



Storage Architectures

Types of Storage Architectures

File Storage

- One of the oldest types of storage, where data is stored in files and folders.
- Suitable for small to medium-sized applications, often used for web servers, etc.
- Pros: Simplicity, durability.
- Cons: Limited metadata. Not scalable. Not ideal for complex queries, large datasets, or high concurrency.

Object Stores

- Highly scalable storage option where data is bundled with metadata and a unique identifier.
- Ideal for: Unstructured data like photos, videos, logs.
- Examples: Amazon S3, Google Cloud Storage.
- Pros: Highly scalable, metadata features, API accessible
- Cons: Latency, cost.

Database Storage

- Various databases come with their storage architectures optimized for specific types of queries.
- Examples: MySQL, PostgreSQL for relational data; MongoDB, Cassandra for NoSQL, Redis for key-value caches.
- Importance: Critical for application performance, especially in large, data-intensive applications.

Database Storage in detail

- Understanding database storage options is critical for any software architecture.
- Each storage option has its own unique set of features, pros, and cons.

SQL (Relational) Databases

- **Definition:** SQL databases are based on a schema, use SQL (Structured Query Language) for defining and manipulating the data.
- **Examples:** MySQL, PostgreSQL, OracleDB.
- **Pros**
- ACID compliant (Atomicity, Consistency, Isolation, Durability).
- **Atomicity**
 - All or Nothing: Atomicity ensures that a transaction is treated as a single unit.
 - If any part of the transaction fails, the entire transaction is rolled back to its initial state, ensuring data integrity.
- **Consistency**
 - Rule-keeping: After a transaction, the database should remain in a consistent state.
 - All data should obey predefined rules and constraints like unique keys, foreign keys, and so on.
- **Isolation**
 - Independence: Isolation ensures that transactions occur independently without interference.
 - Other operations can't read or write into the ongoing transaction.
- **Durability**
 - Permanence: Once a transaction has been committed, it will remain in the system, even in the event of system failures.
 - Data is saved in a safe, durable storage layer.
- ACID compliance is especially crucial in systems where data integrity and accuracy are mandatory, like banking systems or medical records.
- Good for complex queries.
- Strongly typed schema.
- **Cons**
- Schema changes are challenging.
- May become slow with very large datasets.
- Can be overkill for simple, non-relational data.
- Not as scaleable as NoSQL databases

NoSQL Databases

- **Definition:** Non-relational databases that can store and process a large amount of unstructured data.
- **Examples:** MongoDB, Cassandra, DocumentDB.
- **Pros**

- Schema-less.
 - Highly scalable horizontally.
 - Good for hierarchical data storage.
- **Cons**
 - Not as mature as SQL databases.
 - Lack of standardization.
 - Typically not ACID compliant.

In-memory Databases

- **Definition:** Data is stored in the main memory (RAM) rather than disk.
- **Examples:** Redis, Memcached.
- **Pros**
 - Extremely fast read and write.
 - Good for caching and session storage.
- **Cons**
 - Data can be volatile.
 - Limited structuring and querying options (key-value store).
 - More expensive.

Vector Databases

- **Definition:** Databases designed to efficiently handle vector data, often used for machine learning and data analytics.
- **Examples:** Weaviate.
- **Pros**
 - Fast querying for similarity.
 - Optimized for large datasets.
 - Often used in machine learning for nearest neighbor searches.
- **Cons**
 - Highly specialized; not for general-purpose storage.
 - Could be overkill for simple use-cases.

Graph Databases (also called Knowledge Graphs)

- **Definition:** Optimized to handle relationships like a graph.
- **Examples:** Neo4j, ArangoDB.
- **Pros**
 - Great for social networks, recommendation systems.
 - Optimized for complex relationships.
- **Cons**
 - Can be slower for operations that aren't graph-oriented.
 - Learning curve for specialized query languages.

Time-series Databases

- **Definition:** Optimized for handling time-series data.
- **Examples:** InfluxDB, TimescaleDB.
- **Pros**
 - Good for IoT, real-time analytics.
 - Optimized for write-heavy workloads.
- **Cons**
 - Not suitable for all types of data.
 - Specialized query language may be needed.

Guidelines for Picking the Right Storage

- Don't need to pick just one: often a combination of multiple types of storage.
- **Leadspotr**
 - SQL database for the main objects (users, quizzes, etc)
 - Object storage for file uploads such as profile pictures
 - Don't use cache at the moment, but you can imagine using cache for lead collection analytics.
- Evaluate your project needs: speed, scalability, data complexity, consistency requirements.
- What type of data are you dealing with, who needs access to it, and how long do you need to store it?
 - Files → depending on who needs access: local file system, object store
 - Short-term/intermediate data such as analytics results → local files or key-value store if you need search and high performance
 - Long-term data → database with backup
- My advice specific to databases
 - SQL is a great starting point if your app works with data that you know the shape of and you need limited scalability.
 - NoSQL is great for more open-ended applications that might need more scale in the future.
 - Don't add caching unless you really need it (it's expensive!). And make sure cache is really a cache: your system should be able to function if you completely clear the cache (cache should only contain data that you can regenerate)
- High performance usually comes at a higher cost.
- Make sure you think about the future as well. Scalability, adaptability for unforeseen needs.
- Storage architecture choice is linked with architectural patterns as well
 - Microservices: separate storage architectures per microservice
 - Pipeline: key-value store for storing intermediate/cached results

Real-world Examples

- Let's talk about a few real-world applications to see how they leverage various types of database storage architectures to meet their unique needs.
- As you'll see many applications use a hybrid approach, combining different storage architectures for various purposes.
- **E-commerce Platforms (e.g., Amazon)**
 - **SQL Databases:** Used for structured data like inventory management, customer data, and order tracking.
 - **NoSQL Databases:** Used for handling customer reviews, product recommendations, and other types of unstructured data.
 - **In-memory Databases:** Often used as a caching layer to speed up data retrieval.
 - **Object store:** product media (images, videos)
- **Social Media Platforms (e.g., Twitter)**
 - **NoSQL Databases:** Ideal for unstructured data like tweets, likes, and follows.
 - **SQL Databases:** Used for structured data, such as user credentials and payment information.
 - **Time-series Databases:** For logging and analyzing user interaction metrics over time.
- **Search Engines and Recommendation Systems**
 - **Vector Databases:** Employed to quickly find similar items among large datasets.
 - **NoSQL Databases:** For storing metadata associated with indexed items.
 - **SQL Databases:** For storing user profiles and settings.
- **High-Frequency Trading Platforms**
 - **In-memory Databases:** Required for extremely quick data retrieval and storage.
 - **SQL Databases:** For storing historical trading data and user accounts.
 - **File storage:** for logging trading server activity
- **Professional Networking Platforms (e.g., LinkedIn)**
 - **Graph Databases:** Used to model and analyze connections between people.
 - **SQL Databases:** For storing user profiles and credentials.
 - **Object stores:** uploaded profile photos and images

Summary and Takeaways

- Understanding the landscape of storage architectures helps make informed decisions.
- Consider factors like requirements, cost, and compatibility when selecting storage for your project.
- Consider a hybrid approach when building complex applications, as it's often necessary to use more than one type of database to efficiently meet varied data storage and retrieval needs.