Normalizing a Database

Database Normalization

Database normalization is a process by which database and table structures are created or modified in order to address inefficiencies/complexities related to the following:

- data storage
- data modification
- querying database tables

Repeating Column Groups

Repeating groups of columns in a database table can create inefficiencies and errors related to data storage, querying, and modification. For example, consider a `songs` table with the following columns:

- `id`
- `title`
- `artist1_id`
- `artist1_name`
- `artist2_id`
- `artist2_name`

The repeating artist-related columns likely contain duplicated data. It would also be difficult to sort this table by artist.

Independent Columns

In a relational database, columns that are not dependent on the primary key of a table can create inefficiencies related to data storage and modification, while also increasing the potential for future data errors. This is often because columns that are not dependent on the primary key contain duplicated information. For example, in a `books` table with columns `isbn`, `title`, `length`, `author_id`, and `author_name`, the author-related columns will contain duplicated data if the same author has written multiple books; moving author-related information to a separate table would solve this problem.

Updating Duplicated Data

If the same information is stored in multiple locations in a database table, a database manager needs to be careful when updating the table. For example, in the database table shown here, each customer's email address is stored in multiple rows. Therefore, in order to update a customer email, multiple fields will need to be changed. Normalizing the table gets rid of duplicated data and therefore makes data errors less likely.

order_id	price	cust_id	cust_email
1	20.43	1	hello@email.com
2	51.33	1	hello@email.com
3	80.01	2	dbeng@email.com
4	33.27	2	dbeng@email.com

Inserting Data Without a Primary Key

Problems can occur when updating a database table if new information needs to be inserted before the associated primary key is known. This can happen if columns are not dependent on the primary key. For example, consider a `songs` table with the following columns:

- `song_id` (primary key)
- `title`
- `length`
- `artist_id`
- `artist_name`

It would be impossible to add artist information to this table without also adding a value for `song_id` , which could be problematic.

Database Efficiency and Use

The efficiency of any database schema is dependent on how the database is going to be used. While normalization solves many problems related to data storage, modification, and querying, it can also make some things more difficult. For example, tables in a normalized database will need to be joined back together if a query relies on information in multiple tables. It is therefore not always beneficial to normalize every database table. Decisions about schema design should be made with future use in mind!

1NF Databases

A 1NF database is an atomic database. In this case, atomic means that each cell contains one value and each row is unique. In the given example, we can see that the non-atomic table has cells with more than one value and non-unique rows.

```
// This would be a non-atomic table
| Title            | Length      | Type |
|------------------|-------------|------|
| Example A, B, C  | 125 inches  | B, C |
| Example A, B, C  | 125 inches  | B, C |
// This would be an atomic table
| ID | Title      | Length      | Type |
|----|------------|-------------|------|
| 1  | Example A  | 125 inches  | B    |
| 2  | Example A  | 125 inches  | C    |
```

A 2NF Database

When a database is said to be 2NF, that means the database is both 1NF and contains no partial dependencies. A partial dependency is when an attribute depends partly on the table's primary key.

```
// A 1NF Database with partial
dependencies
| ID | Name     | Address ID | Address
|
|----|----------|------------|-------------
----------|
| 1  | Logan    | 1          | 7777 Willow
Drive     |
| 2  | Charlie  | 2          | 8888 Blue
Bonnet Road |
| 3  | Johanna  | 1          | 7777 Willow
Drive     |
// A 2NF database
*Name.db
| ID | Name     | Address ID |
|----|----------|------------|
| 1  | Logan    | 1          |
| 2  | Charlie  | 2          |
| 3  | Johanna  | 1          |
*Address.db
| Address ID | Address               |
|------------|-----------------------|
| 1          | 7777 Willow Drive     |
| 2          | 8888 Blue Bonnet Road |
```

A 3NF database

When making a 3NF database, two goals need to be accomplished. The first being that the database is already 2NF, and the second being that the database contains no transitive functional dependencies. A transitive functional dependency is when a non-prime attribute is dependent on another non-prime attribute.

```
// A 2NF database with transitive
functional dependency.
| ID | Name    | Address                |
|----|---------|------------------------|
| 1  | Logan   | 7777 Willow Drive      |
| 2  | Charlie | 4444 Blue Bonnet Road |
| 4  | Hannah  | 4444 Blue Bonnet Road |
// A 3NF database with no transitive
functional dependencies.
*name.db
| ID | Name    | Address ID |
|----|---------|------------|
| 1  | Logan   | 1          |
| 2  | Charlie | 2          |
| 4  | Hannah  | 2          |
*Address.db
| Address ID | Address               |
|------------|-----------------------|
| 1          | 7777 Willow Drive     |
| 2          | 4444 Blue Bonnet Road |
```

Update Anomalies

When updating data in a non-normalized database, sometimes not all of the data can get updated due to the lack of normalization. Another possible problem can be updating the wrong data. This is called an update anomaly and can be fixed by making sure the database has a higher level of normalization.

```
| ID | Name   | City          | Total
Sales |
|----|--------|---------------|-----------
--|
| 1  | Arthur | New York City | $26,000
|
| 2  | Sarah  | Cincinnati    | NULL
|
| 3  | Josh   | Cincinnati    | $20,000
|
// In the table above, if we updated the
sales of an employee in cincinnati without
specifics, we would end up updating either
both of the sales for Sarah or Josh, or by
possibly updating the wrong employee.
```

Insertion Anomalies

Sometimes, when working with a non-normalized database, incomplete data being added to the database can lead to NULL values existing within the database. This is called an insertion anomaly and can be prevented by making sure the database has a higher level of normalization.

```
| ID | Name   | City          | Total Sales |
|----|--------|---------------|-------------|
| 1  | Arthur | New York City | $26,000     |
| 2  | Sarah  | Cincinnati    | NULL        |
| 3  | Josh   | Los Angeles   | $20,000     |
// In the table above, new employees won't have any sales yet meaning that there will be a number of NULL values in the database until a set number exists.
```

Deletion Anomalies

When working with a non-normalized database, a deletion anomaly can occur. A deletion anomaly is when a query ends up deleting more data from the database than was intended due to a lack of normalization.

Print Share ▾