Normalization

Normalization is the process of organizing data to minimize data redundancy(duplication) and to improve data integrity.

Problems of data redundancy

- Disk space Wastage
- Data Inconsistency
- DML queries will be slow

Advantages

- **Minimizes Data Redundancy:** Reduces duplicate data, saving storage space.
- **Avoids Anomalies:** Prevents issues with data insertion, update, and deletion.
- **Improves Query Performance:** Simplifies queries and makes the database more efficient

Disadvantages

- **Performance Issues:** The need for multiple joins can slow down read operations, especially in high normalized databases

Although there are 6 NF , I have discussed only 3NF . In real time most databases are designed to meet 3NF and going beyond 3NF is not necessary.

1st Normal Form

A table is in first normal form when the columns have atomic values(Atomic value is a value that is stored in single column of a table).

Ex: If you want to update middle name from below De-normalized table , you need to write weird SQL query, so you will be having DML (Insert, Update, Delete) anomalies.

So we will have separate columns for first, middle and last names as shown in below normalized table.

Now you know how simple it is to write SQL query to update middle name on Normalized table.

ID	Name
1	Anil, Reddy, Chenchu
2	Satheesh, Kumar, Jeggajetti

DeNormalized

Data is in De-normalized form when multiple values of a single attribute are stored in single column rather than in separate columns

ID	First_Name	Middle_Name	Last_Name
1	Anil	Reddy	Chenchu
2	Satheesh	Kumar	Jeggajetti

Normalized

Each cell contains only one value

2nd Normal Form

First normal form should be satisfied and all non key columns should be fully dependent on the primary key(PK).

Joint P.K.
(Composite Key)

ID	Country_ID	First_Name	Middle_Name	Last_Name	Country
1	1	Anil	Reddy	Chenchu	India
2	2	Satheesh	Kumar	Jeggajetti	U.K

In the above table **Country** column(Non key value) is not fully dependent on Primary Key(P.K) **country ID**, it is partially dependent on **country ID.**

So we move out **Country** in to another table as shown below, now Country column is fully dependent on primary key(P.K) country_ID.

Country_ID	Country
1	India
2	U.K

→ **2 NF satisfied**

3rd Normal Form

First and Second normal forms should be satisfied and no transient dependency should be present.

Joint P.K.
(Composite Key)

ID	Country_ID	First_Name	Middle_Name	Last_Name	Product	Cost
1	1	Anil	Reddy	Chenchu	Shoes	10
2	2	Satheesh	Kumar	Jeggajetti	Shirts	20

Transient dependency meance a non key column should not depend on another non key column, in the above table cost column is dependent on product column.

So we move cost column to another table as shown below

Product	Cost	Size
Shoes	10	11
Shirts	20	XL

→ **3rd NF Satisfied**

www.linkedin.com/in/chenchuanil

De - Normalization

Denormalization is a database technique to improve search performances.

There is a intentional data redundancy meance sorting all necessa data in a large single tables which in turn reduces joins, so query results will be faster.

Which one to Choose

Choose normalization when you have write heavy operations such as customer is ordering every 2 or 3 sec. Ex: OLTP

Choose De-normalization when you have OLAP analysis like B.I reports, trend analysis reports etc.

ACID POPERTIES

The ACID properties ensure the reliability of transactions in a database. They stand for **Atomicity, Consistency, Isolation, and Durability.**

Let's break down each property with an example to illustrate how they work in practice.

1. Atomicity

Atomicity ensures that all operations within a transaction are completed successfully. If any part of the transaction fails, the entire transaction is rolled back, and no changes are made to the database.

```
BEGIN TRANSACTION;

UPDATE accounts
SET balance = balance - 100
WHERE  account_id = 1;

UPDATE accounts
SET balance = balance + 100
WHERE  account_id = 2;

COMMIT;
```

If either of the UPDATE statements fail in above SQL query(e.g., insufficient funds in account 1), the transaction is rolled back, ensuring no partial updates occur.

2. Consistency

Consistency ensures that a transaction brings the database from one valid state to another, complying with all predefined rules (such as constraints, triggers, or integrity conditions). If a transaction violates any rule, the entire transaction is aborted, and the database remains unchanged.

Example:

Consider a scenario where a bank database has a rule that account balances must always be non-negative (i.e., there's a CHECK constraint that prevents balances from falling below zero)

```
BEGIN TRANSACTION;

UPDATE  accounts
SET  balance = balance - 500
WHERE  account_id = 1;

-- Suppose account 1 has a balance of  300
-- This operation would violate the CHECK constraint
-- that ensures the balance must always be >= 0.

COMMIT;
```

If the UPDATE operation attempts to reduce the balance below zero, the transaction will fail due to the CHECK constraint, and no changes will be applied to the database.

Consistency ensures that the database maintains its integrity, meaning no invalid data (like a negative balance) will ever be stored. The transaction will either complete successfully or roll back to maintain this state.

3. Isolation

Isolation ensures that transactions are executed in isolation from one another. The intermediate state of a transaction should not be visible to other concurrent transactions.

Example:

Consider two transactions running concurrently

- **Transaction A: Withdraw ₹100 from account 1.**
- **Transaction B: Query the balance of account 1.**

```
-- Transaction A
BEGIN  TRANSACTION;
UPDATE  accounts
SET  balance = balance - 100
WHERE  account_id = 1;
-- Transaction not yet committed

-- Transaction B
SELECT  balance FROM accounts
WHERE  account_id = 1;
```

If isolation is maintained properly, Transaction B will either see the balance before or after Transaction A's update, but not the intermediate state.

4. Durability

Durability ensures that once a transaction is committed, its effects are permanent. Even in the case of a system crash, power failure, or other catastrophic events, the changes made by committed transactions will persist in the database.

Example:

Consider a scenario where you transfer ₹200 from one account to another. Once the transaction is committed, the database guarantees that the changes (the deduction from one account and the addition to the other) will not be lost, even if there's a power outage or a system failure.

```
BEGIN TRANSACTION;

UPDATE accounts
SET balance = balance - 200
WHERE account_id = 1;

UPDATE accounts
SET balance = balance + 200
WHERE account_id = 2;

COMMIT;
```

After the COMMIT statement, the changes to the accounts table are durable. Even if the system crashes or there's a power failure immediately after the commit, the database will ensure that these changes remain intact when the system comes back online.

Durability ensures that no committed transactions are lost, making it crucial for financial systems where reliability is paramount.

Slowly Changing Dimensions (SCD)

In data warehousing and business intelligence, Slowly Changing Dimensions (SCD) are a concept used to manage and track changes to dimension data over time. Dimension tables store the attributes or characteristics of business entities, such as customers, products, employees, etc. Slowly changing dimensions refer to scenarios where these attributes change frequently, and there are different ways to handle these changes, known as SCD types.

Here is the list of Slowly Changing Dimension (SCD) types:

- SCD Type 0: Retain Original (No Change)
- SCD Type 1: Overwrite
- SCD Type 2: Add New Row (Track History)
- SCD Type 3: Add New Attribute (Track Limited History)

Let us discuss each type one by one

SCD Type 0: Retain Original (No Change)

Design : The dimension table is designed to store static data, and no provisions are made for changes.

- **Purpose:** No changes are tracked.
- **Method:** Data is never updated, meaning once a record is inserted, it remains static. Even if the dimension attributes change in the source system, the data remains the same in the data warehouse.
- **Use Case:** Useful when historical accuracy is required, and changes to attributes are irrelevant.

Example

```sql
CREATE TABLE customer_dimension (
  customer_id INT PRIMARY KEY,
  name VARCHAR(100),
  address VARCHAR(200),
  city VARCHAR(100),
  state VARCHAR(100),
  zip_code VARCHAR(10)
);
```

Key Point: Data never gets updated, so no need to worry about managing historical data or tracking changes.

SCD Type 1: Overwrite

Purpose: The old data is overwritten with new data.

Method: When an attribute changes, the new value simply overwrites the
old one. This method does not retain any history of changes.

Use Case: When keeping history is not important, and you only care about the most current information.

Example: A customer changes their address, and the old address is overwritten with the new one.

```sql
CREATE TABLE customer_dimension (
  customer_id INT PRIMARY KEY,
  name VARCHAR(100),
  address VARCHAR(200),
  city VARCHAR(100),
  state VARCHAR(100),
  zip_code VARCHAR(10)
);
```

When updating the table, execute SQL like shown below:

```sql
UPDATE customer_dimension
SET address = 'New Address',
    city = 'New City',
    state = 'New State',
    zip_code = 'New Zip'
WHERE customer_id = 1;
```

Key Point: The previous values are lost, so no history is retained.

SCD Type 2: Add New Row (Track History)

Purpose: Historical changes are tracked by adding new records.

Method: Instead of overwriting the existing record, a new row is inserted with the new attribute values, and the old row is retained with additional metadata (e.g., effective dates, active/inactive flags) to track historical versions.

Use Case: When it's important to track the history of changes over time.

Example: If a customer changes their address, a new row is added to the dimension table with the new address, while the old row remains unchanged, representing the historical record.

```sql
CREATE TABLE customer_dimension (
  customer_dimension_id INT PRIMARY KEY AUTO_INCREMENT,
  customer_id INT,
  name VARCHAR(100),
  address VARCHAR(200),
  city VARCHAR(100),
  state VARCHAR(100),
  zip_code VARCHAR(10),
  effective_start_date DATE,
  effective_end_date DATE DEFAULT '9999-12-31',
  is_current BOOLEAN
);
```

To insert a new record when a change happens:

```sql
-- 1. Mark the current record as inactive (end the historical record)

UPDATE   customer_dimension
SET    effective_end_date = CURDATE(),          -- Set the end date to today
   is_current = FALSE                           -- Mark it as inactive
WHERE     customer_id = 1                       -- Filter by the specific customer
 AND    is_current = TRUE;                      -- Only update the current active record

-- 2. Insert a new record with the updated address (start a new version)

INSERT INTO    customer_dimension (
   customer_id, name, address, city, state, zip_code, effective_start_date, is_current
)
VALUES (
   1, 'John Doe', '123 New Address', 'New City', 'New State', '12345', CURDATE(), TRUE );
```

Key Point: Each new version of the data gets a new row, and historical versions are retained by setting the end date and status flag.

SCD Type 3: Add New Attribute (Track Limited History)

Purpose: Track limited history by adding new columns (attributes) for changes.

Method: Instead of adding new rows, a new column (attribute) is added to the table to store the previous value of the attribute. This approach only tracks a limited number of changes (usually just the previous value and the current value).

Use Case: Useful when only a few changes need to be tracked, and full historical tracking is not necessary.

Example: A "previous address" column is added to store the old address, while the current address is stored in a separate column.

Design:
Add columns to the dimension table for storing both the current and previous values of attributes. This method only tracks limited historical changes, typically just one previous version.

When updating the table:

```sql
CREATE TABLE customer_dimension (
  customer_id INT PRIMARY KEY,
  current_address VARCHAR(200),
  previous_address VARCHAR(200),
  current_city VARCHAR(100),
  previous_city VARCHAR(100),
  current_state VARCHAR(100),
  previous_state VARCHAR(100),
  current_zip_code VARCHAR(10),
  previous_zip_code VARCHAR(10)
);
```

```sql
UPDATE customer_dimension
SET previous_address = current_address,
    previous_city = current_city,
    previous_state = current_state,
    previous_zip_code = current_zip_code,
    current_address = 'New Address',
    current_city = 'New City',
    current_state = 'New State',
    current_zip_code = 'New Zip'
WHERE customer_id = 1;
```

Key Point: Only a limited amount of historical information is stored (e.g., the previous value).