DATABASE NORMALIZATION

Description of normalization

Normalization is a systematic approach to organize data within a database to reduce redundancy and eliminate undesirable characteristics such as **insertion**, **update**, and **deletion** anomalies. The process involves breaking down large tables into smaller, well-structured ones and defining relationships between them. This not only reduces the chances of storing duplicate data but also improves the overall efficiency of the database.

Why is Normalization Important?

- Reduces Data Redundancy: Duplicate data is stored efficiently, saving disk space and reducing inconsistency.
- Improves Data Integrity: Ensures the accuracy and consistency of data by organizing it in a structured manner.
- Simplifies Database Design: By following a clear structure, database designs become easier to maintain and update.
- Optimizes Performance: Reduces the chance of anomalies and increases the efficiency of database operations.

First normal form(1NF)

A table is in 1NF if it satisfies the following conditions:

A. All columns contain atomic values (i.e., indivisible values).
B. Each row is unique (i.e., no duplicate rows).
C. Each column has a unique name.
D. The order in which data is stored does not matter.

What 1NF works;

- Eliminate repeating groups in individual tables.
- Create a separate table for each set of related data.
- Identify each set of related data with a primary key.

Don't use multiple fields in a single table to store similar data. For example, to track an inventory item that may come from two possible sources, an inventory record may contain fields for Vendor Code 1 and Vendor 2.

What happens when you add a third vendor? Adding a field isn't the answer; it requires program and table modifications and doesn't smoothly accommodate a dynamic number of vendors. Instead, place all vendor information in a separate table called Vendors, then link inventory to vendors with an item number key, or vendors to inventory with a vendor code key.

Second normal form

- Create separate tables for sets of values that apply to multiple records.
- Relate these tables with a foreign key.

Records shouldn't depend on anything other than a table's primary key (a compound key, if necessary). For example, consider a customer's address in an accounting system. The address is needed by the Customers table, but also by the Orders, Shipping, Invoices, Accounts Receivable, and Collections tables. Instead of storing the customer's address as a separate entry in each of these tables, store it in one place, either in the Customers table or in a separate Addresses table.

Third normal form

- Eliminate fields that don't depend on the key.

Values in a record that aren't part of that record's key don't belong in the table. In general, anytime the contents of a group of fields may **apply to more than a single record in the table**, consider placing those fields in a separate table.

For example, in an Employee Recruitment table, a candidate's university name and address may be included. But you need a complete list of universities for group mailings. If university information is stored in the Candidates table, there is no way to list universities with no current candidates. Create a separate Universities table and link it to the Candidates table with a university code key.

<u>EXCEPTION:</u> Adhering to the third normal form, while theoretically desirable, isn't always practical. If you have a Customers table and you want to eliminate all possible interfield dependencies, you must create separate tables for cities, ZIP codes, sales representatives, customer classes, and any other factor that may be duplicated in multiple records. In theory, normalization is worth pursuing. However, many small tables may degrade performance or exceed open file and memory capacities.

It may be more feasible to apply third normal form only to data that changes frequently. If some dependent fields remain, design your application to require the user to verify all related fields when any one is changed.

Normalizing an example table

These steps demonstrate the process of normalizing a fictitious student table.

1. ***Unnormalized table:***

Student#	Advisor	Adv_Room	Class1	Class2	Class3
1022	Jones	412	101-07	143-01	159-02

4123	Smith	216	101-07	143-01	179-04

2. *First normal form: No repeating groups*

Tables should have only two dimensions. Since one student has several classes, these classes should be listed in a separate table.

Fields Class1, Class2, and Class3 in the above records are indications of design trouble.

Spreadsheets often use the third dimension, but tables shouldn't. Another way to look at this problem is with a **one-to-many relationship**, don't put the one side and the many sides in the same table. Instead, create another table in first normal form by eliminating the repeating group (Class#), as shown in the following example:

Student#	Advisor	Adv_Room	Class#
1022	Jones	412	101-07
1022	Jones	412	143-01
1022	Jones	412	159-02
4123	Smith	216	101-07
4123	Smith	216	143-01
4123	Smith	216	179-04

3. *Second normal form: Eliminate redundant data*

Note the multiple Class# values for each Student# value in the above table. Class# isn't functionally dependent on Student# (primary key), so this relationship isn't in second normal form.

The following tables demonstrate second normal form:

Students#	**Advisor**	**Adv-Room**
1022	Jones	412
4123	Smith	216

Registration:

Students#	Class#
1022	101-07
4123	143-01
1022	159-02
4123	101-07
4123	143-01
4123	179-04

4. ***Third normal form: Eliminate data not dependent on key***

In the last example, Adv-Room (the advisor's office number) is functionally dependent on the Advisor attribute. The solution is to move that attribute from the Students table to the Faculty table, as shown below:

Students:

Student#	Advisor
1022	Jones
4123	Smith

Faculty:

Name	Room	Dept
Jones	412	42
Smith	216	42

Applications of Normal Forms in DBMS

➔ **Ensures Data Consistency:** Prevents data anomalies by ensuring each piece of data is stored in one place, reducing inconsistencies.

➔ **Reduces Data Redundancy:** Minimizes repetitive data, saving storage space and avoiding errors in data updates or deletions.
➔ **Improves Query Performance:** Simplifies queries by breaking large tables into smaller, more manageable ones, leading to faster data retrieval.
➔ **Enhances Data Integrity:** Ensures that data is accurate and reliable by adhering to defined relationships and constraints between tables.
➔ **Easier Database Maintenance:** Simplifies updates, deletions, and modifications by ensuring that changes only need to be made in one place, reducing the risk of errors.
➔ **Facilitates Scalability:** Makes it easier to modify, expand, or scale the database structure as business requirements grow.
➔ **Supports Better Data Modeling:** Helps in designing databases that are logically structured, with clear relationships between tables, making it easier to understand and manage.
➔ **Reduces Update Anomalies:** Prevents issues like insertion, deletion, or modification anomalies that can arise from redundant data.
➔ **Improves Data Integrity and Security:** By reducing unnecessary data duplication, normal forms help ensure sensitive information is securely and correctly maintained.
➔ **Optimizes Storage Efficiency:** By organizing data into smaller tables, storage is used more efficiently, reducing the overhead for large databases

ACID PROPERTIES OF A DATABASE.

ACID stands for **Atomicity**, **Consistency**, **Isolation**, and **Durability**.

These four key properties define how a transaction should be processed in a reliable and predictable manner, ensuring that the database remains consistent, even in cases of failures or concurrent accesses.

What Are Transactions in DBMS? A transaction in DBMS refers to a sequence of operations performed as a single unit of work. These operations may involve reading or writing data to the database.

To maintain data integrity, DBMS ensures that each transaction adheres to the ACID properties. Think of a transaction like an ATM withdrawal. When we withdraw money from our account, the transaction involves several steps:

- Checking your balance.
- Deducting the money from your account.
- Adding the money to the bank's record.

For the transaction to be successful, all steps must be completed. If any part of this process fails (e.g., if there’s a system crash), the entire transaction should fail, and no data should be altered. This ensures the database remains in a consistent state.

The Four ACID Properties

1. **Atomicity:** "All or Nothing" Atomicity ensures that a transaction is atomic, it means that either the entire transaction completes fully or doesn't execute at all.

   There is no in-between state i.e. transactions do not occur partially. If a transaction has multiple operations, and one of them fails, the whole transaction is rolled back, leaving the database unchanged.

   This avoids partial updates that can lead to inconsistency.

Commit: If the transaction is successful, the changes are permanently applied.
Abort/Rollback: If the transaction fails, any changes made during the transaction are discarded. Example: Consider the following transaction T consisting of T1 and T2 : Transfer of $100 from account X to account Y .

Example If the transaction fails after completion of T1 but before completion of T2 , the database would be left in an inconsistent state. With Atomicity, if any part of the transaction fails, the entire process is rolled back to its original state, and no partial changes are made.

2. **Consistency:** Maintaining Valid Data States Consistency ensures that a database remains in a valid state before and after a transaction. It guarantees that any transaction will take the database from one consistent state to another, maintaining the rules and constraints defined for the data. In simple terms, a transaction should only take the database from one valid state to another. If a transaction violates any database rules or constraints, it should be rejected, ensuring that only consistent data exists after the transaction.

Example: Suppose the sum of all balances in a bank system should always be constant. Before a transfer, the total balance is $700. After the transaction, the total balance should remain $700. If the transaction fails in the middle (like updating one account but not the other), the system should maintain its consistency by rolling back the transaction

Total before T occurs = 500 + 200 = 700 . Total after T occurs = 400 + 300 = 700 .

3. **Isolation**: Ensuring Concurrent Transactions Don't Interfere This property ensures that multiple transactions can occur concurrently without leading to the inconsistency of the database state. Transactions occur independently without interference. Changes occurring in a particular transaction will not be visible to any other transaction until that particular change in that transaction is written to memory or has been committed.

This property ensures that when multiple transactions run at the same time, the result will be the same as if they were run one after another in a specific order. This property prevents issues such as dirty reads (reading uncommitted data), non-repeatable reads (data changing between two reads in a transaction), and phantom reads (new rows appearing in a result set after the transaction starts).

Example: Let’s consider two transactions:Consider two transactions;

T and T''.

X = 500, Y = 500

Transaction T:

- T wants to transfer $50 from X to Y.
- T reads Y (value: 500), deducts $50 from X (new X = 450), and adds $50 to Y (new Y = 550).

Transaction T'':

- T'' starts and reads X (value: 500) and Y (value: 500), then calculates the sum: 500 + 500 = 1000.

But, by the time T finishes, X and Y have changed to 450 and 550 respectively, so the correct sum should be 450 + 550 = 1000. Isolation ensures that T'' should not see the old values of X and Y while T is still in progress.

Both transactions should be independent, and T'' should only see the final state after T commits. This prevents inconsistent data like the incorrect sum calculated by T''

4. **Durability:** Persisting Changes This property ensures that once the transaction has completed execution, the updates and modifications to the database are stored in and written to disk and they persist even if a system failure occurs.

   These updates now become permanent and are stored in non-volatile memory. In the event of a failure, the DBMS can recover the database to the state it was in after the last committed transaction, ensuring that no data is lost.

Example: After successfully transferring money from Account A to Account B, the changes are stored on disk. Even if there is a crash immediately after the commit, the transfer details will still be intact when the system recovers, ensuring durability.

How ACID Properties Impact DBMS Design and Operation

The ACID properties, in totality, provide a mechanism to ensure the **correctness** and **consistency** of a database in a way such that each transaction is a group of operations that acts as a single unit, produces consistent results, acts in isolation from other operations, and updates that it makes are durably stored.

1. **Data Integrity and Consistency;** ACID properties safeguard the data integrity of a DBMS by ensuring that transactions either complete successfully or leave no trace if interrupted. They prevent partial updates from corrupting the data and ensure that the database transitions only between valid states.

2. **Concurrency Control;** ACID properties provide a solid framework for managing concurrent transactions. Isolation ensures that transactions do not interfere with each other, preventing data anomalies such as lost updates, temporary inconsistency, and uncommitted data.

3. **Recovery and Fault Tolerance;** Durability ensures that even if a system crashes, the database can recover to a consistent state. Thanks to the Atomicity and Durability properties, if a transaction fails midway, the database remains in a consistent state.

Advantages of ACID Properties in DBMS

1. ***Data Consistency:*** ACID properties ensure that the data remains consistent and accurate after any transaction execution.
2. ***Data Integrity:*** It maintains the integrity of the data by ensuring that any changes to the database are permanent and cannot be lost.
3. ***Concurrency Control:*** ACID properties help to manage multiple transactions occurring concurrently by preventing interference between them.
4. ***Recovery:*** ACID properties ensure that in case of any failure or crash, the system can recover the data up to the point of failure or crash.

Disadvantages of ACID Properties in DBMS

1. ***Performance Overhead:*** ACID properties can introduce performance costs, especially when enforcing isolation between transactions or ensuring atomicity.
2. ***Complexity:*** Maintaining ACID properties in distributed systems (like microservices or cloud environments) can be complex and may require sophisticated solutions like distributed locking or transaction coordination.
3. ***Scalability Issues:*** ACID properties can pose scalability challenges, particularly in systems with high transaction volumes, where traditional relational databases may struggle under load.\

ACID in the Real World: Where Is It Used?

In modern applications, ensuring the **reliability** and **consistency** of data is crucial.

ACID properties are fundamental in sectors like:

- **Banking:** Transactions involving money transfers, deposits, or withdrawals must maintain strict consistency and durability to prevent errors and fraud.

- **E-commerce:** Ensuring that inventory counts, orders, and customer details are handled correctly and consistently, even during high traffic, requires ACID compliance.
- **Healthcare:** Patient records, test results, and prescriptions must adhere to strict consistency, integrity, and security standards.

Conclusion

The ACID properties in DBMS provide the backbone for maintaining data consistency, integrity, and reliability in the face of transaction failures, concurrent operations, and system crashes.

In today's digital world, ACID properties ensure that database systems can **handle complex transactions securely**, **reliably,** and **efficiently**, which is why they remain a cornerstone of data management systems used in a variety of critical applications.

By understanding how ACID works, developers and system administrators can design better systems and make informed decisions about database management, especially when it comes to balancing **consistency**, **performance**, and **scalability.**