SQL vs NoSQL

Structured Query Language databases store data in tables with predefined schemas. Data is organized into rows and columns with relationships between tables.

Popular SQL Databases:

- PostgreSQL (most feature-rich, open source)
- MySQL (widely used, open source)
- Oracle (enterprise, expensive)
- Microsoft SQL Server (enterprise, Windows-focused)
- SQLite (embedded, single-file)

When to use SQL 🏦

- **Complex relationships:** Data has many interconnected entities (users, orders, products, reviews).
- **ACID transactions:** Financial systems, inventory management, & where consistency is critical.
- **Complex queries:** Need to run analytics, generate reports, perform aggregations across multiple tables.
- **Data integrity:** Enforced constraints, foreign keys, data validation at database level.
- **Mature ecosystem:** Well-understood patterns, extensive tooling, experienced developers.
- **Structured, stable data:** Schema doesn't change frequently, data structure is well-defined.

SQL (ACID Transactions):

- **Atomicity:** All operations succeed or all fail
- **Consistency:** Data always in valid state
- **Isolation:** Concurrent transactions don't interfere
- **Durability:** Committed data survives crashes

NoSQL databases store data in various formats other than tables—documents, key-value pairs, graphs, or wide columns. Schemas are flexible or non-existent.

Types & examples:

- **Key-Value** → Redis, DynamoDB
- **Document** → MongoDB, CouchDB
- **Column** → Cassandra
- **Graph** → Neo4j

When to use NoSQL 🚀

- Logs, events, metrics
- Caching
- Real-time analytics
- Large-scale distributed systems
- Flexible schema
- Unstructured data
- High availability
- Horizontal scaling

NoSQL (BASE):

- **Basically Available:** System mostly available
- **Soft state:** State may change without input
- **Eventual consistency:** Eventually becomes consistent

Detailed Comparison

Feature	SQL	NoSQL
Schema	Fixed, predefined	Flexible, dynamic
Scalability	Vertical (mainly)	Horizontal (native)
Relationships	JOINs, foreign keys	Embedded or manual
Transactions	ACID, multi-row	Limited, eventual consistency
Query complexity	Very powerful (JOINs, subqueries)	Simple, key-based
Consistency	Strong consistency	Eventual consistency
Data integrity	Enforced constraints	Application-level
Learning curve	Steeper (SQL syntax)	Easier (JSON-like)
Maturity	Very mature (40+ years)	Newer (10-20 years)
Use cases	Complex business logic	High-scale, flexible data

Scalability

SQL:

- **Vertical scaling** (bigger server) traditionally
- **Horizontal scaling** (more servers) is challenging
- Can add read replicas for read scaling
- Sharding possible but complex

NoSQL:

- **Horizontal scaling** built-in
- Designed for distributed systems
- Easy to add nodes to cluster
- Automatic data distribution (sharding)

Real-World Hybrid Approach (Most large systems use **both)** :

E-commerce example:

PostgreSQL (SQL):

- User accounts
- Product catalog
- Orders and payments
- Inventory

Redis (Key-Value NoSQL):

- Session storage
- Shopping cart data
- Rate limiting
- Caching

MongoDB (Document NoSQL):

- Product reviews
- User activity logs
- Personalization data

Elasticsearch (Document NoSQL):

- Product search
- Full-text search

Each database is chosen for what it does best.

CAP Theorem Context

SQL databases typically choose:

- Consistency + Availability (within single datacenter)
- Sacrifice partition tolerance

NoSQL databases typically choose:

- Availability + Partition tolerance
- Sacrifice consistency (eventual consistency)

Modern Trends

NewSQL: Databases trying to combine SQL benefits with NoSQL scalability:

- Google Spanner
- CockroachDB
- VoltDB

Multi-model databases: Support multiple data models in one system:

- ArangoDB (document + graph)
- OrientDB (document + graph + key-value)

Postgres extensions: PostgreSQL adding NoSQL features:

- JSONB support (document-like)
- Still maintains ACID guarantees