Exploring Database Design: Normalization, Denormalization, Optimization & their Applications in OLTP and OLAP Systems

Pawan Kumar Ganjhu · Follow
12 min read · Jun 4, 2023

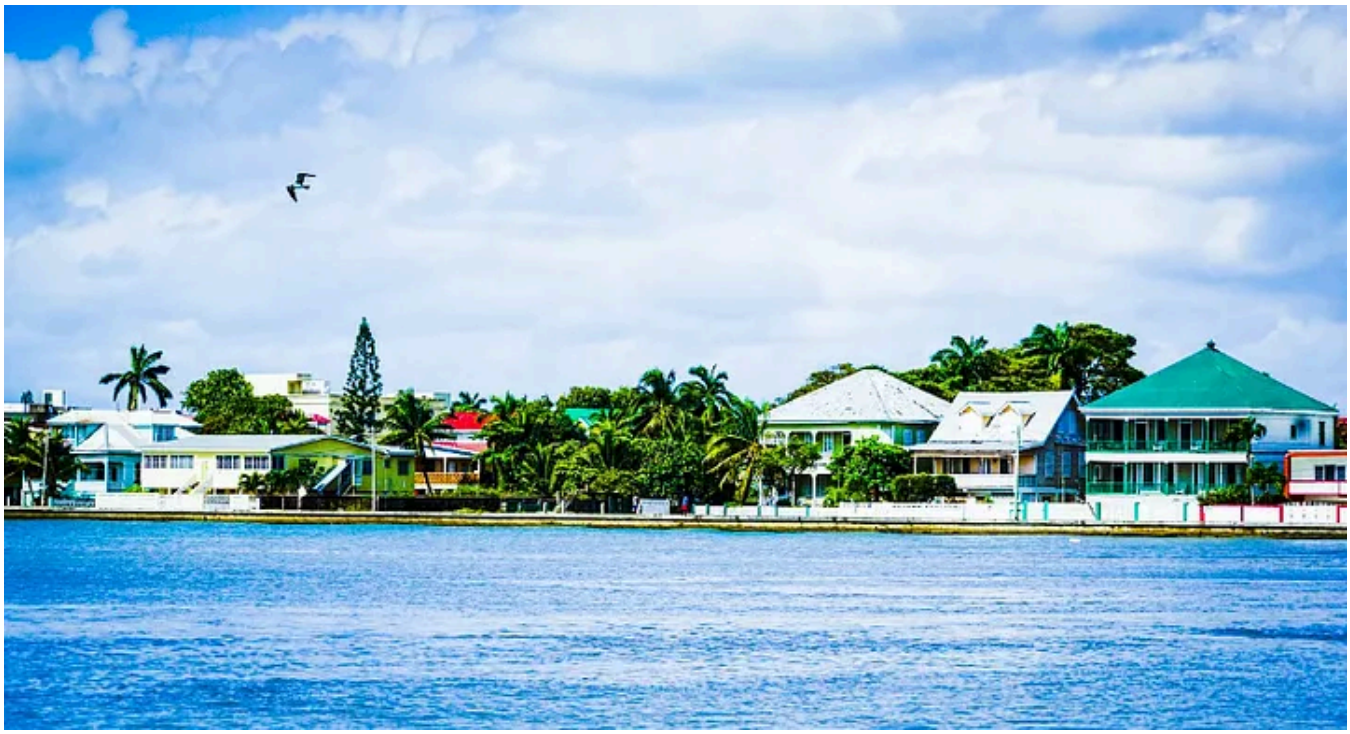

Image Source

Normalization and denormalization are techniques used in database design to optimize data storage and improve query performance. Let's understand these concepts by using a sample table and its data.

Consider a sample table called "Customers" with the following columns:

```
+----+-----------+--------------+--------+
| ID | FirstName | LastName     | Email  |
+----+-----------+--------------+--------+
| 1  | John      | Doe          |        |
| 2  | Jane      | Smith        |        |
| 3  | David     | Anderson     |        |
+----+-----------+--------------+--------+
```

This table represents customer data, where each row corresponds to a unique customer and the columns store specific information about them.

Normalization: Normalization is a process that organizes data into separate tables to reduce redundancy and dependency. The goal is to eliminate data duplication and ensure data integrity. Normalization is achieved by breaking down the table into multiple smaller tables based on the relationships between the data.

In our example, we can normalize the “Customers” table into two tables: “CustomerInfo” and “CustomerEmail.”

```
CustomerInfo table:
+----+-----------+--------------+
| ID | FirstName | LastName     |
+----+-----------+--------------+
| 1  | John      | Doe          |
| 2  | Jane      | Smith        |
| 3  | David     | Anderson     |
+----+-----------+--------------+

CustomerEmail table:
+----+------------------+
| ID | Email            |
+----+------------------+
| 1  | john@example.com |
| 2  | jane@example.com |
| 3  | david@example.com|
+----+------------------+
```

In this normalized form, the customer’s personal information is stored in the “CustomerInfo” table, and the email addresses are stored in the “CustomerEmail” table. The common field “ID” acts as a primary key in the “CustomerInfo” table and

a foreign key in the “CustomerEmail” table to establish the relationship between the tables.

Denormalization: Denormalization, on the other hand, involves combining normalized tables to improve performance by reducing the number of joins required in queries. It sacrifices some of the normalization benefits in favor of faster query execution.

Using the normalized tables from the previous example, we can denormalize them back into a single table:

```
+----+-----------+--------------+------------------+
| ID | FirstName | LastName     | Email            |
+----+-----------+--------------+------------------+
| 1  | John      | Doe          | john@example.com |
| 2  | Jane      | Smith        | jane@example.com |
| 3  | David     | Anderson     | david@example.com|
+----+-----------+--------------+------------------+
```

In this denormalized form, all the information is stored in a single table. This eliminates the need for joining multiple tables when querying the data, which can improve query performance.

It’s important to note that denormalization should be used judiciously, considering the trade-off between performance and data integrity. It may be suitable for read-intensive applications or situations where the overhead of joins outweighs the benefits of normalization. However, in write-intensive applications, maintaining data consistency might be more critical, and normalization should be preferred.

Overall, normalization and denormalization are techniques used to optimize data storage and query performance based on the specific requirements of the application.

Normalization & Types

Normalization is a process in database design that aims to eliminate redundancy and dependency by organizing data into separate tables. There are several normal

forms, each representing a different level of normalization. Let's explore the different types of normalization with examples and their corresponding output.

1. First Normal Form (1NF): First Normal Form ensures that each column in a table contains only atomic values, meaning that there are no repeating groups or arrays.

Example: Consider a table called "Students" with the following columns:

```
+------+----------------------+
| ID   | Courses              |
+------+----------------------+
| 1    | Math, Science, Music |
| 2    | English, History     |
+------+----------------------+
```

In this example, the "Courses" column contains multiple values separated by commas. To bring this table to 1NF, we need to separate the courses into individual rows:

```
+------+---------+
| ID   | Courses |
+------+---------+
| 1    | Math    |
| 1    | Science |
| 1    | Music   |
| 2    | English |
| 2    | History |
+------+---------+
```

2. Second Normal Form (2NF): Second Normal Form requires that each non-key column is fully dependent on the primary key, meaning that no partial dependencies exist.

Example: Consider a table called "Orders" with the following columns:

```
+------+---------+-------+----------+
| ID   | Product | Price | Category |
+------+---------+-------+----------+
| 1    | Apple   | 1.99  | Fruits   |
| 1    | Orange  | 0.99  | Fruits   |
| 2    | Shirt   | 19.99 | Clothing |
| 2    | Pants   | 29.99 | Clothing |
+------+---------+-------+----------+
```

In this example, the “Category” column depends only on the “ID” column, and not on the “Product” column. To achieve 2NF, we split the table into two tables, ensuring that each non-key column depends on the entire primary key:

```
Orders table:
+------+---------+-------+
| ID   | Product | Price |
+------+---------+-------+
| 1    | Apple   | 1.99  |
| 1    | Orange  | 0.99  |
| 2    | Shirt   | 19.99 |
| 2    | Pants   | 29.99 |
+------+---------+-------+

Categories table:
+------+----------+
| ID   | Category |
+------+----------+
| 1    | Fruits   |
| 2    | Clothing |
+------+----------+
```

3. Third Normal Form (3NF): Third Normal Form states that no transitive dependencies should exist, where a non-key column depends on another non-key column.

Example: Consider a table called “Books” with the following columns:

```
+------+------+-------------+---------+
| ID   | Name | Author      | Country |
+------+------+-------------+---------+
```

```
| 1    | Book1| Author1     | USA     |
| 2    | Book2| Author2     | UK      |
| 3    | Book3| Author3     | USA     |
+------+------+-------------+---------+
```

In this example, the “Country” column depends on the “Author” column, which is not the primary key. To achieve 3NF, we split the table into two tables, removing the transitive dependency:

```
Books table:
+------+------+-------------+
| ID   | Name | Author      |
+------+------+-------------+
| 1    | Book1| Author1     |
| 2    | Book2| Author2     |
| 3    | Book3| Author3     |
+------+------+-------------+

Authors table:
+---------+---------+
| Author  | Country |
+---------+---------+
| Author1 | USA     |
| Author2 | UK      |
| Author3 | USA     |
+---------+---------+
```

These are the three most commonly used normalization forms (1NF, 2NF, and 3NF) in database design. Higher normal forms, such as Boyce-Codd Normal Form (BCNF) and Fourth Normal Form (4NF), exist as well, and they further refine the normalization process by eliminating additional anomalies and dependencies.

Boyce-Codd Normal Form (BCNF) and Fourth Normal Form (4NF)

Let’s explore Boyce-Codd Normal Form (BCNF) and Fourth Normal Form (4NF) with examples and their corresponding output.

1. Boyce-Codd Normal Form (BCNF): BCNF is a higher level of normalization that addresses functional dependencies within a table. It ensures that there are no

non-trivial dependencies of non-key attributes on the candidate keys.

Example: Consider a table called “Employees” with the following columns:

```
+------+-----------+-----------+
| ID   | Dept      | ManagerID |
+------+-----------+-----------+
| 1    | Finance   | 2         |
| 2    | HR        | 1         |
| 3    | Finance   | 1         |
+------+-----------+-----------+
```

In this example, the “ManagerID” column depends on the “ID” column, which is not a candidate key. To bring this table to BCNF, we split it into two tables, removing the dependency:

```
Employees table:
+------+-----------+
| ID   | Dept      |
+------+-----------+
| 1    | Finance   |
| 2    | HR        |
| 3    | Finance   |
+------+-----------+

Managers table:
+------+-----------+
| ID   | ManagerID |
+------+-----------+
| 1    | 2         |
| 2    | 1         |
| 3    | 1         |
+------+-----------+
```

In the BCNF form, the “ManagerID” column is now placed in a separate table, eliminating the functional dependency.

2. Fourth Normal Form (4NF): Fourth Normal Form aims to address multi-valued dependencies between attributes in a table. It ensures that there are no non-trivial multi-valued dependencies.

Example: Consider a table called “Products” with the following columns:

```
+------+--------+---------------------+
| ID   | Name   | Attributes          |
+------+--------+---------------------+
| 1    | Phone  | Color: Black, White |
| 2    | Laptop | Color: Silver       |
+------+--------+---------------------+
```

In this example, the “Attributes” column contains multiple values, representing different attributes of the products. To achieve 4NF, we split the table into three tables, eliminating the multi-valued dependency:

```
Products table:
+------+--------+
| ID   | Name   |
+------+--------+
| 1    | Phone  |
| 2    | Laptop |
+------+--------+

Attributes table:
+------+-----------+
| ID   | Attribute |
+------+-----------+
| 1    | Color     |
+------+-----------+

ProductAttributes table:
+------+-------+-------------+
| ID   | Value | AttributeID |
+------+-------+-------------+
| 1    | Black | 1           |
| 1    | White | 1           |
| 2    | Silver| 1           |
+------+-------+-------------+
```

In the 4NF form, the “Attributes” column is extracted into a separate table, and the multi-valued dependencies are stored in the “ProductAttributes” table.

BCNF and 4NF are advanced levels of normalization that further refine the database design by eliminating additional anomalies and dependencies. These normal forms help ensure data integrity, reduce redundancy, and improve the efficiency of database operations.

Denormalization & Types

Denormalization is the process of combining normalized tables into fewer tables or even a single table to improve query performance by reducing the need for joins. Here are some common types of denormalization along with examples and their corresponding output:

1. Flattening Denormalization: Flattening denormalization involves combining multiple related tables into a single denormalized table. This reduces the need for joins and simplifies queries.

Example: Consider the following normalized tables: “Customers” and “Orders.”

Customers table:

```
+----+-----------+--------------+
| ID | FirstName | LastName     |
+----+-----------+--------------+
| 1  | John      | Doe          |
| 2  | Jane      | Smith        |
+----+-----------+--------------+
```

Orders table:

```
+-----+--------+------------+
| ID  | Amount | Date       |
+-----+--------+------------+
| 101 | 50     | 2022-01-01 |
| 102 | 75     | 2022-02-01 |
+-----+--------+------------+
```

By denormalizing these tables, we can combine them into a single table, “CustomersOrders,” with all the relevant information:

CustomersOrders table:

```
+----+-----------+--------------+--------+--------+------------+
| ID | FirstName | LastName     | OrderID | Amount | Date       |
+----+-----------+--------------+--------+--------+------------+
| 1  | John      | Doe          | 101    | 50     | 2022-01-01 |
| 1  | John      | Doe          | 102    | 75     | 2022-02-01 |
| 2  | Jane      | Smith        | NULL   | NULL   | NULL       |
+----+-----------+--------------+--------+--------+------------+
```

In this denormalized form, the “CustomersOrders” table includes all the customer and order information, even if a customer has no orders. It eliminates the need for joins when querying customer and order details together.

2. Column Denormalization: Column denormalization involves duplicating data from related tables into a single table, reducing the need for joins and improving query performance.

Example: Consider the following normalized tables: “Customers” and “Orders.”

Customers table:

```
+----+-----------+--------------+
| ID | FirstName | LastName     |
+----+-----------+--------------+
| 1  | John      | Doe          |
| 2  | Jane      | Smith        |
+----+-----------+--------------+
```

Orders table:

```
+-----+--------+------------+
| ID  | Amount | Date       |
+-----+--------+------------+
```

```
| 101 | 50     | 2022-01-01 |
| 102 | 75     | 2022-02-01 |
+-----+--------+------------+
```

By denormalizing these tables, we can duplicate the customer information into the “Orders” table:

Denormalized Orders table:

```
+-----+-----------+--------------+--------+--------+---------+
| ID  | FirstName | LastName     | Amount | Date             |
+-----+-----------+--------------+--------+--------+---------+
| 101 | John      | Doe          | 50     | 2022-01-01       |
| 102 | John      | Doe          | 75     | 2022-02-01       |
+-----+-----------+--------------+--------+--------+---------+
```

In this denormalized form, the customer information is duplicated in the “Orders” table, avoiding the need for joins when retrieving customer details along with order information.

3. Application-Based Denormalization: Application-based denormalization involves denormalizing the data based on the specific requirements of an application. It optimizes performance by precalculating and storing derived or aggregated data.

Example: Consider a normalized table called “Sales” with the following columns:

```
+------+-------+--------+------------+
| Year | Month | Amount | Product    |
+------+-------+--------+------------+
| 2022 | Jan   | 100    | Product A  |
| 2022 | Jan   | 150    | Product B  |
| 2022 | Feb   | 200    | Product A  |
| 2022 | Feb   | 100    | Product B  |
+------+-------+--------+------------+
```

Suppose the application frequently needs to retrieve the total sales amount per month. In that case, we can denormalize the data and create a summary table with

precalculated values:

Denormalized SalesSummary table:

```
+------+-------+--------------+
| Year | Month | TotalAmount  |
+------+-------+--------------+
| 2022 | Jan   | 250          |
| 2022 | Feb   | 300          |
+------+-------+--------------+
```

In this denormalized form, the “SalesSummary” table stores the aggregated total sales amount per month, eliminating the need for recalculating it every time a query is executed.

Denormalization is a technique used to optimize database performance in scenarios where read operations are more frequent or time-sensitive. However, it should be used judiciously, considering the trade-off between query performance and data redundancy.

Choosing the Right Database Structure: Normalized Tables and Denormalized Tables in OLTP and OLAP Systems

Normalized tables and denormalized tables have different use cases and are often associated with different types of database systems, such as OLTP (Online Transaction Processing) and OLAP (Online Analytical Processing). Let’s explore when to use each type:

1.Normalized Tables (OLTP): Normalized tables are commonly used in OLTP systems, which are designed for transactional processing and handling day-to-day operations. OLTP systems focus on efficient data modification, such as inserting, updating, and deleting records. The goal is to maintain data integrity and minimize redundancy.

Benefits of normalized tables in OLTP systems:

- Reduced data redundancy: Normalization eliminates redundant data, ensuring consistency and reducing storage requirements.
- Data integrity: Normalization helps maintain data integrity by enforcing referential integrity constraints and avoiding update anomalies.
- Efficient data modification: Normalized tables provide a structured and efficient way to handle frequent data modifications in OLTP systems.

Example: Consider an OLTP system for an e-commerce website. The normalized tables could include “Customers,” “Orders,” “Products,” and “ShippingDetails.” These tables would be designed to efficiently handle customer registrations, order processing, inventory management, and other transactional operations.

2. Denormalized Tables (OLAP): Denormalized tables are commonly used in OLAP systems, which are designed for analytical processing and complex queries. OLAP systems focus on aggregating and analyzing large volumes of data to support decision-making and reporting.

Benefits of denormalized tables in OLAP systems:

- Improved query performance: Denormalization reduces the need for joins and allows for faster query execution, particularly when dealing with complex analytical queries involving aggregations and multi-table joins.
- Simplified data model: Denormalized tables provide a flattened structure that simplifies querying and reporting, as all relevant data is consolidated in a single table.
- Better user experience: OLAP systems often involve interactive querying and reporting, and denormalized tables can provide faster response times, enhancing the user experience.

Example: In an OLAP system for business intelligence, denormalized tables can be used to store aggregated data, such as sales summaries, customer segmentation, or financial metrics. These tables would be optimized for query performance and would often involve data from multiple sources.

It’s important to note that the decision to use normalized or denormalized tables depends on the specific requirements of the system and the trade-offs between data

consistency, query performance, and storage. In some cases, a hybrid approach may be employed, where a combination of normalized and denormalized tables is used to achieve the desired balance.

In this discussion, we explored the concepts of normalization and denormalization in the context of database design. Normalization involves organizing data into separate tables to eliminate redundancy and dependency, while denormalization focuses on combining or duplicating data to improve query performance. We discussed different levels of normalization, such as 1NF, 2NF, and 3NF, as well as advanced forms like BCNF and 4NF.

Normalization ensures data integrity, reduces redundancy, and is typically used in OLTP systems that handle transactional operations. On the other hand, denormalization optimizes query performance, simplifies data models, and is often applied in OLAP systems for analytical processing.

By understanding the trade-offs between normalized and denormalized tables, we can make informed decisions when designing databases based on the specific requirements of the system. Choosing the appropriate approach depends on factors like data modification frequency, query complexity, and the need for real-time transactional processing versus analytical reporting.

Ultimately, an effective database design strikes a balance between normalization and denormalization, aiming to achieve data consistency, efficiency, and optimal performance.

Normalization | Denormalization | Optimization | Oltp | Olap

Follow

Written by Pawan Kumar Ganjhu