SQL Workshop

By: Kyaw Swar Ye Myint

Before we begin...

Please complete this survey

https://forms.gle/jvt7rJ2AFwJRtBJo6

Kyaw Swar Ye Myint

Senior
Data Science and Economics
Fall 2023

SQL Basics

*MUST BE IN THIS ORDER

SELECT <column expression list>	-	choose columns
FROM <single table>	-	choose table(s)
WHERE <predicate>	-	choose row(s)
GROUP BY <column list>	-	group values based on a column(s)
HAVING <predicate>	-	choose group(s)
ORDER BY <column list>	-	order results by column(s)
LIMIT <integer>;	-	number of results

SQL Basics - SELECT, FROM

SELECT <column expression list>

FROM <single table>

WHERE <predicate>

GROUP BY <column list>

HAVING <predicate>

ORDER BY <column list>

LIMIT <integer>;

SELECT, FROM

- Mandatory as part of every SQL statement
- **SELECT** columnA, columnB
  - We can do arithmetic and apply functions to the columns when selecting as well
  - Asterisk (*) denotes “all”
- **FROM** table1, table2

```
SELECT * FROM table;

SELECT table.weight, table.height FROM table;

SELECT max(income) FROM salaries_table;

SELECT speed / time FROM physics_table;
```

SQL Basics - WHERE

SELECT [DISTINCT] <column expression list>

FROM <single table>

WHERE <predicate>

GROUP BY <column list>

HAVING <predicate>

ORDER BY <column list>

LIMIT <integer>;

WHERE

The WHERE clause allows us to specify certain constraints for the returned data

- Constraints are often referred to as **predicates**

Like a SELECT but for the rows

We can also use the operators AND, OR, and NOT to further constrain our SQL query.

WHERE

- Possible boolean operators: >, <, >=, <=, !
- Combining conditions:
    - AND
    - OR
    - NOT

```
SELECT * FROM table1
WHERE column1_name BOOL OPERATOR column1_value
  AND column2_name BOOL OPERATOR column2_value;
```

SQL Basics - DISTINCT, COUNT, ORDER BY, LIMIT

SELECT [DISTINCT] <column expression list>

FROM <single table>

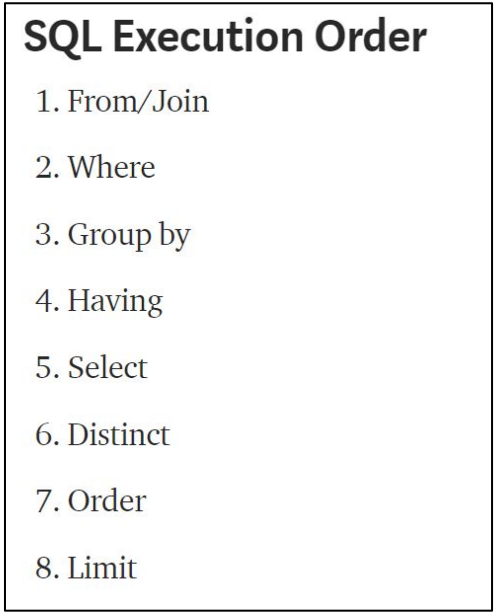
WHERE <predicate>

GROUP BY <column list>

HAVING <predicate>

ORDER BY <column list>

LIMIT <integer>;



SQL Execution Order

1. From/Join
2. Where
3. Group by
4. Having
5. Select
6. Distinct
7. Order
8. Limit

ORDER BY

- Orders the rows in our resulting table based on given column(s)
    - **ASC** by default; can flag **DESC**
- Can order by multiple columns

```
SELECT * FROM table1
ORDER BY ColA DESC, ColB ASC;
```

(breaks ties with column B)

LIMIT

LIMIT <integer>

Specifies a limited number of rows in the result set to be returned based on <integer>

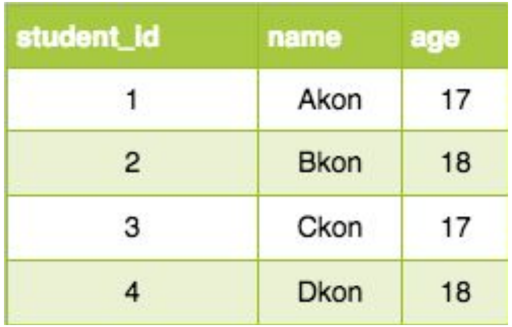
For example, **LIMIT 10** would return the first 10 rows matching the SELECT criteria → happens last

```
SELECT * FROM table1
ORDER BY colA
LIMIT 5;
```

Intro to Aggregations

- So far, we've only worked with data from the existing rows in the table; that is, our queries return some subset of the entries found in the table
- To conduct data analysis, we'll want to aggregate/summarize our data
- In SQL, this is done using **aggregate functions.**
  - Max, min, avg, sum, count are some common ones

```
SELECT MAX(age)
FROM students
```



student_Id	name	age
1	Akon	17
2	Bkon	18
3	Ckon	17
4	Dkon	18

SQL Basics - GROUP BY, HAVING

SELECT [DISTINCT] <column expression list>

FROM <single table>

WHERE <predicate>

GROUP BY <column list>

HAVING <predicate>

ORDER BY <column list>

LIMIT <integer>;

GROUP BY

GROUP BY takes in a column, and returns a row for each unique value in that column → groups each of the other columns by an aggregate function

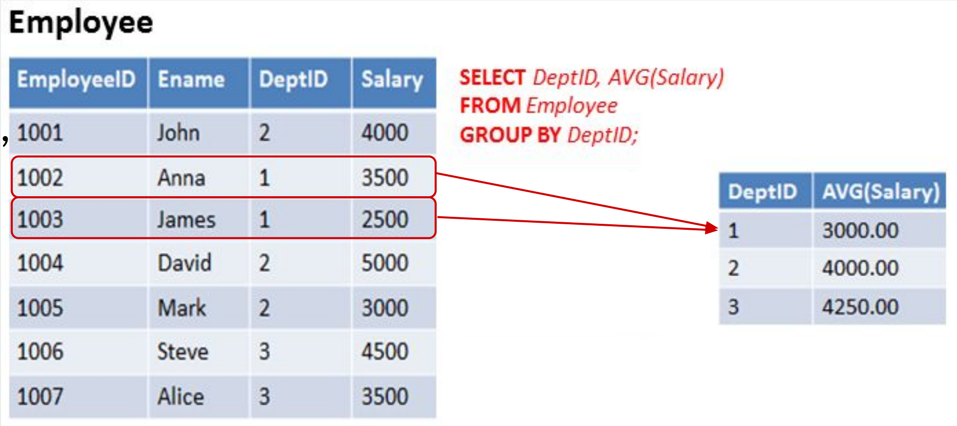
The grouping is specified in the **SELECT** statement

GROUP BY

GROUP BY takes in a column, and returns a row for each unique value in that column → groups each of the other columns by an aggregate function

The grouping is specified in the **SELECT** statement

- Notice that we can still select `DeptID` without aggregating it, but we can't select other columns without applying an aggregate function



Employee

EmployeeID	Ename	DeptID	Salary
1001	John	2	4000
1002	Anna	1	3500
1003	James	1	2500
1004	David	2	5000
1005	Mark	2	3000
1006	Steve	3	4500
1007	Alice	3	3500

```
SELECT DeptID, AVG(Salary)
FROM Employee
GROUP BY DeptID;
```

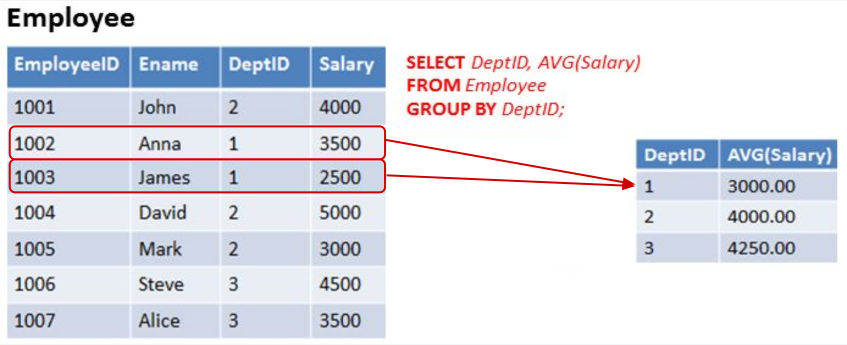
DeptID	AVG(Salary)
1	3000.00
2	4000.00
3	4250.00

GROUP BY

Q: What if the select statement became:

```
SELECT DeptID, AVG(Salary), AVG(Ename)
```

A: For MySQL, it would return a column of 0’s! But in general, the behavior of AVG(text) is undefined, so it may error for other versions of SQL.



Employee

EmployeeID	Ename	DeptID	Salary
1001	John	2	4000
1002	Anna	1	3500
1003	James	1	2500
1004	David	2	5000
1005	Mark	2	3000
1006	Steve	3	4500
1007	Alice	3	3500

```
SELECT DeptID, AVG(Salary)
FROM Employee
GROUP BY DeptID;
```

DeptID	AVG(Salary)
1	3000.00
2	4000.00
3	4250.00

HAVING

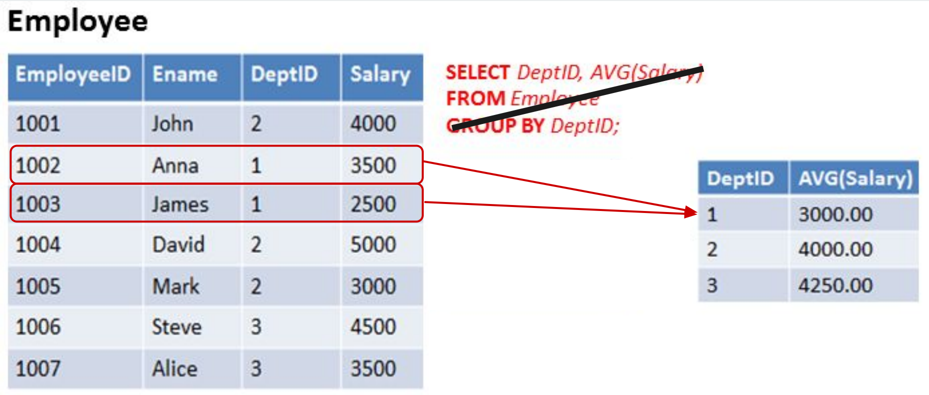
- **HAVING** is functionally similar to WHERE, but is used exclusively to apply predicates to aggregated data.
- In order to use HAVING, it **must** be preceded by a GROUP BY clause

HAVING

- **HAVING** is functionally similar to WHERE, but is used exclusively to apply predicates to aggregated data.
- In order to use HAVING, it **<u>must</u>** be preceded by a GROUP BY clause

What if we used this query instead?

```
SELECT DeptId, AVG(Salary)
FROM Employee
GROUP BY DeptId
Having AVG(Salary) > 3000;
```



Employee

EmployeeID	Ename	DeptID	Salary
1001	John	2	4000
1002	Anna	1	3500
1003	James	1	2500
1004	David	2	5000
1005	Mark	2	3000
1006	Steve	3	4500
1007	Alice	3	3500

~~SELECT DeptID, AVG(Salary)~~
~~FROM Employee~~
~~GROUP BY DeptID;~~

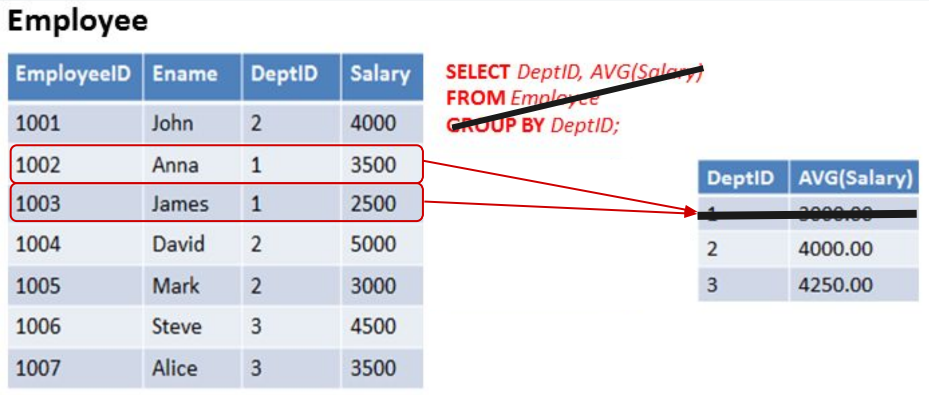
DeptID	AVG(Salary)
1	3000.00
2	4000.00
3	4250.00

HAVING

- **HAVING** is functionally similar to WHERE, but is used exclusively to apply predicates to aggregated data.
- In order to use HAVING, it **<u>must</u>** be preceded by a GROUP BY clause

What if we used this query instead?

```
SELECT DeptId, AVG(Salary)
FROM Employee
GROUP BY DeptId
Having AVG(Salary) > 3000;
```



Employee

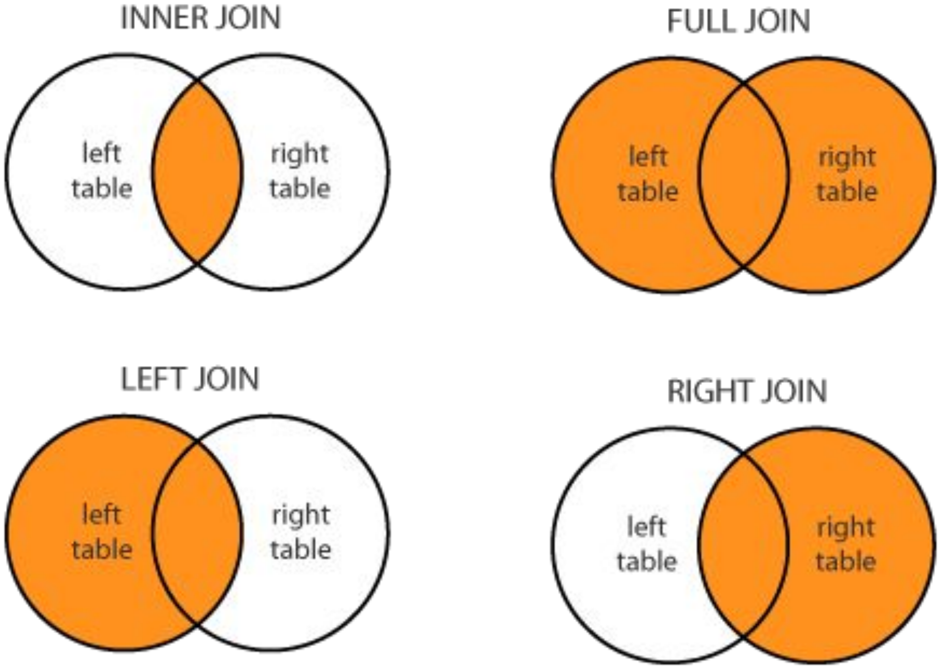
EmployeeID	Ename	DeptID	Salary
1001	John	2	4000
1002	Anna	1	3500
1003	James	1	2500
1004	David	2	5000
1005	Mark	2	3000
1006	Steve	3	4500
1007	Alice	3	3500

~~SELECT DeptID, AVG(Salary)~~
~~FROM Employee~~
~~GROUP BY DeptID;~~

DeptID	AVG(Salary)
~~1~~	~~3000.00~~
2	4000.00
3	4250.00

SQL Joins

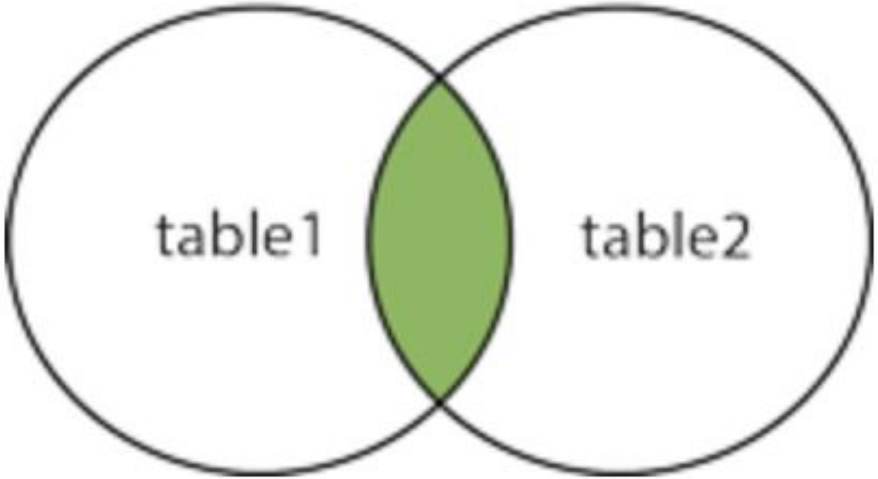
- Inner
- Left
- Right
- Full aka Outer



SQL Joins

- **Inner**
- Left
- Right
- Outer

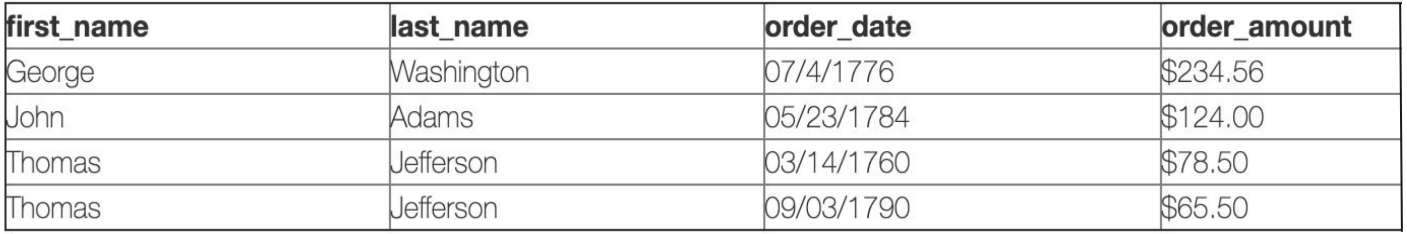
INNER JOIN



```
SELECT * FROM table1 INNER JOIN table2
ON table1.col = table2.col;
```

Inner Join Example:

List of those customers who placed an order and the details of the order they placed



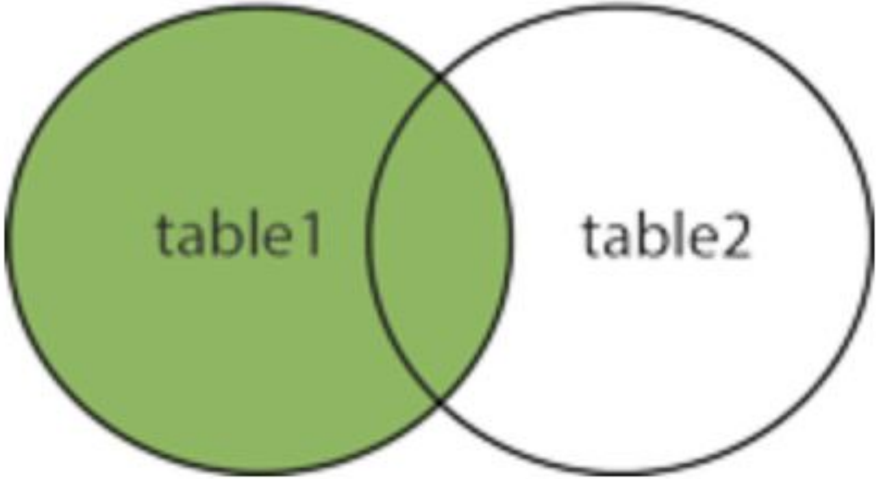
first_name	last_name	order_date	order_amount
George	Washington	07/4/1776	$234.56
John	Adams	05/23/1784	$124.00
Thomas	Jefferson	03/14/1760	$78.50
Thomas	Jefferson	09/03/1790	$65.50

```
SELECT first_name, last_name, order_date, order_amount

FROM customers c

INNER JOIN orders o ON c.customer_id = o.customer_id
```

SQL Joins

- Inner
- **Left**
- Right
- Outer

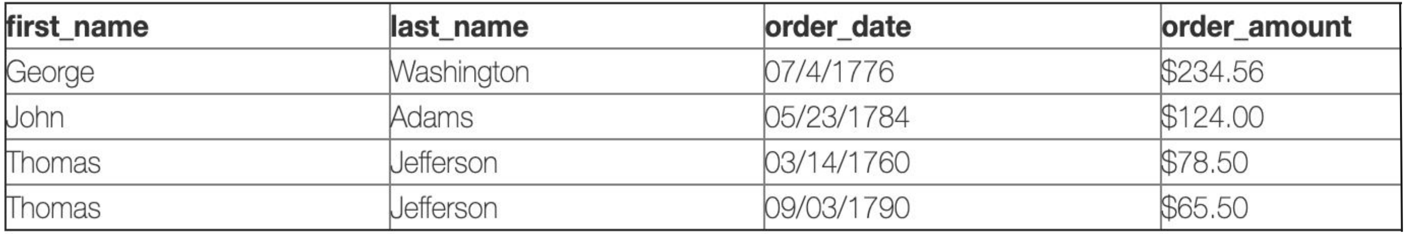
LEFT JOIN



```
SELECT * FROM table1 LEFT JOIN table2
ON table1.col = table2.col;
```

Left Join Example:

append information about orders to our customers table, regardless of whether a customer placed an order or not, we would use a left join.



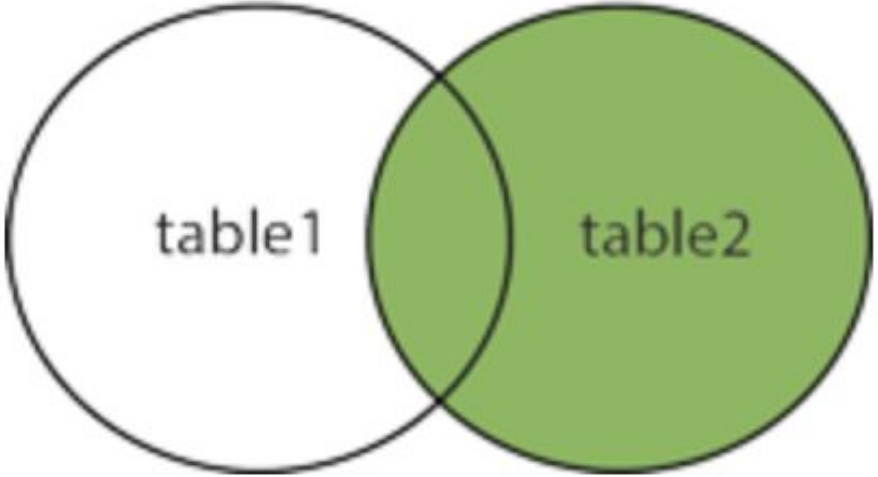
first_name	last_name	order_date	order_amount
George	Washington	07/4/1776	$234.56
John	Adams	05/23/1784	$124.00
Thomas	Jefferson	03/14/1760	$78.50
Thomas	Jefferson	09/03/1790	$65.50

```
select first_name, last_name, order_date, order_amount

from customers c

left join orders o

on c.customer_id = o.customer_id

where order_date is NULL
```

SQL Joins

- Inner
- Left
- **Right**
- Outer

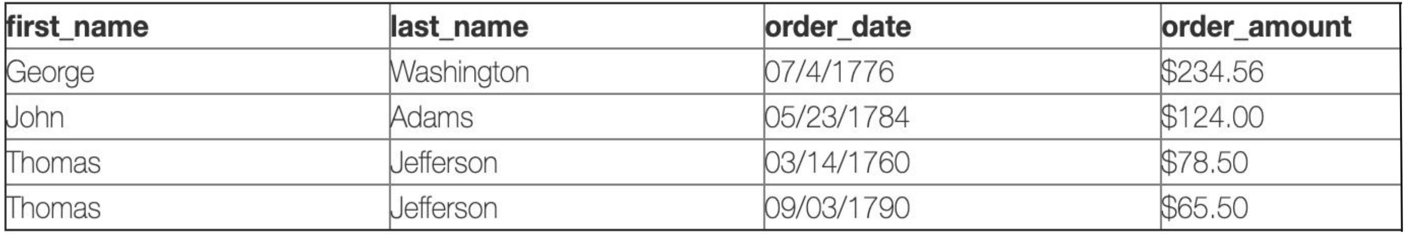
RIGHT JOIN



```
SELECT * FROM table1 RIGHT JOIN table2
ON table1.col = table2.col;
SELECT * FROM table2 LEFT JOIN table1
ON table1.col = table2.col;
```

Right Join Example:

a mirror version of the left join and allows to get a list of all orders, appended with customer information.



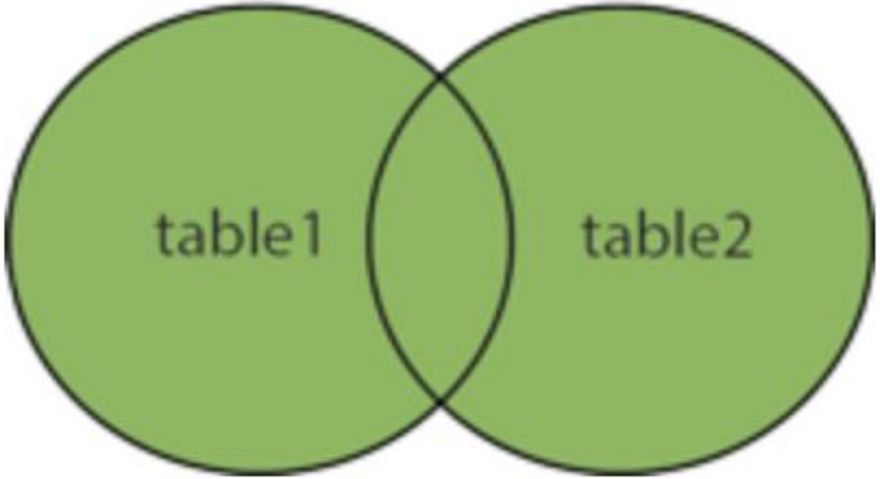
first_name	last_name	order_date	order_amount
George	Washington	07/4/1776	$234.56
John	Adams	05/23/1784	$124.00
Thomas	Jefferson	03/14/1760	$78.50
Thomas	Jefferson	09/03/1790	$65.50

```
SELECT first_name, last_name, order_date, order_amount

FROM customers c

RIGHT JOIN orders o ON c.customer_id = o.customer_id
```

SQL Joins

- Inner
- Left
- Right
- **Outer**

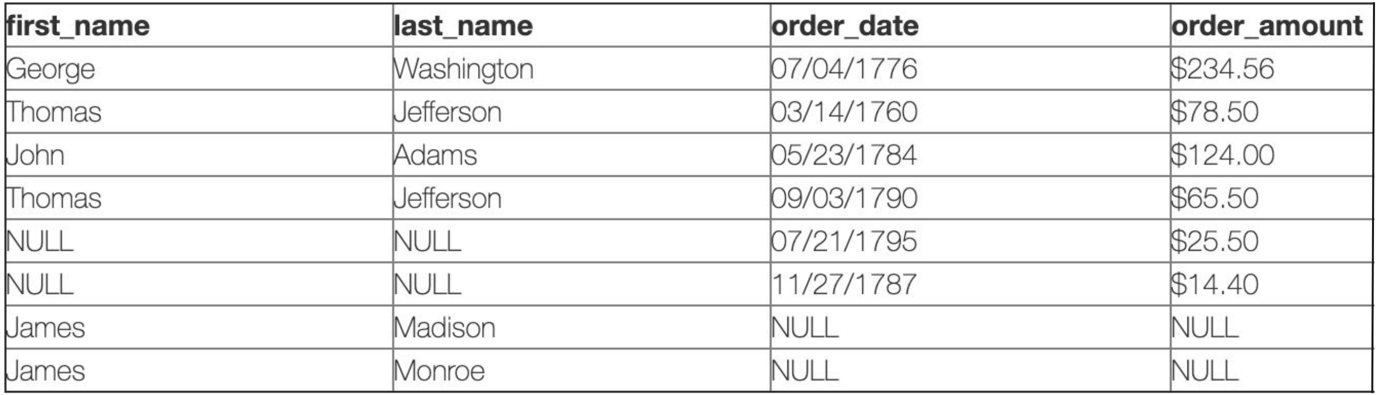
OUTER JOIN



```
SELECT * FROM table1 OUTER JOIN table2
ON table1.col = table2.col;
SELECT * FROM table1, table2
WHERE table1.col = table2.col;
```

Outer Join Example:

for a list of all records from both tables, we can use an outer join.



first_name	last_name	order_date	order_amount
George	Washington	07/04/1776	$234.56
Thomas	Jefferson	03/14/1760	$78.50
John	Adams	05/23/1784	$124.00
Thomas	Jefferson	09/03/1790	$65.50
NULL	NULL	07/21/1795	$25.50
NULL	NULL	11/27/1787	$14.40
James	Madison	NULL	NULL
James	Monroe	NULL	NULL

```
SELECT first_name, last_name, order_date, order_amount

FROM customers c

OUTER JOIN orders o

on c.customer_id = o.customer_id
```

LIKE Operator

- **Used in the WHERE clause to search for a specified pattern in a column**
  - % - represents zero, one, or multiple characters
  - _ - represents a single character
- **Using %**
  - WHERE Names LIKE ‘a%’ - any number of characters after ‘a’ (starts with ‘a’)
  - WHERE Names LIKE ‘%a’ - any number of characters before ‘a’ (ends with ‘a’)
  - WHERE Names LIKE ‘%a%’ - any entries with the letter ‘a’ in it
- **Using _**
  - WHERE Names LIKE ‘a__%’ - where ‘a’ is the first character followed by AT LEAST two more characters

SUBSTRING() Function

- **Extract characters from a string**
  - SUBSTRING(string, start, length)
  - string: the string to extract from
  - start: the starting position you want to extract
  - length: how many characters you want
- **SELECT SUBSTRING(‘SQL Substring’, 1, 3)**
  - Output will be ‘SQL’
- **SQL indexes start at 1*****

Views

- **CREATE VIEW creates a virtual table that you can access in your query**
  - Helpful when you want to create a new table that isn't in your database
  - Can use this database to join with another existing table
  - Similar syntax
- CREATE VIEW *view_name* AS

  SELECT *column1*, *column2*, ...

  FROM *table_name*

  WHERE *condition*;

  SELECT *

  FROM *view_name*, *table_name*

Subqueries

- **A subquery is a SQL query nested inside a larger query**
    - **Usually in the FROM and/or WHERE clauses**
- **FROM clause:**
    - Subquery where the output will be another table
    - ex) FROM (SELECT *column_x*, FROM *table_name* WHERE *predicate_x*) AS *table2*, *table1*
- **WHERE clause:**
    - Subquery where the output is a value
    - ex) WHERE max_length = (SELECT MAX(duration) FROM films)

Thank you!

CONTACT

Data Peer Consultants

- Drop-in hours: 12PM - 4PM, Monday - Friday
  - http://data.berkeley.edu/dpc-drop-in
- Email: ds-peer-consulting@berkeley.edu

D-Lab

- Virtual Front-Desk hours: 9AM - 5PM, Monday - Friday
  - https://dlab.berkeley.edu/frontdesk