Real-World Applications with Database Examples

---

1. SQL Database Example – Banking System

Real-World Application

Online Banking System
(account management, fund transfer, transaction history)

Database Used

SQL (Relational Database – PostgreSQL / MySQL)

Data Format

- Data is stored in tables (rows and columns)
- Uses a fixed schema
- Relationships are maintained using primary keys and foreign keys

---

Example Table Structure (Customer Table)

customer_id	name	email	phone
1	Ankita Advitot	ankita@gmail.com	8765434567
2	Sanjana Mutkiri	sanjana@gmail.com	8765435693
3	Sakshi Jain	sakshi@gmail.com	9876543456

Why SQL is Suitable

Banking systems require ACID properties:

- **Atomicity:**
  A transaction either completes fully or not at all
- **Consistency:**
  Account balances remain correct
- **Isolation:**
  Multiple transactions don't interfere with each other
- **Durability:**
  Data is permanently saved even after crashes

One Example: Sending Money Using a Mobile Banking App

Imagine you are sending ₹1,000 to your friend using a banking app.

1. Atomicity – "It should fully work or not work at all"

When you send ₹1,000:

- The money should be removed from your account
- And added to your friend's account

If something goes wrong (network issue, app crash), then no money should be taken from you. You should never see a situation where money is deducted from you but your friend doesn't receive it.

---

2. Consistency – "The system should always make sense"

Before sending money:

- You have ₹5,000 in your account

After sending ₹1,000:

- Your balance should become ₹4,000

The system should never:

- Show a wrong balance
- Allow you to send more money than you have

This means the system always keeps correct and sensible data.

---

3. Isolation – “Others should not affect your transaction”

At the same time:

- You are sending ₹1,000 to a friend
- Someone else is also using the bank app

Your transaction should happen independently.
What others are doing should not disturb your payment.
This prevents confusion and mistakes when many people use the system together.

---

4. Durability

Durability means that once a transaction is completed, it will not be lost, even if the system crashes or power fails.

After you send money and see:
“Transaction Successful”

Even if:

- Your phone battery dies
- The app closes
- There is a power cut

The bank will not forget the transaction.
It remains recorded permanently.

How to Achieve ACID Property

Below is a **clean, simple, documentation-ready explanation** you can paste into your project README or design doc.

How Databases Achieve ACID

1. Transactions (All-or-Nothing Execution)

A transaction groups many operations into one unit.

Example:

```
BEGIN;
UPDATE A;
UPDATE B;
COMMIT;
```

This means:

- Either all updates succeed
- Or none of them happen

If anything fails, the database **rolls everything back**.

This supports **Atomicity**.

---

2. Write-Ahead Log (WAL)

Before changing real data, the database writes the change into a **log file**.

This log is saved safely to disk first.

Only after the log is stored does the database update the actual tables.

Why this matters:

- If the system crashes, the database reads this log

- It can **restore finished work**
- And **remove half-done work**

This gives:

- **Atomicity** (no half updates)
- **Durability** (saved data never disappears)

---

3. Crash Recovery

When the database restarts after a crash:

1. It reads the WAL (log)
2. It re-applies all **committed** changes
3. It removes all **uncommitted** changes

So the database always comes back in a **correct state**.

This protects:

- **Atomicity**
- **Durability**

---

4. Locks and MVCC (Isolation)

These systems prevent users from disturbing each other.

Lock-based system

- Readers lock data when reading
- Writers lock data when writing
- Others must wait

This prevents:

- Dirty reads
- Overwritten data

But it can be slow.

MVCC (Modern Databases)

Instead of blocking people:

- The database keeps **multiple versions** of rows
- Each user sees their own **snapshot**

Readers read old versions
Writers create new versions

They do not block each other.

This gives:

- High performance
- No dirty or broken data

This ensures **Isolation**.

---

5. Constraints (Consistency Rules)

The database enforces rules like:

- Balance must be ≥ 0
- Email must be unique
- Customer must exist for an order

Before saving anything, the database checks all rules.

If any rule is broken:

- The whole transaction is cancelled

So the database always stays in a **valid state**.

This ensures **Consistency**.

SQL Databases Ensure Data Integrity Using Constraints

Data integrity means the data is correct, meaningful, and trustworthy.
SQL databases use rules (called constraints) to make sure wrong or incomplete data does not enter the system.

Ideal for Structured Data and Complex Queries

Structured data means information that is well-organized and follows a fixed format.

Examples:

- Customer name
- Account number
- Balance
- Transaction date

SQL databases are good at handling this kind of organized data because everything is stored neatly in rows and columns.

Mandatory for Legal, Audit, and Compliance Requirements

Banks and companies must follow laws and rules set by the government.
This means:

- Every transaction must be recorded
- Data must be accurate and traceable
- Records must not be changed secretly

SQL databases help because they:

- Keep clear and permanent records
- Make it easy to check past data
- Support audits where authorities verify transactions

In short, SQL databases help organizations prove that everything is correct and legal.

---

2. NoSQL Database Example – Social Media Application

1. Netflix – Cassandra (Column-Family NoSQL Database)

Why Netflix Needs Cassandra

- **High Traffic:**
  Netflix has millions of users watching videos at the same time. Cassandra can handle a very large number of requests without slowing down because it distributes data across many servers.
- **High Availability:**
  Netflix cannot afford downtime. Even if one server fails, Cassandra continues to work because data is copied across multiple machines. Users can still stream content without interruption.
- **Fast Write Performance:**
  Every click, pause, and watch history update must be saved instantly. Cassandra is designed to write data very quickly, which is important for real-time user activity.
- **Scalability:**
  As Netflix gains more users, Cassandra allows new servers to be added easily without changing the system. This helps Netflix grow smoothly.

Use at Netflix

- Saves user viewing history quickly
- Supports recommendation systems
- Tracks device and session activity

---

2. LinkedIn – Graph Database (Neo4j)

Why LinkedIn Needs a Graph Database

- **Relationship-Focused Data:**
  LinkedIn is built around connections between people. A graph database stores data as nodes and relationships, which matches LinkedIn's structure naturally.
- **Fast Relationship Queries:**
  Features like "People You May Know" require checking mutual connections. Graph databases do this faster than traditional databases because relationships are stored directly.
- **Complex Network Analysis:**
  LinkedIn often needs to find connection paths (1st, 2nd, 3rd degree connections). Graph databases are designed specifically for this type of analysis.

- **Flexible Data Structure:**
  Users can add new skills, roles, or connections without changing the entire database structure, making it easy to evolve the platform.

Use at LinkedIn

- Mutual connections
- Connection recommendations
- Professional network mapping

---

3. In-Memory Database Example – Real-Time Chat Application

Application

Real-Time Chat Application
(such as WhatsApp, Messenger)

Type of Database

In-Memory Database – Redis

Why an In-Memory Database Is Used

Real-Time Messaging

Chat applications require messages to be delivered instantly.
In-memory databases store data in RAM instead of disk, which makes reading and writing data extremely fast.

Low Latency

Users expect messages to appear immediately after sending.
Since Redis works in memory, it provides very quick response time, avoiding delays.

Reduced Load on Main Database

Important data like chat history can be saved in a main database later, while Redis temporarily holds active messages.
This improves overall performance.

Example Data Stored in Redis (Simple Format)

user:101:status → online
chat:room_5 → ["Hi", "Hello", "How are you?"]

Conclusion

Each database type is designed to solve a specific real-world problem:

- SQL ensures accuracy and consistency
- NoSQL enables scalability and flexibility
- In-Memory databases deliver real-time speed

Choosing the correct database improves performance, reliability, and user experience.
