SQL: An Introduction

October 2015

Objectives

- Understand the *what* and *why* of Relational Database Management Systems.
- Use SQL to get data out of a Relational Database.

What is Relational Database?

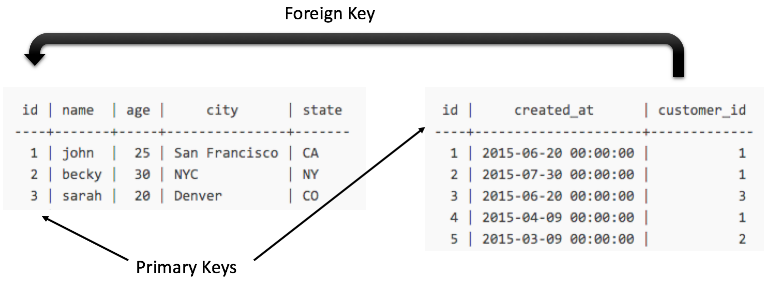
- A relational database happens to be one way of storing *persistent* data. i.e. data which
  - survives after the process in which it was created has ended.
  - is written to non-volatile storage.
  - is infrequently accessed and unlikely to be changed.
- For all intents and purposes, *most* business data is stored in a relational database.
- 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'

Types of Databases

- Network Databases (think lots of pointers and hierarchy)
- Document Store (raw text files)
- Lots more...
- *Relational Databases*
  - E.F. Codd win the Turing prize for this.

Relational Databases

Everything is a table



```
id | name  | age |     city      | state
----+-------+-----+---------------+-------
  1 | john  |  25 | San Francisco | CA
  2 | becky |  30 | NYC           | NY
  3 | sarah |  20 | Denver        | CO
```

```
id |      created_at      | customer_id
----+---------------------+-------------
  1 | 2015-06-20 00:00:00 |           1
  2 | 2015-07-30 00:00:00 |           1
  3 | 2015-06-20 00:00:00 |           3
  4 | 2015-04-09 00:00:00 |           1
  5 | 2015-03-09 00:00:00 |           2
```

- Each row = one record
- Each column (field) has a specified column type. (E.g. text, numeric, date, etc.)

Relational Algebra

- Technically, we can think of tables as *sets* and there is a rich mathematics behind how we can operate on them. In particular, there is an *algebra of sets*.
- We can take unions, subsets, projections, etc..
- Most importantly, there are some algebraic identities. For example,

$$\sigma_{p\wedge q}(R \bowtie S) = \sigma_p(R) \bowtie \sigma_q(S)$$

Don't worry about what these mean - the point is that they exist.

SQL

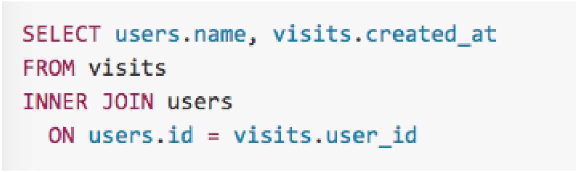
SQL is the language used to specify mathematical set operations

SELECT	What data do you want? (Projection)
FROM	Where do you want to get the data from?
WHERE	Under what conditions? (Subset)
JOIN	Combine two tables. (Cross-product... sort of)

The clincher is that the database (Postgres, Oracle, etc.) knows all the algebraic identities. So you just tell it what you want, and it rearranges things so that the query will be optimal. That is, SQL is *Declarative* rather than *Imperative*.

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
```

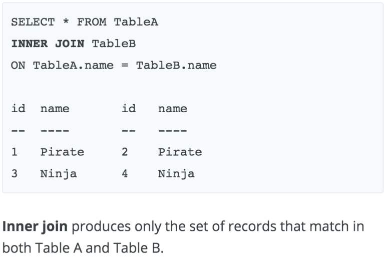
Each visit has a **user_id** that corresponds to the **id** column in the users table. For each match that is found, a row is inserted into the result set.

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.

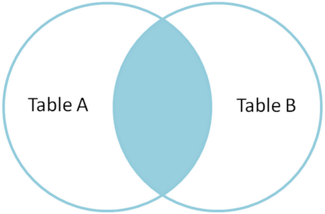
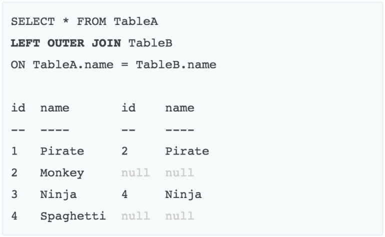
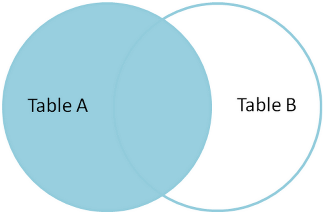


Image copied from
http://blog.codinghorror.com/a-visual-explanation-of-sql-joins/

Left Join



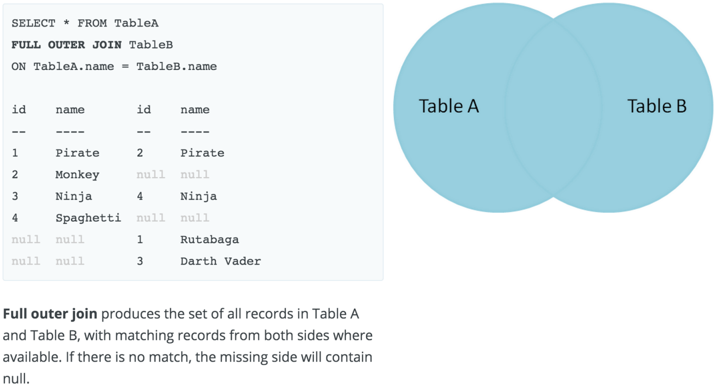
```
SELECT * FROM TableA
LEFT OUTER JOIN TableB
ON TableA.name = TableB.name

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