February 14, 2019

# Lab 2 Discussion

We can end up with a lot of tables when we normalize our data

* If we were to not normalize the data, we would end up with one relation to hold all of our data
  + Don’t try to game the system: just follow the BCNF instructions

1. Write down all functional dependencies
   1. Order # -> everything
   2. Customer ID -> Customer Name
   3. Serial # -> everything
   4. Model # -> speed, ram, hd, price
2. Pick a violation of BCNF and start to decompose:
   1. Customer ID | Customer Name (Table 1)
   2. Customer ID | Everything EXCEPT Customer Name (Table 2)
3. Since Table 2 is still a violation of the Model # functional dependency, we decompose on table 2
   1. Model # | Speed | Ram| HD | Price (Table 1)
   2. Model # | Everything EXCEPT Speed, ram, hd, price (Table 2)
4. Because we decomposed on table 2 from step 2, we no longer ‘need’ that table
5. So, we end with three tables
   1. Customer name | Cust ID
   2. Model # | Speed | Ram| HD | Price
   3. Model # | Cust ID | Order # | Date | Shipping | SN
6. DONE

# SQL Deep Dive

Since we now put our tables in BCNF, we need to be able to put our tables back together using some SQL

**Cross Product**

* Most fundamental, least useful *directly*
* Defined as each instance in T is paired with each instance in S
* **Doesn’t necessarily create meaningful data**
* The result can be a **massive** result set
  + Probably not exactly what we want to do
    - Likely another step to get to the result that we want

**Join Operation**

* Calculate the cross product *without* extraneous data

*Natural Join*

* Resulting rows must agree on attributes
  + ONLY work if you happen to have the same column names in the two tables you want to join
* Can result in odd things if you have two columns that mean different things that have the same name

*Theta Join*

* Conditional join
  + We are specifying the condition for which the result set is based on
* Much better than natural join in most applications (if not all)

How do we do this in SQL?

SELECT **columns**

FROM **relations**

WHERE **condition**

Cross productions and joins are added to the FROM clause

**Cross Product**

SELECT **columns**

FROM **A, B**

WHERE **condition**

* The first thing the computer does is build the cross product of A, B
* Applies the where clause to get all the rows we want
* Then gets only the columns we want

**Natural Join**

SELECT **columns**

FROM **A NATURAL JOIN B**

WHERE **condition**

**Theta Join**

SELECT **columns**

FROM **A JOIN B ON *condition(s)***

WHERE **condition**

**Examples:**

SELECT \* FROM Orders, PC WHERE Orders.Model# = PC.Model# AND Date < ‘2014/1/1’

SELECT \* FROM Orders NATURAL JOIN PC WHERE Date < ‘2014/1/1’

SELECT \* FROM Orders JOIN PC ON Orders.Model# = PC.Model# AND Date < ‘2014/1/1’

The where clause filters real data -> this will filter real data

On clause -> creates/defines the real data

* **WHERE IS FOR FILTERING, ON IS FOR JOINING**

An attribute in a given row could be empty

* This might lead to issues when we try to join on certain things and not realize that our query isn’t returning everything that we think it is

So, we could specify that something is not null:

CREATE TABLE Persons {

ID int NOT NULL,

LastName varchar(255) NOT NULL,

…

}

This means that we don’t need to account for null attributes in our application code!

On actually refers to an inner join

* An inner join means that the two things being joined on must match completely

We can add the keyword left to our join

SELECT \*

FROM Movie

LEFT JOIN Studio ON Movie.studioName == Studio.name

Full outer join returns a result set that is potentially larger than left join and join

…

…

FULL OUTER JOIN …

## Order of Results

ORDER BY clause acts as the sorting operator

* Followed by comma separated list of attributes to order by
* Comes after the where clause (always the last clause)
* Default order is ascending – can include DESC to reverse

This can be taxing on the database

## Set Operations

Set operations require that both relations have the ***same schema***

* The where clause only examines one row at a time
* Hard to detect issues related to this
  + Need to consider if were asking a question that encapsulates a set of rows instead of one row at a time!

EXCEPT will connect one result to another result and return the subtraction of the smaller result from the larger

* Have to have exactly the same column names

## Renaming:

SELECT StarsIn.starName as name

If we modify a column, the as key is required\*\*\*

## Subqueries

**Scalar Values**

Atomic Value

* One attribute of one row

**Boolean Operators**

* EXISTS R
* … IN R
* … > ALL R
* … > ANY R
* NOT works with all of these

*See presentation for a bunch of examples on subqueries*

## Pattern Matching

LIKE keyword

* Search for strings without an exact match
  + Similar pattern in text

**%**

* Matches any sequence of 0 or more characters in string

**\_**

* Matches any single character in string

**All other characters are exact matches**

If we want to search for %:

* We must define an escape character

s LIKE ‘x%%x%’ ESCAPE ‘x’

This is like a regular expression

* Never search for patters with regular expressions if you can use the like keyword because this is resource intensive on a database

**Text searching is a huge topic**

* Word-Stemming
  + Branch of being able to look at the search term and find fields based on whether or not the search terms are like other search terms or not
* Etc.

Some databases are good for this, others are not

## Aggregation

**Act of collapsing a set of values into fewer values**

SELECT DISTINCT title FROM Movies

SELECT AVG(netWorth) as Average FROM Movie Exec

SELECT COUNT(DISTINCT title) as num FROM Movies

GROUP BY ….

* We don’t need to compute aggregations over the entire relation
  + We can instead define groups
* So, group by essentially lets us reference multiple rows at a time

SELECT studioName, SUM(length) AS total

FROM Movies

GROUP BY studioName

* Returns a sum of the length of all of movies produced by a studio for all of the studios that are present

Group by comes after the for clause\*

We can select certain groups with the HAVING clause

SELECT studioName

FROM Movies

GROUP BY studioName

HAVING COUNT (title) > 6

## Updating and Deleting Data

**Deletion**

Deletion is performed as a part of a selection

Different than:

DROP TABLE *table*

ALTER TABLE *table* DROP *column*

DELETE FROM R WHERE *condition*

**RULE OF THUMB**:

* Before performing a delete, replace it with SELECT \* and check the results to make sure your statement is doing what you think it is

When deleting we cannot use JOIN or cross products

BUT WE CAN DO THIS:

DELETE FROM Movies

WHERE producerCert# IN

(SELECT cert# FROM MovieExec WHERE name = ‘John Smith’)

**Updating data**

UPDATE R SET *new values* WHERE condition

* Without the where clause, the database will update ***everything***

UPDATE Movies SET length=175, genre=’comedy’

WHERE title=’Star Wars’ AND year = 1977