# Atomicity Consistency Isolation Durability

Set of Expectations/principles

* to ensure any database transactions are processed in a reliable way, resulting in correctness.

## Atomicity

Atomicity is all based around this idea of togetherness. When carrying out any kind of database transaction, it often consists of multiple operations. With atomicity, either every operation succeeds or none of them do. This is important because the operations can have an impact on each other, so one failing can lead to unexpected results.

Think of a financial transaction, for example. You are paying a friend $250 for a holiday you are going on. The whole transaction would consist of the money leaving your account and arriving in the recipient’s account. If there was no atomicity, it is possible that money leaves your account but doesn’t arrive at the other end, resulting in you being debited the money but still owing the recipient.

## Consistency

Consistency is about ensuring that changes made as part of a transaction are consistent with any database constraints. If the data at any stage goes against these constraints, the whole transaction will fail.

Unless you have an agreed overdraft, banks, for example, will expect your balance to be positive. So if you tried to withdraw more money than you have available, this would break a constraint and fail, rolling back all operations in that transaction.

## Isolation

Isolation is there to make sure that all transactions are run in an isolated environment without interfering with each other.

Sticking with the financial example, imagine you have a bank balance of $200 and you try to withdraw $100 at an ATM. At the same time, a standing order you have set up comes out for $100. With isolation, these transactions can occur concurrently, ensuring that your ending balance is $0, not $100, because the transactions impacted each other.

## Durability

Durability is another important element of ACID because it ensures that no matter what happens, once a transaction is complete, the changes in that transaction are written to the database. This makes sure that data changes are persisted, even in the event of a power failure or system crash.

### Can NoSQL databases be ACID-compliant? No but Mongo DB is.

* ACID has been something many relational databases have offered for a while. [MongoDB is an ACID-compliant database](https://www.mongodb.com/basics/acid-transactions).
* MongoDB, cannot be BASE-compliant because transactions are consistent and not eventually consistent, as per the rules of BASE.

[Which Database Is Right for Your Use Case? | Integrate.io](https://www.integrate.io/blog/which-database/#:~:text=Here%20are%20our%20top%20takeaways%20about%20choosing%20the,you%20choose%20the%20right%20database%20option.%20More%20items)

# BAbasicallyAvailable SoftState EventuallyConsistent

With BASE, availability is prioritized over consistency

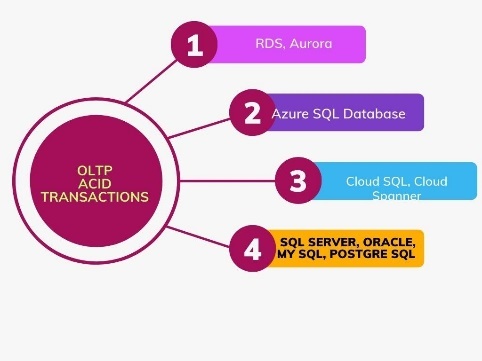
# When to choose which Database

Understanding the application’s data

## 1. Structured data(Tabular data)

Structured data refers to organized data presented in a **tabular format** with predefined schemas, where individual columns represent specific attributes, and each row contains corresponding values.

**Recommended Database Type**: Relational Database Management System (RDBMS) (Highly stable, Reliable but hard to scale)



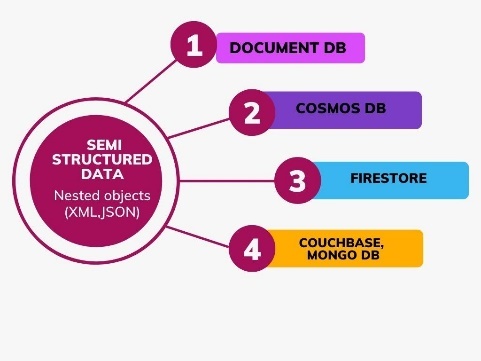
A diagram of a data center

AI-generated content may be incorrect.

## 2. Semi-Structured Data:

semi-structured data lacks a strict adherence to a fixed tabular schema but possesses some degree of structure. It may contain nested fields, arrays, or key-value pairs. When dealing with semi-structured data, **NoSQL databases that offer flexible schema and document-based storage prove to be well-suited.**

**Recommended Database Type**: Document Store or Wide-Column Store

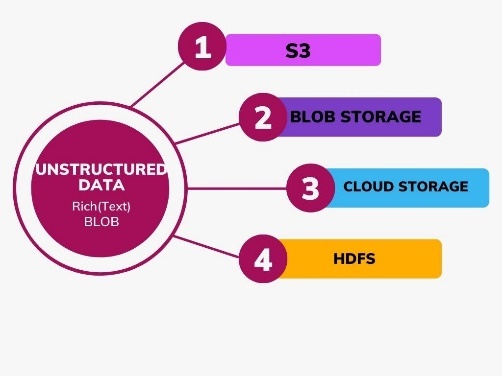


## 3. Unstructured Data:

Unstructured data lacks a predefined data model and may include multimedia files, text documents, emails, etc. Storing and retrieving unstructured data efficiently requires specialized databases designed for object storage or file systems.

**Recommended Database Type**: Object Storage or File Systems

Examples: Amazon S3, Google Cloud Storage, Hadoop Distributed File System (HDFS)



A diagram of data being used

AI-generated content may be incorrect.

## 4. Time-Series Data:

Time-series data is organized based on timestamps and is frequently generated by sensors, logs, or IoT devices. The data is typically appended with new entries and queried based on time intervals. Databases optimized for handling time-series data offer excellent performance.

**Recommended Database Type**: Time-Series Database

A diagram of data processing

AI-generated content may be incorrect.

A diagram of data analysis

AI-generated content may be incorrect.

## 5. Graph Data:

Graph data represent relationships between entities, making it valuable for social networks, recommendation systems, and data dependency analysis. Graph databases excel at traversing complex relationships efficiently.

**Recommended Database Type**: Graph Database

Examples: Neo4j, Amazon Neptune, ArangoDB

A diagram of data

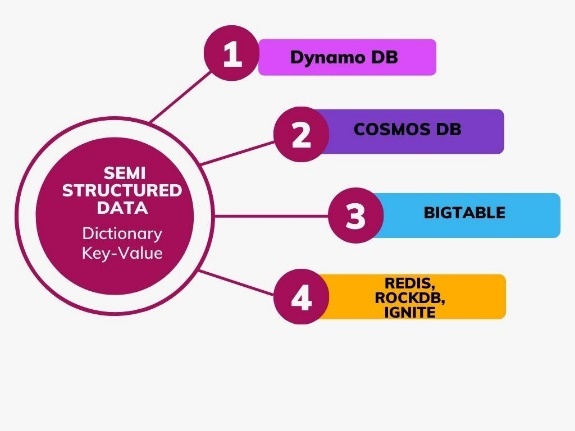
AI-generated content may be incorrect.

**6. Key-Value Data:**

Key-value data stores data as simple key-value pairs, suitable for caching, session management, and high-performance data access. They are known for their lightning-fast retrieval of data by key.

**Recommended Database Type**: Key-Value Store

Examples: Redis, Amazon DynamoDB, Riak



A diagram of a computer hardware company

AI-generated content may be incorrect.

A diagram of a data system

AI-generated content may be incorrect.

## 7. Geospatial Data:

Geospatial data represents geographic information, such as coordinates and polygons. Spatial databases enable efficient handling of geospatial data and support location-based queries.

**Recommended Database Type**: Spatial Database

Examples: PostGIS (extension for PostgreSQL), MongoDB (with geospatial indexes)

A diagram of data

AI-generated content may be incorrect.

## 8. Hybrid Data:

In some cases, your application may require the use of multiple database types to handle various data types effectively. This approach is known as polyglot persistence, where different databases are used for different parts of the application based on the data characteristics.

# Evaluating Application Requirements

Once you understand your data type, thoroughly evaluate your application’s requirements. Consider the following factors:

1. **Scalability:** Assess the scalability demands of your application. Will your database need to handle a growing amount of data and an increased workload? Evaluate the potential for both vertical(RDBMS) and horizontal scalability(NoSQL).

If your data strategy rests on vertical scaling, a relational database is fine. Vertical scaling adds more compute power to a server instead of adding more servers to the system. It’s ideal when there are a limited number of users and not a lot of querying involved. In that sense, vertical scaling might be suitable for business-facing startups. The basic advantages of vertical scaling are speed and simplicity.

On the other hand, if you are expecting higher loads of users or querying, horizontal scaling is a much cheaper solution. NoSQL databases employ horizontal scaling. Instead of adding more compute power to a server, they distribute the load across servers. Horizontal scaling and, in turn, NoSQL databases, give businesses more elasticity. However, running joint operations is difficult on these systems.

1. **Performance**: Analyze the read and write performance of different databases for your application’s expected workload. Look for benchmarks and real-world performance data.
2. **Data Integrity and Consistency**: Determine the level of data consistency required by your application. Some databases offer strong consistency with ACID transactions, while others provide eventual consistency.
3. **Query Complexity**: Consider the types of queries your application will frequently perform. Different databases have varying strengths in handling specific query types.
4. **Community and Support**: Gauge the size of the database’s community and the availability of documentation, tutorials, and support channels.
5. **Security and Compliance**: Prioritize data security and compliance with relevant data privacy regulations if handling sensitive data.
6. **Cost Considerations**: Factor in licensing fees, hosting expenses, and operational costs when assessing the overall database expense.
7. **Future Flexibility**: Evaluate the chosen database’s flexibility in accommodating potential changes in application requirements.
8. **Cloud-Native Considerations:** If building a cloud-native application, explore managed database services provided by cloud providers for streamlined management.
9. **Atomicity:** If atomicity is a top priority for you, stick to a relational database. Atomicity in database management promotes consistency. It rests on the principle of atomic transactions. These are a series of operations that are considered composite operations. In other words, either all operations in an atomic transaction occur or none at all.

The simplest example of an atomic transaction is transferring money from account A to account B. The balance from account A should be deducted and the requisite number should be added to account B. Both operations need to be successful for the transaction to be successful.

1. **Speed:** If speed is more important than ACID compliance, a non-relational database is a better bet. In the case of real-time data, such as sensor data, some compromise in data integrity can be tolerated in favor of speed. In a non-relational database, each record is an independent entity. Thus, it is possible to run multiple queries simultaneously irrespective of the size of the database.

# Considerations for Cloud-Native Applications

For cloud-native applications, additional considerations come into play:

1. **Managed Services**: Cloud providers offer managed database services, which can significantly reduce operational overhead and simplify database management.
2. **Serverless Databases**: Consider serverless database options, where you only pay for actual usage, making it cost-effective for applications with varying workloads.
3. **Vendor Lock-in**: Be aware of potential vendor lock-in when using cloud provider-specific databases. Ensure you can migrate your data easily if needed.

# Prototype and Testing

Before committing fully to a specific database, consider creating a prototype or conducting small-scale tests. This allows you to evaluate the database’s performance, compatibility with your application, and ease of development.