

Redis Cluster:

- Automatic sharding
- 1000+ nodes supported
- No single point of failure

Real Examples:

- **Twitter**: Timeline storage, rate limiting
- **Instagram**: Feed cache, session storage
- **Pinterest**: Follower lists, board caching
- **GitHub**: Job queues, caching

When NOT to Use:

- ❌ Primary data store (not durable enough)
- ❌ Complex queries

- ❌ Large objects (> 512 MB limit)
- ❌ Data larger than RAM

---

3. MongoDB (Document Store)

Type: Document-Oriented NoSQL

Best For:

- Content management systems
- Catalogs (product, content)
- User profiles with varying fields
- Mobile applications
- Real-time analytics

Characteristics:

- **Flexible schema**: Each document can have different fields
- **JSON documents**: Natural data model
- **Rich queries**: Support for complex queries
- **Aggregation pipeline**: MapReduce-style processing
- **Horizontal scaling**: Sharding built-in

Data Model:

```
// User document
{
  "_id": ObjectId("507f1f77bcf86cd799439011"),
  "username": "alice",
  "email": "alice@example.com",
  "profile": {
    "age": 25,
    "location": "San Francisco",
    "interests": ["coding", "music"]
  },
  "posts": [
    {"title": "First post", "created_at": ISODate("2025-01-01")}
  ]
}

// Product document (different structure)
{
  "_id": ObjectId("..."),
  "name": "Laptop",
  "price": 999.99,
  "specs": {
    "ram": "16GB",
    "storage": "512GB SSD",
    "processor": "Intel i7"
  },
  "reviews": [...]
}
```

Scaling:

- **Sharding**: Automatic data distribution
- **Replica sets**: High availability (primary + secondaries)
- **Read preference**: Route reads to secondaries

Use Cases:

- **Content Management**: WordPress, blogs
- **Catalogs**: E-commerce product catalogs
- **Mobile Apps**: User data with varying structure
- **Real-time Analytics**: Log aggregation

Real Examples:

- **Uber**: Store trip data, user profiles
- **eBay**: Product catalog
- **MetLife**: Insurance customer data
- **Codecademy**: User progress tracking

When NOT to Use:

- ❌ Need multi-document ACID transactions (< v4.0)
- ❌ Complex JOINs across collections
- ❌ Extremely high write throughput (use Cassandra)

---

4. Cassandra (Wide-Column Store)

Type: Distributed Wide-Column NoSQL

Best For:

- Time-series data (logs, metrics, events)
- Write-heavy workloads
- High availability requirements
- Massive datasets (petabytes)
- Geographic distribution

Characteristics:

- **Masterless**: No single point of failure
- **Linear scalability**: Add nodes linearly
- **Tunable consistency**: Choose CP or AP per query
- **Write-optimized**: Append-only commit log
- **No single point of failure**: P2P architecture

Data Model (Column Family):

```
CREATE TABLE tweets (
    user_id uuid,
```

```
    tweet_id timeuuid,
    content text,
    created_at timestamp,
    like_count counter,
    PRIMARY KEY (user_id, tweet_id)
) WITH CLUSTERING ORDER BY (tweet_id DESC);

-- Partition key: user_id (data co-located)
-- Clustering key: tweet_id (sorted within partition)
```

Consistency Levels:

```
Write Consistency:
- ONE: Write to 1 replica (fast, less durable)
- QUORUM: Write to majority (balanced)
- ALL: Write to all replicas (slow, most durable)

Read Consistency:
- ONE: Read from 1 replica (fast, might be stale)
- QUORUM: Read from majority (balanced)
- ALL: Read from all replicas (slow, most consistent)
```

Use Cases:

- **Time-Series**: IoT sensor data, application logs
- **Write-Heavy**: Twitter tweets, Instagram posts
- **Messaging**: Chat history, notifications
- **Activity Tracking**: User analytics, click streams

Real Examples:

- **Netflix**: Viewing history, recommendations
- **Apple**: iMessage chat logs
- **Instagram**: Photo metadata, user activity
- **Twitter**: Tweets, timeline

When to Use:

- ✅ Writes >> Reads
- ✅ Time-series data
- ✅ Need geo-distribution
- ✅ No complex transactions needed

When NOT to Use:

- ❌ Need ACID transactions
- ❌ Complex queries with aggregations
- ❌ Small dataset (< 1 TB)
- ❌ Frequent schema changes

---

5. DynamoDB (Managed NoSQL)

Type: Managed Key-Value/Document Store (AWS)

Best For:

- Serverless applications
- Gaming leaderboards
- Session storage
- Mobile backends
- IoT applications

Characteristics:

- **Fully managed**: No ops overhead
- **Auto-scaling**: Scales automatically
- **Single-digit millisecond latency**
- **Event-driven**: DynamoDB Streams
- **Global tables**: Multi-region replication

Data Model:

```
{
  "PK": "USER#12345",              // Partition Key
  "SK": "METADATA",                 // Sort Key
  "username": "alice",
  "email": "alice@example.com",
  "created_at": "2025-01-08"
}

// Access pattern: GetItem(PK, SK)
// Query: Query(PK, SK begins_with "ORDER#")
```

Access Patterns:

- **GetItem**: O(1) lookup by primary key
- **Query**: Range query on sort key
- **Scan**: Full table scan (expensive)
- **BatchGetItem**: Multiple items in one call

Use Cases:

- **Gaming**: Player profiles, leaderboards, game state
- **Mobile**: User preferences, app data
- **Ad Tech**: Real-time bidding, user segments
- **E-commerce**: Shopping carts, session data

Real Examples:

- **Lyft**: Trip data, real-time tracking
- **Duolingo**: User progress, streaks

- **Redfin**: Property listings
- **Samsung**: Smart Things IoT data

When to Use:

- ✅ Need managed service (no ops)
- ✅ Simple key-value access patterns
- ✅ Serverless architecture
- ✅ Predictable performance

When NOT to Use:

- ❌ Complex queries (no JOINs, limited filtering)
- ❌ Need flexibility to change DB later
- ❌ Cost-sensitive (can be expensive at scale)

---

6. Elasticsearch (Search Engine)

Type: Distributed Search and Analytics Engine

Best For:

- Full-text search
- Log analytics
- Application monitoring
- E-commerce product search
- Autocomplete/suggestions

Characteristics:

- **Full-text search**: Fuzzy matching, relevance scoring
- **Near real-time**: Indexing delay < 1 second
- **Distributed**: Scales horizontally
- **Aggregations**: Complex analytics queries
- **RESTful API**: Easy HTTP interface

Index Structure:

```
{
  "mappings": {
    "properties": {
      "title": {
        "type": "text",
        "analyzer": "standard"
      },
      "price": {
        "type": "float"
      },
      "created_at": {
        "type": "date"
      },
      "tags": {
```

```
        "type": "keyword"  // Exact match
      }
    }
  }
}
```

Query Types:

```
// Full-text search
{
  "query": {
    "match": {
      "title": "system design"
    }
  }
}

// Fuzzy search (typo tolerance)
{
  "query": {
    "fuzzy": {
      "username": {
        "value": "alcie",  // Will match "alice"
        "fuzziness": "AUTO"
      }
    }
  }
}

// Aggregations
{
  "aggs": {
    "popular_tags": {
      "terms": {"field": "tags", "size": 10}
    }
  }
}
```

Use Cases:

- **E-commerce**: Product search with filters
- **Social Media**: Search tweets, users, hashtags
- **Logging**: ELK stack (Elasticsearch, Logstash, Kibana)
- **Analytics**: Application metrics, user behavior

Real Examples:

- **GitHub**: Code search
- **LinkedIn**: Job search, people search
- **Uber**: Trip search, driver search
- **Netflix**: Content discovery

When to Use:

- ✅ Need full-text search
- ✅ Need autocomplete/suggestions
- ✅ Log aggregation and analysis
- ✅ Complex filtering and faceting

When NOT to Use:

- ❌ Primary data store (use with another DB)
- ❌ Need strong consistency
- ❌ ACID transactions
- ❌ Simple key-value lookups (use Redis)

---

7. Neo4j (Graph Database)

Type: Native Graph Database

Best For:

- Social networks (followers, friends)
- Recommendation engines
- Fraud detection (pattern recognition)
- Knowledge graphs
- Network topology

Characteristics:

- **Nodes and Relationships**: Natural graph model
- **Cypher query language**: Pattern matching
- **Index-free adjacency**: O(1) traversal
- **ACID transactions**: Full ACID support

Data Model:

```
// Create nodes
CREATE (alice:User {name: "Alice", age: 25})
CREATE (bob:User {name: "Bob", age: 30})
CREATE (post:Post {content: "Hello World"})

// Create relationships
CREATE (alice)-[:FOLLOWS]->(bob)
CREATE (bob)-[:FOLLOWS]->(alice)
CREATE (alice)-[:POSTED]->(post)
CREATE (bob)-[:LIKED]->(post)

// Query: Find mutual friends
MATCH (me:User {name: "Alice"})-[:FOLLOWS]->(friend)-[:FOLLOWS]->(fof)
WHERE NOT (me)-[:FOLLOWS]->(fof) AND me <> fof
RETURN fof.name, COUNT(*) as mutual_friends
ORDER BY mutual_friends DESC
```

```
// Query: Friend-of-friend recommendations
MATCH (me:User)-[:FOLLOWS]->(friend)-[:FOLLOWS]->(fof)
WHERE NOT (me)-[:FOLLOWS]->(fof) AND me <> fof
WITH fof, COUNT(*) as connections
ORDER BY connections DESC
LIMIT 10
RETURN fof
```

Use Cases:

- **Social Networks**: Facebook friend graph, LinkedIn connections
- **Recommendations**: "People you may know", "Similar products"
- **Fraud Detection**: Money laundering patterns
- **Identity Management**: Access control graphs
- **Supply Chain**: Product dependencies

Real Examples:

- **LinkedIn**: Connection degrees, job recommendations
- **eBay**: Product recommendations
- **Walmart**: Supply chain optimization
- **Airbnb**: Location-based recommendations

When to Use:

- ✅ Data is naturally a graph (nodes + relationships)
- ✅ Need to traverse relationships (friend-of-friend)
- ✅ Pattern matching queries
- ✅ Need to analyze network structure

When NOT to Use:

- ❌ Simple key-value lookups
- ❌ Tabular data with no relationships
- ❌ Need extreme write scalability
- ❌ Data doesn't have many relationships

---

8. Amazon S3 (Object Storage)

Type: Distributed Object Storage

Best For:

- File storage (images, videos, documents)
- Backup and archival
- Static website hosting
- Data lakes
- Content distribution

Characteristics:

- **11 9's durability** (99.999999999%)

- **Infinite scalability**
- **Object-level storage** (files, not blocks)
- **Metadata support**
- **Versioning and lifecycle policies**

Storage Classes:

```
S3 Standard:
- Hot data, frequently accessed
- $0.023/GB/month
- Millisecond latency

S3 Intelligent-Tiering:
- Automatic cost optimization
- Moves data between tiers based on access

S3 Infrequent Access (IA):
- Data accessed < once/month
- $0.0125/GB/month
- Millisecond latency

S3 Glacier:
- Archive, accessed rarely
- $0.004/GB/month
- Minutes to hours retrieval

S3 Glacier Deep Archive:
- Long-term archive (7-10 years)
- $0.00099/GB/month
- 12 hours retrieval
```

Use Cases:

- **Media Storage**: Instagram photos, YouTube videos
- **Backups**: Database backups, file backups
- **Data Lakes**: Raw data for analytics
- **Static Assets**: Website CSS, JS, images

Real Examples:

- **Netflix**: Video storage, transcoded files
- **Airbnb**: Property photos
- **Dropbox**: File storage
- **Spotify**: Music files

Best Practices:

- Use CloudFront CDN in front of S3
- Enable versioning for critical data
- Lifecycle policies to move old data to Glacier
- Use pre-signed URLs for temporary access

When NOT to Use:

- ❌ Need file system operations (use EFS)
- ❌ Need low latency (< 10ms)
- ❌ Frequently updated small files

---

9. Apache Kafka (Event Streaming)

Type: Distributed Event Streaming Platform

Best For:

- Event sourcing
- Stream processing
- Log aggregation
- Real-time analytics
- Microservices communication

Characteristics:

- **High throughput**: Millions of messages/sec
- **Durable**: Persists messages to disk
- **Scalable**: Add brokers linearly
- **Replay capability**: Re-process old events
- **Ordering guarantees**: Within partition

Architecture:

```
Producer → Kafka Broker → Consumer
             (Topics)

Topic: "user-events"
Partitions: 10 (for parallelism)
Replication: 3 (for durability)
Retention: 7 days
```

Use Cases:

```
1. Event Sourcing
   - All state changes as events
   - Can rebuild state from event log
   - Example: Order placed, Payment processed, Item shipped

2. Stream Processing
   - Real-time analytics
   - Fraud detection
   - Example: Detect unusual login patterns

3. Change Data Capture (CDC)
   - Capture DB changes
```

```
   - Sync multiple datastores
   - Example: MySQL → Kafka → Elasticsearch

4. Log Aggregation
   - Centralize logs from all services
   - Feed to ELK stack for analysis
```

Real Examples:

- **LinkedIn**: Activity streams, metrics
- **Netflix**: Real-time recommendations
- **Uber**: Trip events, surge pricing
- **Airbnb**: Booking events, pricing updates

When to Use:

- ✅ Need durable message queue
- ✅ Multiple consumers for same data
- ✅ Stream processing required
- ✅ Event sourcing architecture

When NOT to Use:

- ❌ Simple request-response (use REST)
- ❌ Need transactions
- ❌ Low latency requirement (< 1ms)

---

10. Amazon Redshift / Google BigQuery (Data Warehouse)

Type: Columnar Data Warehouse

Best For:

- Business intelligence
- Analytics and reporting
- Historical data analysis
- OLAP (Online Analytical Processing)
- Data science

Characteristics:

- **Columnar storage**: Efficient for aggregations
- **MPP (Massively Parallel Processing)**
- **Optimized for reads**: Not for transactional workloads
- **SQL interface**: Familiar query language
- **Petabyte scale**: Handle massive datasets

Column-Oriented vs Row-Oriented:

```
Row-Oriented (MySQL):
Row 1: [ID=1, Name="Alice", Age=25, City="SF"]
```

```
Row 2: [ID=2, Name="Bob", Age=30, City="NY"]
- Good for: SELECT * FROM users WHERE id = 1
- Bad for: SELECT AVG(age) FROM users

Column-Oriented (Redshift):
ID column: [1, 2, 3, ...]
Name column: ["Alice", "Bob", "Charlie", ...]
Age column: [25, 30, 28, ...]
- Bad for: SELECT * WHERE id = 1 (read all columns)
- Good for: SELECT AVG(age) (read only age column)
```

Use Cases:

```
1. Business Analytics
   - Daily/monthly revenue reports
   - User growth metrics
   - Sales forecasting

2. Data Science
   - Train ML models on historical data
   - Feature engineering
   - A/B test analysis

3. Compliance
   - Audit logs retention
   - Historical data queries
```

ETL Pipeline:

```
[Operational DB] → [Kafka/Airflow] → [Data Warehouse] → [BI Tools]
 (MySQL/Postgres)                     (Redshift)        (Tableau/Looker)
```

Real Examples:

- **Netflix**: Viewing analytics, recommendations training
- **Spotify**: User listening analytics
- **Pinterest**: Pin analytics, growth metrics

When to Use:

- ✅ OLAP (analytics) not OLTP (transactions)
- ✅ Aggregate queries (SUM, AVG, COUNT)
- ✅ Historical data analysis
- ✅ Join large datasets

When NOT to Use:

- ❌ Transactional workload
- ❌ Real-time updates (batch-oriented)

- ❌ Need low latency (queries take seconds)

Database Comparison Matrix

Database	Type	Consistency	Scalability	Best For	Avoid For
MySQL/PostgreSQL	SQL	Strong (ACID)	Vertical + Read Replicas	Structured data, transactions, relationships	Massive writes, unstructured data
Redis	Key-Value	Eventual	Horizontal (Cluster)	Caching, sessions, real-time	Primary storage, complex queries
MongoDB	Document	Eventual (Tunable)	Horizontal (Sharding)	Flexible schema, mobile apps	Complex JOINs, high-write volume
Cassandra	Wide-Column	Tunable	Linear Horizontal	Time-series, write-heavy, HA	ACID transactions, complex queries
DynamoDB	Key-Value/Document	Eventual	Auto	Serverless, key-value, mobile	Complex queries, cost-sensitive
Elasticsearch	Search Engine	Eventual	Horizontal	Full-text search, analytics	Primary storage, strong consistency
Neo4j	Graph	Strong (ACID)	Vertical	Social graphs, recommendations	Non-graph data, write-heavy
S3	Object Store	Eventual	Infinite	Files, backups, static assets	Frequently updated files, low latency
Redshift/BigQuery	Data Warehouse	Strong	Horizontal (MPP)	Analytics, reporting, OLAP	OLTP, real-time updates

---

Real-World Use Cases

Case Study 1: E-Commerce Platform (Amazon-like)

Requirements:

- User accounts and authentication
- Product catalog
- Shopping cart
- Order management
- Search functionality
- Recommendations

Database Choice:

```
1. User Data & Orders: PostgreSQL
   Why: ACID transactions, complex queries, relationships
   Scale: Master-slave replication, 10 read replicas

2. Product Catalog: MongoDB
   Why: Flexible schema (different product types)
   Scale: Sharding by product_id

3. Shopping Cart: Redis
   Why: Fast, temporary data, TTL support
   Scale: Redis Cluster

4. Product Search: Elasticsearch
   Why: Full-text search, faceting, autocomplete
   Scale: 5-node cluster with sharding

5. Recommendation Engine: Neo4j
   Why: "Users who bought X also bought Y"
   Scale: Read replicas for queries

6. Order Analytics: Redshift
   Why: Business intelligence, reporting
   Scale: MPP for large aggregations
```

Case Study 2: Social Media (Twitter-like)

Requirements:

- User profiles
- Post tweets (500M/day)
- Timeline generation
- Search tweets
- Trending topics
- Direct messaging

Database Choice:

```
1. User Profiles: MySQL
   Why: Structured, ACID, relationships
   Scale: Shard by user_id
```

```
2. Tweets: Cassandra
   Why: Write-heavy (500M/day), time-series
   Scale: 100+ nodes, linear scaling

3. Timelines (Cache): Redis
   Why: Fast reads, sorted sets, TTL
   Scale: Redis Cluster

4. Social Graph: Cassandra or Neo4j
   Why: Fast follower lookup, traversal
   Scale: Partition by user_id

5. Search: Elasticsearch
   Why: Full-text search on tweets
   Scale: 20-node cluster

6. Direct Messages: Cassandra
   Why: High write volume, time-series
   Scale: Partition by conversation_id

7. Analytics: Kafka + Flink → Redshift
   Why: Stream processing, trend detection
```

Case Study 3: Uber/Lyft (Ride-Sharing)

Requirements:

- Driver/rider locations (real-time)
- Trip history
- Payments
- Pricing
- Analytics

Database Choice:

```
1. User Accounts: PostgreSQL
   Why: ACID for payments, structured
   Scale: Master-slave, shard by user_id

2. Trip Data: Cassandra
   Why: High write volume, time-series
   Scale: Geo-distributed

3. Real-time Locations: Redis Geo
   Why: Geo-spatial queries, fast updates
   Scale: Redis Cluster

4. Payment Transactions: PostgreSQL
   Why: ACID required, cannot lose money
   Scale: Master-master with conflict resolution
```

```
5. Driver/Rider Search: Elasticsearch
   Why: Geo-search, filtering
   Scale: Multi-region clusters

6. Analytics: S3 + Redshift
   Why: Store raw data (S3), analyze (Redshift)
```

Decision Tree

```
START: What are your requirements?

├─ Need ACID transactions?
│  ├─ YES → Consider SQL
│  │  ├─ Complex queries/JOINs?
│  │  │  ├─ YES → PostgreSQL/MySQL
│  │  │  └─ NO → Could use NoSQL with transactions
│  │  └─ Geographic distribution?
│  │     ├─ YES → CockroachDB, Spanner
│  │     └─ NO → PostgreSQL/MySQL
│  │
│  └─ NO → Consider NoSQL
│     ├─ What's your access pattern?
│     │  ├─ Key-value lookups → Redis/DynamoDB
│     │  ├─ Document storage → MongoDB
│     │  ├─ Time-series data → Cassandra/InfluxDB
│     │  ├─ Graph relationships → Neo4j
│     │  └─ Full-text search → Elasticsearch
│     │
│     ├─ What's your workload?
│     │  ├─ Write-heavy → Cassandra
│     │  ├─ Read-heavy → MongoDB with replicas
│     │  └─ Balanced → DynamoDB
│     │
│     └─ What's your scale?
│        ├─ Small (< 1 TB) → Any SQL/NoSQL
│        ├─ Medium (1-100 TB) → Sharded SQL or NoSQL
│        └─ Large (> 100 TB) → Cassandra, DynamoDB

├─ Need full-text search?
│  └─ YES → Elasticsearch (+ another DB for source)

├─ Need graph traversal?
│  └─ YES → Neo4j or graph layer over SQL

├─ Storing files/blobs?
│  └─ YES → S3, Google Cloud Storage

└─ Analytics/Reporting?
   └─ YES → Redshift, BigQuery, Snowflake
```

Hybrid Database Architectures

Most production systems use multiple databases!

Pattern 1: Polyglot Persistence

```
[Application]
     ├──→ PostgreSQL (User data, transactions)
     ├──→ Redis (Caching, sessions)
     ├──→ Cassandra (Time-series events)
     ├──→ Elasticsearch (Search)
     ├──→ S3 (Media files)
     └──→ Neo4j (Social graph)
```

Benefits:

- Use right tool for each job
- Optimize for specific use cases
- Independent scaling

Challenges:

- Increased complexity
- Data synchronization
- Multiple systems to manage

Pattern 2: CQRS (Command Query Responsibility Segregation)

```
Writes → [PostgreSQL] → [Kafka] → [Elasticsearch] ← Reads (Search)
                           ↓
                      [Redis] ← Reads (Cache)
                           ↓
                      [Cassandra] ← Reads (Analytics)
```

Benefits:

- Optimize writes and reads separately
- Scale independently
- Different data models for different needs

Pattern 3: Lambda Architecture

```
[Data Source] → [Kafka]
                  ├──→ [Speed Layer] → [Redis] (Real-time)
                  └──→ [Batch Layer] → [Redshift] (Historical)
                                            ↓
                                    [Serving Layer] (Combined view)
```

Use Case: Real-time + historical analytics

Interview Tips

How to Choose Database in Interview

Step 1: Clarify Requirements

```
Ask these questions:
- What's the data structure? (structured vs unstructured)
- What's the access pattern? (read-heavy vs write-heavy)
- What's the scale? (MB, GB, TB, PB)
- What's the consistency requirement? (strong vs eventual)
- Need transactions? (ACID vs BASE)
- Latency requirements? (ms vs seconds)
```

Step 2: Start with SQL, Justify NoSQL

```
Default to SQL (PostgreSQL/MySQL) UNLESS:
- Scale exceeds SQL capacity (> 10TB single table)
- Write throughput > 10K QPS
- Need flexible schema
- Simple key-value access pattern
- Geographic distribution required

Interviewer wants to hear: "I'd start with PostgreSQL for
user data because we need ACID transactions, but as we scale
beyond 10M users, we'd shard. For the tweet feed which is
write-heavy with 500M tweets/day, I'd use Cassandra."
```

Step 3: Explain Trade-offs

```
For every database choice, mention:
1. Why this database? (strengths)
2. What are the drawbacks? (weaknesses)
3. How to scale it?
4. Alternative options?

Example:
"I'd use Redis for caching because it's in-memory (< 1ms
latency) and supports sorted sets for timelines. The trade-off
is it's not durable, so we need PostgreSQL as source of truth.
We'd scale Redis with Redis Cluster. Alternative would be
Memcached, but Redis has richer data structures."
```

Common Interview Scenarios

Scenario: "Design Twitter"

```
Interviewer Expectation:

User Data → SQL (MySQL/PostgreSQL)
  - Why: Structured, relationships, ACID
  - Scale: Shard by user_id

Tweets → Cassandra
  - Why: Write-heavy (500M/day), time-series
  - Scale: Linear with nodes

Timeline Cache → Redis
  - Why: Fast reads, sorted sets
  - Scale: Redis Cluster

Social Graph → Cassandra or Neo4j
  - Why: Fast fan-out, relationship queries
  - Scale: Partition by user_id

Search → Elasticsearch
  - Why: Full-text search
  - Scale: Horizontal with shards

Don't say: "I'll use MongoDB for everything"
This shows lack of understanding of trade-offs
```

Scenario: "Design Instagram"

```
Similar to Twitter but emphasize:

Media Storage → S3
  - Why: Durable (11 9s), scalable, cheap
  - Scale: Infinite, built-in
  - With CDN (CloudFront) for delivery

Photo Metadata → Cassandra
  - Why: Write-heavy, many photos
  - Scale: Geographic distribution

Rest similar to Twitter architecture
```

Scenario: "Design Uber"

```
Emphasize:

Payment Transactions → PostgreSQL
  - Why: MUST have ACID, cannot lose money
```

```
  - Scale: Master-master with conflict resolution

Real-time Locations → Redis Geo
  - Why: Geo-spatial queries, fast updates
  - Commands: GEOADD, GEORADIUS

Trip History → Cassandra
  - Why: Time-series, write-heavy
  - Scale: Geo-partitioned

Always mention: "Payments need SQL with ACID because
financial transactions require strong consistency"
```

Quick Reference Guide

When Interviewer Asks: "What database would you use for..."

User Profiles/Accounts:
→ PostgreSQL/MySQL

- Structured, relationships, ACID

Session Storage:
→ Redis

- Fast, TTL support, in-memory

Product Catalog (varying fields):
→ MongoDB

- Flexible schema, JSON documents

Social Media Posts (Twitter/Instagram):
→ Cassandra

- Write-heavy, time-series, massive scale

Search Functionality:
→ Elasticsearch

- Full-text search, autocomplete

Recommendation System:
→ Neo4j or collaborative filtering in Redis

- Graph relationships

File/Media Storage:
→ S3

- Durable, scalable, with CDN

Analytics/Reporting:
→ Redshift/BigQuery

- OLAP, aggregations, historical analysis

Real-time Leaderboard:
→ Redis Sorted Sets

- Fast, atomic operations, ranking

Chat Messages:
→ Cassandra or MongoDB

- Time-series, high volume

IoT Sensor Data:
→ Cassandra or InfluxDB

- Time-series, write-heavy, massive scale

Scaling Patterns Summary

SQL Scaling Patterns

1. Vertical Scaling (Scale Up):

```
Small: 2 CPU, 8 GB RAM
Medium: 8 CPU, 32 GB RAM
Large: 32 CPU, 128 GB RAM
Extra Large: 96 CPU, 768 GB RAM

Limit: ~1 TB RAM per instance
Cost: Expensive at large sizes
```

2. Read Replicas (Master-Slave):

```
[Master] ———→ Writes
   ↓
   ├→ [Slave 1] ———→ Reads
   ├→ [Slave 2] ———→ Reads
   └→ [Slave 3] ———→ Reads

Benefit: Scale reads linearly
Limitation: Doesn't scale writes
```

3. Sharding (Horizontal Partitioning):

```
Users 1–1M   → Shard 1
Users 1M–2M  → Shard 2
```

```
Users 2M-3M  → Shard 3

Benefit: Scale both reads and writes
Limitation: Complex queries, no JOINs across shards
```

4. Federation (Vertical Partitioning):

```
[Users DB] —→ User data
[Products DB] —→ Product catalog
[Orders DB] —→ Order history

Benefit: Smaller DBs, better cache locality
Limitation: Can't JOIN across databases
```

NoSQL Scaling Patterns

Cassandra:

- Add nodes → Linear scalability
- No downtime during scaling
- Handles petabytes

MongoDB:

- Sharding by shard key
- Automatic balancing
- Replica sets for HA

Redis:

- Redis Cluster (automatic sharding)
- Master-slave replication
- Sentinel for failover

Performance Characteristics

Latency Comparison

Operation	MySQL	Redis	Cassandra	MongoDB	Elasticsearch
Point Query (by PK)	1-5ms	< 1ms	1-10ms	5-15ms	10-50ms
Range Query	10-100ms	1-5ms	5-50ms	10-100ms	20-200ms
Complex Query	100-1000ms	N/A	N/A	50-500ms	50-500ms
Write	1-10ms	< 1ms	< 1ms	1-5ms	10-100ms
Bulk Write	10-100ms	1-10ms	1-10ms	10-50ms	100-1000ms

**Latencies are approximate and depend on data size, indexing, hardware*

Throughput Comparison

Database	Reads/Second	Writes/Second	Notes
MySQL (single instance)	10K-50K	1K-10K	With proper indexing
PostgreSQL (single instance)	10K-50K	5K-20K	Better write performance
Redis (single instance)	100K-1M	100K-1M	In-memory speed
Cassandra (cluster)	1M+	1M+	Linear with nodes
MongoDB (cluster)	100K-1M	50K-500K	Depends on sharding
Elasticsearch (cluster)	100K-500K	10K-100K	Index-heavy writes

---

Common Mistakes in Interviews

❌ Mistake 1: "I'll use MongoDB for everything"

Why Bad: Shows you don't understand trade-offs
Better: "I'll use PostgreSQL for user data (need ACID), Cassandra for tweets (write-heavy), and Redis for cache"

❌ Mistake 2: "NoSQL is always faster than SQL"

Why Bad: Not true for all workloads
Better: "NoSQL can be faster for simple key-value lookups and high write throughput, but SQL is often faster for complex JOINs with proper indexing"

❌ Mistake 3: Not mentioning caching

Why Bad: Cache is critical for scalability
Better: "I'd put Redis in front of PostgreSQL to handle 80-90% of reads from cache"

❌ Mistake 4: Choosing NoSQL without justification

Why Bad: SQL is default for most use cases
Better: "I'd start with PostgreSQL, but if writes exceed 10K QPS or we need to store > 10TB, I'd migrate to Cassandra"

❌ Mistake 5: Not discussing consistency

Why Bad: Shows incomplete understanding
Better: "For user profiles we need strong consistency (SQL), but for the feed eventual consistency is acceptable (Cassandra with eventual consistency)"

Interview Cheat Sheet

Quick Decision Matrix

Requirement	Database Choice

Requirement	Database Choice
Transactions required	PostgreSQL, MySQL
Write-heavy (> 10K QPS)	Cassandra, DynamoDB
Flexible schema	MongoDB, DynamoDB
Caching	Redis, Memcached
Full-text search	Elasticsearch, Algolia
Graph data	Neo4j, Amazon Neptune
Time-series	Cassandra, InfluxDB, TimescaleDB
File storage	S3, Google Cloud Storage
Analytics	Redshift, BigQuery, Snowflake
Geographic distribution	Cassandra, DynamoDB Global Tables
Real-time data	Redis, Firebase Realtime DB
Audit logs	Cassandra, S3 + Redshift
Leaderboards	Redis Sorted Sets
Counting	Redis Counters, Cassandra Counters

Scaling Thresholds

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

Read/Write Patterns

```
Read-Heavy (100:1):
- SQL with read replicas
- Cache aggressively (Redis)
- Consider denormalization

Write-Heavy (1:10):
- Cassandra (optimized for writes)
- Queue writes (Kafka → batch insert)
- Use append-only patterns

Balanced:
- MongoDB, DynamoDB
- PostgreSQL with good indexing
```

Additional Databases Worth Knowing

InfluxDB (Time-Series Database)

Use For: Metrics, monitoring, IoT sensors
Example: Prometheus metrics, Grafana dashboards

TimescaleDB (Time-Series on PostgreSQL)

Use For: Time-series with SQL familiarity
Example: Financial tick data, sensor readings

CockroachDB (Distributed SQL)

Use For: Global distribution with ACID
Example: Multi-region user data

Google Spanner (Distributed SQL)

Use For: Global consistency with ACID
Example: Google's internal services

Firebase (Real-time Database)

Use For: Mobile apps, real-time sync
Example: Chat apps, collaborative editing

Apache HBase (Wide-Column on Hadoop)

Use For: Hadoop ecosystem, MapReduce
Example: Large-scale batch processing

Summary & Key Takeaways

The Golden Rules

1. **Start with SQL (PostgreSQL/MySQL)** unless you have a specific reason not to
    - It handles 90% of use cases
    - Well-understood, mature tooling
    - Easy to reason about
2. **Add NoSQL for specific needs**:
    - Caching → Redis
    - Search → Elasticsearch
    - Write-heavy → Cassandra
    - Files → S3
3. **Most systems use 3-5 databases**:

- Primary: SQL or MongoDB
- Cache: Redis
- Search: Elasticsearch
- Files: S3
- Analytics: Redshift

4. **Always mention scaling strategy**:
   - How will this database scale?
   - What are the limits?
   - When do we need to shard/add replicas?

5. **Discuss consistency requirements**:
   - Strong consistency for money/inventory
   - Eventual consistency for social feeds
   - Know the trade-offs

Interview Answer Template

```
"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: [TB of data, QPS needed]
4. Consistency needs: [strong/eventual]
5. Special requirements: [transactions/search/real-time]

To scale this, I would [replication/sharding/clustering strategy].

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

Alternative options would be [other database], but I prefer
[chosen database] because [specific advantage for this use case]."
```

Red Flags to Avoid

❌ "MongoDB scales better than MySQL" (too vague)
✅ "MongoDB scales writes horizontally with sharding, while MySQL requires application-level sharding"

❌ "We'll use NoSQL because it's faster" (not always true)
✅ "We'll use Cassandra for tweets because it handles 500M writes/day better than SQL, which would require complex sharding"

❌ "One database for everything" (unrealistic)
✅ "We'll use PostgreSQL for users, Cassandra for tweets, and Redis for cache - each optimized for its purpose"

References & Further Reading

Books

1. **"Designing Data-Intensive Applications"** by Martin Kleppmann
    - Comprehensive database comparison
    - Distributed systems concepts
    - Must-read for system design
2. **"Database Internals"** by Alex Petrov
    - How databases work internally
    - Deep dive into storage engines
3. **"Seven Databases in Seven Weeks"** by Eric Redmond
    - Hands-on with different database types

Online Resources

1. **System Design Primer** - https://github.com/donnemartin/system-design-primer
2. **Database Rankings** - https://db-engines.com/en/ranking
3. **High Scalability Blog** - http://highscalability.com
4. **AWS Database Decision Guide** - AWS Architecture Center
5. **CAP Theorem Explained** - https://robertgreiner.com/cap-theorem-revisited/

Company Engineering Blogs

- **Netflix Tech Blog**: Cassandra usage
- **Uber Engineering**: Database sharding
- **Instagram Engineering**: PostgreSQL scaling
- **Twitter Engineering**: Manhattan database
- **LinkedIn Engineering**: Kafka architecture

Appendix: Database Feature Comparison

ACID Support

- **Full ACID**: PostgreSQL, MySQL, Neo4j
- **Limited ACID**: MongoDB (v4.0+, single document only)
- **No ACID**: Cassandra, Redis, Elasticsearch
- **Eventually consistent**: DynamoDB, Cassandra

Query Capabilities

- **SQL Support**: PostgreSQL, MySQL, Redshift, BigQuery
- **Secondary Indexes**: MongoDB, DynamoDB, Cassandra
- **Full-Text Search**: Elasticsearch, PostgreSQL (limited)
- **Graph Queries**: Neo4j (Cypher), Amazon Neptune
- **Aggregations**: MongoDB, Elasticsearch, Redshift

Operational Characteristics

- **Fully Managed**: DynamoDB, BigQuery, Firebase
- **Self-Hosted**: MySQL, PostgreSQL, Cassandra, MongoDB
- **Hybrid**: RDS (managed MySQL/Postgres), DocumentDB (managed MongoDB API)

Cost (Approximate Monthly)

```
For 1 TB storage + moderate traffic:

Redis (AWS ElastiCache): $1,000–2,000
PostgreSQL (RDS): $500–1,500
MongoDB Atlas: $500–2,000
Cassandra (self-hosted): $300–1,000
DynamoDB: $500–3,000 (varies widely)
S3: $23
Elasticsearch: $1,000–3,000
Redshift: $2,000–5,000
```

Document Version: 1.0
Last Updated: January 8, 2025
For: System Design Interview Preparation
Status: Complete & Interview-Ready ✅

Remember: In interviews, justify EVERY database choice with specific requirements and trade-offs!