DATABASE NORMALIZATION AND ACID PROPERTIES

NORMALIZATION

Database Normalization refers to the process of organizing data in a relational database to reduce data redundancy and improve data integrity.  
It involves dividing large tables into smaller, non-redundant tables and establishing relationships between them.  
Normalization checks for data dependencies and restructures tables to remove unnecessary dependencies.  
Based on functional dependency theory, the process is repeated until no more data dependency issues are left.

Why do we need Normalization?

The primary objective for normalizing the relations is to eliminate the below anomalies. Failure to reduce anomalies results in data redundancy, which may threaten data integrity and cause additional issues as the database increases. Normalization consists of a set of procedures that assist you in developing an effective database structure.

* **Insertion Anomalies:** Insertion anomalies occur when it is not possible to insert data into a database because the required fields are missing or because the data is incomplete. For example, if a database requires that every record has a primary key, but no value is provided for a particular record, it cannot be inserted into the database.
* **Deletion anomalies:** Deletion anomalies occur when deleting a record from a database and can result in the unintentional loss of data. For example, if a database contains information about customers and orders, deleting a customer record may also delete all the orders associated with that customer.
* **Updation anomalies:** Updation anomalies occur when modifying data in a database and can result in inconsistencies or errors. For example, if a database contains information about employees and their salaries, updating an employee’s salary in one record but not in all related records could lead to incorrect calculations and reporting.

**Importance Of Normalization**

1. Highlights constraints and data dependencies, helping better understand data structure.
2. Reduces data redundancy, saving storage space and simplifying maintenance.
3. Each normal form eliminates a specific type of data dependency.
4. Allows for simpler and more efficient data retrieval.
5. A database in higher normal forms (like 3NF) is more flexible and independent.

**Problems with Data Redundancy**

1. Wasted storage: Repeating the same data across the database consumes space.
2. Insertion issues: Same data must be entered in multiple places.
3. Deletion issues: deleting in one place may require deleting elsewhere.
4. Update issues: Data must be updated in multiple locations.

**Database Normal Forms**

**First Normal Form (1NF)**

It lays the foundation for higher levels of normalization by enforcing atomicity and eliminating duplicate and multivalued data.

**Rules:**

A relation (table) is said to be in First Normal Form (1NF) if:

* Each table cell contains a single (atomic) value.
* Each record is unique.
* There are no null values.
* There are no repeating groups or arrays.
* Primary key is clearly defined.

Goal is to eliminate repeating groups and ensure atomicity.

Example: Consider the following Employees Table.

**Before 1NF:**

|  |  |  |
| --- | --- | --- |
| **Employee** | **Age** | **Department** |
| Melvin | 32 | Marketing, Sales |
| Edward | 45 | Quality Assurance |
| Alex | 36 | IT |

**After 1NF:**

|  |  |  |
| --- | --- | --- |
| **Employee** | **Age** | **Department** |
| Melvin | 32 | Marketing |
| Melvin | 32 | Sales |
| Edward | 45 | Quality Assurance |
| Alex | 36 | IT |

**Second Normal Form (2NF)**

Second Normal Form (2NF) is based on the concept of fully functional dependency.

**Fully Functional Dependency** means a non-key attribute depends on the **entire** primary key, not just part of it.

**Rules:**

For a table to be in 2NF, it must first meet the following requirements;

1. Meet 1NF requirements: The table must be in 1NF
2. Eliminate **partial dependencies**: A **partial dependency** occurs when a **non-prime attribute** (not part of the candidate key) depends only on a part of a composite primary key, rather than the entire key.

Example: Consider the following table.

**Before 2NF:**

|  |  |  |  |
| --- | --- | --- | --- |
| **StudentID** | **ProjectNO** | **StudentName** | **ProjectName** |
| S01 | 199 | Faiza | Statistics |
| S02 | 120 | Claire | Psychology |

In the above table;

The StudentName can be determined by StudentID, which makes the relation Partial Dependent.

* The ProjectName can be determined by ProjectNo, which makes the relation Partial Dependent.
* Therefore, the relation violates the 2NF in Normalization and is considered a bad database design.

To remove Partial Dependency and violation on 2NF, decompose the tables.

**After 2NF:**

**<Student info>**

|  |  |  |
| --- | --- | --- |
| **StudentID** | **ProjectNO** | **StudentName** |
| S01 | 199 | Faiza |
| S02 | 120 | Claire |

**<Project info>**

|  |  |
| --- | --- |
| **ProjectNO** | **ProjectName** |
| 199 | Statistics |
| 120 | Psychology |

**Third Normal Form (3NF)**

A relation is in **Third Normal Form (3NF)** if it satisfies the following two conditions:

1. It is in **Second Normal Form (2NF)**: This means the table has no partial dependencies (i.e., no non-prime attribute is dependent on a part of a candidate key).
2. There is **no transitive dependency** for **non-prime attributes**: In simpler terms, no non-key attribute should depend on another non-key attribute. Instead, all non-key attributes should depend directly on the primary key.

**What is Transitive Dependency?**

When an indirect relationship causes functional dependency, it is called Transitive Dependency.

If P -> Q (Q is dependent on P) and Q -> R (Q is dependent on R) is true, then P-> R (R is also dependent on P).

This is known as transitive dependency.

Example: Consider the following table;

|  |  |  |  |
| --- | --- | --- | --- |
| **Movie\_ID** | **Listing\_ID** | **Listing\_Type** | **Dvd\_Price** |
| M08 | L09 | Crime | 180 |
| M03 | L05 | Drama | 250 |
| M05 | L09 | Crime | 180 |

The above table is not in 3NF because it has a transitive functional dependency.

Movie\_ID -> Listing\_ID

Listing\_ID -> Listing\_Type

Therefore, the following has transitive functional dependency.

Movie\_ID -> Listing\_Type

To remove the violation, you need to split the tables and remove the transitive functional dependency.

|  |  |  |
| --- | --- | --- |
| **Movie\_ID** | **Listing\_ID** | **Dvd\_Price** |
| M08 | L09 | 180 |
| M03 | L05 | 250 |
| M05 | L09 | 180 |

|  |  |
| --- | --- |
| **Listing\_ID** | **Listing\_Type** |
| L09 | Crime |
| L05 | Drama |
| L09 | Crime |

ACID PROPERTIES

What are ACID properties?

ACID is an acronym for Atomicity, Consistency, Isolation, and Durability.

ACID properties maintain the integrity, accuracy, and stability of the database even in cases of system failures, errors, or concurrent access.

**A – Atomicity**

Atomicity ensures that a transaction is treated as a single unit, which either completes in full or does not happen at all.

* If any part of the transaction fails, the entire transaction is **rolled back**.
* Prevents partial updates to the database.
* Maintains **all-or-nothing** rule.

Example:

A bank transaction where Ksh100 is transferred from Account A to Account B:

* Debit Ksh100 from A
* Credit Ksh100 to B

If credit fails, the debit must be **reversed**.

**C – Consistency**

Consistency ensures that a transaction takes the database **from one valid state to another valid state**, maintaining all **defined rules and constraints.**

* Data integrity constraints (like primary key, foreign key, unique) must always be met.
* No data should become corrupt due to transaction execution.

Example:

If a student’s age must be between 5 and 25:

* A transaction that inserts age = 30 would be rejected to maintain **consistency**.

**I – Isolation**

Isolation ensures that **concurrent execution** of transactions leaves the database in the same state as if the transactions were executed **one after the other (serially)**.

* Transactions must not **interfere** with each other.
* Intermediate results of one transaction must not be visible to others.

Example:

Two customers booking the **last ticket** at the same time:

* Isolation ensures that only one booking is allowed; the other will be processed **after** or **rolled back**.

**D – Durability**

Durability ensures that once a transaction is **committed**, the changes it made to the database are **permanent**, even if a **system crash** occurs.

* Data is saved to **non-volatile storage**.
* Committed data is not lost, even in the case of power failure or restart.

Example:

If money is transferred and the system crashes, the transaction is **not lost**; the updated account balances are still available after restart.