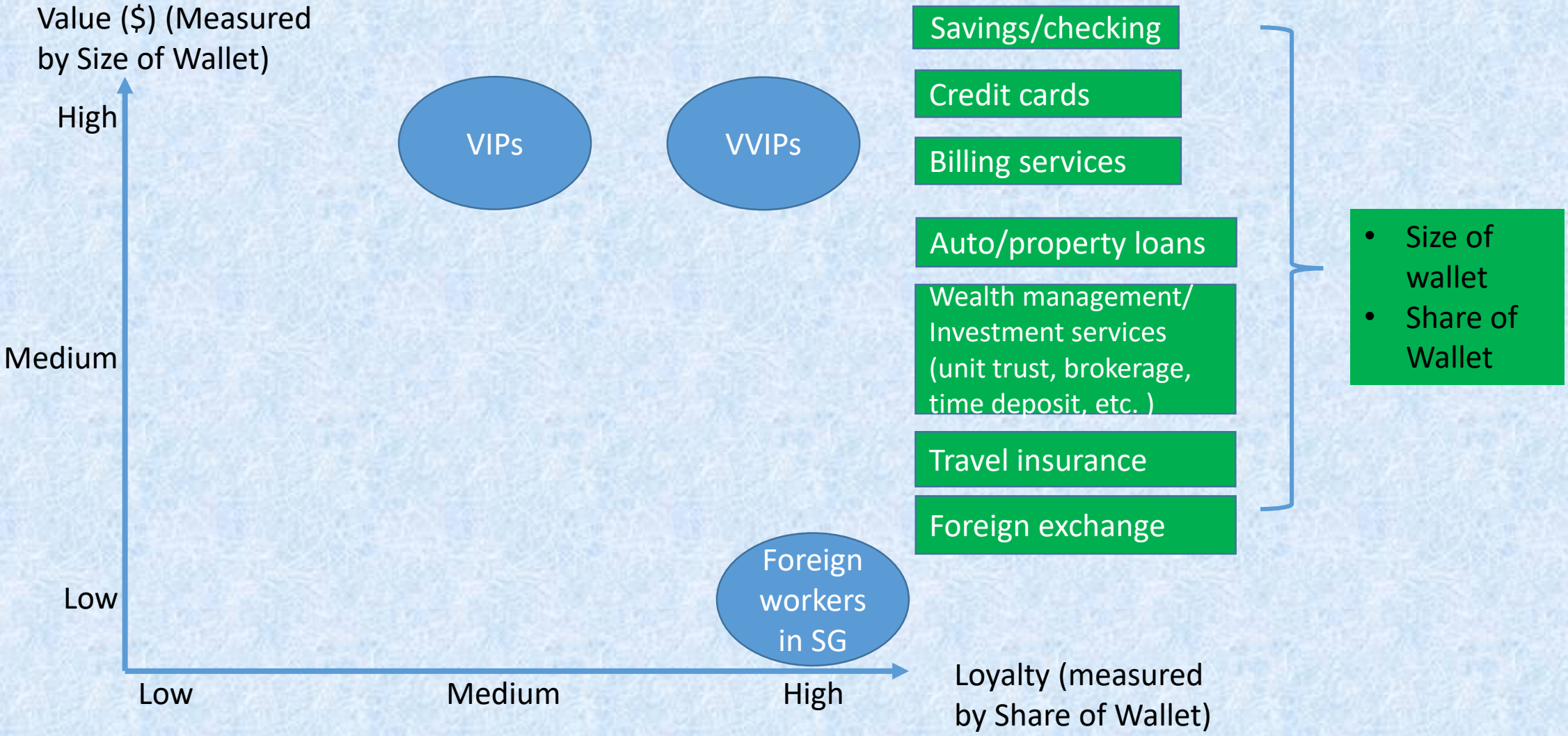
SQL (Structured Query Language) in Customer Relationship Management

Chu Junhong

NUS Business School

Junhong.chu@nus.edu.sg

Value vs. Loyalty: Two Dimensions for Customer Segmentation (DBS)



Online Resources

- I use **Jupyter Notebook** as a text editor for SQL

  (you can use any other text editors (e.g., Atom, EditPlus) that you are comfortable with)

- I follow the instruction on classroom.UDAcity.com
  - https://classroom.udacity.com/courses/ud198

- Other useful websites for SQL:
  - https://www.w3schools.com/sql/
  - https://www.khanacademy.org/computing/computer-programming/sql
  - https://www.khanacademy.org/computer-programming/new/sql
  - https://www.edx.org/course/advanced-topics-in-sql

Table of Content

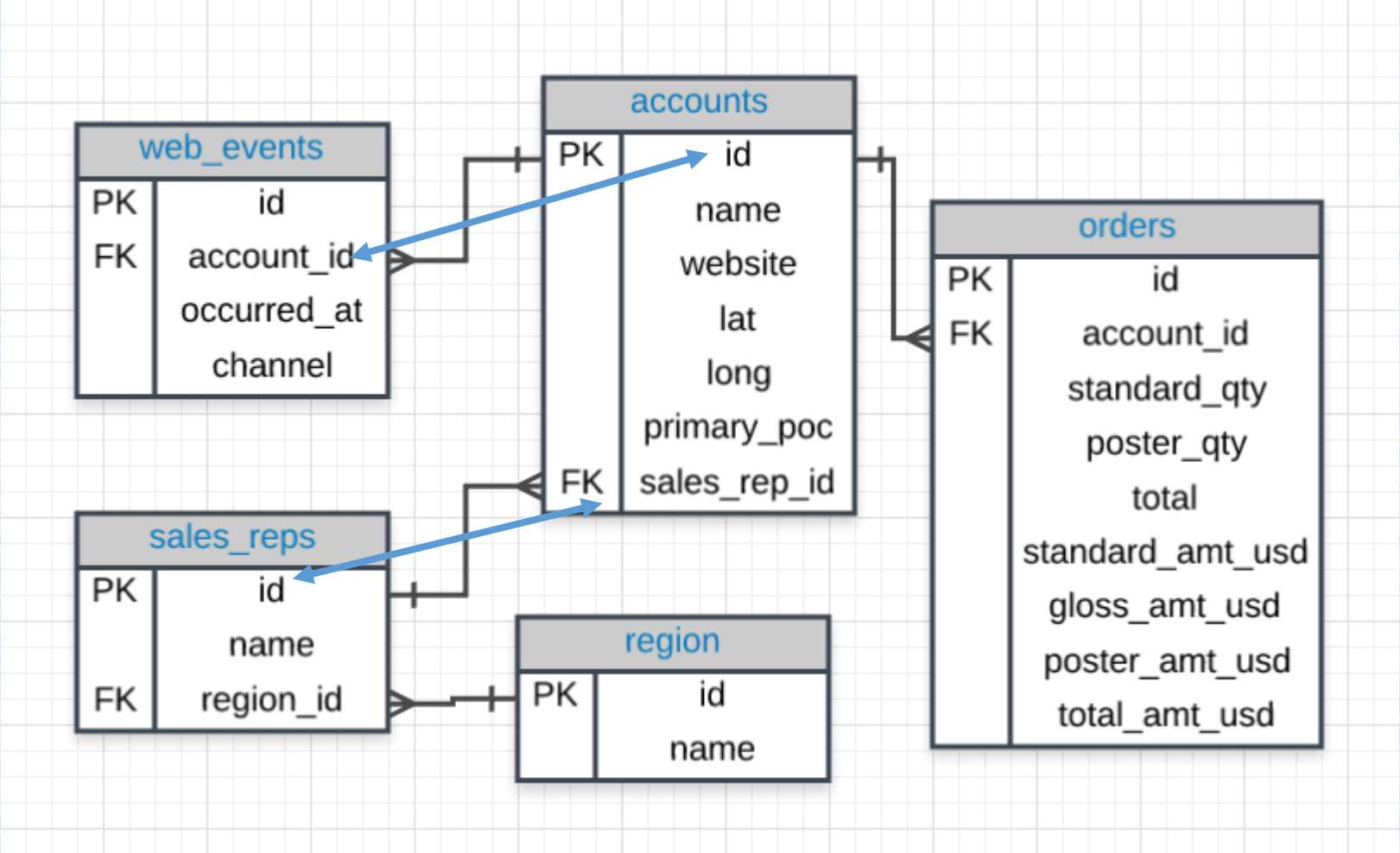
Basics of Database: Entity Relationship Diagrams (ERD)

In the Parch & Posey database there are 5 tables:

- **web_events**
- **accounts**
- **orders**
- **sales_reps**
- **Region**

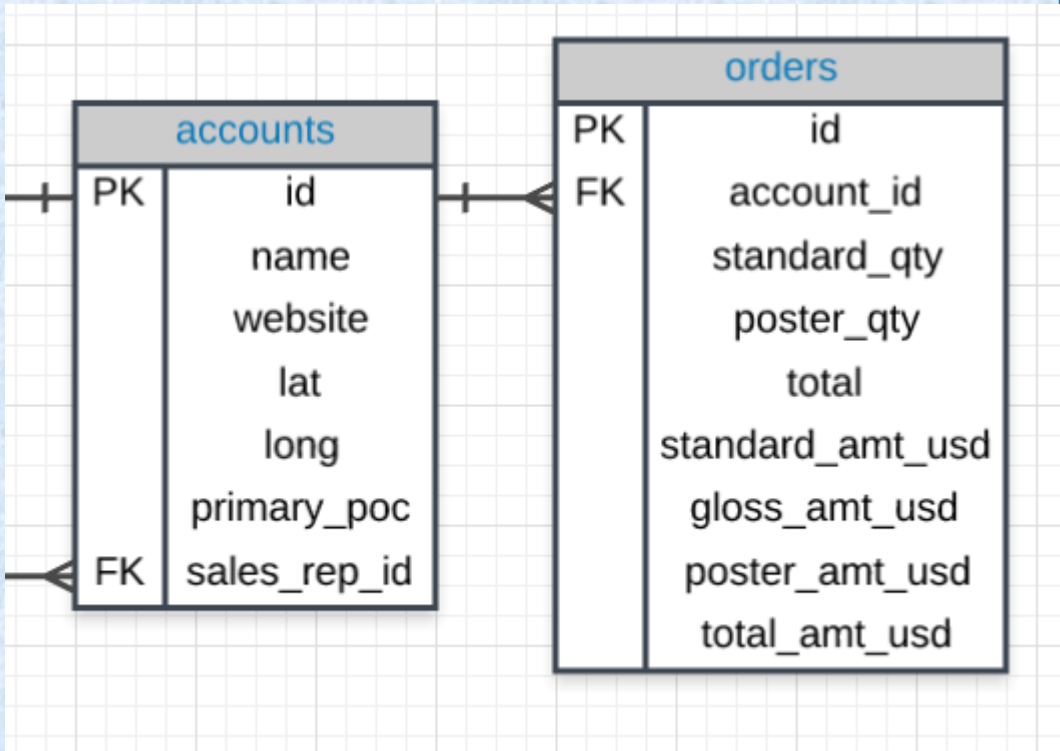
- PK: Primary key
- FK: Foreign key

(FK = PK in another table)



Keys

- **Primary Key (PK)**
  - A **primary key** is a unique column in a particular table. This is the first column in each of our tables. Here, those columns are all called **id**, but that doesn't necessarily have to be the name. **It is common that the primary key is the first column in our tables in most databases.**
- **Foreign Key (FK)**
  - A **foreign key** is a column in one table that is a primary key in a different table. We can see in the Parch & Posey ERD that the foreign keys are:
  - **region_id**
  - **account_id**
  - **sales_rep_id**



Some of the Most Popular Databases

- MySQL (next week)
- Access
- Oracle
- Microsoft SQL Server
- Postgres (Used in the web lessons on UDAcity.com)

Some Basic SQL Formatting Conventions

- SQL is **NOT** case sensitive, but character variables are case sensitive
- Conventionally,
  - Use **UPPER CASE** for SQL functions or keywords
  - Use **lower case** for variable and table names
- Ending with “;” (a good practice, though some databases do not require it)

Part I SQL Basic Statements

Statement	How to Use It	Other Details
CREATE	CREATE table_name	Create tables
DROP	DROP table_name	Drop/delete tables
SELECT	SELECT **Col1**, **Col2**, ...	Provide the columns you want
FROM	FROM **Table**	Provide the table where the columns exist
LIMIT	LIMIT **10**	Limits the number of rows returned
ORDER BY	ORDER BY **Col**	Orders table based on the column. Used with **DESC**.
WHERE	WHERE **Col > 5**	A conditional statement to filter your results
LIKE	WHERE **Col LIKE '%me%'**	Only pulls rows where column has 'me' within the text
IN	WHERE **Col IN ('Y', 'N')**	A filter for only rows with column of 'Y' or 'N'
NOT	WHERE **Col NOT IN ('Y', 'N')**	**NOT** is frequently used with **LIKE** and **IN**
AND	WHERE **Col1 > 5 AND Col2 < 3**	Filter rows where two or more conditions must be true
OR	WHERE **Col1 > 5 OR Col2 < 3**	Filter rows where at least one condition must be true
BETWEEN	WHERE **Col BETWEEN 3 AND 5**	Often easier syntax than using an **AND**

Typical SQL Block

```
/*typical SQL block */
SELECT          /* variable names separated by ","  */
FROM            /* file names separated by ","      */
WHERE           /* logical conditions               */
GROUP BY        /* do aagregation                   */
ORDER BY        /* sort the table by variables      */
LIMIT           /* limit the # of output            */
```

SELECT … FROM …; ORDER BY, LIMIT

```
SELECT id, occurred_at                    -- columns / variables in "orders"
FROM orders                               -- table name
ORDER BY occurred_at DESC                 -- must appear AFTER FROM and BEFORE LIMIT
LIMIT 15;                                 -- limit the # of output
```

```
SELECT id, occurred_at,  total_amt_usd
FROM orders
ORDER BY occurred_at, total_amt_usd DESC   -- order by 2 variables
LIMIT 20;

SELECT id, occurred_at,  total_amt_usd
FROM orders
ORDER BY 2, 3 DESC                         -- can use numbers (order of appearance in SELECT
LIMIT 20;
```

WHERE statement to filter data

Common symbols used in **WHERE** statements include:

1. `>` (greater than)
2. `<` (less than)
3. `>=` (greater than or equal to)
4. `<=` (less than or equal to)
5. `=` (equal to)
6. `!=` (not equal to)

```sql
-- Pulls the first 5 rows and all columns from the orders table
-- that have a dollar amount of gloss_amt_usd >= 1000.
SELECT *
FROM orders
WHERE gloss_amt_usd >= 1000
LIMIT 5;

-- Pulls the first 10 rows and all columns from the orders table
-- that have a total_amt_usd less than 500.
SELECT *
FROM orders
WHERE total_amt_usd < 500
LIMIT 10;

-- Pulls the 10 rows and all columns from the orders table
-- that have the largest total_amt_usd and that are less than 500.
SELECT *
FROM orders
WHERE total_amt_usd < 500
ORDER BY total_amt_usd DESC
LIMIT 10;
```

WHERE statement with non-numericals

- =, !=,
- **LIKE, NOT, IN, IS**

```sql
SELECT name, website, primary_poc
FROM accounts
WHERE name = 'Exxon Mobil';

SELECT name, website, primary_poc
FROM accounts
WHERE name != 'Exxon Mobil'
LIMIT 15;

SELECT name, website, primary_poc
FROM accounts
WHERE name IN ('Exxon Mobil', 'Walmart', 'Apple');

SELECT name, website, primary_poc
FROM accounts
WHERE name NOT IN ('Exxon Mobil', 'Walmart', 'Apple')
LIMIT 15;

SELECT name, website, primary_poc, sales_rep_id
FROM accounts
WHERE name NOT IN ('Exxon Mobil', 'Walmart', 'Apple') AND sales_rep_id >1100
LIMIT 15;

SELECT *
FROM accounts
WHERE name NOT IN ('Exxon Mobil', 'Walmart', 'Apple') and sales_rep_id >1100
LIMIT 15;
```

Logic Operators in WHERE statement

1. **LIKE**

   This allows you to perform operations similar to using **WHERE** and `=`, but for cases when you might **not** know **exactly** what you are looking for.

2. **IN**

   This allows you to perform operations similar to using **WHERE** and `=`, but for more than one condition.

3. **NOT**

   This is used with **IN** and **LIKE** to select all of the rows **NOT LIKE** or **NOT IN** a certain condition.

4. **AND & BETWEEN**

   These allow you to combine operations where all combined conditions must be true.

5. **OR**

   This allow you to combine operations where at least one of the combined conditions must be true.

LIKE operator

- The **LIKE** operator is extremely useful for working with text.
- The **LIKE** operator is frequently used with % (wild key)

```sql
--      All the companies whose names start with 'C'.
SELECT name
FROM accounts
WHERE name LIKE 'C%';

--      All companies whose names contain the string 'one' somewhere in the name
SELECT name
FROM accounts
WHERE name LIKE '%one%';

--      All companies whose names end with 's'.
SELECT name
FROM accounts
WHERE name LIKE '%s';
```

IN, NOT IN, LIKE, NOT LIKE

```sql
-- Use the web_events table to find all info for individuals contacted via organic or adwords channel
SELECT *
FROM web_events
WHERE channel IN ('organic', 'adwords');

-- Use the web_events table to find all info for individuals not contacted via organic or adwords channel
SELECT *
FROM web_events
WHERE channel NOT IN ('organic', 'adwords');

-- All the companies whose names do not start with 'C'.
SELECT name
FROM accounts
WHERE name NOT LIKE 'C%';

-- All the companies whose names start with 'C'.
SELECT name
FROM accounts
WHERE name LIKE 'C%';

-- All companies whose names do not contain the string 'one' somewhere in the name.
SELECT name
FROM accounts
WHERE name NOT LIKE '%one%';
```

AND, BETWEEN, OR operators

```sql
SELECT *
FROM orders
WHERE (standard_qty > 1000) and (poster_qty + gloss_qty) = 0;

-- all the companies whose names do not start with 'C' and do end with 's'.
SELECT name
FROM accounts
WHERE name NOT LIKE 'C%' AND name LIKE '%s';

-- From the orders table, find all gloss_qty between 24 and 29
SELECT occurred_at, gloss_qty
FROM orders
WHERE gloss_qty BETWEEN 24 AND 29;   --inclusive

-- Use the web_events table to find all info regarding individuals contacted
-- via the organic or adwords channels, and started their account at any point
-- in 2016, sorted from newest to oldest.
SELECT *
FROM web_events
WHERE channel IN ('organic', 'adwords') AND occurred_at BETWEEN '2016-01-01' AND '2017-01-01'
ORDER BY occurred_at DESC;

-- or
SELECT *
FROM web_events
WHERE channel IN ('organic', 'adwords') AND (DATE_PART('year', occurred_at) = 2016)
ORDER BY occurred_at DESC;
```

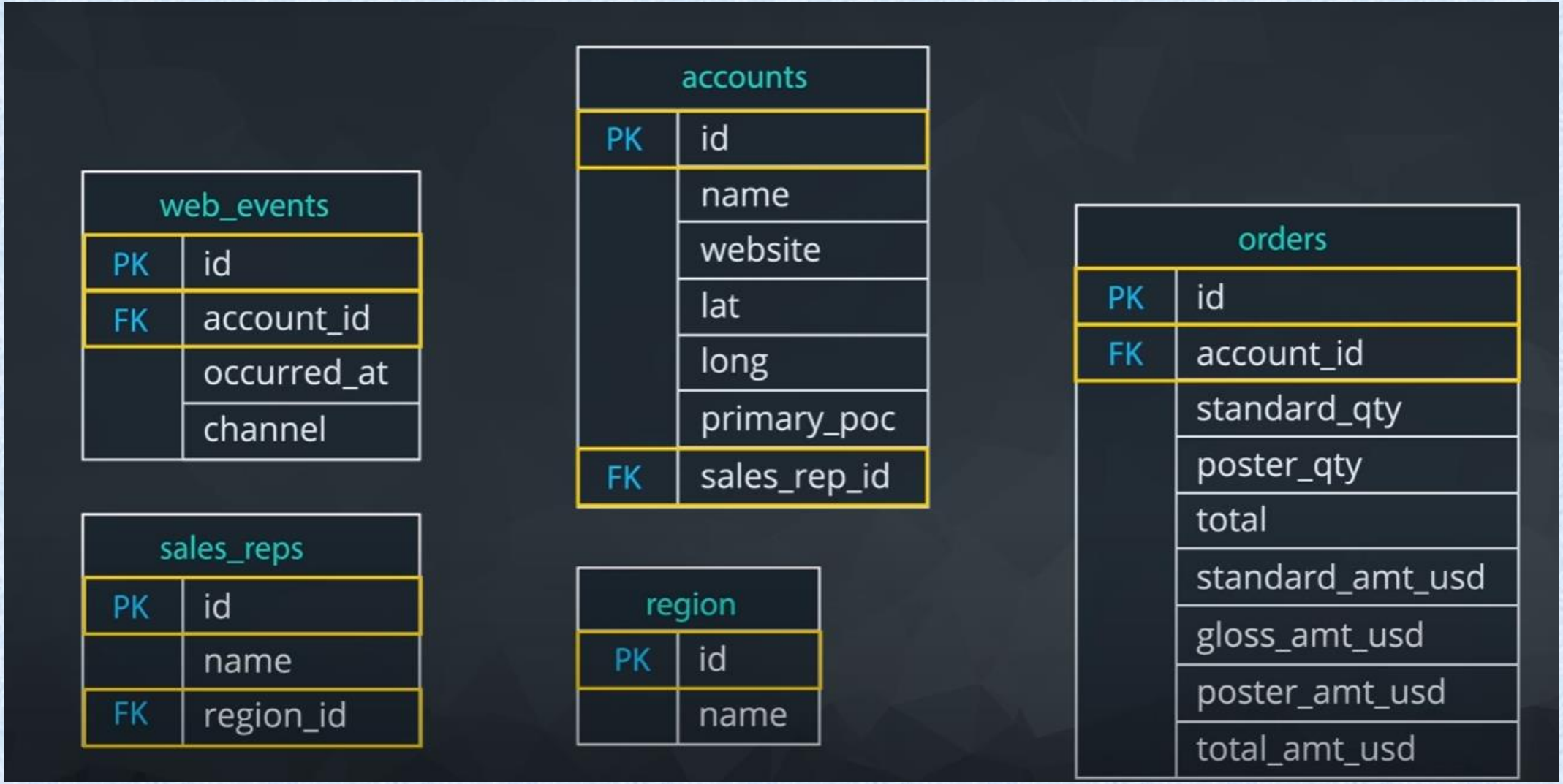
```sql
-- Find all the company names that start with a 'C' or 'W',
-- and the primary contact contains 'ana' or 'Ana',
-- but it doesn't contain 'eana'.
SELECT *
FROM accounts
WHERE (name LIKE 'C%' OR name LIKE 'W%')
           AND ((primary_poc LIKE '%ana%' OR primary_poc LIKE '%Ana%')
           AND primary_poc NOT LIKE '%eana%');
```

Use functions to derive columns

- +, -, *, /
- Aggregation functions: SUM, AVG, COUNT, MIN, MAX

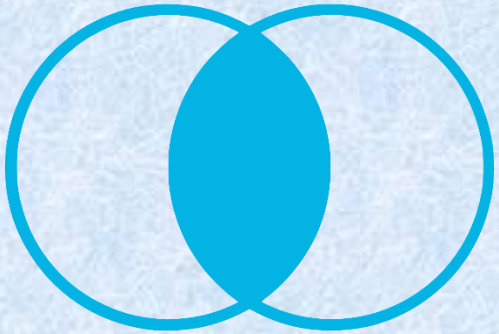
```
SELECT id,
       account_id,
       poster_amt_usd / total*100 AS poster_pct
FROM orders
LIMIT 10;

SELECT id, account_id,
   poster_amt_usd/(standard_amt_usd + gloss_amt_usd + poster_amt_usd) AS post_per
FROM orders
LIMIT 10;
```

Part 2 JOINs: Working with Multiple Tables

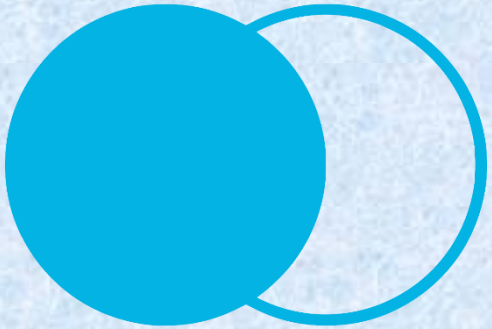


Types of JOIN

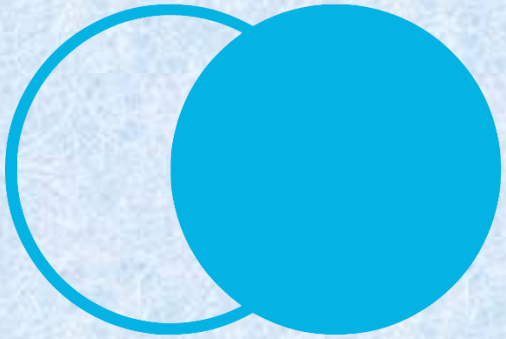
INNER JOIN (default)



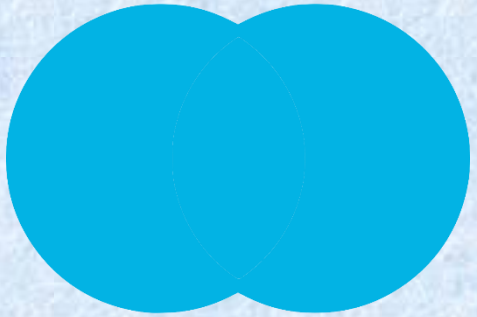
LEFT JOIN

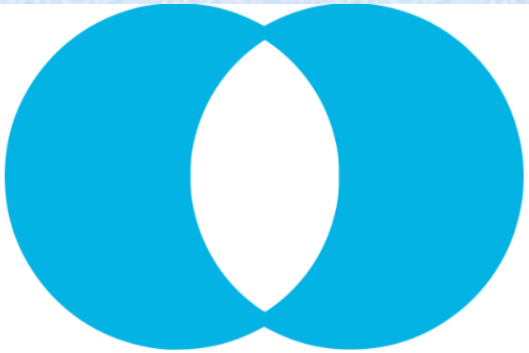


RIGHT JOIN

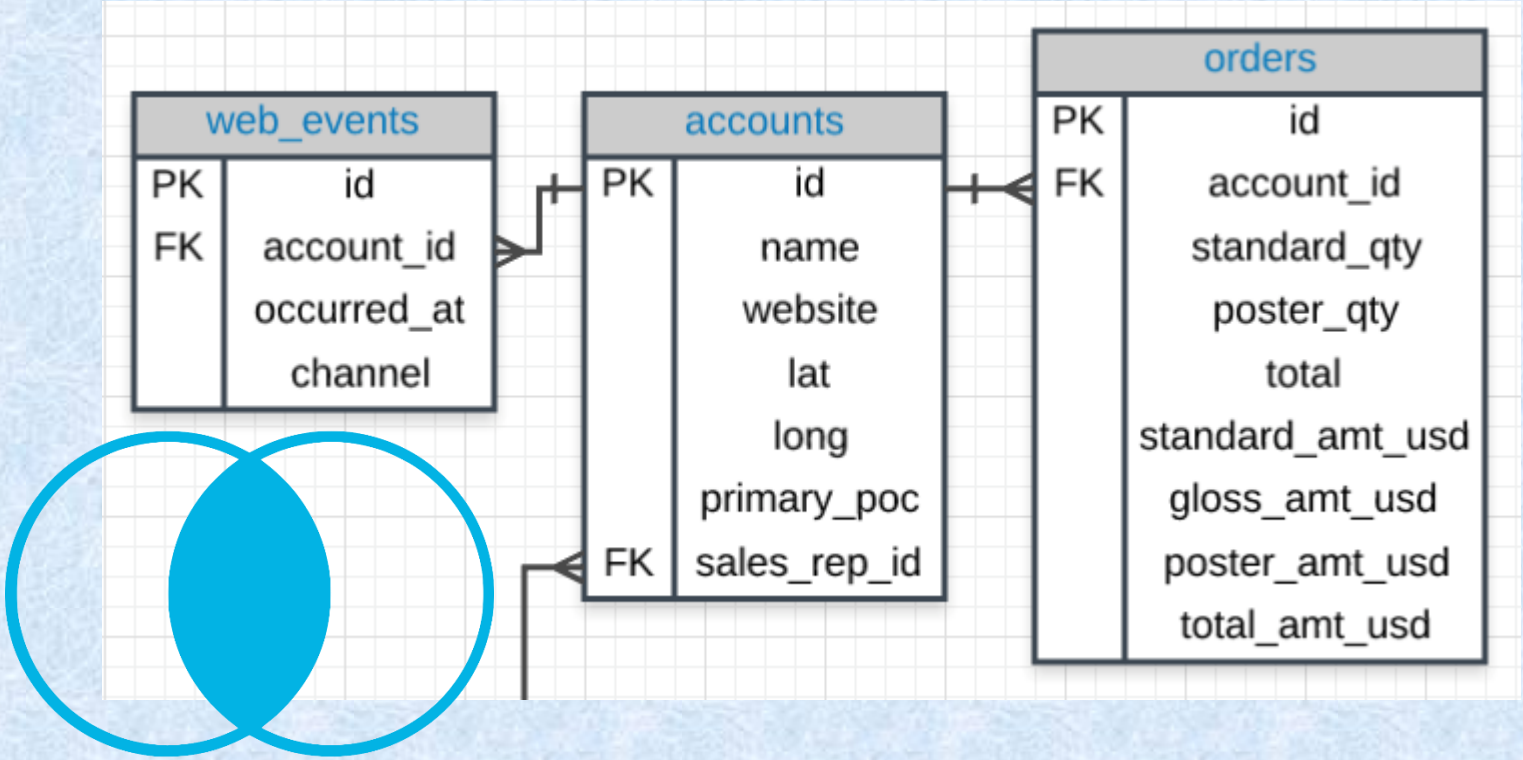


FULL OUTER JOIN





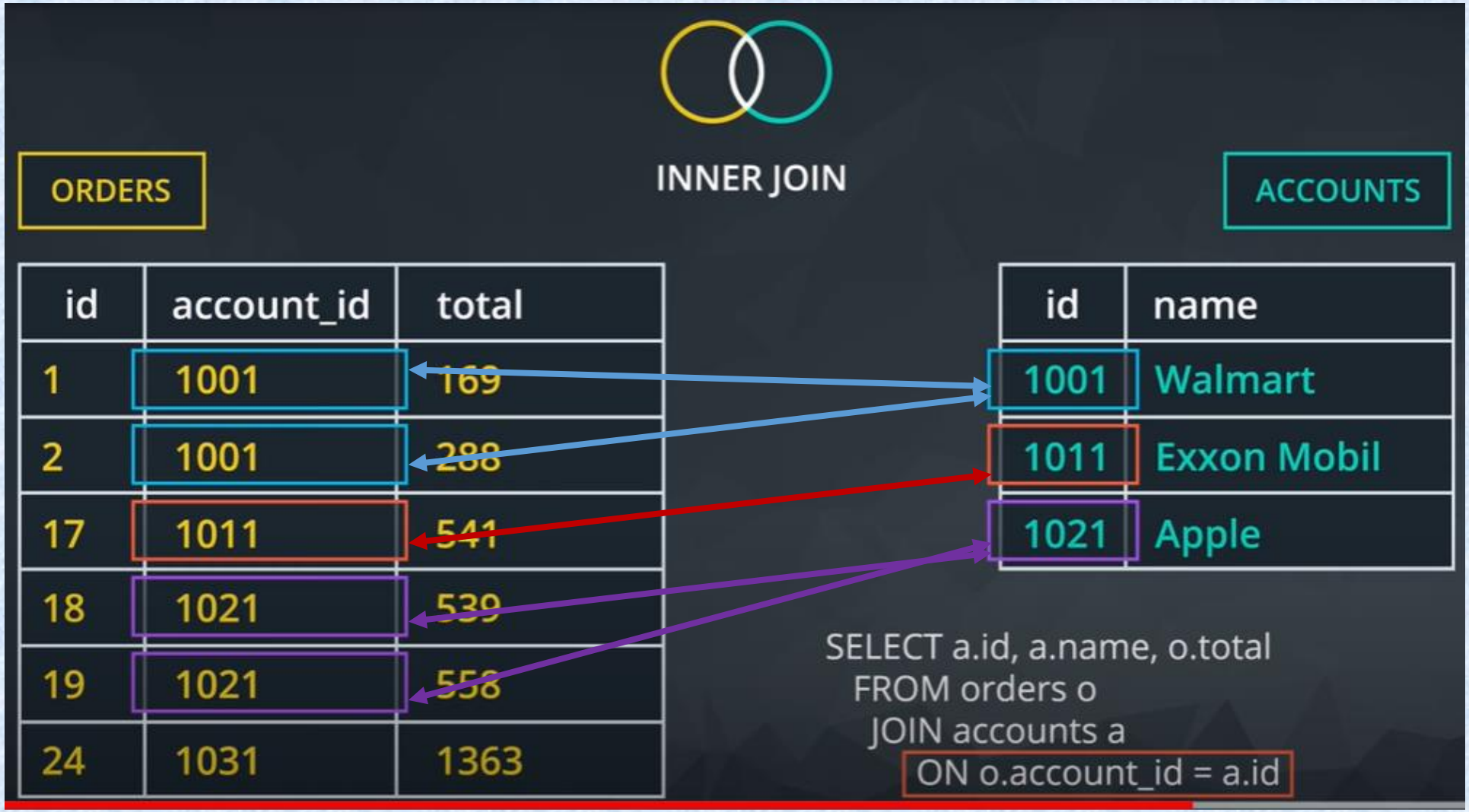
`FULL OUTER JOIN` with `WHERE A.Key IS NULL OR B.Key IS NULL` Venn Diagram



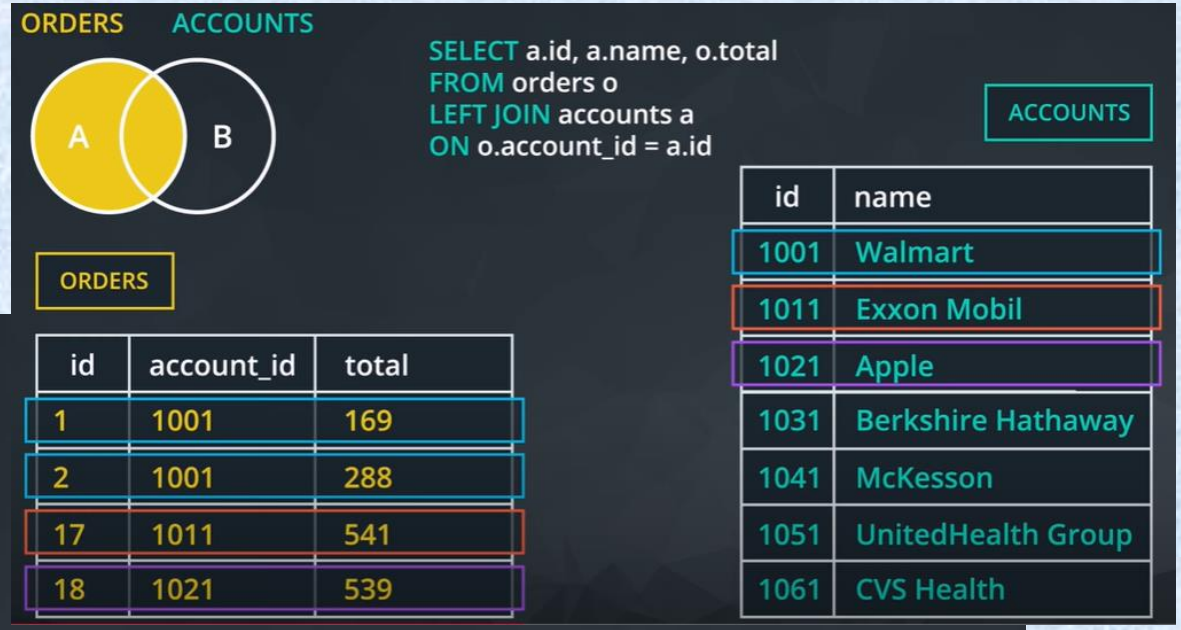

```sql
SELECT *
FROM web_events
JOIN accounts
ON web_events.account_id = accounts.id
JOIN orders
ON accounts.id = orders.account_id;
```

```sql
SELECT *
FROM web_events, accounts, orders
WHERE web_events.account_id = accounts.id
      AND accounts.id = orders.account_id;
```

Inner Join: The Intersection



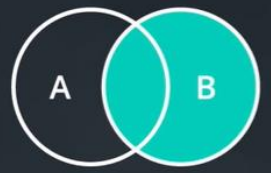
LEFT JOIN


ORDERS
ACCOUNTS
A
B
SELECT a.id, a.name, o.total
FROM orders o
LEFT JOIN accounts a
ON o.account_id = a.id
ACCOUNTS
id name
1001 Walmart
1011 Exxon Mobil
1021 Apple
1031 Berkshire Hathaway
1041 McKesson
1051 UnitedHealth Group
1061 CVS Health
ORDERS
id account_id total
1 1001 169
2 1001 288
17 1011 541
18 1021 539


ORDERS
ACCOUNTS
A
B
SELECT a.id, a.name, o.total
FROM orders o
LEFT JOIN accounts a
ON o.account_id = a.id
id name total
1001 Walmart 169
1001 Walmart 288
1011 Exxon Mobil 541
1021 Apple 539

RIGHT JOIN

ORDERS ACCOUNTS




```
SELECT a.id, a.name, o.total
FROM orders o
RIGHT JOIN accounts a
ON o.account_id = a.id
```

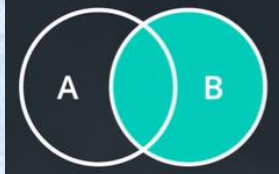
ORDERS

id	account_id	total
1	1001	169
2	1001	288
17	1011	541
18	1021	539

ACCOUNTS

id	name
1001	Walmart
1011	Exxon Mobil
1021	Apple
1031	Berkshire Hathaway
1041	McKesson
1051	UnitedHealth Group
1061	CVS Health

ORDERS ACCOUNTS




```
SELECT a.id, a.name, o.total
FROM orders o
RIGHT JOIN accounts a
ON o.account_id = a.id
```

NOT MATCHED →

RIGHT JOIN RESULT

id	name	total
1001	Walmart	169
1001	Walmart	288
1011	Exxon Mobil	541
1021	Apple	539
1031	Berkshire Hathaway	
1041	McKesson	
1051	UnitedHealth Group	
1061	CVS Health	

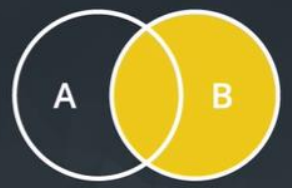
RIGHT JOIN can be accomplished by **LEFT JOIN** with Tables A and B swapped

LEFT JOIN and RIGHT JOIN are Interchangeable

ACCOUNTS

id	name
1001	Walmart
1011	Exxon Mobil
1021	Apple
1031	Berkshire Hathaway
1041	McKesson
1051	UnitedHealth Group
1061	CVS Health

```
SELECT a.id, a.name, o.total
FROM accounts a
LEFT JOIN orders o
ON o.account_id = a.id
```



ORDERS

id	account_id	total
1	1001	169
2	1001	288
17	1011	541
18	1021	539

THESE GIVE THE SAME RESULTING TABLE

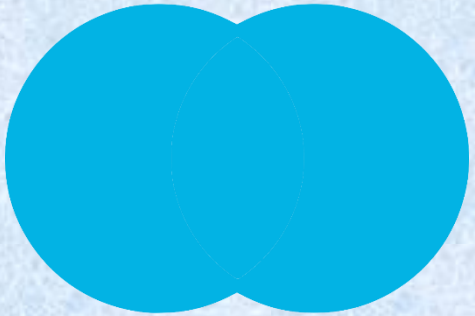
```
SELECT a.id, a.name, o.total
FROM orders o
RIGHT JOIN accounts a
ON o.account_id = a.id
```

```
SELECT a.id, a.name, o.total
FROM accounts a
LEFT JOIN orders o
ON o.account_id = a.id
```

RESULT

id	name	total
1001	Walmart	169
1001	Walmart	288
1011	Exxon Mobil	541
1021	Apple	539
1031	Berkshire Hathaway	
1041	McKesson	
1051	UnitedHealth Group	
1061	CVS Health	

OUTER JOIN



```
SELECT *
FROM table1
FULL OUTER JOIN table2
ON table1.column_name=table2.column_name;

-- or equivalently,
SELECT *
FROM table1
OUTER JOIN table2
ON table1.column_name=table2.column_name;
```

table_A

A	M
1	m
2	n
4	o

table_B

A	N
2	p
3	q
5	r

```
SELECT *
FROM table_A
OUTER JOIN table_B
ON table_A.A=table_B.A;
```

A	M	A	N
2	n	2	p
1	m	-	-
4	o	-	-
-	-	3	q
-	-	5	r

Using Alias

When we JOIN tables together, it is nice to give each table an alias. Frequently an alias is just the first letter of the table name, or use a, b, c, t1, t2, t3,…

```sql
-- Provide a table that provides the region for each sales_rep along with their associated accounts.
-- Your final table should include 3 columns: region name, sales rep name, and account name.
-- Sort the accounts alphabetically (A-Z) according to account name.

SELECT r.name region, s.name rep, a.name account
FROM sales_reps s
JOIN region r
ON s.region_id = r.id
JOIN accounts a
ON a.sales_rep_id = s.id
ORDER BY a.name;

-- Provide the name for each region for every order, as well as the account name
-- and the unit price they paid (total_amt_usd/total) for the order.
-- Your final table should have 3 columns: region name, account name, and unit price.
-- A few accounts have 0 for total, so I divided by (total + 0.01) to assure not dividing by zero.

SELECT r.name region, a.name account,
       o.total_amt_usd/(o.total + 0.01) AS unit_price
FROM region r
JOIN sales_reps s
ON s.region_id = r.id
JOIN accounts a
ON a.sales_rep_id = s.id
JOIN orders o
ON o.account_id = a.id;

-- or, equivalently,
SELECT r.name region, a.name account,
       o.total_amt_usd/(o.total + 0.01) AS unit_price
FROM region r, sales_reps s, accounts a, orders o
WHERE (s.region_id = r.id) AND (a.sales_rep_id = s.id) AND (o.account_id = a.id);
```

Recap of JOINs

- **Primary and Foreign Keys**
  - A key element for **JOIN**ing tables in a database has to do with primary and foreign keys:
  - **primary keys** - are unique for every row in a table. These are generally the first column in our database (like you saw with the **id** column for every table in the Parch & Posey database).
  - **foreign keys** - are the **primary key** appearing in another table, which allows the rows to be non-unique.
- **JOINs**
  - To combine data from multiple tables using **JOIN**s. The three **JOIN** statements you are most likely to use are:
  - **JOIN** - an **INNER JOIN** that only pulls data that exists in both tables.
  - **LEFT JOIN** - pulls all the data that exists in both tables, as well as all of the rows from the table in the **FROM** even if they do not exist in the **JOIN** statement.
  - **RIGHT JOIN** - pulls all the data that exists in both tables, as well as all of the rows from the table in the **JOIN** even if they do not exist in the **FROM** statement.
- **Alias**
  - You can alias tables and columns using **AS** or not using it. This allows you to be more efficient in the number of characters you need to write, while at the same time you can assure that your column headings are informative of the data in your table.

Part III SQL Aggregations

- **COUNT**: counts how many rows are in a particular column
- **SUM**: add all values in a particular column
- **MIN** and **MAX**:
- **AVG**
- **STD**
- **MEDIAN**
- **MODE**
- **NTILE** (**NTILE(4)** for **quartile, NTILE(100)** for **percentile**)

These are better used together with "GROUP BY"

COUNT

```sql
SELECT COUNT(*) AS no_obs
FROM accounts;

SELECT COUNT(accounts.id) AS id_count
FROM accounts;

SELECT COUNT(primary_poc) AS primary_poc_count    -- exclude NULL
from accounts;

SELECT *
FROM accounts
WHERE primary_poc IS NULL;
```

SUM

```sql
-- SUM Questions and Solutions
-- Find the total amount of poster_qty paper, standard_qty paper, total $ amount of sales ,
-- average price of standard paper in the orders table.

SELECT SUM(poster_qty) AS total_poster_sales ,
       SUM(standard_qty) AS total_standard_sales ,
       SUM(total_amt_usd) AS total_dollar_sales ,
       SUM(standard_amt_usd)/SUM(standard_qty) AS standard_price_per_unit
FROM orders;

-- Find the total amount for each individual order that was spent on standard and gloss paper in the orders table.
-- This should give a dollar amount for each order in the table.

SELECT id, standard_amt_usd + gloss_amt_usd AS total_standard_gloss
FROM orders;
```

MAX, MIN, and AVG

```
SELECT MAX(occurred_at)   AS last_order,    MIN(occurred_at) AS first_order,
       AVG(standard_qty)  AS mean_standard, AVG(gloss_qty)   AS mean_gloss,
       AVG(poster_qty)    AS mean_poster,   AVG(standard_amt_usd) AS mean_standard_usd,
       AVG(gloss_amt_usd) AS mean_gloss_usd, AVG(poster_amt_usd) AS mean_poster_usd
FROM orders;
```

We will learn “MEDIAN” when we come to NTILE (NTILE(2) = MEDIAN)

GROUP BY and ORDER BY

- **GROUP BY** can be used to aggregate data within subsets of the data (equivalent to cross-tabulation in SAS and Pivot Table in Excel)
- Any column in the **SELECT** statement that is not within an aggregator must be in the **GROUP BY** clause.
- The **GROUP BY** always goes between **WHERE** and **ORDER BY**.
- **ORDER BY** works like **SORT** in spreadsheet software.
- SQL does GROUP BY **before** the **LIMIT** clause. If you group by a column with enough unique values that exceed the **LIMIT** number, the aggregates will be calculated, and then some rows will simply be omitted from the results

GROUP BY

```sql
SELECT a.name, SUM(total_amt_usd) AS total_sales
FROM orders o
JOIN accounts a
ON a.id = o.account_id
GROUP BY a.name
ORDER BY a.name DESC;
```

```sql
SELECT s.name, w.channel, COUNT(*) num_events
FROM accounts a
JOIN web_events w
ON a.id = w.account_id
JOIN sales_reps s
ON s.id = a.sales_rep_id
GROUP BY s.name, w.channel
ORDER BY num_events DESC;
```

```sql
SELECT r.name, w.channel, COUNT(*) AS num_events
FROM accounts a
JOIN web_events w
ON a.id = w.account_id
JOIN sales_reps s
ON s.id = a.sales_rep_id
JOIN region r
ON r.id = s.region_id
GROUP BY r.name, w.channel
ORDER BY num_events DESC;
```

```sql
SELECT a.name, AVG(o.standard_amt_usd) AS avg_stand,
               AVG(o.gloss_amt_usd) AS avg_gloss,
               AVG(o.poster_amt_usd) avg_post
FROM accounts a
JOIN orders o
ON a.id = o.account_id
GROUP BY a.name
ORDER BY a.name;
```

DISTINCT to get unique values

```
SELECT DISTINCT id, name
FROM accounts;

SELECT DISTINCT a.id AS "account id", r.id AS "region id",
       a.name AS "account name", r.name AS "region name"
FROM accounts a
JOIN sales_reps s
ON s.id = a.sales_rep_id
JOIN region r
ON r.id = s.region_id;
```

HAVING clause vs. WHERE clause

- **WHERE** is to filter data **before** aggregation
- **HAVING** is to filter data **after** aggregation: It is the “clean” way to filter a query that has been aggregated, but this is also commonly done using a **subquery**.
- Any time you want to perform a **WHERE** on an element of your query that was created by an aggregation, you need to use **HAVING** instead.

```sql
-- How many of the sales reps have more than 5 accounts that they manage?
SELECT DISTINCT sales_rep_id, COUNT(sales_rep_id) AS num_accounts
FROM accounts
GROUP BY sales_rep_id
HAVING COUNT(sales_rep_id) > 5          |      -- cannot use the alias num_accounts
ORDER BY num_accounts DESC;

-- or

SELECT s.id, s.name, COUNT(*) num_accounts
FROM accounts a
JOIN sales_reps s
ON s.id = a.sales_rep_id
GROUP BY s.id, s.name
HAVING COUNT(*) > 5
ORDER BY num_accounts;

-- or use subquery
SELECT COUNT(*) num_reps_above5
FROM (SELECT s.id, s.name, COUNT(*) num_accounts
      FROM accounts a
      JOIN sales_reps s
      ON s.id = a.sales_rep_id
      GROUP BY s.id, s.name
      HAVING COUNT(*) > 5
      ORDER BY num_accounts
     ) AS Table1;
```

subquerry

Working with “date” – DATE_TRUNC

Input: 2017-04-16 12:15:01

Results	Function
2017-04-16 12:15:01	DATE_TRUNC(‘second’, 2017-04-16 12:15:01)
2017-04-16 00:00:00	DATE_TRUNC(‘day’, 2017-04-16 12:15:01)
2017-04-01 00:00:00	DATE_TRUNC(‘month’, 2017-04-16 12:15:01)
2017-01-01 00:00:00	DATE_TRUNC(‘year’, 2017-04-16 12:15:01)

```sql
-- In which month of which year did Walmart spend the most
-- on gloss paper in terms of dollars?
SELECT DATE_TRUNC('month', o.occurred_at) AS ord_date,
       SUM(o.gloss_amt_usd) AS tot_spent
FROM orders o
JOIN accounts a
ON a.id = o.account_id
WHERE a.name = 'Walmart'
GROUP BY 1
ORDER BY 2 DESC
LIMIT 1;
```

Another way to reference

SQL date format: yyyy-mm-dd

Working with “date” – DATE_PART (get some part)

```sql
-- Find the sales in terms of total dollars for all orders in each year, ordered from greatest to least
 SELECT DATE_PART('year', occurred_at) ord_year,  SUM(total_amt_usd) total_spent
 FROM orders
 GROUP BY 1
 ORDER BY 2 DESC;

-- Which month did Parch & Posey have the greatest sales in terms of total dollars?
SELECT DATE_PART('month', occurred_at) ord_month, SUM(total_amt_usd) total_spent
FROM orders
WHERE occurred_at BETWEEN '2014-01-01' AND '2017-01-01'
GROUP BY 1
ORDER BY 2 DESC;

-- Which year did Parch & Posey have the greatest sales in terms of total number of orders? A
SELECT DATE_PART('year', occurred_at) ord_year,  COUNT(*) total_sales
FROM orders
GROUP BY 1
ORDER BY 2 DESC;

-- Which month did Parch & Posey have the greatest sales in terms of total number of orders?
SELECT DATE_PART('month', occurred_at) ord_month, COUNT(*) total_sales
FROM orders
WHERE occurred_at BETWEEN '2014-01-01' AND '2017-01-01'
GROUP BY 1
ORDER BY 2 DESC;
```

Input: 2017-04-16 12:15:01

Results	Function
1	DATE_PART(‘second’, 2017-04-16 12:15:01)
16	DATE_PART(‘day’, 2017-04-16 12:15:01)
4	DATE_PART(‘month’, 2017-04-16 12:15:01)
2017	DATE_PART(‘year’, 2017-04-16 12:15:01)

```sql
SELECT DATE_PART('dow',occurred_at) AS day_of_week,
       SUM(total) AS total_qty
  FROM demo.orders
 GROUP BY 1
 ORDER BY 2 DESC
```

“CASE” statement

- CASE statement: “if… then…” to Add columns
- The CASE statement always goes in the SELECT clause.
- CASE must include the following components: **WHEN, THEN**, and **END. ELSE** is an optional component to catch cases that didn’t meet any of the other previous CASE conditions.
- You can make any conditional statement using any conditional operator (like **WHERE**) between WHEN and THEN. This includes stringing together multiple conditional statements using **AND** and **OR**.
- You can include multiple WHEN statements, as well as an ELSE statement again, to deal with any unaddressed conditions.

CASE and AGGREGATION examples

```sql
-- Write a query to display for each order, the account ID, total amount of the order,
-- and the level of the order - 'Large' or 'Small' - depending on if the order is
-- $3000 or more, or smaller than $3000.
SELECT account_id, total_amt_usd,
       CASE WHEN total_amt_usd > 3000 THEN 'Large'
       ELSE 'Small' END AS order_level
FROM orders;

-- Write a query to display the number of orders in each of three categories,
-- based on the total number of items in each order. The three categories are:
-- 'At Least 2000', 'Between 1000 and 2000' and 'Less than 1000'.
SELECT CASE WHEN total >= 2000 THEN 'At Least 2000'
            WHEN total >= 1000 AND total < 2000 THEN 'Between 1000 and 2000'
            ELSE 'Less than 1000'
       END AS order_category,  -- "order_cateogry is the name given for the grouped
       COUNT(*) AS order_count
FROM orders
GROUP BY 1;    -- GROUP BY "order_category"
```

Values of the new variable

New variable name

Create a new variable conditional on the value of an existing variable

```sql
SELECT a.name, SUM(total_amt_usd) total_spent,
     CASE WHEN SUM(total_amt_usd) > 200000 THEN 'top'
          WHEN  SUM(total_amt_usd) > 100000 THEN 'middle'
          ELSE 'low'
     END AS customer_level
FROM orders o
JOIN accounts a
ON o.account_id = a.id
GROUP BY a.name
ORDER BY 2 DESC;
```

Part IV Subqueries and Temporary Tables

```sql
-- Find the number of events in each day and each channel

SELECT DATE_TRUNC('day',occurred_at) AS day,
       channel, COUNT(*) as events
FROM web_events
GROUP BY 1,2
ORDER BY 3 DESC;

-- create a query that simply produces all the data from your 1st query
SELECT *
FROM (SELECT DATE_TRUNC('day',occurred_at) AS day,
           channel, COUNT(*) as events
     FROM web_events
     GROUP BY 1,2
     ORDER BY 3 DESC) sub;

-- find the average number of events for each channel
SELECT channel, AVG(events) AS average_events
FROM (SELECT DATE_TRUNC('day',occurred_at) AS day,
             channel, COUNT(*) as events
      FROM web_events
      GROUP BY 1,2) sub
GROUP BY channel
ORDER BY 2 DESC;
```

Subqueries can become very complicated, so subquery formatting is very important

```sql
-- Pull the max for each region, and then use this to pull those rows
SELECT region_name, MAX(total_amt) total_amt
     FROM (SELECT s.name rep_name, r.name region_name,
                  SUM(o.total_amt_usd) total_amt
             FROM sales_reps s
             JOIN accounts a
             ON a.sales_rep_id = s.id
             JOIN orders o
             ON o.account_id = a.id
             JOIN region r
             ON r.id = s.region_id
             GROUP BY 1, 2) t1   -- t1 is subquery name
     GROUP BY 1;
```

WITH Statement to create temporary tables

- The **WITH** statement is often called a **Common Table Expression** or **CTE**
- **QUESTION:** You need to find the average number of events for each channel per day.

Solution 1

```
SELECT channel, AVG(events) AS average_events
FROM (SELECT DATE_TRUNC('day',occurred_at) AS day,
        channel, COUNT(*) as events
      FROM web_events
      GROUP BY 1,2) sub
GROUP BY channel
ORDER BY 2 DESC;
```

Solution 2

```
WITH events AS (
        SELECT DATE_TRUNC('day',occurred_at) AS day,
                channel, COUNT(*) as events
        FROM web_events
        GROUP BY 1,2)

SELECT channel, AVG(events) AS average_events
FROM events
GROUP BY channel
ORDER BY 2 DESC;
```

You can have > 1 table with “WITH” statement

```
WITH t1 AS (
        SELECT s.name rep_name, r.name region_name, SUM(o.total_amt_usd) total_amt
        FROM sales_reps s
        JOIN accounts a
        ON a.sales_rep_id = s.id
        JOIN orders o
        ON o.account_id = a.id
        JOIN region r
        ON r.id = s.region_id
        GROUP BY 1,2
        ORDER BY 3 DESC),

   t2 AS (
       SELECT region_name, MAX(total_amt) total_amt
       FROM t1
       GROUP BY 1)

SELECT t1.rep_name, t1.region_name, t1.total_amt
FROM t1
JOIN t2
ON t1.region_name = t2.region_name AND t1.total_amt = t2.total_amt;
```