1. Basic Definition

A transaction is a single logical unit of work comprising multiple database operations (e.g., `INSERT`, `UPDATE`, `DELETE`) that must either fully succeed or have no effect at all. Example: Transferring $50 between bank accounts requires two operations (debit one account, credit another) to be treated as one transaction[1][4].

2. ACID Properties

Transactions follow four critical guarantees:

Property	Description	Example
Atomicity	All operations in the transaction complete or *none do*.	If a power outage occurs mid-transfer, the partial changes are rolled back[5].
Consistency	The database remains valid (follows constraints) before/after the transaction.	A transaction cannot leave an account balance negative if a `CHECK(balance >= 0)` constraint exists[2][8].
Isolation	Concurrent transactions don't interfere with each other.	Two users updating the same product inventory won't corrupt the data[4][7].
Durability	Committed changes survive system failures.	After a "Payment Successful" message, the transaction is saved to disk[4][9].

3. Transaction Lifecycle

1. Begin: Mark the start of a transaction (`BEGIN TRANSACTION` in SQL).

2. Operations: Execute SQL commands (reads/writes).
3. Commit: Permanently save changes (`COMMIT`)[67].
4. Rollback: Undo changes if an error occurs (`ROLLBACK`)[39].

Example in SQL:

```sql
BEGIN TRANSACTION;
UPDATE accounts SET balance = balance - 50 WHERE id = 123;  -- Debit
UPDATE accounts SET balance = balance + 50 WHERE id = 456;  -- Credit
COMMIT;  -- Save changes (or ROLLBACK if an error occurs)
```

4. Why Transactions Matter

Scenario	Without Transactions	With Transactions
System crash mid-operation	Partial updates corrupt data	All changes rolled back automatically
Concurrent writes	Data races cause incorrect results	Isolation ensures sequential consistency
Complex business logic	Manual error handling required	Atomicity guarantees all-or-nothing

Real-world impact:

- Transactions prevent overselling inventory during high-traffic sales[1].
- They ensure financial systems never "lose" money during transfers[4].

5. Key Implementation Details

- Logging: Databases use write-ahead logs (WAL) to track changes for recovery[3][6].
- Locking: Transactions acquire locks to enforce isolation (e.g., row-level locks)[4][7].
- Savepoints: Allow partial rollbacks within large transactions (`SAVEPOINT` in SQL)[9].

By mastering these concepts, you can explain how transactions ensure reliability and consistency in relational databases.

What is a database?

Databases are structured sets of data that are stored within computers. Oftentimes, databases are stored on entire server farms filled with computers that were made specifically for the purpose of handling that data and the processes necessary for making use of it.

Modern databases are such complex systems that management systems have been designed to handle them. These database management systems (DBMS) seek to optimize and manage the storage and retrieval of data within databases.

The ACID approach is one of the guiding stars leading organizations to successful database management.

What are ACID transactions?

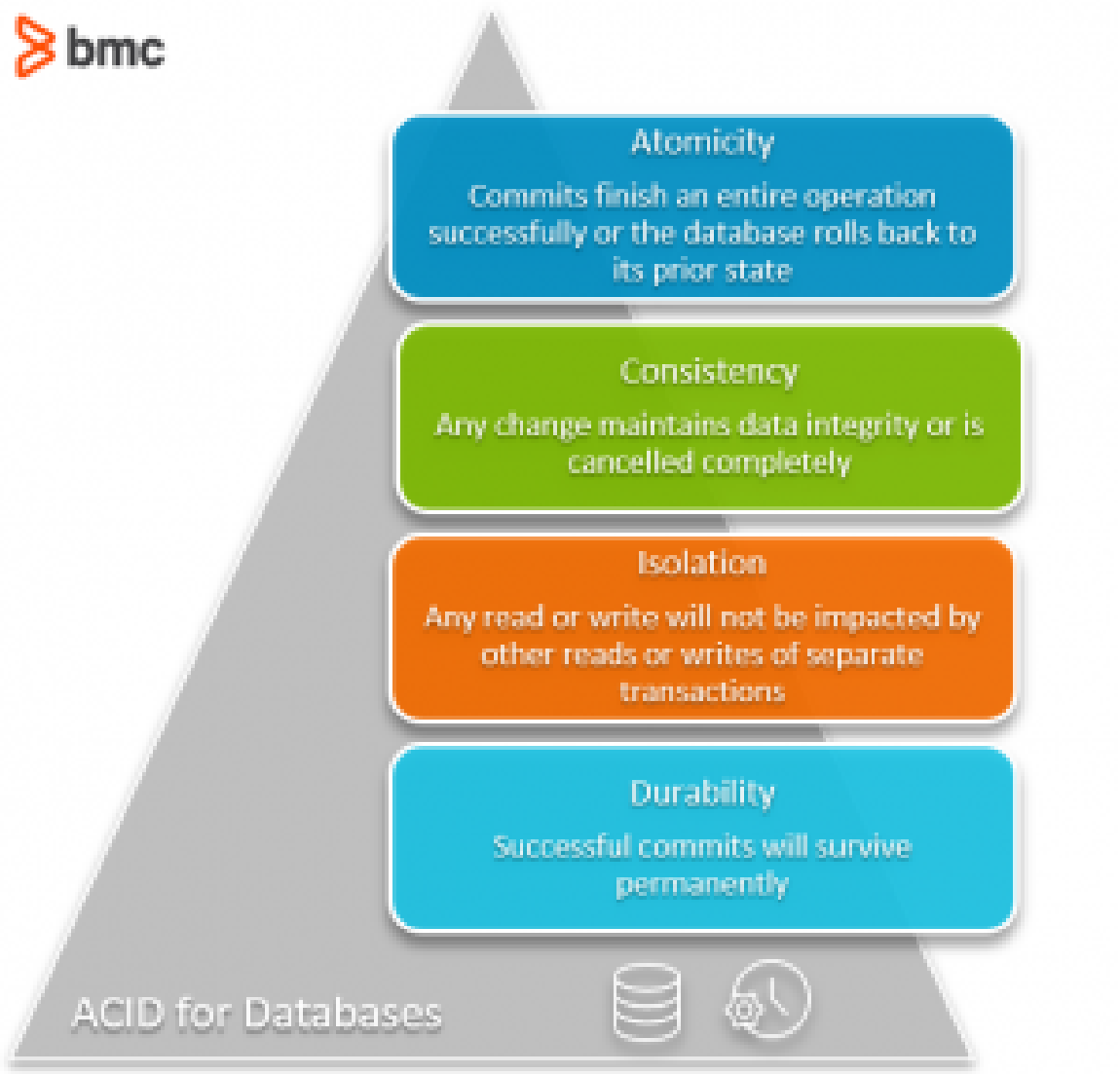
In the context of computer science and databases, ACID stands for:

Atomicity
Consistency
Isolation
Durability

ACID is a set of guiding principles that ensure database transactions are processed reliably. A database transaction is any operation performed within a database, such as creating a new record or updating data within one.

ACID transactions are operations made within a database that need to be performed with care to ensure the data doesn’t become corrupted. Applying the ACID properties to each database modification is the best way to maintain its accuracy and reliability.

Let’s look at each component to understand the full meaning of ACID in database management.



Atomicity

In the context of ACID properties of a database, atomicity means that you either:

Commit to the entirety of the transaction occurring
Have no transaction at all

Essentially, an atomic transaction ensures that any commit you make finishes the entire operation successfully. In case of a lost connection in the middle of an operation, the database is rolled back to its state prior to the commit being initiated.

This is important for preventing crashes or outages from creating cases where the transaction was partially finished to an unknown overall state. If a crash occurs during a transaction with no atomicity, you can't know exactly how far along the process was before the transaction was interrupted. Using atomic transactions principle, you ensure that either the entire transaction is successfully completed—or that none of it was.

Consistency

In ACID database management, consistency refers to maintaining data integrity constraints.

A consistent transaction will not violate integrity constraints placed on the data by the database rules. Enforcing consistency ensures that if a database enters into an illegal state (if a violation of data integrity constraints occurs) the process will be aborted and changes rolled back to their previous legal state.

Another way of ensuring consistency within a database throughout each transaction is by also enforcing declarative constraints placed on the database.

An example of a declarative constraint might be that all customer accounts must have a positive balance. If a transaction would bring a customer account into a negative balance, that transaction would be rolled back. This ensures changes are successful at maintaining data integrity or they are canceled completely.

Isolation

Isolated transactions are considered to be “serializable”, meaning each ACID transaction happens in a distinct order without any transactions occurring in tandem.

Any reads or writes performed on the database will not be impacted by other reads and writes of separate transactions occurring on the same database. A global order is created, with each transaction queueing up in line to ensure the transactions complete in their entirety before another one begins.

Importantly, this doesn’t mean two operations can’t happen at the same time. Multiple transactions can occur as long as those transactions have no possibility of impacting the other transactions occurring at the same time.

Doing this can have impacts on the speed of transactions as it may force many operations to wait before they can initiate. However, this tradeoff is worth the added data security provided by isolation.

In an ACID database, isolation can be accomplished through the use of a sliding scale of permissiveness that goes between what are called optimistic transactions and pessimistic transactions:

> **An optimistic transaction schema assumes that other transactions will complete without reading or writing to the same place twice.** With the optimistic schema, both transactions will be aborted and retried in the case of a transaction hitting the same place twice.
> **A pessimistic transaction schema provides less liberty and will lock down resources, assuming that transactions will impact others.** This results in fewer aborts and retries, but it also means that transactions are forced to wait in line for their turn more often than with the optimistic transaction approach.

Finding a sweet spot between these two ideals is often where you'll find the best overall result.

Durability

The final aspect of the ACID approach to database management is durability.

Durability ensures that changes made to the database (transactions) that are successfully committed will survive permanently, even in the case of system failures. This ensures that the data within the database will not be corrupted by:

- Service outages
- Crashes
- Other cases of failure

Durability is achieved through the use of changelogs that are referenced when databases (or portions of the database) are restarted.

ACID supports data integrity & security

When every aspect of the ACID approach is brought together successfully, databases are maintained with the utmost data integrity and data security to ensure that they continuously provide value to the organization. A database with corrupted data can present costly issues due to the huge emphasis that organizations place on their data for both day-to-day operations as well as strategic analysis.

Using ACID properties with your database will ensure your database continues to deliver valuable data throughout operations.

ACID properties in practice

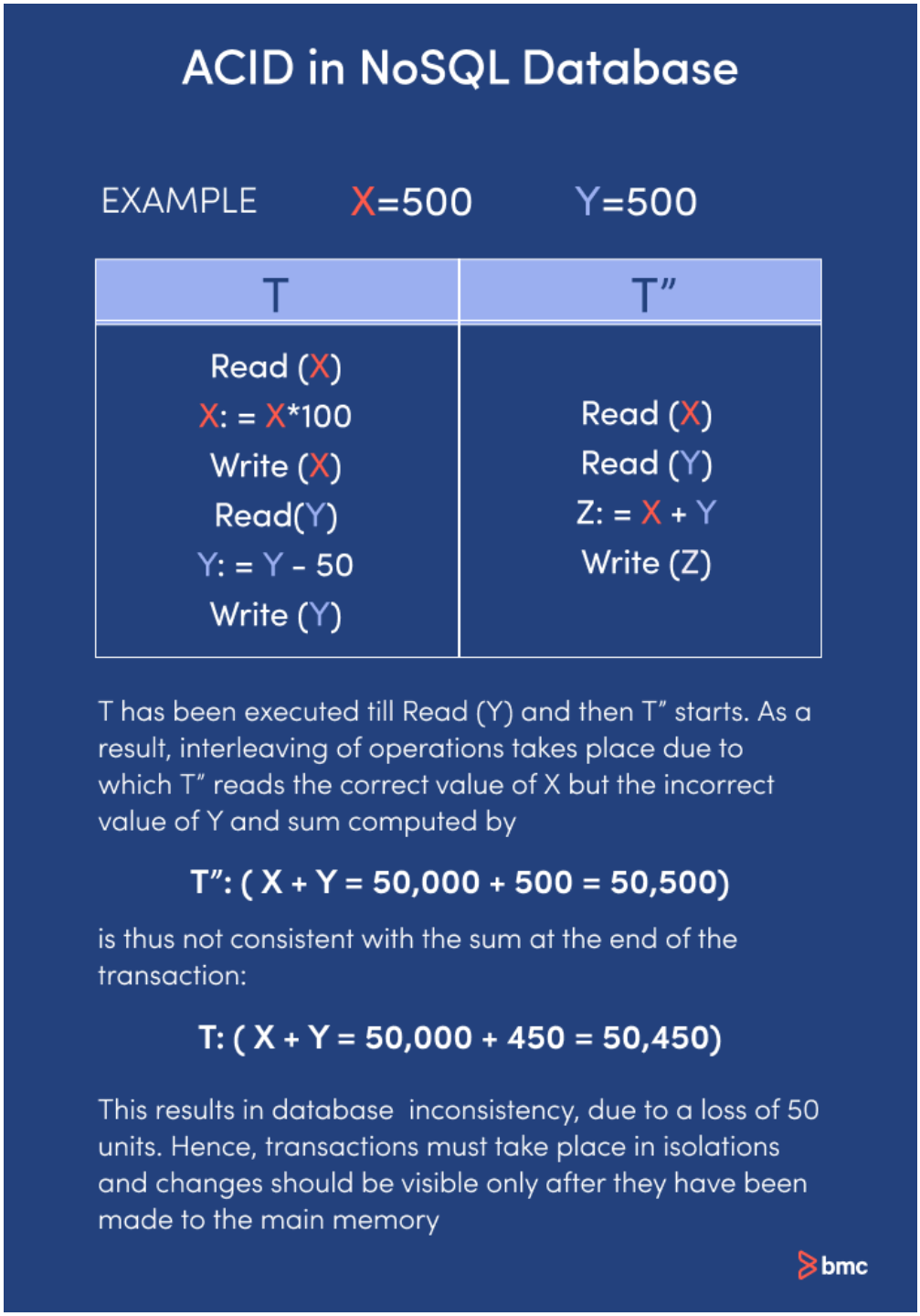
ACID in relational database

Using ACID properties in real-world applications such as relational databases is crucial for maintaining data integrity.

SQL Server fully complies with ACID principles, making it ideal for guaranteeing strong data integrity. Using MySQL, ACID compliance is storage-engine dependent.

ACID in noSQL database

MongoDB, a NoSQL database oriented around documents, was not originally designed around ACID properties. However, newer versions incorporate ACID at the document level, so it is fast approaching SQL Server in terms of data integrity.



Best practices for using ACID transactions

To effectively use ACID guiding principles, follow these best practices:

Model data to store related data together. You will reduce the risk of inconsistencies, promote scalability, and support more efficient access and update efficiency.
Break long-running transactions into smaller pieces. Shorter transactions require fewer resources.
Limit transactions to 1,000 document modifications. This allows the system to handle large data volumes while maintaining stability and performance.
Configure appropriate read and write concerns to better balance data consistency, system performance, and availability.
Handle errors and retry transactions that fail due to transient errors. You will reduce the number of failed operations and ensure that more ACID transactions fully complete to preserve data consistency.
Be aware of performance costs for transactions affecting multiple shards. To reduce latency and potential failures, you may wish to localize related transactions on one share.
Enforce data integrity using constraints that prevent invalid data entry and validate data before it gets to the database.
Use logging and backups to prevent data loss, schedule frequent backups and test them thoroughly.