What are ACID Properties in Database Management Systems?

ACID transactions guarantee that a database will be in a consistent state after running a group of operations. Most databases offer transactional guarantees for operations that impact a single record. However, not all databases support transactions that work across multiple records, which can be concerning to developers who are accustomed to using them.

MongoDB's data model allows related data to be stored together in a single document. We estimate that 80%-90% of applications that model their data in a way that leverages the document model will not require multi-document transactions. However, MongoDB supports multi-document ACID transactions for the use cases that require them. Developers appreciate the flexibility of being able to model their data in a way that does not typically require multi-document transactions but having multi-document transaction capabilities available in the event they do.

In this article, we'll explore what an ACID transaction is, how to implement a transaction in MongoDB, and why and when to use one.

What are ACID transactions?

A transaction is a group of database read and write operations that only succeeds if all the operations within succeed. Transactions can impact a single record or multiple records.

Let's walk through an example transaction that impacts multiple records. Imagine you were building a function to transfer money from one account to another where each account is its own record. If you successfully take money from the source account, but never credit it to the destination, you have a serious accounting problem. You'd have just as big a problem (if not bigger) if you instead credited the destination, but never took money out of the source to cover it. These two write operations have to either both happen, or both not happen, to keep the system consistent.

```
await session.withTransaction(async () => {
   const subtractMoneyResults = await accountsCollection.updateOne(
       { _id: account1 },
       { $inc: { balance: amount * -1 } },
       { session });
   if (subtractMoneyResults.modifiedCount !== 1) {
       await session.abortTransaction();
       return;
   }

   const addMoneyResults = await accountsCollection.updateOne(
       { _id: account2 },
       { $inc: { balance: amount } },
       { session });
   if (addMoneyResults.modifiedCount !== 1) {
       await session.abortTransaction();
       return;
   }
});
```

In this transactions code sample, money is being moved from one account to another. Updates to both accounts must succeed or the transaction will be aborted. Visit the Node.js Quick Start GitHub repository to get a copy of the full code sample and run it yourself.

This is the kind of problem that multi-record transactions were invented to solve: We tell the database we're doing a transaction, and it keeps track of every update that it makes along the way. If the connection breaks before the transaction is complete, or if any command in the transaction fails, then the database rolls back (undoes) all of the changes it had written in the course of the transaction.

The downside of using a transaction is that the database has to "lock" the involved resources to prevent concurrent writes from interfering with one another. That means other

clients trying to write data might be stuck waiting for the transaction to complete, affecting application latency and, ultimately, user experience.

What are the ACID properties of a transaction?

ACID is an acronym that stands for atomicity, consistency, isolation, and durability.

Together, these ACID properties ensure that a set of database operations (grouped together in a transaction) leave the database in a valid state even in the event of unexpected errors.

Atomicity

Atomicity guarantees that all of the commands that make up a transaction are treated as a single unit and either succeed or fail together. This is important as in the case of an unwanted event, like a crash or power outage, we can be sure of the state of the database. The transaction would have either completed successfully or been rollbacked if any part of the transaction failed.

If we continue with the above example, money is deducted from the source and if any anomaly occurs, the changes are discarded and the transaction fails.

Consistency

Consistency guarantees that changes made within a transaction are consistent with database constraints. This includes all rules, constraints, and triggers. If the data gets into an illegal state, the whole transaction fails.

Going back to the money transfer example, let's say there is a constraint that the balance should be a positive integer. If we try to overdraw money, then the balance won't meet the constraint. Because of that, the consistency of the ACID transaction will be violated and the transaction will fail.

Isolation

Isolation ensures that all transactions run in an isolated environment. That enables running transactions concurrently because transactions don't interfere with each other.

For example, let's say that our account balance is $200. Two transactions for a $100 withdrawal start at the same time. The transactions run in isolation which guarantees that when they both complete, we'll have a balance of $0 instead of $100.

Durability

Durability guarantees that once the transaction completes and changes are written to the database, they are persisted. This ensures that data within the system will persist even in the case of system failures like crashes or power outages.

The ACID characteristics of transactions are what allow developers to perform complex, coordinated updates and sleep well at night knowing that their data is consistent and safely stored.

In MongoDB, single-document updates have always been atomic. The document model lends itself to storing related data that is accessed together in a single document (compared to the relational model, where related data may be normalized and split between multiple tables). In most cases, a well-designed document schema allows you to work without the need for multi-document transactions.

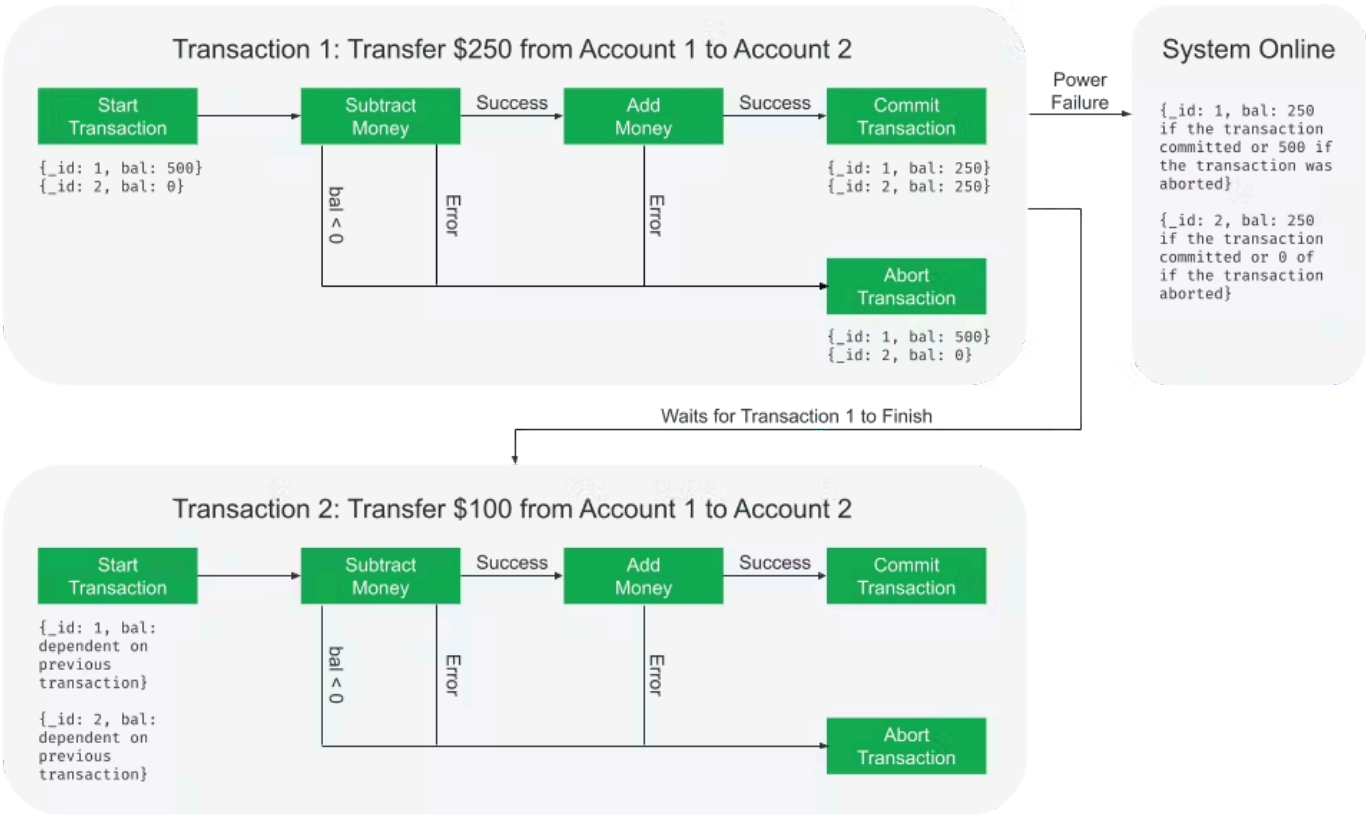
When you do need multi-document ACID compliance, MongoDB transactions work across your cluster and operate the way you'd expect. There is performance overhead to using transactions in a distributed system, so you'll want to be mindful of your resource constraints and performance goals.

What is an ACID transaction example?

Let's continue with the banking example we discussed earlier in the article where money is being transferred from one account to another. Let's examine each of the ACID properties in this example:

- **Atomicity**: Money needs to both be removed from one account and added to the other, or the transaction will be aborted. Removing money from one account without adding it to another would leave the data in an inconsistent state.
- **Consistency**: Consider a database constraint that an account balance cannot drop below zero dollars. All updates to an account balance inside of a transaction must leave the account with a valid, non-negative balance, or the transaction should be aborted.

- **Isolation:** Consider two concurrent requests to transfer money from the same bank account. The final result of running the transfer requests concurrently should be the same as running the transfer requests sequentially.
- **Durability:** Consider a power failure immediately after a database has confirmed that money has been transferred from one bank account to another. The database should still hold the updated information even though there was an unexpected failure.



The diagram demonstrates how the ACID properties impact the flow of transferring money from one bank account to another.

Why are ACID transactions important?

ACID transactions ensure data remains consistent in a database. In data models where related data is split between multiple records or documents, multi-record or multi-document ACID transactions can be critical to an application's success.

MongoDB's document model allows developers to store related data together in a single document using arrays and embedded objects. We estimate that 80%-90% of applications don't need to use multi-document transactions when they leverage the document model to store related data together. MongoDB supports multi-document transactions for use cases where related data is not contained in a single document.

How do ACID transactions work in MongoDB?

MongoDB added support for multi-document ACID transactions in version 4.0 in 2018 and extended that support for distributed multi-document ACID transactions in version 4.2 in 2019.

MongoDB's document model allows related data to be stored together in a single document. The document model, combined with atomic document updates, obviates the need for transactions in a majority of use cases. Nonetheless, there are cases where true multi-document, multi-collection MongoDB transactions are the best choice.

MongoDB transactions work similarly to transactions in other databases. To use a transaction, start a MongoDB session through a driver. Then, use that session to execute your group of database operations. You can run any of the CRUD (create, read, update, and delete) operations across multiple documents, multiple collections, and multiple shards.

For specific code samples of how to implement transactions, view the Quick Starts on the MongoDB Developer Hub:

- Multi-Document ACID Transactions in MongoDB with Golang
- Multi-Document ACID Transactions in MongoDB with Java
- Multi-Document ACID Transactions in MongoDB with Node.js
- Multi-Document ACID Transactions in MongoDB with Python

Visit the MongoDB Drivers documentation for language-specific guides for each of the languages officially supported by MongoDB. See the MongoDB documentation for a list of transactions best practices and production considerations.

When should I use MongoDB multi-document transactions?

As we mentioned earlier, we estimate that 80%-90% of applications that leverage the document model will not need to utilize transactions in MongoDB.

For the other 10%-20% of applications, multi-document transactions are available.

Applications that require transactions typically have use cases where values are exchanged between different parties. These are typically "System of Record" or "Line of Business" applications.

Example applications that may benefit from multi-document transactions include:

- Systems that move funds, like banking applications, payment processing systems, and trading platforms.
- Supply chain and booking systems where ownership of goods and services is transferred from one party to another.
- Billing systems that store information in detailed records as well as summary records.

What are the best practices for transactions in MongoDB?

In general, we recommend modeling data in a way that data that is accessed together is stored together. When data is modeled this way, you will have better performance and transactions will not be required.

For applications that do require transactions, we recommend the following best practices:

- Break long-running transactions into smaller pieces so they don't exceed the default 60-second timeout. (Note that this timeout can be extended.) Ensure the operations in your transaction are using indexes so they run quickly.
- Limit each transaction to 1,000 document modifications.
- Ensure the appropriate read and write concerns are configured (note that beginning in version 5.0, MongoDB defaults to the necessary majority write concern).
- Add appropriate error handling and retry transactions that fail due to transient errors.
- Be aware that transactions that affect multiple shards will incur a performance cost.

For more information on these best practices, check out the Multi-Document ACID Transactions on MongoDB white paper as well as the MongoDB documentation on transactions.

Summary

ACID transactions provide developers with the comfort of knowing that their database is in a consistent state after running a set of operations. Relational databases rely on transactions because of the way related data is split between multiple tables. Other databases like MongoDB rarely need transactions but provide the capability in the event transactions are required.

MongoDB Atlas is MongoDB's fully managed Database-as-a-Service and is the easiest way to begin using MongoDB. Get started with transactions by creating a free MongoDB Atlas cluster.

FAQ

What are ACID transactions?

Transactions are groups of database read and write operations that need to all succeed or all fail together. ACID transactions follow the ACID properties: atomicity, consistency, isolation, and durability.

What are the ACID properties of a transaction?

The ACID properties of a transaction are atomicity, consistency, isolation, and durability.

ACID vs BASE

The ACID (Atomicity, Consistency, Isolation, Durability) consistency model describes database transactions that always leave the database in a consistent, valid state. The BASE consistency model (Basically Available, Soft state, Eventually consistent) describes a database system that is eventually consistent.

While relational databases tend to follow the ACID consistency model and NoSQL databases tend to follow the BASE consistency model, this is not always true. Some NoSQL databases, like MongoDB, support strong consistency and multi-document ACID transactions.

English

About

Careers

Investor Relations

Legal Notices

Privacy Notices

Security Information

Trust Center

Support

Contact Us

Customer Portal

Atlas Status

Customer Support

Manage Cookies

Social

GitHub

Stack Overflow

LinkedIn

YouTube

X

Twitch

Facebook