Relational Data Models

Wednesday, April 29, 2020

12:17 AM

Importance of Relational Databases:

* **Standardization of data model:**Once your data is transformed into the rows and columns format, your data is standardized, and you can query it with SQL
* **Flexibility in adding and altering tables:**Relational databases gives you flexibility to add tables, alter tables, add and remove data.
* **Data Integrity:**Data Integrity is the backbone of using a relational database.
* **Structured Query Language (SQL):**A standard language can be used to access the data with a predefined language.
* **Simplicity:** Data is systematically stored and modeled in tabular format.
* **Intuitive Organization:**The spreadsheet format is intuitive but intuitive to data modeling in relational databases.

OLAP VS OLTP

**Online Analytical Processing (OLAP):**

Databases optimized for these workloads allow for complex analytical and ad hoc queries, including aggregations. These types of databases are optimized for reads.

**Online Transactional Processing (OLTP):**

Databases optimized for these workloads allow for less complex queries in large volume. The types of queries for these databases are read, insert, update, and delete.

Structuring the Database:

Objectives of Normal Form:

1. To free the database from unwanted insertions, updates, & deletion dependencies
2. To reduce the need for refactoring the database as new types of data are introduced
3. To make the relational model more informative to users
4. To make the database neutral to the query statistics

Normal Forms

1. How to reach First Normal Form (1NF):
   * Atomic values: each cell contains unique and single values
   * Be able to add data without altering tables
   * Separate different relations into different tables
   * Keep relationships between tables together with foreign keys
2. Second Normal Form (2NF):
   * Have reached 1NF
   * All columns in the table must rely on the Primary Key
3. Third Normal Form (3NF):
   * Must be in 2nd Normal Form
   * No transitive dependencies
   * Remember, transitive dependencies you are trying to maintain is that to get from A-> C, you want to avoid going through B.  
     When to use 3NF:
   * When you want to update data, we want to be able to do in just 1 place. We want to avoid updating the table in the Customers Detail table (in the example in the lecture slide).

Normalization

* The process of trying to improve the read performance of a database at the expense of losing some write performance by adding redundant copies of data

Logical Design Change

* The Designer is in charge of keeping data consistent
* Reads will be faster(select)
* Writes will be slower (insert, update, delete)

**Key Concepts:**

**Normalization** is about trying to increase data integrity by reducing the number of copies of the data. Data that needs to be added or updated will be done in as few places as possible.

**Denormalization** is trying to increase performance by reducing the number of joins between tables (as joins can be slow). Data integrity will take a bit of a potential hit, as there will be more copies of the data (to reduce JOINS).

**Fact and Dimension Tables**

Fact Tables

* Fact table consists of the measurements, metrics or facts of a business process.

Dimension

* A structure that categorizes facts and measures in order to enable users to answer business questions. Dimensions are people, products, place and time.

**Star Schema**

A Star Schema is the simplest style of data mart schema. It consists of one or more fact tables referencing any number of dimension tables.

A screenshot of a cell phone

Description automatically generated

Benefits:

* Denormalized
* Simplified queries
* Fast Aggregations

Drawbacks

* Issues that come with denormalization
* Data Integrity
* Decrease query flexibility
* Many to many relationship – simplified

**Snowflake Schema**

Logical Arrangement of tables in a multidimensional database represented by centralized fact tables which are connected to multiple dimensions.

**PSQL commands**

**Data Definition and Constraints**

The CREATE statement in SQL has a few important constraints that are highlighted below.

**NOT NULL**

The **NOT NULL** constraint indicates that the column cannot contain a null value.

Here is the syntax for adding a NOT NULL constraint to the CREATE statement:

CREATE TABLE IF NOT EXISTS customer\_transactions (

customer\_id int NOT NULL,

store\_id int,

spent numeric

);

You can add **NOT NULL** constraints to more than one column. Usually this occurs when you have a **COMPOSITE KEY**, which will be discussed further below.

Here is the syntax for it:

CREATE TABLE IF NOT EXISTS customer\_transactions (

customer\_id int NOT NULL,

store\_id int NOT NULL,

spent numeric

);

**UNIQUE**

The **UNIQUE** constraint is used to specify that the data across all the rows in one column are unique within the table. The **UNIQUE** constraint can also be used for multiple columns, so that the combination of the values across those columns will be unique within the table. In this latter case, the values within 1 column do not need to be unique.  
  
Let's look at an example.

CREATE TABLE IF NOT EXISTS customer\_transactions (

customer\_id int NOT NULL UNIQUE,

store\_id int NOT NULL UNIQUE,

spent numeric

);

Another way to write a **UNIQUE** constraint is to add a table constraint using commas to separate the columns.

CREATE TABLE IF NOT EXISTS customer\_transactions (

customer\_id int NOT NULL,

store\_id int NOT NULL,

spent numeric,

UNIQUE (customer\_id, store\_id, spent)

);

**PRIMARY KEY**

The **PRIMARY KEY** constraint is defined on a single column, and every table should contain a primary key. The values in this column uniquely identify the rows in the table. If a group of columns are defined as a primary key, they are called a **composite key**. That means the combination of values in these columns will uniquely identify the rows in the table. By default, the **PRIMARY KEY** constraint has the unique and not null constraint built into it.  
  
Let's look at the following example:

CREATE TABLE IF NOT EXISTS store (

store\_id int PRIMARY KEY,

store\_location\_city text,

store\_location\_state text

);

Here is an example for a group of columns serving as **composite key**.

CREATE TABLE IF NOT EXISTS customer\_transactions (

customer\_id int,

store\_id int,

spent numeric,

PRIMARY KEY (customer\_id, store\_id)

);

**Upsert**

In RDBMS language, the term *upsert* refers to the idea of inserting a new row in an existing table or updating the row if it already exists in the table. The action of updating or inserting has been described as "upsert".

The way this is handled in PostgreSQL is by using the INSERT statement in combination with the ON CONFLICT clause.

**INSERT**

The **INSERT** statement adds in new rows within the table. The values associated with specific target columns can be added in any order.

Let's look at a simple example. We will use a customer address table as an example, which is defined with the following **CREATE** statement:

CREATE TABLE IF NOT EXISTS customer\_address (

customer\_id int PRIMARY KEY,

customer\_street varchar NOT NULL,

customer\_city text NOT NULL,

customer\_state text NOT NULL

);

Let's try to insert data into it by adding a new row:

INSERT into customer\_address (

VALUES

(432, '758 Main Street', 'Chicago', 'IL'

);

Now let's assume that the customer moved, and we need to update the customer's address. However, we do not want to add a new customer id. In other words, if there is any conflict on the customer\_id, we do not want that to change.

This would be a good candidate for using the **ON CONFLICT DO NOTHING** clause.

INSERT INTO customer\_address (customer\_id, customer\_street, customer\_city, customer\_state)

VALUES

(

432, '923 Knox Street', 'Albany', 'NY'

)

ON CONFLICT (customer\_id)

DO NOTHING;

Now, let's imagine we want to add more details in the existing address for an existing customer. This would be a good candidate for using the **ON CONFLICT DO UPDATE** clause.

INSERT INTO customer\_address (customer\_id, customer\_street)

VALUES

(

432, '923 Knox Street, Suite 1'

)

ON CONFLICT (customer\_id)

DO UPDATE

SET customer\_street = EXCLUDED.customer\_street;

We recommend checking out these two links to learn other ways to insert data into the tables.