DS 1300 - Introduction to SQL
Part 2

by Michael Hahsler
Based on slides for CS145 Introduction to Databases (Stanford)

**DILBERT**

I accomplished nothing this week because I was in a training class.

I didn't approve any training expenses.

A vendor paid for it.

**BY SCOTT ADAMS**

You didn't ask for permission.

I'm proactive and empowered.

And what was the name of this alleged class?

Advanced scripting structure for internetwork optimization of SQL databases.

That doesn't sound real.

I can't do my job if you don't trust me!

Do you like how I combined aggressiveness with my baseline level of uselessness?

I have a good feeling about this.

You might need more aggressiveness.
Lecture Overview

1. Aggregation & GROUP BY

2. Advanced SQL-izing (set operations, NULL, Outer Joins, etc.)
AGGREGATION, GROUP BY AND HAVING CLAUSE
Aggregation

```
SELECT COUNT(*)
FROM Product
WHERE year > 1995
```

```
SELECT AVG(price)
FROM Product
WHERE maker = 'Toyota'
```

- SQL supports several **aggregation** operations:
  - SUM, COUNT, MIN, MAX, AVG

Except for COUNT, all aggregations apply to a single attribute!
Aggregation: COUNT

COUNT counts the number of tuples including duplicates.

```
SELECT COUNT(category) 
FROM   Product 
WHERE  year > 1995
```

Note: Same as `COUNT(*)`. Why?

We probably want:

```
SELECT COUNT(DISTINCT category) 
FROM   Product 
WHERE  year > 1995
```
More Examples

```
SELECT SUM(price * quantity)
FROM Purchase
```

What do these mean?

```
SELECT SUM(price * quantity)
FROM Purchase
WHERE product = ‘bagel’
```
**Simple Aggregations**

### Purchase

<table>
<thead>
<tr>
<th>Product</th>
<th>Date</th>
<th>Price</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>bagel</td>
<td>10/21</td>
<td>1</td>
<td>20</td>
</tr>
<tr>
<td>banana</td>
<td>10/3</td>
<td>0.5</td>
<td>10</td>
</tr>
<tr>
<td>banana</td>
<td>10/10</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>bagel</td>
<td>10/25</td>
<td>1.50</td>
<td>20</td>
</tr>
</tbody>
</table>

```sql
SELECT SUM(price * quantity) FROM Purchase WHERE product = 'bagel'
```

50 (= 1*20 + 1.50*20)
Grouping and Aggregation

Find total sales after 10/1 per product.

Purchase(product, date, price, quantity)

SELECT product,
    SUM(price * quantity) AS TotalSales
FROM Purchase
WHERE date > '10/01'
GROUP BY product

Let’s see what this means...

Note: Be very careful with dates! Use date/time related functions!
Grouping and Aggregation

Semantics of the query:

1. Compute the **FROM** and **WHERE** clauses

2. Group by the attributes in the **GROUP BY**

3. Compute the **SELECT** clause: grouped attributes and aggregates
1. Compute the **FROM** and **WHERE** clauses

```
SELECT   product, SUM(price*quantity) AS TotalSales
FROM     Purchase
WHERE    date > '10/01'
GROUP BY product
```
2. Group by the attributes in the **GROUP BY**

```
SELECT  product, SUM(price*quantity) AS TotalSales
FROM    Purchase
WHERE   date > '10/01'
GROUP BY product
```

<table>
<thead>
<tr>
<th>Product</th>
<th>Date</th>
<th>Price</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bagel</td>
<td>10/21</td>
<td>1</td>
<td>20</td>
</tr>
<tr>
<td>Bagel</td>
<td>10/25</td>
<td>1.50</td>
<td>20</td>
</tr>
<tr>
<td>Banana</td>
<td>10/3</td>
<td>0.5</td>
<td>10</td>
</tr>
<tr>
<td>Banana</td>
<td>10/10</td>
<td>1</td>
<td>10</td>
</tr>
</tbody>
</table>
3. Compute the **SELECT** clause: grouped attributes and aggregates

```sql
SELECT product, SUM(price*quantity) AS TotalSales
FROM Purchase
WHERE date > '10/01'
GROUP BY product
```

<table>
<thead>
<tr>
<th>Product</th>
<th>Date</th>
<th>Price</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bagel</td>
<td>10/21</td>
<td>1</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>10/25</td>
<td>1.50</td>
<td>20</td>
</tr>
<tr>
<td>Banana</td>
<td>10/3</td>
<td>0.5</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>10/10</td>
<td>1</td>
<td>10</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Product</th>
<th>TotalSales</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bagel</td>
<td>50</td>
</tr>
<tr>
<td>Banana</td>
<td>15</td>
</tr>
</tbody>
</table>
Activity

1) What do the next two queries calculate?

```sql
SELECT SUM(price) AS total, SUM(price) * 1.08 AS totalPlusTax
FROM Product pr
JOIN Purchase p ON pr.PName = p.product
WHERE p.buyer = 'Joe Blow'
```

```sql
SELECT p.buyer, SUM(price) AS total, SUM(price) * 1.08 AS totalPlusTax
FROM Product pr
JOIN Purchase p ON pr.PName = p.product
GROUP BY p.buyer
ORDER BY 1
```

2) Write a query to find the price of the most expensive product in each category.
HAVING Clause

Same query as before, except that we consider only products that have more than 100 buyers.

```
SELECT product, SUM(price*quantity)
FROM Purchase
WHERE date > '10/1/2005'
GROUP BY product
HAVING SUM(quantity) > 100
```

**HAVING clauses contains conditions on aggregates**

**Whereas WHERE clauses condition on individual tuples...**
General form of Grouping and Aggregation

\[
\text{SELECT } S \quad \text{FROM } R_1, \ldots, R_n \\
\text{WHERE } C_1 \\
\text{GROUP BY } a_1, \ldots, a_k \\
\text{HAVING } C_2
\]

- \( S = \) Can ONLY contain attributes \( a_1, \ldots, a_k \) and/or aggregates over other attributes
- \( C_1 = \) is any condition on the attributes in \( R_1, \ldots, R_n \)
- \( C_2 = \) is any condition on the aggregate expressions
General form of Grouping and Aggregation

```
SELECT S
FROM R₁,…,Rₙ
WHERE C₁
GROUP BY a₁,…,aₖ
HAVING C₂
```

Evaluation steps:

1. Evaluate FROM-WHERE: apply condition C₁ on the attributes in R₁,…,Rₙ
2. GROUP BY the attributes a₁,…,aₖ
3. Compute aggregates in S and do projection (SELECT)
4. Apply condition C₂ to each group (may have aggregates)
Activity

1) What does this query do?

```sql
SELECT p.buyer, SUM(price) AS total, COUNT(*) AS purchases
FROM Product pr
JOIN Purchase p ON pr.PName = p.product
GROUP BY p.buyer
HAVING purchases > 2
ORDER BY 1
```

2) What is the revenue per product in the DB?
Subqueries

SELECT *
FROM (SELECT product, COUNT(product) AS count
FROM Purchase GROUP BY product)
WHERE count > 2

SELECT *, (SELECT count(*) FROM Product p1
WHERE p1.category = p2.category) AS '# Prod. in Cat.'
FROM Product p2

Subqueries can appear wherever a table or a value is needed.
ADVANCED SQL: NULLS, CASTING AND OUTER JOINS
NULL VALUES & OTHER DETAILS
NULL Values

• Whenever we do not have a value, we can use NULL

• Can mean many things:
  – Value does not exists
  – Value exists but is unknown (n/a, not available)
  – Value not applicable

• The schema specifies for each attribute if it can be null (nullable attribute) or not with NOT NULL
Null Values and Operators

For numerical operations:
- If \( x = \text{NULL} \) then \( 4*(3-x)/7 \) is also NULL

For boolean operations, in SQL there are three values:

<table>
<thead>
<tr>
<th>FALSE</th>
<th>TRUE</th>
<th>UNKNOWN</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

If \( x = \text{NULL} \) then \( x='Joe' \) is UNKNOWN

Note: comparison in SQL is a single ‘=’

SQLite does not have a boolean datatype. It uses Integer instead!
Try:
- SELECT 2>1
- SELECT 2>NULL
- SELECT 1+NULL
Null Values in the WHERE Clause

```sql
SELECT * 
FROM Person 
WHERE (age < 25) 
AND (height > 6 AND weight > 190)
```

Will not return age=20, height=NULL, weight=200
Since NULL > 6 is UNKNOWN!
Null Values in WHERE Clauses

Unexpected behavior:

```
SELECT *
FROM Person
WHERE age < 25 OR age >= 25
```

Should return all persons, but persons with NULL as age are not included!

You can use CASE with IS NULL, ISNULL(), IFNULL() or COALESCE() to handle NULL values.
CASTing Data Types

SQL is a typed language. I.e., values and columns have a data type.

<table>
<thead>
<tr>
<th>SELECT 3/2</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT 3.0/2</td>
<td>1.5</td>
</tr>
<tr>
<td>SELECT 3/2.0</td>
<td>1.5</td>
</tr>
<tr>
<td>SELECT <code>CAST</code>(3 AS DOUBLE)/2</td>
<td>1.5</td>
</tr>
</tbody>
</table>

Typecasting rules are similar to other typed languages like C++. 
**RECAP: Inner Joins**

**Inner joins** select all rows from both tables as long as there is a match between the columns in both tables. Inner joins are the default in SQL.

**Example:** What stores sell what products?

```
Product(name, category)
Purchase(prodName, store)
```

```
SELECT Product.name, Purchase.store
FROM Product
JOIN Purchase ON Product.name = Purchase.prodName
```

Both equivalent:

```
SELECT Product.name, Purchase.store
FROM Product, Purchase
WHERE Product.name = Purchase.prodName
```
Inner Joins + NULLS = Lost data?

Product(name, category)
Purchase(prodName, store)

SELECT Product.name, Purchase.store
FROM Product
JOIN Purchase ON Product.name = Purchase.prodName

SELECT Product.name, Purchase.store
FROM Product, Purchase
WHERE Product.name = Purchase.prodName

However: Products that were never sold in any store (with no Purchase tuple) will be lost!
Outer Joins

An outer join returns also tuples from the joined relations that do not have a corresponding tuple in the other relations (filled with NULL values).

Left outer joins in SQL:

```sql
SELECT Product.name, Purchase.store
FROM Product
  LEFT OUTER JOIN Purchase
    ON Product.name = Purchase.prodName

Now we’ll get products even if they didn’t sell
```
INNER JOIN:

**Product**

<table>
<thead>
<tr>
<th>name</th>
<th>category</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gizmo</td>
<td>gadget</td>
</tr>
<tr>
<td>Camera</td>
<td>Photo</td>
</tr>
<tr>
<td>OneClick</td>
<td>Photo</td>
</tr>
</tbody>
</table>

**Purchase**

<table>
<thead>
<tr>
<th>prodName</th>
<th>store</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gizmo</td>
<td>Wiz</td>
</tr>
<tr>
<td>Camera</td>
<td>Ritz</td>
</tr>
<tr>
<td>Camera</td>
<td>Wiz</td>
</tr>
</tbody>
</table>

```sql
SELECT Product.name, Purchase.store
FROM Product
INNER JOIN Purchase
ON Product.name = Purchase.prodName
```
LEFT OUTER JOIN:

SELECT Product.name, Purchase.store
FROM Product
LEFT OUTER JOIN Purchase
ON Product.name = Purchase.prodName
Other Outer Joins

• Left outer join:
  – Include the left tuple even if there’s no match

• Right outer join:
  – Include the right tuple even if there’s no match

• Full outer join:
  – Include both left and right tuples even if there’s no match

SQLite currently only supports LEFT OUTER JOIN, but you can easily just change the order of the tables in the query.
Adding Data

`INSERT INTO TABLE_NAME
    [(column1, column2, column3,...columnN)]
VALUES (value1, value2, value3,...valueN);`

Note: column names are optional.

```
INSERT INTO Product
VALUES ('Gizmo', 19, 'Gadgets', 'GWorks')
```
Adding Data

The data can also come from an existing table.

```sql
INSERT INTO first_table_name [(column1, column2, ... columnN)]
SELECT column1, column2, ...columnN
FROM second_table_name
[WHERE condition];
```
Removing a Table

DROP TABLE database_name.table_name
Select Syntax Diagram (SQLite)

http://www.sqlite.org/lang.html
Activity

Review (http://www.tutorialspoint.com/sqlite/):

• Transaction control
• Views
• Indexes
• Date & Time