Data Normalization

What is Normalization?

Normalization is a process used to organise data in a in a database efficiently.

- **Eliminates redundancy:** reduces duplicate data by splitting tables and linking them through relationships.
- **Improves data integrity:** ensures data accuracy by enforcing consistent and reliable structures.
- **Follows a set of rules (Normal Forms):** helps prevent update, insert and delete anomalies by structuring data logically.
- **Optimizes storage:** minimizes unnecessary data storage by avoiding duplicated information.
- **Balances data integrity and performance:** while improving integrity, normalization also considers performance implications, especially in highly transactional systems.

Unnormalized Data (UNF)

Customer_ID	Customer_Name	Address	Product_1	Product_2	Product_3
1	John Doe	123 Elm St	Laptop	Phone	Headphones
2	Jane Smith	456 Oak St	Tablet		
3	Bob Johnson	789 Pine St	Laptop	Phone	
4	Alice Brown	321 Maple St	Headphones	Phone	Laptop

- Null values cannot be indexed
- It is very limiting, customers cannot buy more than 3 items

- **Repeating groups:** products are stored across multiple columns, violating the rule of atomicity.
- **Redundancy:** customer data is duplicated across rows when they buy multiple products.
- **Update anomalies:** if a customer address changes, it would need to be updated in every row where that customer appears.

- **Insertion Anomalies:** inserting a new product for a customer requires adding a new column if the product list grows.
- **Deletion Anomalies:** deleting a product might accidentally remove the customer data if all products for that customer are removed.

The Rule of Atomicity

- **Single Value per Cell:** each cell in a table should contain only one value, not a list or group of values.
- **No Multiple Values:** columns should not hold multiple pieces of information, but rather one piece of data per column.
- **Simplifies Queries:** ensuring atomic values makes querying and updating data easier and more efficient.
- **Avoids Data Redundancy:** atomic data reduces the need for repeating groups, preventing redundancy and ensuring data integrity.
- **Facilitates Accurate Updates:** having atomic values ensures that updates are made in a single place without affecting other data, reducing the risk of errors or anomalies.

What is Unnormalized Data?

- **Unstructured Data Format:** Unnormalized Data (UNF) refers to a database table that has not been organised according to any formal rules of normalization.
- **Contains Redundancy:** it often includes duplicated data and repeating groups, leading to inefficiency in storage and maintenance.
- **Lack of Atomicity:** columns may contain multiple values (non-atomic data), which violates the rule that each column should hold a single, indivisible value.
- **Difficult to Update:** updating unnormalized data can lead to inconsistencies, as the same data might be duplicated across multiple rows.
- **Resulting Issues:** the database is harder to manage, as data updates, deletions and inserts can affect multiple rows, leading to potential data

integrity problems.

First Normal Form (1NF)

Customer_ID	Customer_Name	Address	Product
1	John Doe	123 Elm St	Laptop
1	John Doe	123 Elm St	Phone
1	John Doe	123 Elm St	Headphones
2	Jane Smith	456 Oak St	Tablet
3	Bob Johnson	789 Pine St	Laptop
3	Bob Johnson	789 Pine St	Phone
4	Alice Brown	321 Maple St	Headphones
4	Alice Brown	321 Maple St	Phone
4	Alice Brown	321 Maple St	Laptop

- **Eliminate Repeating Groups:** tables must not have columns with repeating or similar groups of data.
- **Atomic Values:** each column must contain atomic (indivisible) values, meaning that cells cannot hold multiple values or lists.
- **Unique Rows:** every row in the table must be unique, typically achieved through a primary key that identifies each row.
- **Consistent Data Types:** all entries in a column must be of the same data type, ensuring that each column holds a specific kind of data.
- **Foundation of Normalization:** achieving 1NF is the first step in organizing data into a structure that reduces redundancy and ensures data integrity.
- **Simplifies Queries and Updates:** by adhering to 1NF, querying, updating and managing the database becomes easier and more efficient.

Key Changes

- **Repeating Groups Removed:** the products are no longer in separate columns, instead they are listed in separate rows.
- **Atomic Values:** each row contains one customer and one product, ensuring that all data values are atomic.
- **Unique Rows:** even though the customer appears multiple times, each row has a unique combination of *Customer_ID* and *Product*.

This 1NF structure eliminates redundancy and simplifies querying, updates and data integrity management.

Second Normal Form (2NF)

- **1NF Requirement:** the table must already meet the criteria for 1NF.

- **Remove Partial Dependencies:** all non-key attributes must depend on the entire primary key, not just part of it. This means the table should not have any columns that depend only on part of a composite key.
- **Applies to Composite Primary Keys:** 2NF is concerned with tables that have composite keys. If the table has a single-column primary key, it automatically satisfies 2NF.

Issues in the 1NF example

Partial Dependency: *Customer_Name* and *Address* depend only on *Customer_ID*, not the entire composite key (*Customer_ID, Product*). This violates 2NF because non-key attributes should depend on the entire composite key.

Transformation to 2NF

To achieve 2NF, we need to **split** the table into two, separating the customer details from the product purchase details. This removes the partial dependency.

Key Changes

Customers table

Customer_ID	Customer_Name	Address
1	John Doe	123 Elm St
2	Jane Smith	456 Oak St
3	Bob Johnson	789 Pine St

Product_Purchases table

Customer_ID	Product	Product_Price
1	Laptop	1000
1	Phone	500
1	Headphones	100
2	Tablet	700
3	Laptop	1000
3	Phone	500

- **Removed Partial Dependency:** *Customer_Name* and *Address* are now stored in a separate table and only depend on *Customer_ID*.
- **Product Data in a Separate Table:** product information and pricing, which depends on both *Customer_ID* and *Product*, is stored in the other table.
- **No Partial Dependencies:** each non-key attribute now depends on the entire primary key in its table.

Foreign Keys

- **Primary Key in the *Customers* Table:** *Customer_ID* is the primary key in the *Customers* table, uniquely identifying each customer.
- **Foreign Key in the *Product_Purchases* Table:** in the *Product_Purchases* table, *Customer_ID* references the *Customer_ID* in the *Customers* table. This establishes a relationship between the two tables, linking each product purchase to a specific customer.

- **Data Integrity:** ensures that every *Customer_ID* in the *Product_Purchases* table exists in the *Customers* table, maintaining referential integrity.
- **Relationship Link:** allows the two tables to be connected, so product purchases can be associated with the correct customer.
- **One-to-Many Relationship:** one customer can be associated with multiple product purchases, so there's a one-to-many relationship between the *Customers* table and the *Product_Purchases* table.

Functional Dependencies

A **functional dependency** occurs when the value of one attribute (or a set of attributes) uniquely determines the value of another attribute in a relation (table).

Notation: if attribute A determines attribute B, this is written as A → B. This means that for each unique value of A, there is exactly one corresponding value of B.

Example: if *Customer_ID* → *Customer_Name*, then knowing the *Customer_ID* allows to uniquely determine *Customer_Name.* (***Customer_ID*** is the determinant, ***Customer_Name*** is the dependent attribute).

Key Point: functional dependencies are fundamental to understanding how attributes relate to each other in a table, and they are crucial when defining keys in a table (primary keys, candidate keys, ...).

Transitive Dependencies

A **transitive dependency** is a specific type of functional dependency in which a non-key attribute depends on another non-key attribute, which in turn depends

on the primary key. It occurs when an attribute is indirectly dependent on the primary key through another attribute.

Condition for Transitive Dependency: if A → B and B → C, then A → C is a transitive dependency, B must not be a primary key or candidate key.

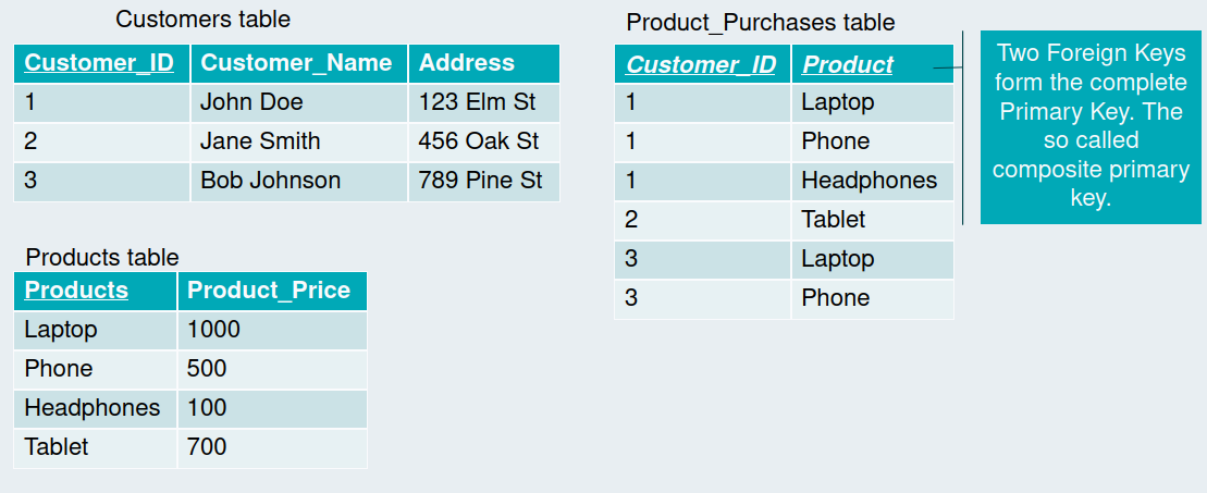
Key Point: transitive dependencies violate 3NF and must be eliminated by separating the table into multiple tables to avoid redundancy and update anomalies.

Third Normal Form (3NF)

- **Prerequisite:** the table must already be in 2NF.
- **Remove Transitive Dependencies:** all non-key columns must depend directly on the primary key. There should be no transitive dependencies, where a non-key column depends on another non-key column.
- **Direct Dependency on Primary Key:** every non-key attribute must depend on the primary key, not on any other non-key attributes. Non-key columns should not store information that can be derived from other non-key columns.

Purpose of 3NF

- **Improves Data Integrity:** ensures that the data structure is free from unnecessary redundancy and improves data integrity.



Surrogate Keys

A surrogate key is a database key that is not derived from the natural key of the table. Instead, it is a unique identifier that is generated by the database system. Surrogate keys are often used when the natural key is not unique or when it is subject to change.

Pros

- **Uniqueness:** guarantees a unique identifier for each record, independent of the actual data.
- **Consistency:** maintains consistency across related tables, as the key is system-generated and not subject to changes in actual attributes.
- **Simplifies Joins:** easier to use in joins, as surrogate keys are typically simple integers, making queries more efficient.
- **Immutable:** surrogate keys don’t change, even if the actual data in the table changes.
- **Avoids Complex Composite Keys:** prevents the need for composite key, which can complicate database structure and indexing.
- **Performance Benefits:** numeric keys, especially integers, are faster to process and compare than strings or multiple fields.

Cons

- **Less Meaningful:** surrogate keys have no real-world meaning, making it harder to understand the data context just by looking at the key.
- **Requires an Additional Unique Constraint:** the actual attributes still need unique constraints, so data integrity is enforced on meaningful data.
- **Increased Redundancy:** in some cases, surrogate keys may introduce redundancy, as you still need to maintain meaningful attributes that uniquely identify data.
- **May Obscure Logic:** when querying or debugging, surrogate keys don’t offer insight into the actual data, which can make understanding the relationships harder.
- **Unnecessary for Small Tables:** in smaller tables or where natural keys are stable, using surrogate keys may add unnecessary complexity.

When to use Surrogate Keys

Good Practice: when natural keys are prone to change, or the table is large and complex.

Not Always Necessary: when a natural key is stable, unique and meaningful, it can serve as a primary key.

In general, using a running number as a primary key (surrogate key) is considered a best practice for most scenarios, especially in large, complex databases, but it's not always the optimal solution for every situation.

Revision on Keys

- **Primary Key:** unique identifier for each record in a table. Ensures that no two rows have the same values for this key. Cannot contain NULL values.
- **Candidate Key:** attribute or set of attributes that can uniquely identify a record in a table. A table can have multiple candidate keys, and one is chosen as the primary key.
- **Super Key:** any set of attributes (one or more) that can uniquely identify a record in a table. Includes primary keys, candidate keys, and composite keys, but may include extra attributes beyond the minimum necessary for uniqueness.
- **Foreign Key:** key in one table that refers to the primary key in another table, creating a relationship between the two tables. Ensures referential integrity between the related tables.
- **Alternate Key:** candidate key that is not selected as the primary key. If a table has more than one candidate key, the ones not chosen as the primary key are alternate keys.
- **Surrogate Key:** artificial, system-generated key (usually an integer) used as the primary key. It has no business meaning and is typically used when no natural primary key exists or to simplify key management.
- **Composite Key:** primary key made up of two or more attributes that together uniquely identify a record. None of the individual attributes can uniquely identify the record on their own.
- **Compound Key:** similar to a composite key, but specifically refers to a key made up of multiple attributes, regardless of whether the individual

components are keys on their own, but at least one is. Often used interchangeably with composite key, but some distinctions exist in certain context.

A table can have **one primary key** and **one or more candidate keys**. Understanding whether individual attributes in a multi-attribute key are **candidate keys** (compound keys) might influence indexing decisions and query optimisation. If one of the attributes is a candidate key on its own, it may be useful for certain queries, and indexing it separately could improve performance.

If an attribute in a **compound key** is a key on its own, it may indicate that this attribute could serve as a **foreign key** in other tables. This can influence **referential integrity** and relationships in a database.

In a **composite key**, because no individual attribute is unique on its own, the database designer might need to ensure that the combination of attributes is enforced to avoid redundancy and ensure data integrity.

Whether one or more of those attributes could serve as a key elsewhere (compound) or not (composite) is often secondary. In practice, this terms are often used interchangeably.

Relevance of the Superkey

A **superkey** is important because it forms the basis for understanding how records are uniquely identified in a table. It plays a crucial role in database design, especially in establishing primary keys, candidate keys, and ensuring data integrity.

- Superkeys help identify candidate keys, and from these, a primary key is chosen. The primary key must be a minimal superkey.

A **superkey** ensures uniqueness in a table, no two rows in a table are identical. The concept of superkeys guarantees that uniqueness is enforced, preventing duplicate records.

- While only one superkey is needed to enforce uniqueness, other superkeys can still serve this purpose, ensuring redundant methods of unique identification.

Query Optimisation and Indexing: Knowing which attributes form superkeys helps in indexing decisions for query optimisation. Indexes are often created on superkeys, especially primary keys and candidate keys, to speed up lookups and joins.

Superkeys help in understanding the dependencies between attributes in a table. This becomes essential when designing **normalized databases** to minimise redundancy and anomalies. By analysing superkeys, a database designer can identify **functional dependencies** and design the database in such a way that unnecessary data duplication is avoided.

Boyce-Codd Normal Form (BCNF)

- **Stricter Form of 3NF:** BCNF is an advanced version of 3NF that deals with certain anomalies not covered by 3NF.
- **Candidate Key Requirement:** for every functional dependency (X → Y), the attribute X (determinant) must be a candidate key.
- **Eliminates Redundancy:** helps eliminate more subtle forms of redundancy that can exist in 3NF.
- **Anomaly Prevention:** prevents anomalies (update, insert and delete) in cases where functional dependencies cause issues in 3NF.
- **Use Case:** BCNF is usually applied when a table in 3NF still has non-trivial dependencies and further decomposition is needed to ensure all functional dependencies are based on candidate keys.

Why stop at 3NF?

- **Efficient Balance:** 3NF provides an efficient balance between eliminating data redundancy and maintaining performance in most practical databases.
- **Data Integrity:** by 3NF, most redundancies, update anomalies and data integrity issues are resolved. Further normalization often brings minimal benefits.
- **Performance Considerations:** beyond 3NF, further normalization (4NF, 5NF, ...) can lead to more fragmented data. This results in more complex joins, which can slow down query performance.

- **Increased Complexity:** normalizing beyond 3NF often results in a larger number of tables, making the database structure more complex to manage and query.
- **Diminishing Returns:** the incremental benefits of further normalization (such as removing rare anomalies) are typically not worth the cost of added complexity in real-world scenarios.
- **Denormalization for Performance:** in some cases, denormalization (intentionally not fully normalizing) is used to improve performance, especially in read-heavy systems like data warehouses.
- **Practical Applicability:** most real-world databases encounter and resolve common redundancy and integrity issues by 3NF or BCNF, so further normalization is often unnecessary and even counterproductive.
- **Trade-off Between Speed and Integrity:** after 3NF, the focus shifts more to performance optimization than integrity, as most integrity issues are addressed by this point.

Normalization in Data Warehouse Design

Denormalization is Preferred: in data warehouse design, denormalization is often used instead of strict normalization. This is because data warehouses prioritize read performance and efficient querying over data integrity.

Normalization in OLTP vs OLAP

- **Online Transaction Processing (OLTP)** systems (like regular operational databases) focus on maintaining data integrity and efficiency through normalization.
- **Online Analytical Processing (OLAP)** systems (data warehouses) focus on high-speed queries and reporting, where denormalized structures, like star and snowflake schemas, are common.

Common Mistakes of Normalization

- **Over-Normalization:** breaking down tables too much, creating excessive table fragmentation and requiring too many joins for simple queries, which

can hurt performance.

- **Under-Normalization:** leaving too much redundant data in the tables, leading to update, insert, and delete anomalies, as well as wasted storage.
- **Forgetting to Define Primary Keys:** not defining clear primary keys in tables, which is essential for uniquely identifying each record and ensuring data integrity.
- **Ignoring Business Logic:** focusing purely on theoretical normalization without considering the real-world usage of the data, resulting in an inefficient database design for business needs.
- **Not Accounting for Performance Needs:** failing to balance normalization with performance considerations, especially in read-heavy systems where denormalization might be more efficient.
- **Inappropriate Use of Composite Keys:** using composite keys unnecessarily, even when a single surrogate or natural key would suffice, leading to overly complex table structures.
- **Misidentifying Functional Dependencies:** incorrectly identifying which attributes depend on the primary key, resulting in a failure to properly normalize tables, often leaving partial or transitive dependencies in the design.
- **Normalization Without Referential Integrity:** normalizing the database but not enforcing foreign key constraints, which can lead to inconsistent data and orphaned records across related tables.
- **Failure to Consider Query Requirements:** ignoring the types of queries that will be run on the database, which may become inefficient or overly complex if the design is too normalized.
- **Normalizing at the Wrong Time:** attempting to fully normalize a database before understanding the full scope of the data and relationships, leading to early design mistakes that are difficult to correct later.

Normalization and SQL

Defining Tables with Primary Keys

SQL supports normalization by allowing you to define primary keys, ensuring each record in a table is uniquely identifiable.

```
create table customers (
    customer_id int primary key,
    customer_name varchar(50)
);
```

SQL uses **foreign keys** to enforce relationships between tables, which helps achieve higher normal forms by linking normalized tables.

```
foreign key (customer_id) references customers(customer_id);
```

SQL supports normalization with constraints like **primary key** and **foreign key** to enforce referential integrity between normalized tables.

```
alter table orders add constraint fk_customer
    foreign key (customer_id) references customers(customer_i
```

Normalization and Data Integrity

SQL constraints like **unique**, **not null** and **check** help ensure data integrity in a normalized database.

```
create table products (
    product_id int primary key,
    product_name varchar(50) not null,
    price decimal check (price > 0)
);
```

SQL allows you to create **indexes** to improve query performance in normalized databases, helping mitigate the performance hit from multiple table joins.

```
create index idx_customer on customers(customer_name);
```

Summary

Normalization Basics: normalization is a process used to organize data in a database, reducing redundancy and improving data integrity.

SQL constraints (primary keys, foreign keys, unique constraints) support normalization, ensuring data integrity and enforcing relationships between tables.

Summary Normal Forms

- **First Normal Form (1NF):** eliminate repeating groups and ensure each column hold **atomic** values (indivisible).
  - **Goal:** ensure every cell contains a single value, no lists or sets in a single column.
  - **Fix:** break rows with multiple values into separate rows.
- **Second Normal Form (2NF):** satisfy 1NF and eliminate **partial dependencies**.
  - **Goal:** ensure that every non-key attribute depends on the entire primary key (if composite).
  - **Fix:** split the table so each attribute depends on the full key.
- **Third Normal Form (3NF):** satisfy 2NF and eliminate **transitive dependencies**.
  - **Goal:** ensure non-key attributes depend only on the primary key and nothing else.
  - **Fix:** move transitive dependencies into a separate table.