Acid Transactions Defined

ACID transactions refer to a set of properties that are designed to ensure the reliability and consistency of database transactions. The term “ACID” stands for Atomicity, Consistency, Isolation, and Durability, which are the four key properties of an ACID transaction. Essentially, ACID transactions guarantee that database operations are executed correctly, and if there is any kind of failure, the database can recover to a previous state without losing any data or impacting the consistency of the data. In other words, ACID transactions provide a high level of assurance that database transactions will be processed reliably and that data will be stored accurately and consistently.

Understanding acid transactions

An ACID transaction is a set of properties that ensure the reliability and consistency of a database transaction. Transactions are a series of operations that are completed as a single unit of work, using read and write operations to access data. Most databases provide transactional guarantees for operations that impact just one record. This section will explain the basic definitions of the characteristics involved in an ACID transaction.

Atomicity

Atomicity in ACID transactions guarantees that a transaction is treated as a single, indivisible unit of work. If any part of the transaction fails, the entire transaction must be rolled back, meaning that any changes made during the transaction are undone. This ensures that the database remains in a consistent state, regardless of any failures that may occur during the transaction.

Consistency

Consistency ensures that the database remains in a valid state before and after the transaction. In other words, the database schema must satisfy all constraints and rules, and any transaction that violates these constraints must be rolled back to maintain the consistency of the database. This ensures that the database maintains its integrity and the data remains accurate and reliable.

Isolation

This property ensures that each transaction operates independently of other transactions, which means that a transaction's effects should only become visible to other transactions after it has been committed. This property prevents interference and conflicts between concurrent transactions, and helps maintain the integrity and consistency of the database. It's important to note that different levels of isolation can be configured for transactions, depending on the specific requirements of the application and the database system being used.

Durability

This characteristic makes sure that, even in a system failure, the changes made to the database during a transaction are irreversible. Any changes made after a transaction is committed must persist, even if the system is destroyed or loses power.

How do acid transactions work?

ACID transactions maintain data integrity by adhering to a set of steps. The steps described are a common way that databases implement ACID transactions, but there could be variations or differences in implementation depending on the specific database system being used.

1. **Begin Transaction**: A **BEGIN TRANSACTION** statement declaration initiates a transaction and establishes a savepoint from which the transaction can be rolled back if necessary.
2. **Execute operations**: All operations within the transaction are executed one at a time. The database validates each operation to ensure it complies with the constraints and schema.

3. **Commit or Rollback**: Following the successful completion of all operations, the transaction is committed using the **COMMIT** statement. The transaction is rolled back to the savepoint established at the beginning of the transaction if any operation fails.

Example of an acid transaction in action

Consider a banking app where a user wishes to transfer funds from one account to another, where the operation’s transaction might look like this:

- **BEGIN TRANSACTION** – An example of withdrawing money from the bank using a cheque, pay order, or through an ATM.
- Deduct the transfer amount from the source account.
- Add the transfer amount to the destination account.
- **COMMIT** – Updating the record of the transaction carried out by the customer.

The transaction is rolled back, and the database is restored to its initial state if any of its operations fail, such as if the source account does not have enough funds.

Acid transactions use cases

ACID transactions are suitable for many use cases. Following are some examples:

Banking

Banks use ACID transactions to ensure that payments and other financial transactions are handled accurately and securely. For example, when a customer withdraws money from an ATM, an ACID transaction is executed to update their account balance and record the transaction. The transaction is atomic, meaning it succeeds or fails, and the account balance remains constant.

For an outline of the right Redis tools to handle transactions, and learn how Redis Enterprise can help you scale your customer experience check out our whitepaper Ensure Customer Satisfaction with Redis

Healthcare systems

Healthcare systems use ACID transactions to help guarantee that patient records are updated accurately and that private medical data is protected. Electronic health records (EHRs) contain personal information about patients that must be accurate and consistent. For instance, an ACID transaction occurs when a doctor updates a patient’s medication in the EHR to ensure the data is updated atomically, consistently, and durably.

E-commerce applications

E-commerce applications use ACID transactions to make sure that customer orders are processed correctly, and that inventory levels are updated correctly. For example, an ACID transaction is carried out when a customer purchases an item to update the inventory records and guarantee that the transaction is atomic, consistent, isolated, and durable.

Advantages and disadvantages of using acid transactions

Let’s compare the benefits and drawbacks of ACID transactions and identify their potential. It’s worth noting that the disadvantages listed are not always inherent drawbacks of ACID transactions. They can vary depending on the specific implementation and the requirements of the application. For example, while overhead and scalability might be concerns in some scenarios, they may not be significant issues in others. Similarly, while deadlocks can be a problem in some cases, proper design and management of transactions can often mitigate or prevent them.

Advantages	**Disadvantages**
Data integrity– Even if a transaction fails, ACID transactions guarantee that the database will remain in a consistent state. It contributes to data reliability and integrity.	**Overhead**– The performance of databases is affected by the extra processing overhead required by ACID transactions.
Consistency– The validity of the database is ensured both before and after an ACID transaction. It contributes to database consistency.	**Deadlocks**– Multiple transactions waiting for each other to release resources can cause deadlocks. Deadlocks can be difficult to resolve and have an impact on database reading and retrieval performance.
Isolation– ACID transactions ensure that each transaction is independent of the others. It also contributes to preserving data integrity by preventing interference between concurrent transactions.	**Scalability**– ACID transactions can be difficult to implement in large-scale distributed systems that require performance and scalability.
Durability– ACID transactions ensure that database changes made during a transaction are irreversible, even in a system failure. It contributes to data reliability.	**Similar data update-** When several transactions are running concurrently, they might clash if they attempt to modify the same data at the same time. Thus,one transaction might need to wait for another transaction to finish before it can move forward, which would reduce system performance and increase latency.

Alternatives to acid transactions

ACID transactions provide several benefits for ensuring data reliability, consistency, isolation, and durability. However, they may not be the best fit for all applications. In such cases, there are various alternative transaction models and theorems available that can be considered instead of ACID. These include:

BASE (Basically Available, Soft state, Eventually consistent)

BASE isn't necessarily a replacement for ACID transactions, but rather an alternative model for handling distributed systems that cannot guarantee immediate consistency. BASE emphasizes availability and partition tolerance more than consistency. This model trades off near-term consistency for long-term stability. Although it assumes that all data will eventually become consistent, it cannot guarantee this. This approach is appropriate for high-volume distributed systems because it provides greater scalability and availability. It is often used in conjunction with NoSQL databases, which prioritize scalability and availability over strict consistency requirements.

CAP (Consistency, Availability, Partition tolerance)

The CAP theorem states that in a distributed system, it is impossible to guarantee all three of Consistency, Availability, and Partition tolerance. However, it does not suggest sacrificing consistency for availability and partition tolerance. In fact, the theorem poses a trade-off between consistency and partition tolerance. This means that in the event of a network partition, you must choose between consistency and partition tolerance. The CAP theorem is not truly an alternative transaction model but a theoretical framework for understanding the limitations of distributed systems. Transaction models, such as BASE, can be used in conjunction with the principles of CAP to design and implement distributed systems.

NoSQL databases

NoSQL databases do not impose rigid consistency standards and prioritize performance and scalability over immediate consistency. They are often used in applications that require high throughput, and where data consistency is not critical. While relational databases ensure desirable ACID properties, NoSQL databases are more effective at handling large and complex data sets. Additionally, BASE properties can perform even better for a wide range of applications, although ACID is not always guaranteed in this case.

Acid transactions in distributed systems

Distributed systems comprise numerous computers that interact with each other to deliver a single service. ACID transactions can be difficult to implement in a distributed system because they are made up of multiple nodes that are geographically dispersed and communicate over a network.

Challenges of implementing acid transactions in distributed systems

Distributed systems consist of multiple computers that collaborate to provide a single service. ACID transactions can be challenging to implement in distributed systems because they comprise multiple nodes that are geographically dispersed and communicate over a network.

Challenges of implementing ACID transactions in distributed systems include:

Network latency: In distributed systems, network latency can impact the performance of ACID transactions. Longer transaction times and higher overhead can result from network communication delays.

Consistency: Maintaining consistency across all nodes in a distributed system can be challenging. A distributed system’s nodes might store different versions of the same data, which could result in discrepancies.

Availability: Keeping a distributed system available can be difficult. Nodes could fail, and it might be challenging to maintain the system’s responsiveness.

Scalability: As the number of nodes in a distributed system increases, it becomes more difficult to maintain consistency and availability.

Solutions for maintaining acid properties in distributed systems

ACID transactions are difficult to maintain in distributed systems due to factors such as network latency, consistency, availability, and scalability. To address these challenges, several solutions have been developed, including:

•**Two-phase commit:** This protocol ensures that all nodes in a distributed system agree to commit a transaction before it is committed, ensuring data consistency and agreement on the transaction’s outcome.

•**Multi-Version Concurrency Control (MVCC):** This technique manages data concurrency in a distributed system by allowing each transaction to access the appropriate data version, enabling multiple versions of the same data to coexist.

•**Replication:** In a distributed system, replication involves keeping multiple copies of the same data on various nodes, reducing network latency and increasing availability.

•**Sharding:** This process involves dividing data across multiple nodes in a distributed system, improving performance and scalability but increasing the complexity of maintaining data consistency.

ACID transactions are a fundamental concept in database management, providing benefits such as data integrity, consistency, isolation, and durability. They ensure that the database remains in a consistent state even if the server fails. However, they do have limitations and challenges, such as overhead, deadlocks, and scalability problems in distributed systems.

Although ACID transactions provide strong consistency and reliability, they may not always be the best option for every use case. Organizations must carefully evaluate their unique needs and requirements to determine whether ACID transactions or alternative transaction models, such as BASE or CAP, are a better fit for their systems.