MIS2502: Data Analytics

SQL – Getting Information Out of a Database

Jing Gong

gong@temple.edu

http://community.mis.temple.edu/gong
The relational database

- Core of Online Transaction Processing (OLTP)
- A series of tables
- Linked together through primary/foreign key relationships
What do we want to do?

**Put information into the database (change)**

**Get information out of the database (retrieve)**

Database Management System

- **Customer**
  - CustomerID
  - FirstName
  - LastName
  - City
  - State
  - Zip

- **Order**
  - OrderNumber
  - OrderDate
  - CustomerID

- **Order-Product**
  - OrderProductID
  - OrderNumber
  - ProductID
  - Quantity

- **Product**
  - ProductID
  - ProductName
  - Price
To do this we use SQL

- Structured Query Language (SQL)

- A high-level set of statements (commands) that let you communicate with the database

- With SQL, you can
  - Retrieve records
  - Join (combine) tables
  - Insert records
  - Delete records
  - Update records
  - Add and delete tables

A statement is any SQL command that interacts with a database.

A SQL statement that retrieves information is referred to as a query.
Some points about SQL

- It’s not a true programming language
- It is used by programming languages to interact with databases

- There is no standard syntax
- MySQL, Oracle, SQL Server, and Access all have slight differences

- There are a lot of statements and variations among them
- We will be covering the basics, and the most important ones

This is a great online reference for SQL syntax:

http://www.w3schools.com/sql

Here’s the one specifically for MySQL, but it’s not as well-written:

SELECT statement

SELECT column_name(s)
FROM schema_name.table_name;

Example:
SELECT FirstName FROM orderdb.Customer;

<table>
<thead>
<tr>
<th>CustomerID</th>
<th>FirstName</th>
<th>LastName</th>
<th>City</th>
<th>State</th>
<th>Zip</th>
</tr>
</thead>
<tbody>
<tr>
<td>1001</td>
<td>Greg</td>
<td>House</td>
<td>Princeton</td>
<td>NJ</td>
<td>09120</td>
</tr>
<tr>
<td>1002</td>
<td>Lisa</td>
<td>Cuddy</td>
<td>Plainsboro</td>
<td>NJ</td>
<td>09123</td>
</tr>
<tr>
<td>1003</td>
<td>James</td>
<td>Wilson</td>
<td>Pittsgrove</td>
<td>NJ</td>
<td>09121</td>
</tr>
<tr>
<td>1004</td>
<td>Eric</td>
<td>Foreman</td>
<td>Warminster</td>
<td>PA</td>
<td>19111</td>
</tr>
</tbody>
</table>

Your schema will use your misx MySQL ID (i.e., mis80orderdb.Customer)

This returns the FirstName column for every row in the Customer table. Called a “View.”
Retrieving multiple columns

`SELECT FirstName, State FROM orderdb.Customer;`  

<table>
<thead>
<tr>
<th>FirstName</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greg</td>
<td>NJ</td>
</tr>
<tr>
<td>Lisa</td>
<td>NJ</td>
</tr>
<tr>
<td>James</td>
<td>NJ</td>
</tr>
<tr>
<td>Eric</td>
<td>PA</td>
</tr>
</tbody>
</table>

The * is called a **wildcard**. It means “return every column.”

`SELECT * FROM orderdb.Customer;`  

<table>
<thead>
<tr>
<th>CustomerID</th>
<th>FirstName</th>
<th>LastName</th>
<th>City</th>
<th>State</th>
<th>Zip</th>
</tr>
</thead>
<tbody>
<tr>
<td>1001</td>
<td>Greg</td>
<td>House</td>
<td>Princeton</td>
<td>NJ</td>
<td>09120</td>
</tr>
<tr>
<td>1002</td>
<td>Lisa</td>
<td>Cuddy</td>
<td>Plainsboro</td>
<td>NJ</td>
<td>09123</td>
</tr>
<tr>
<td>1003</td>
<td>James</td>
<td>Wilson</td>
<td>Pittsgrove</td>
<td>NJ</td>
<td>09121</td>
</tr>
<tr>
<td>1004</td>
<td>Eric</td>
<td>Foreman</td>
<td>Warminster</td>
<td>PA</td>
<td>19111</td>
</tr>
</tbody>
</table>

It’s good practice to end every statement with a **semicolon**, especially when entering multiple statements.
Capitalization and spacing

- SQL syntax is **not sensitive** to capitalization and spacing

<table>
<thead>
<tr>
<th>SELECT FirstName, State FROM orderdb.Customer;</th>
<th>✓ Correct</th>
<th>✓ Best Practice</th>
</tr>
</thead>
<tbody>
<tr>
<td>Select firstname, state from orderdb.customer;</td>
<td>✓ Correct</td>
<td></td>
</tr>
<tr>
<td>SELECT FirstName, State FROM orderdb.Customer;</td>
<td>✓ Correct</td>
<td></td>
</tr>
</tbody>
</table>

- For this class:
  - We will write all SQL keywords (e.g., SELECT and FROM) in upper case
  - When referring to a table or column, we match the case (upper or lower) of each character
  - Use space appropriately for readability
Retrieving unique values

**SELECT DISTINCT State**
FROM orderdb.Customer;

Returns only one occurrence of each value in the column.

**SELECT DISTINCT City, State**
FROM orderdb.Customer;

In this case, each combination of City AND State is unique, so it returns all of them.

<table>
<thead>
<tr>
<th>City</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>Princeton</td>
<td>NJ</td>
</tr>
<tr>
<td>Plainsboro</td>
<td>NJ</td>
</tr>
<tr>
<td>Pittsgrove</td>
<td>NJ</td>
</tr>
<tr>
<td>Warminster</td>
<td>PA</td>
</tr>
</tbody>
</table>
Returning only certain records

- We don’t always want every record in the table

<table>
<thead>
<tr>
<th>CustomerID</th>
<th>FirstName</th>
<th>LastName</th>
<th>City</th>
<th>State</th>
<th>Zip</th>
</tr>
</thead>
<tbody>
<tr>
<td>1001</td>
<td>Greg</td>
<td>House</td>
<td>Princeton</td>
<td>NJ</td>
<td>09120</td>
</tr>
<tr>
<td>1002</td>
<td>Lisa</td>
<td>Cuddy</td>
<td>Plainsboro</td>
<td>NJ</td>
<td>09123</td>
</tr>
<tr>
<td>1003</td>
<td>James</td>
<td>Wilson</td>
<td>Pittsgrove</td>
<td>NJ</td>
<td>09121</td>
</tr>
<tr>
<td>1004</td>
<td>Eric</td>
<td>Foreman</td>
<td>Warminster</td>
<td>PA</td>
<td>19111</td>
</tr>
</tbody>
</table>

**Use:** SELECT * FROM schema_name.table_name **WHERE** condition;
so SELECT * FROM orderdb.Customer **WHERE** State= 'NJ';
returns this:

<table>
<thead>
<tr>
<th>CustomerID</th>
<th>FirstName</th>
<th>LastName</th>
<th>City</th>
<th>State</th>
<th>Zip</th>
</tr>
</thead>
<tbody>
<tr>
<td>1001</td>
<td>Greg</td>
<td>House</td>
<td>Princeton</td>
<td>NJ</td>
<td>09120</td>
</tr>
<tr>
<td>1002</td>
<td>Lisa</td>
<td>Cuddy</td>
<td>Plainsboro</td>
<td>NJ</td>
<td>09123</td>
</tr>
<tr>
<td>1003</td>
<td>James</td>
<td>Wilson</td>
<td>Pittsgrove</td>
<td>NJ</td>
<td>09121</td>
</tr>
</tbody>
</table>
More conditional statements

SELECT * FROM orderdb.Customer WHERE State <> 'NJ';

<table>
<thead>
<tr>
<th>CustomerID</th>
<th>FirstName</th>
<th>LastName</th>
<th>City</th>
<th>State</th>
<th>Zip</th>
</tr>
</thead>
<tbody>
<tr>
<td>1004</td>
<td>Eric</td>
<td>Foreman</td>
<td>Warminster</td>
<td>PA</td>
<td>19111</td>
</tr>
</tbody>
</table>

= means “equal to”
> means “greater than”
>= means “greater than or equal to”
< means “less than”
<= means “less than or equal to”
<> means “not equal to”

SELECT * FROM orderdb.Product WHERE Price > 2;

<table>
<thead>
<tr>
<th>ProductID</th>
<th>ProductName</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>2251</td>
<td>Cheerios</td>
<td>3.99</td>
</tr>
<tr>
<td>2505</td>
<td>Eggo Waffles</td>
<td>2.99</td>
</tr>
</tbody>
</table>

Put single quotes around string (non-numeric) values. For example, 'NJ'
The quotes are optional for numeric values.
More conditional statements: AND & OR Operators

SELECT * FROM orderdb.Product WHERE Price > 2 AND Price<=3.5;

<table>
<thead>
<tr>
<th>ProductID</th>
<th>ProductName</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>2505</td>
<td>Eggo Waffles</td>
<td>2.99</td>
</tr>
</tbody>
</table>

The **AND** operator displays a record if both the first condition AND the second condition are true.

SELECT * FROM orderdb.Customer
WHERE City = 'Princeton' OR City = 'Pittsgrove';

<table>
<thead>
<tr>
<th>CustomerID</th>
<th>FirstName</th>
<th>LastName</th>
<th>City</th>
<th>State</th>
<th>Zip</th>
</tr>
</thead>
<tbody>
<tr>
<td>1001</td>
<td>Greg</td>
<td>House</td>
<td>Princeton</td>
<td>NJ</td>
<td>09120</td>
</tr>
<tr>
<td>1003</td>
<td>James</td>
<td>Wilson</td>
<td>Pittsgrove</td>
<td>NJ</td>
<td>09121</td>
</tr>
</tbody>
</table>

The **OR** operator displays a record if either the first condition OR the second condition is true.
Function: Counting records

SELECT `COUNT`(FirstName) FROM orderdb.Customer;

4

Total number of records in the table where the field is not empty. *(don’t forget the parentheses!)*

SELECT `COUNT`(CustomerID) FROM orderdb.Customer;

4

Why is this the same number as the previous query?

SELECT `COUNT`(*) FROM orderdb.Customer;

?  

What number would be returned?
Functions: Retrieving highest, lowest, average, and sum

<table>
<thead>
<tr>
<th>ProductID</th>
<th>ProductName</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>2251</td>
<td>Cheerios</td>
<td>3.99</td>
</tr>
<tr>
<td>2282</td>
<td>Bananas</td>
<td>1.29</td>
</tr>
<tr>
<td>2505</td>
<td>Eggo Waffles</td>
<td>2.99</td>
</tr>
</tbody>
</table>

SELECT MAX(Price) FROM orderdb.Product;

SELECT MIN(Price) FROM orderdb.Product;

SELECT AVG(Price) FROM orderdb.Product;

SELECT SUM(Price) FROM orderdb.Product;
**GROUP BY**

Ask: How many customers from each state are there in the Customer table?

```sql
SELECT State, COUNT(FirstName)
FROM orderdb.Customer
GROUP BY State;
```

<table>
<thead>
<tr>
<th>CustomerID</th>
<th>FirstName</th>
<th>LastName</th>
<th>City</th>
<th>State</th>
<th>Zip</th>
</tr>
</thead>
<tbody>
<tr>
<td>1001</td>
<td>Greg</td>
<td>House</td>
<td>Princeton</td>
<td>NJ</td>
<td>09120</td>
</tr>
<tr>
<td>1002</td>
<td>Lisa</td>
<td>Cuddy</td>
<td>Plainsboro</td>
<td>NJ</td>
<td>09123</td>
</tr>
<tr>
<td>1003</td>
<td>James</td>
<td>Wilson</td>
<td>Pittsgrove</td>
<td>NJ</td>
<td>09121</td>
</tr>
<tr>
<td>1004</td>
<td>Eric</td>
<td>Foreman</td>
<td>Warminster</td>
<td>PA</td>
<td>19111</td>
</tr>
</tbody>
</table>

So it looks for unique State values and then counts the number of records for each of those values.

**GROUP BY** is used in conjunction with the aggregate functions (COUNT, MAX, MIN, AVG, SUM), to group the result-set by one or more columns.
**Another GROUP BY**

Ask: What is the quantity sold per product?

<table>
<thead>
<tr>
<th>OrderProductID</th>
<th>OrderNumber</th>
<th>ProductID</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>101</td>
<td>2251</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>101</td>
<td>2282</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>101</td>
<td>2505</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>102</td>
<td>2251</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>102</td>
<td>2282</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>103</td>
<td>2505</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>104</td>
<td>2505</td>
<td>8</td>
</tr>
</tbody>
</table>

**SELECT** ProductID, SUM(Quantity) **FROM** orderdb.`Order-Product` **GROUP BY** ProductID;
When to use back quotes?

• Back quotes are to be used for table and column names
  
  Where is the back quote key on the keyboard?

• Only necessary when the table/column name
  1) Contains **space or special characters**
  2) Is a **reserved word**
Back quotes for space or special characters

When the table/column name contains **space or special characters**:

- SELECT * FROM `orderdb.`Order-Product`

- SELECT `Last Name` FROM hospitaldb.Patient

If you are not sure, include back quotes doesn’t hurt.
Back quotes for reserved words

When the table/column name is a **reserved word**:

```
SELECT * FROM orderdb.`Order`;
```

- **Order** is a **reserved word** in SQL. It is a command.
  - As in “**ORDER BY**”
- The back quotes tell MySQL to treat ``Order` as a database object and not a command.

Again, if you are not sure, include back quotes doesn’t hurt.

For a list of reserved words in MySQL, go to:
Sorting using ORDER BY

```
SELECT * FROM orderdb.Product
WHERE Price > 2
ORDER BY Price;
```

ORDER BY sorts results from lowest to highest based on a field (in this case, Price)

<table>
<thead>
<tr>
<th>ProductID</th>
<th>ProductName</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>2505</td>
<td>Eggo Waffles</td>
<td>2.99</td>
</tr>
<tr>
<td>2251</td>
<td>Cheerios</td>
<td>3.99</td>
</tr>
</tbody>
</table>
Counting and sorting

SELECT State, COUNT(FirstName) FROM orderdb.Customer
GROUP BY State
ORDER BY COUNT(FirstName);

<table>
<thead>
<tr>
<th>State</th>
<th>COUNT(FirstName)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PA</td>
<td>1</td>
</tr>
<tr>
<td>NJ</td>
<td>3</td>
</tr>
</tbody>
</table>

GROUP BY organizes the results by column values.

ORDER BY sorts results from lowest to highest based on COUNT(FirstName)
ORDER BY ASC and DESC

SELECT State, COUNT(FirstName) FROM orderdb.Customer GROUP BY State ORDER BY COUNT(FirstName) DESC;

Forces the results to be sorted in DESCending order

<table>
<thead>
<tr>
<th>State</th>
<th>COUNT(FirstName)</th>
</tr>
</thead>
<tbody>
<tr>
<td>NJ</td>
<td>3</td>
</tr>
<tr>
<td>PA</td>
<td>1</td>
</tr>
</tbody>
</table>

SELECT State, COUNT(FirstName) FROM orderdb.Customer GROUP BY State ORDER BY COUNT(FirstName) ASC;

Forces the results to be sorted in ASCending order

<table>
<thead>
<tr>
<th>State</th>
<th>COUNT(FirstName)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PA</td>
<td>1</td>
</tr>
<tr>
<td>NJ</td>
<td>3</td>
</tr>
</tbody>
</table>
Combining WHERE and COUNT

SELECT COUNT(FirstName) FROM orderdb.Customer WHERE State= 'NJ';

Asks: How many customers live in New Jersey?

3

SELECT COUNT(ProductName) FROM orderdb.Product WHERE Price < 3;

Asks: How many products cost less than $3?

2

Review: Does it matter which field in the table you use in the SELECT COUNT query?
One more note: Combining WHERE, GROUP BY, and ORDER BY

Recall the Customer table:

<table>
<thead>
<tr>
<th>CustomerID</th>
<th>FirstName</th>
<th>LastName</th>
<th>City</th>
<th>State</th>
<th>Zip</th>
</tr>
</thead>
<tbody>
<tr>
<td>1001</td>
<td>Greg</td>
<td>House</td>
<td>Princeton</td>
<td>NJ</td>
<td>09120</td>
</tr>
<tr>
<td>1002</td>
<td>Lisa</td>
<td>Cuddy</td>
<td>Plainsboro</td>
<td>NJ</td>
<td>09123</td>
</tr>
<tr>
<td>1003</td>
<td>James</td>
<td>Wilson</td>
<td>Pittsgrove</td>
<td>NJ</td>
<td>09121</td>
</tr>
<tr>
<td>1004</td>
<td>Eric</td>
<td>Foreman</td>
<td>Warminster</td>
<td>PA</td>
<td>19111</td>
</tr>
</tbody>
</table>

Ask:
How many customers are there in each city in New Jersey? Order the results alphabetically by city.
One more note: Combining WHERE, GROUP BY, and ORDER BY

```
SELECT City, COUNT(*)
FROM orderdb.Customer
WHERE State='NJ'
GROUP BY City
ORDER BY City ASC;
```

This is the correct SQL statement

```
SELECT City, COUNT(*)
FROM orderdb.Customer
WHERE State='NJ';
GROUP BY City
ORDER BY City ASC;
```

This won’t work

When combining WHERE, GROUP BY, and ORDER BY, write the WHERE condition first, then GROUP BY, then ORDER BY.
Querying multiple tables

• Right now, you can answer
  – How many customers live in New Jersey?
  – What is the most expensive product sold?

• Because those two questions can be answered looking at only a single table.

• But what if we want to find out the orders a customer placed?

• You need a construct a query that combines two (or more) tables.
The (Inner) Join

• We’ve seen this before

We matched the Order and Customer tables based on the common field (CustomerID)

• We can construct a SQL query to do this
Joining tables using WHERE


Returns this:

<table>
<thead>
<tr>
<th>Customer.CustomerID</th>
<th>FirstName</th>
<th>LastName</th>
<th>City</th>
<th>State</th>
<th>Zip</th>
<th>Order Number</th>
<th>OrderDate</th>
<th>Order.CustomerID</th>
</tr>
</thead>
<tbody>
<tr>
<td>1001</td>
<td>Greg</td>
<td>House</td>
<td>Princeton</td>
<td>NJ</td>
<td>09120</td>
<td>101</td>
<td>2011-3-2</td>
<td>1001</td>
</tr>
<tr>
<td>1002</td>
<td>Lisa</td>
<td>Cuddy</td>
<td>Plainsboro</td>
<td>NJ</td>
<td>09123</td>
<td>102</td>
<td>2011-3-3</td>
<td>1002</td>
</tr>
<tr>
<td>1001</td>
<td>Greg</td>
<td>House</td>
<td>Princeton</td>
<td>NJ</td>
<td>09120</td>
<td>103</td>
<td>2011-3-4</td>
<td>1001</td>
</tr>
<tr>
<td>1004</td>
<td>Eric</td>
<td>Foreman</td>
<td>Warminster</td>
<td>PA</td>
<td>19111</td>
<td>104</td>
<td>2011-3-6</td>
<td>1004</td>
</tr>
</tbody>
</table>

Note that all the fields are there, but depending on the database system, the field order may be different.
A closer look at the JOIN syntax

SELECT * FROM orderdb.Customer, orderdb.Order
WHERE Customer.CustomerID=Order.CustomerID;

<table>
<thead>
<tr>
<th>SELECT *</th>
<th>Return all the columns from both tables</th>
</tr>
</thead>
<tbody>
<tr>
<td>FROM m1orderdb.Customer, m1orderdb.Order</td>
<td>The two tables to be joined</td>
</tr>
<tr>
<td>WHERE Customer.CustomerID = Order.CustomerID</td>
<td>Only choose records where the CustomerID exists in both tables</td>
</tr>
</tbody>
</table>

Another way to say it:
Choose customers that have placed an order

The “.” notation is table_name.column_name
We need this when two tables have the same column name.
A more complex join

- Question: What products did each customer ordered?
- We want to wind up with this view of the database

<table>
<thead>
<tr>
<th>OrderNumber</th>
<th>FirstName</th>
<th>LastName</th>
<th>ProductName</th>
<th>Quantity</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>Greg</td>
<td>House</td>
<td>Cheerios</td>
<td>2</td>
<td>3.99</td>
</tr>
<tr>
<td>101</td>
<td>Greg</td>
<td>House</td>
<td>Bananas</td>
<td>3</td>
<td>1.29</td>
</tr>
<tr>
<td>101</td>
<td>Greg</td>
<td>House</td>
<td>Eggo Waffles</td>
<td>1</td>
<td>2.99</td>
</tr>
<tr>
<td>102</td>
<td>Lisa</td>
<td>Cuddy</td>
<td>Cheerios</td>
<td>5</td>
<td>3.99</td>
</tr>
<tr>
<td>102</td>
<td>Lisa</td>
<td>Cuddy</td>
<td>Bananas</td>
<td>2</td>
<td>1.29</td>
</tr>
<tr>
<td>103</td>
<td>Greg</td>
<td>House</td>
<td>Eggo Waffles</td>
<td>3</td>
<td>2.99</td>
</tr>
<tr>
<td>104</td>
<td>Eric</td>
<td>Foreman</td>
<td>Eggo Waffles</td>
<td>8</td>
<td>2.99</td>
</tr>
</tbody>
</table>
How to do it?

- We need information from Customer and Product (and Order-Product)
- So we need to link all of the tables together
  - To associate Customers with Products we need to follow the path from **Customer** to **Product**

<table>
<thead>
<tr>
<th>Customer</th>
<th>Order</th>
<th>Order-Product</th>
<th>Product</th>
</tr>
</thead>
<tbody>
<tr>
<td>CustomerID</td>
<td>OrderNumber</td>
<td>OrderProductID</td>
<td>ProductID</td>
</tr>
<tr>
<td>FirstName</td>
<td>OrderDate</td>
<td>OrderNumber</td>
<td>ProductName</td>
</tr>
<tr>
<td>LastName</td>
<td>CustomerID</td>
<td>OrderNumber</td>
<td>Price</td>
</tr>
<tr>
<td>City</td>
<td></td>
<td>ProductID</td>
<td></td>
</tr>
<tr>
<td>State</td>
<td></td>
<td>Quantity</td>
<td></td>
</tr>
<tr>
<td>Zip</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note: We will need to use back quotes for `Order` and `Order-Product`
Here’s the query

```sql
SELECT `Order`.OrderNumber, Customer.FirstName, Customer.LastName, Product.ProductName, `Order-Product`.Quantity, Product.Price
WHERE Customer.CustomerID=`Order`.CustomerID
AND `Order`.OrderNumber=`Order-Product`.OrderNumber
AND Product.ProductID=`Order-Product`.ProductID;
```

It looks more complicated than it is!
Note that we have three conditions in the WHERE clause, and we have three relationships in our schema.
Now there are endless variations

- The total cost of all products bought by the customer “Greg House”?

    ```sql
    SELECT SUM(Product.Price*`Order-Product`.Quantity)
    FROM orderdb.Customer, orderdb.`Order`, orderdb.Product,
    orderdb.`Order-Product` WHERE
    Customer.CustomerID=`Order`.CustomerID
    AND `Order`.OrderNumber=`Order-Product`.OrderNumber
    AND Product.ProductID=`Order-Product`.ProductID
    AND Customer.CustomerID=1001;
    ```

    **Answer:** 23.81

    You could have also said
    Customer.LastName='House', but it’s better to use the unique identifier.
What’s with the SUM() function?

• Notice that we’ve introduced something new

```sql
SELECT SUM(Product.Price*`Order-Product`.Quantity)
```

• This multiplies price by quantity for each returned record, and then adds them together.

• You **can** perform arithmetic operations as long as the fields are numeric

**Question:** What do you think would get returned if you left off the SUM() and just had

```sql
SELECT Product.Price * Product.Quantity
```
LIMITing Results

We know that this...

```
SELECT * FROM orderdb.Product
ORDER BY Price DESC;
```

Gives us this...

<table>
<thead>
<tr>
<th>ProductID</th>
<th>ProductName</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>2251</td>
<td>Cheerios</td>
<td>3.99</td>
</tr>
<tr>
<td>2505</td>
<td>Eggo Waffles</td>
<td>2.99</td>
</tr>
<tr>
<td>2282</td>
<td>Bananas</td>
<td>1.29</td>
</tr>
</tbody>
</table>

What if we want the two most expensive products?

```
SELECT * FROM orderdb.Product WHERE Price >= 2.99;
```

Only works if we know all the prices beforehand...

...but then we wouldn’t need the query!
The LIMIT clause...

```sql
SELECT * FROM orderdb.Product
ORDER BY Price DESC
LIMIT 2;
```

This says:

- Give me all the columns
- Put rows in descending order by price
- But only give me the first two results

<table>
<thead>
<tr>
<th>ProductID</th>
<th>ProductName</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>2251</td>
<td>Cheerios</td>
<td>3.99</td>
</tr>
<tr>
<td>2505</td>
<td>Eggo Waffles</td>
<td>2.99</td>
</tr>
</tbody>
</table>

What would we get if we left out DESC?
SQL Subselects

We could also try to use LIMIT to find the least expensive product:

```sql
SELECT * FROM orderdb.Product
ORDER BY Price ASC LIMIT 1;
```

But what if there is more than one product with the lowest value for price AND we don’t know how many there are?
Where MIN() alone fails us...

SELECT MIN(price)
FROM orderdb.Product;

BUT

SELECT MIN(price),ProductName
FROM orderdb.Product;

So what’s going on??
What’s wrong...

SELECT MIN(price), ProductName
FROM orderdb.Product;

It returns the MIN(price)

MIN() will always return only one row

It chooses the first row in the Product column

And it will do this for any function (AVG, SUM, etc.)

<table>
<thead>
<tr>
<th>Price</th>
<th>ProductName</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.29</td>
<td>Cheerios</td>
</tr>
</tbody>
</table>
So we need a SQL subselect statement

It’s where you have a SELECT statement nested inside another SELECT statement!

```
SELECT Price, ProductName
FROM orderdb.Product
WHERE Price =
  (SELECT MIN(Price) FROM orderdb.Product);
```

Now you get all records back with that (lowest) price and avoid the quirk of the MIN() function.

This is a **temporary table** from the database with one column and one row.
Subselects come in handy in other situations too...

We want to get a COUNT of how many DISTINCT states there are in the table.

```sql
SELECT COUNT(*) FROM (SELECT DISTINCT State FROM orderdb.Customer) AS tmp1;
```

- To see how this works:
  - Start with the SELECT DISTINCT...
  - ...then COUNT those values

<table>
<thead>
<tr>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>NJ</td>
</tr>
<tr>
<td>PA</td>
</tr>
</tbody>
</table>

2
Why do we need AS?

SELECT COUNT(*) FROM (SELECT DISTINCT State FROM orderdb.Customer) AS tmp1;

• You’re basically SELECTing from the temporary table generated by the nested query.

• But since you’re SELECTing FROM that temporary table you have to give it a name (i.e., tmp1)
More about AS: SQL Aliases

SQL aliases are used to give a table or a column a temporary name.

• SELECT column_name(s)
  FROM schema_name.table_name AS alias_name;

• SELECT column_name AS alias_name
  FROM schema_name.table_name;
Summary: The full syntax for SELECT

```
SELECT [DISTINCT] expression(s) [AS alias_name]
FROM table(s) [AS alias_name]
[WHERE condition(s)]
[GROUP BY expression(s)]
[ORDER BY expression(s) [ ASC | DESC ]]
[LIMIT number_rows];
```

<table>
<thead>
<tr>
<th>Element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>expression(s)</td>
<td>The column(s) or function(s) that you wish to retrieve.</td>
</tr>
<tr>
<td>table(s)</td>
<td>The table(s) that you wish to retrieve records from.</td>
</tr>
<tr>
<td>DISTINCT</td>
<td>Optional. Return unique values.</td>
</tr>
<tr>
<td>WHERE condition(s)</td>
<td>Optional. The conditions that must be met for the records to be selected.</td>
</tr>
<tr>
<td>GROUP BY expression(s)</td>
<td>Optional. Organize the results by column values.</td>
</tr>
<tr>
<td>ORDER BY expression(s)</td>
<td>Optional. Sort the records in your result set</td>
</tr>
<tr>
<td>LIMIT number_rows</td>
<td>Optional. Restrict the maximum number of records to retrieve.</td>
</tr>
<tr>
<td>AS alias_name</td>
<td>Optional. Give a table or a column a temporary name</td>
</tr>
</tbody>
</table>

The [] means the element is optional.
Summary

• Given a schema of a database, be able to create a SQL statement to answer a question

• Understand how to use
  – SELECT ... FROM ...
  – DISTINCT
  – WHERE (and how to specify conditions)
  – AND/OR
  – Functions: COUNT, AVG, MIN, MAX, SUM
  – GROUP BY
  – ORDER BY (ASC/DESC)
  – LIMIT
  – Joins
  – Subselects
  – AS (aliases)