Transactions

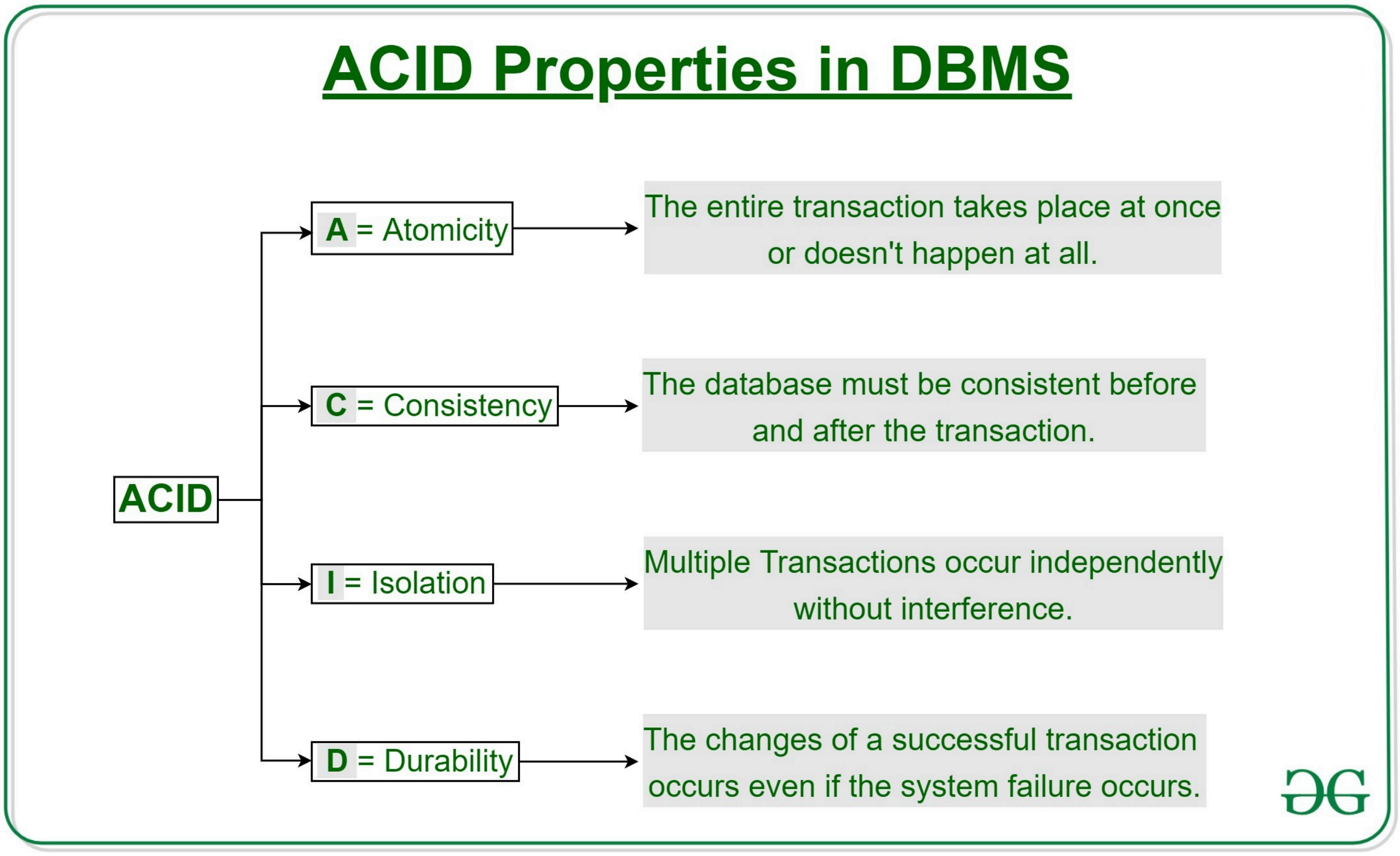
Transactions in a DBMS represent a sequence of operations that must be executed as a single, atomic unit. They are essential for maintaining database consistency and ensuring that data remains accurate and reliable, even in multi-user environments

See 14.1 Transactions, Recovery, and Concurrency Control

Transactions group a set of database operations. A transaction is Atomic, where the transaction should always succeed or fail.
During transaction execution, if an error occurs, and the transaction should uphold Consistency without data loss afterwards.
Transactions are executed concurrently, on partially the same data - this activity known as concurrency control (Isolation).

- Can be run sequentially or concurrently Trade-off between consistency (data integrity) and performance (latency)

ACID


ACID Properties in DBMS
ACID
A = Atomicity
The entire transaction takes place at once or doesn't happen at all.
C = Consistency
The database must be consistent before and after the transaction.
I = Isolation
Multiple Transactions occur independently without interference.
D = Durability
The changes of a successful transaction occurs even if the system failure occurs.

Atomicity

For Recovery
Each transaction runs to completion or has no effect at all.

Consistency

For Constraints, Triggers
After a transactions completes, the integrity constraints are satisfied.

Isolation

Each transaction appears as though it is the only one interacting with the database.

Durability

For Recovery
Persistence of successful transactions even through systems failure.

Distributed Transactions

Guaranteeing ACID properties in a distributed transaction across a distributed database, where no single node is responsible for all data affecting a transaction, presents additional complications. Network connections might fail, or one node might successfully complete its part of the transaction and then be required to roll back its changes because of a failure on another node.
The two-phase commit protocol (not to be confused with Two Phase locking [two-phase locking] provides Atomicity for distributed transactions to ensure that each participant in the transaction agrees on whether the transaction should be

commited or not. Briefly, in the first phase, one node (the coordinator) interrogates the other nodes (the participants), and only when all reply that they are prepared does the coordinator, in the second phase, formalize the transaction.

Transaction Lifecycle

14.2 Delineating Transactions and the Transaction Lifecycle
A database application can manage multiple operations and transactions simultaneously. To ensure the database system recognizes which operations belong to which transaction, transaction **boundaries** must be defined.

Transaction Boundaries

- Can be **explicitly** defined using `BEGIN_TRANSACTION` and `END_TRANSACTION`.
- **Implicitly** assumed by the DBMS: the first executable SQL statement in a thread starts a new transaction.

Transaction States

1. **Active**: The transaction starts when the first operation is executed.
2. **Scheduled**: Operations are organized by the transaction manager alongside other ongoing transactions. *(This state is sometimes referred to as having **uncommitted changes**.)*
3. **Completion**:

   #### Commit

   If the transaction is successful, all changes are made permanent and become visible to other users. Changes cannot be reversed (unless explicitly negated by a subsequent transaction).
   Note: A **commit** does not guarantee that changes are immediately written to disk; this depends on the system's durability policies.

   #### Abort

   If an error occurs at any level (application, OS, DBMS), the transaction fails. Partial changes are reverted through a **rollback**, leaving the database as if the transaction never happened.

Logfile

The logfile is essential in transaction management and recovery. It records every operation in a transaction, capturing `before` and `after` states for all data changes. Key elements in each log entry include

Buffer Management (RAM vs. Disk)

Manages data exchange between memory and physical storage. The Buffer management policies determine when committed changes are moved from RAM to disk.

The buffer manager and recovery manager work closely to ensure transaction durability and consistency. Checkpoints are set periodically, with the buffer manager flushing the committed data to disk. This helps in efficient Recovery by reducing the need to UNDO or REDO changes that occurred before the checkpoint.

Force and Steal Policies

Force / No Force affect data pages of **committed transactions.**

- **Force**: Ensures that all modified pages of a transaction are written to disk at commit time. This guarantees Durability but increases disk I/O operations, impacting performance.

- **No-Force**: Modified pages remain in the buffer even after transaction commit, leading to fewer disk I/O operations. However, this requires REDO logging to recover changes if a failure occurs.

Steal / No Steal affect data pages of **uncommitted transactions.**

- **Steal**: Allows pages from uncommitted transactions to be replaced if necessary and changes to disk before commit. While flexible, it demands UNDO logging to ensure data integrity in the event of an aborted transaction.
- **No-Steal**: Ensures that pages modified by active transactions are not replaced. This simplifies Recovery but limits buffer space, possibly leading to increased I/O for committed transactions. REDO Still has to be done.

No Force / Steal is the *best* and most common buffer management policy.

Recovery

Recovery mechanisms are crucial for ensuring database reliability and integrity in the event of failures. These mechanisms work by reverting or reapplying changes to bring the database to a consistent state. They are implemented to handle different types of failures, ensuring that the ACID properties, especially atomicity and durability, are maintained even when unexpected interruptions occur.

Types of Failures

- **Transaction Failure**: Errors within the transaction process (e.g., division by zero).
- **System Failure**: Issues with hardware, software, or the operating system.
- **Media Failure**: Problems with storage devices, such as disk crashes.

Recovery Techniques

Checkpoints

- **Definition**: A checkpoint is a periodic snapshot of the database state, marking a point where all committed changes have been applied to the database.
- **Purpose**: Checkpoints reduce recovery time by providing a clear point to begin recovery, rather than going through the entire log from the beginning.
- **Implementation**: At each checkpoint, all dirty pages (pages in memory that have been modified but not yet written to disk) are flushed, and all committed transactions are saved to the disk. During recovery, the system only needs to process logs generated after the last checkpoint, speeding up the process.

UNDO

Reverts changes made by failed transactions.

- **Purpose**: Reverts changes made by any transaction that did not complete successfully.
- **When Applied**: During transaction failure or system crash recovery, all uncommitted transactions undergo an UNDO operation.
- **Process**: The log file is scanned in reverse order to revert any uncommitted changes, ensuring that the database state is consistent.

REDO

Reapplies changes for committed transactions that were not saved due to failure

- **Purpose**: Reapplies changes made by transactions that were committed but not fully saved due to a system crash.

- **When Applied**: During recovery after a system failure, REDO ensures that committed transactions are retained.
- **Process**: The log is scanned, and committed transactions that lack durable updates are re-executed based on log entries

Write-Ahead Logging (WAL Protocol)

1. Before any changes are written to disk, we force the corresponding log record to disk
2. Before a transaction is committed, we force all log records for the transaction to disk
   This ensures Atomic and Durability.

Locking

Transactions must *wait* for each other to access the Critical section (data), even if unrelated. This is where *locks* prevent two processes on accessing the same data.

See Mutual Exclusion, Deadlock, Principles of database management the practical guide to storing, managing and analyzing big and small data, page 991

Two Phase locking (2PL)

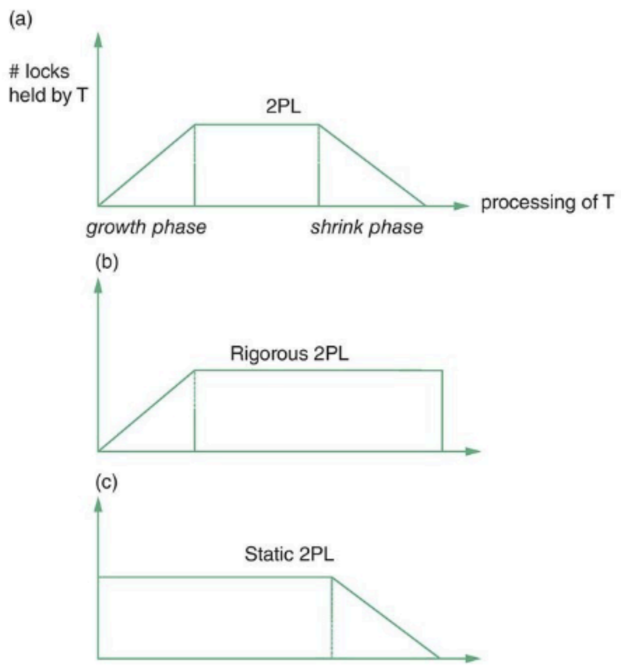
The **Two Phase locking (2PL)** protocol applies a compatibility matrix and determines when lock and unlock instructions are allowed in a transaction’s lifecycle; specifies that acquiring and releasing locks occurs in two phases for each transaction.

1. Transactions must acquire a **shared lock** to read and an **exclusive lock** to update a database object.
2. The lock manager grants locks based on a compatibility matrix, putting transactions on hold if requests conflict.
3. Locking occurs in two phases
   - **Growth phase**: Transactions can acquire locks but cannot release them.
   - **Shrinking phase**: Once a transaction releases its first lock, it cannot acquire new locks.

In most DBMS a there are the following 2PL variants:

1. ***Rigorous 2PL*** is a 2PL variant that specifies that a transaction **holds all its locks until it is committed.**
   - Completely preventing Cascading rollbacks.
   - May lead to Deadlocks,
2. ***Static 2PL* (conservative 2PL)** is a 2PL variant that specifies that a transaction acquires all its locks right at the start of the transaction, and hold all their exclusive locks until they commit or abort.
   - Does not guarantee Cascading rollbacks

- Deadlocks are avoided



Cascading rollbacks

Releasing a lock to early, before a transaction is committed - This allows other transactions to access data modified by an uncommitted transaction, creating unconfirmed changes.