SQL: An Introduction

July 2015

Objectives

- Understand the *what* and *why* of Relational Database Management Systems.
- *Generally*, how to create and populate a RDBMS.
- *Specifically*, how to extract data from a RDBMS (SQL).

At the end of this unit, you will be able to connect to a Postres database, answer questions using data and/or download data for further questioning.

What is RDBMS?

- A RDBMS happens to be one way of storing *persistent* data. i.e. data which
  - survives after the process in which it was created has ended.
  - is written to non-volatile storage.
  - is infrequently accessed and unlikely to be changed.
- For all intents and purposes, *most* business data is stored in a RDBMS
- Relational Databases are the *de facto* standard for storing data. (e.g., Oracle, MySQL, MSSQL Server, etc...)
  - It could be argued that this is only now starting to change with the advent of 'Big Data'

Why RDBMS?

An RDBMS provides the ability to

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

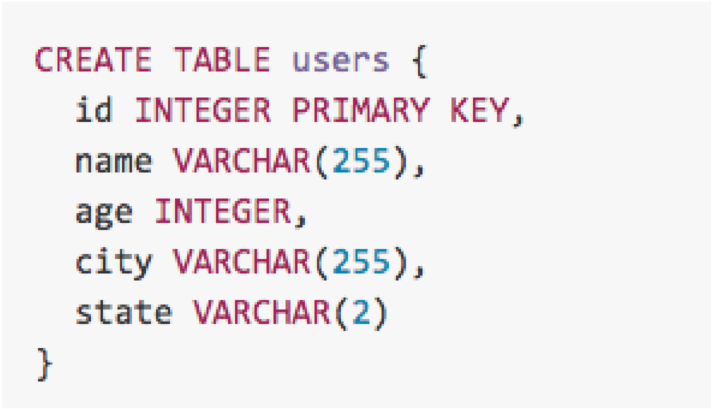
RBDMS Data Model

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 (a.k.a fields)** and **rows (a.k.a records)**
- 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

When confronted with a new data source, your first task is typically to understand the schema (and often this is not trivial).

Database Table Example



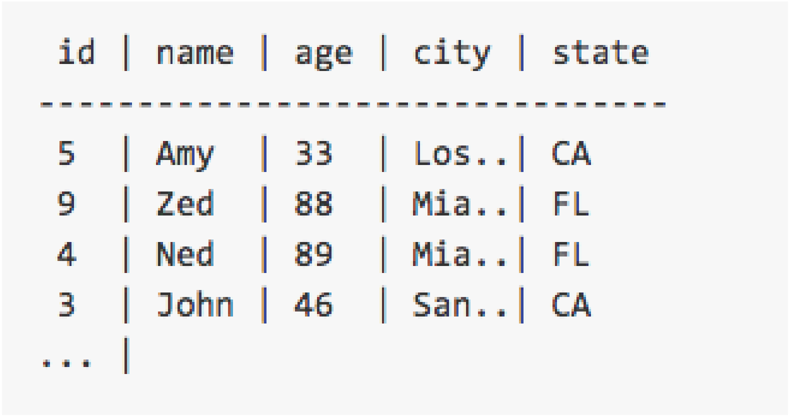
```
CREATE TABLE users {
  id INTEGER PRIMARY KEY,
  name VARCHAR(255),
  age INTEGER,
  city VARCHAR(255),
  state VARCHAR(2)
}
```

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:



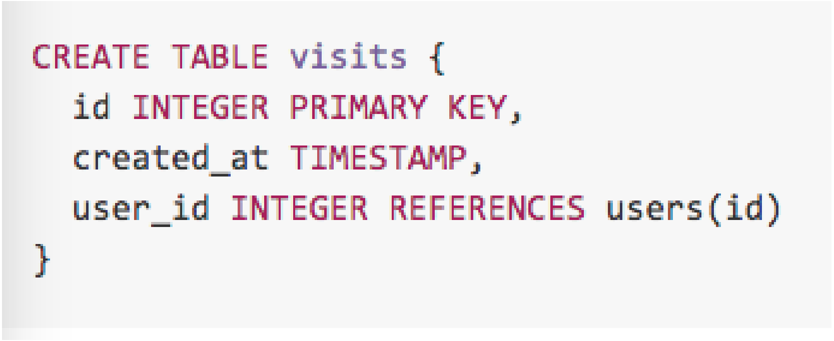
```
 id | name | age | city | state
--------------------------------
 5  | Amy  | 33  | Los..| CA
 9  | Zed  | 88  | Mia..| FL
 4  | Ned  | 89  | Mia..| FL
 3  | John | 46  | San..| CA
... |
```

A primary key is not always an integer—it could be a timestamp, a hash, a combination of columns, etc...

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)
}
```

Schema Normalization

Minimize Redundancy. For example:

- Details about a user (address, age, etc...) are only stored once (in a *users* table).
- Any other table (e.g. *purchases*), where this data might be relevant, only references the user_id.

Structured Query Language (SQL)

- As a data scientist your main interaction with RDBMS will be to *extract* information that already exists in a database.
- 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.
- Even non-relational databases like Hadoop usuall have a SQL-like interface available.

SQL

All SQL queries have three main ingredients

SELECT	What data do you want?
FROM	Where do you want to get the data from?
WHERE	Under what conditions?

SQL is *Declarative* rather than *Imperative*. That is, you tell the machine what you want and it (the database optimizer) decides how to do it.

SQL Queries

Select the columns *name, age* from the table *users*.

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

SQL always returns a table, so the output (to the console) is the sub-table of *users* with two columns.

```sql
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

Let's see some examples..

More SQL

- Joins
- Subqueries
- Order of operations

JOIN

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

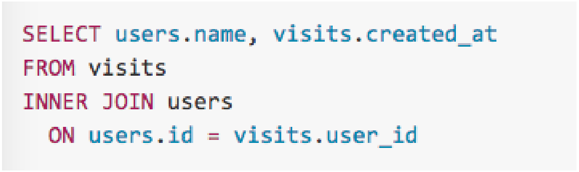
Every **JOIN** clause has two segments:

- 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
```

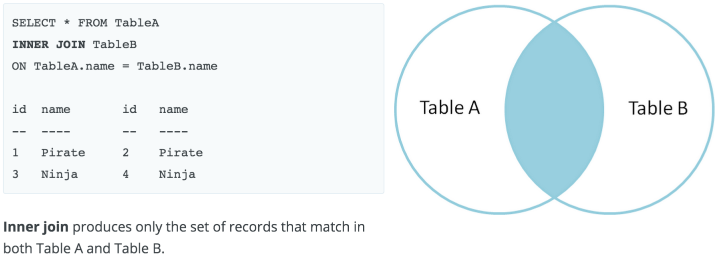
Each visit has a **user_id** that corresponds to the **id** column in the users table. For each match that is found, a row is inserted into the result set.

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

Inner Join



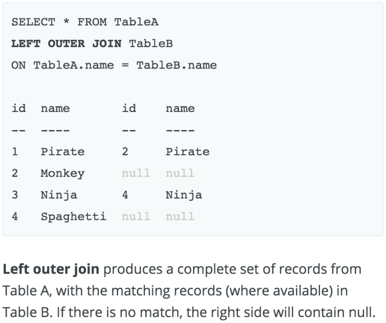
```
SELECT * FROM TableA
INNER JOIN TableB
ON TableA.name = TableB.name

id  name       id   name
--  ----       --   ----
1   Pirate     2    Pirate
3   Ninja      4    Ninja
```

Inner join produces only the set of records that match in both Table A and Table B.

Image copied from
http://blog.codinghorror.com/a-visual-explanation-of-sql-joins/

Left Join



```
SELECT * FROM TableA
LEFT OUTER JOIN TableB
ON TableA.name = TableB.name

id  name       id    name
--  ----       --    ----
1   Pirate     2     Pirate
2   Monkey     null  null
3   Ninja      4     Ninja
4   Spaghetti  null  null
```

Left outer join produces a complete set of records from Table A, with the matching records (where available) in Table B. If there is no match, the right side will contain null.

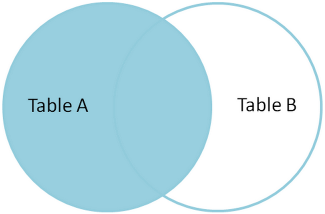
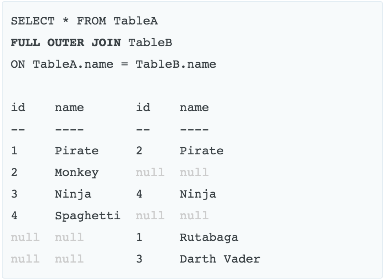


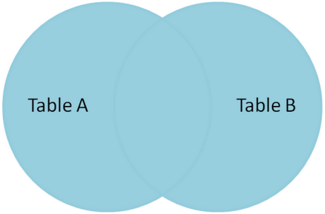
Image copied from
http://blog.codinghorror.com/a-visual-explanation-of-sql-joins/

Outer Join



```
SELECT * FROM TableA
FULL OUTER JOIN TableB
ON TableA.name = TableB.name

id    name       id    name
--    ----       --    ----
1     Pirate     2     Pirate
2     Monkey     null  null
3     Ninja      4     Ninja
4     Spaghetti  null  null
null  null       1     Rutabaga
null  null       3     Darth Vader
```



Full outer join produces the set of all records in Table A and Table B, with matching records from both sides where available. If there is no match, the missing side will contain null.

Image copied from
http://blog.codinghorror.com/a-visual-explanation-of-sql-joins/

Subqueries

- In general, you can replace any table name with a SELECT statement.
  - SELECT ...FROM (SELECT ...)
- If a query returns a **single value**, you can treat it as such.
  - WHERE var1 = (SELECT ...)
- If a query returns a **single column**, you can treat it sort of like a vector
  - WHERE var1 IN (SELECT ...)

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