SQL: An Introduction

Clayton W. Schupp, Galvanize

June 2015

Data Persistence

Persistence is the continuance of an effect after its cause is removed

In context of storing data, this means that data

- survives after the process with which it was created has ended
- is written to non-volatile storage

Relational Database Management Systems (RDBMS)

An RDBMS provides the ability to

- model relations in data
- query data and their relations efficiently
- maintain data consistency and integrity

RBDMS Data Model

RDBMS have a **schema** that defines the structure of the data

- a database is composed of a number of user-defined **tables**, each with **columns** and **rows**
- a column is of a certain **data type** such as integer, string, or date
- a row is an entry in a table with data for each column of that table

Database Table Example

```
CREATE TABLE users {
  id INTEGER PRIMARY KEY,
  name VARCHAR(255),
  age INTEGER,
  city VARCHAR(255),
  state VARCHAR(2)
}
```

The data types available vary from system to system. The above is for PostgreSQL where **VARCHAR** is a string data type.

Primary Key

A primary key is a special column of a table that uniquely identifies that entry.

Here's an example from our users table:

id	name	age	city	state
5	Amy	33	Los..	CA
9	Zed	88	Mia..	FL
4	Ned	89	Mia..	FL
3	John	46	San..	CA
...				

Foreign Keys

Part of the power of RDBMS are their ability to model relations in data using **foreign keys** which is a column that references some other entry in a database. That foreign key entry could be in the same table or in some other table.

Example:

```
CREATE TABLE visits {
  id INTEGER PRIMARY KEY,
  created_at TIMESTAMP,
  user_id INTEGER REFERENCES users(id)
}
```

One-to-One

For a one-to-one relationship, place the foreign key on either side of the relationship

Example: User and Subscription

```
---subscriptions
user_id INTEGER
```

To find the user for a particular subscription:

```
SELECT *
FROM users
WHERE id=?
LIMIT 1
```

One-to-One

For a one-to-one relationship, place the foreign key on either side of the relationship

Example: User and Subscription

```
---subscriptions
user_id INTEGER
```

To find the subscription for a particular user:

```
SELECT *
FROM subscriptions
WHERE user_id=?
LIMIT 1
```

One-to-Many

For a one-to-many relationship, place the foreign key on the many side of the relationship

Example: User and Posts

```
---posts
user_id INTEGER REFERENCES users(id)
```

To find the posts for a particular user:

```
SELECT *
FROM posts
WHERE user_id=?
```

Many-to-Many

For a many-to-many relationship, create a table that contains two foreign keys, one to each side of the relationship.

Example: Posts and Tags

```
---post_tags
post_id INTEGER REFERENCES posts(id)
tag_id INTEGER REFERENCES tags(id)
```

To find the tags for a particular post:

```
SELECT tags.*
FROM tags
JOIN post_tags
  ON post_tags.tag_id = tags.id
WHERE post_tags.post_id=?
```

Schema Normalization

When designing a database schema, we aim to minimize redundancy.

This comes into play with relations between data.

You want to use a simple foreign key (i.e. the smallest piece of information that you can keep about another entry)

Example: Posts and Users

In a fully normalized schema, the post entry would have a **user_id** which would relate the post to a user. A denormalized way would be to have a column in the **posts** table called **author_name** which would duplicate the **name** column in the **users** table.

Structured Query Language (SQL)

SQL is the language used to query relational databases.

All RDBMS use SQL and the syntax and keywords are for the most part the same across systems.

SQL is used to interact with RDBMS, allowing you to create tables, alter tables, insert records, update records, delete records, and query records within and across tables

SQL Queries

SQL is a declarative language, meaning the query describes the set of results

```
SELECT name, age
FROM users
```

This query returns name and age for every user in the **users** table

```
SELECT name, age
FROM users
WHERE state = 'CA'
```

This query returns name and age for every user in the **users** table who also live in CA

SQL Queries

SQL queries are composed of **clauses** and each clause begins with a **keyword**.

Every query begins with the **SELECT** clause followed by the **FROM** and **JOIN** clauses.

You then have the ability to apply filtering, aggregation, and ordering clauses.

SELECT specifies which columns should be returned.

Continue in Afternoon...lets now look at a general SQL style guide and some example queries

JOIN

The **JOIN** clause enables querying based on relations in the data by making use of foreign keys.

Every **JOIN** clause has two segments:

- specifying the tables to join
- specifying the columns to match up

There are different types of **JOIN** clauses:
INNER JOIN , **LEFT OUTER JOIN** , **RIGHT OUTER JOIN**, **FULL OUTER JOIN**
are a few commonly used ones

INNER JOIN (or simply JOIN

Heres an example of simple **INNER JOIN** :

```
SELECT users.name, visits.created_at
FROM visits
INNER JOIN users
  ON users.id = visits.user_id
```

Each visit has a **user_id** that corresponds to the **id** column in the users table. For each match that is found, a row is inserted into the result set.

JOIN types

The various **JOIN** types specify how to deal with different circumstances regarding the primary and foreign key matchings.

- **INNER JOIN** discards any entries that do not have a match between the keys specified in the **ON** clause
- **LEFT OUTER JOIN** keeps all entries in the left table regardless of whether a match is found in the right table.
- **RIGHT OUTER JOIN** is the same except it keeps all the entries in the right table instead of the left
- **FULL OUTER JOIN** will keep the rows of both tables no matter what

Conceptual Order of Evaluation of a SQL **SELECT** Statement

1. **FROM** + **JOIN**: first the product of all tables is formed
2. **WHERE**: the where clause is used to filter rows that do not satisfy search condition
3. **GROUP BY** + (**COUNT**, **SUM**, etc): the rows are grouped using the columns in the group by clause and the aggregation functions are applied on the grouping
4. **HAVING**: like the **WHERE** clause, but can be applied after aggregation
5. **SELECT**: the targeted list of columns are evaluated and returned
6. **DISTINCT**: duplicate rows are eliminated
7. **ORDER BY**: the resulting rows are sorted

Lets walk through a full SQL Query and look at some more complicated example queries