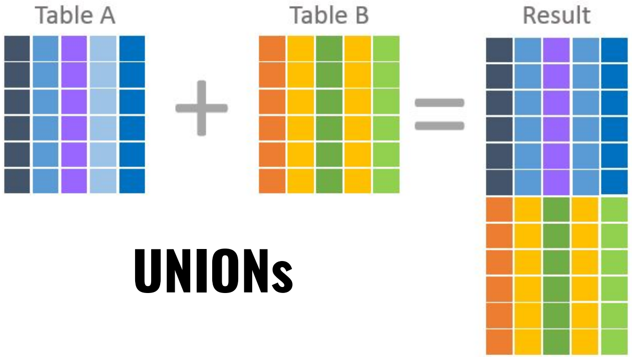
GENERAL ASSEMBLY

MULTIPLE JOINS IN SQL

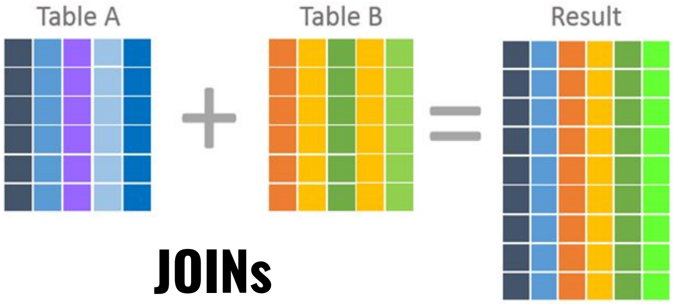
OPENING: LAST SESSION RECAP

Let’s review the following questions from our previous class:

- What’s the difference between a JOIN and a UNION?
- What Excel function is similar to JOIN?
- How do aliases work with JOINS?



UNIONs



JOINs

MULTIPLE RELATIONSHIPS IN SQL

LEARNING OBJECTIVES

In today’s lesson, we’ll learn how to:

1. Create relationships between tables using:
   1. `INNER`, `RIGHT`, and `LEFT JOINs`
   2. `FULL OUTER JOINs`
   3. `EXCEPTION JOINs`
   4. `CROSS JOIN`
2. Optimize queries using `WHERE`, `LIMIT`, and `COALESCE`.

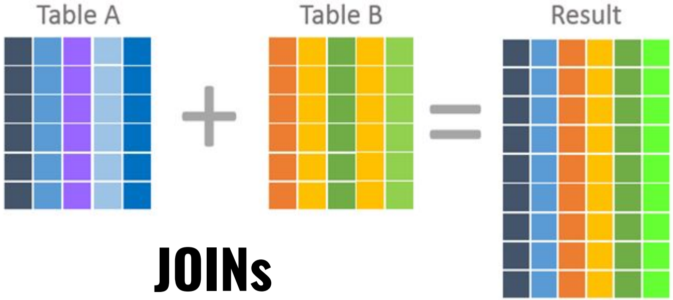
INTRODUCTION: TYPES OF JOINS

TYPES OF JOINS

As we’ve seen, SQL is written from left to right. The code is also read from left to right.

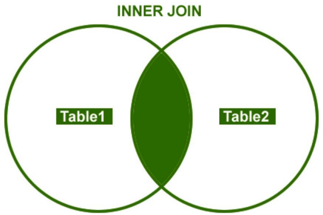
When you create `JOINs`, your first (or primary) table is known as the `LEFT` table and the second table is known as the `RIGHT` table.

`OUTER` and `EXCEPTION JOINs` assist you in handling **missing data** between tables.

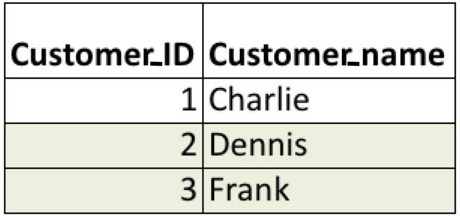


JOINs

EXAMPLE: INNER JOIN

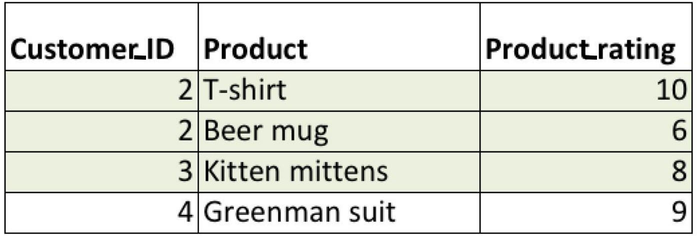


Current customers



Customer_ID	Customer_name
1	Charlie
2	Dennis
3	Frank

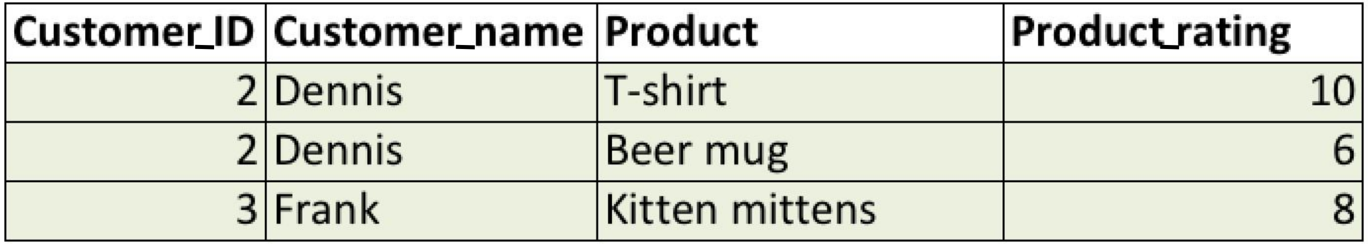
Product ratings



Customer_ID	Product	Product_rating
2	T-shirt	10
2	Beer mug	6
3	Kitten mittens	8
4	Greenman suit	9

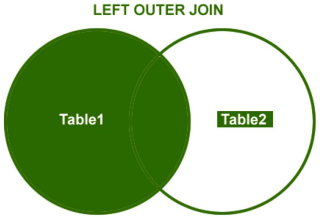
```
cc.select customer_id
, cc.customer_name
, pr.product
, pr.product_rating

from current_customers
cc

inner join
product_ratings pr on
pr.customer_id =
cc.customer_id
```

Inner join on Customer_ID result:

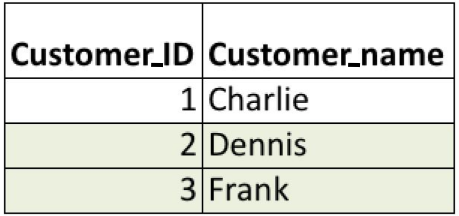


Customer_ID	Customer_name	Product	Product_rating
2	Dennis	T-shirt	10
2	Dennis	Beer mug	6
3	Frank	Kitten mittens	8

EXAMPLE: LEFT JOIN

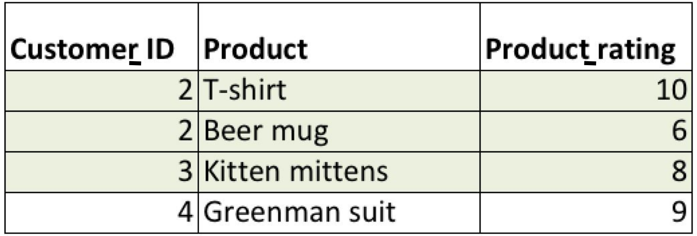


Current customers



Customer_ID	Customer_name
1	Charlie
2	Dennis
3	Frank

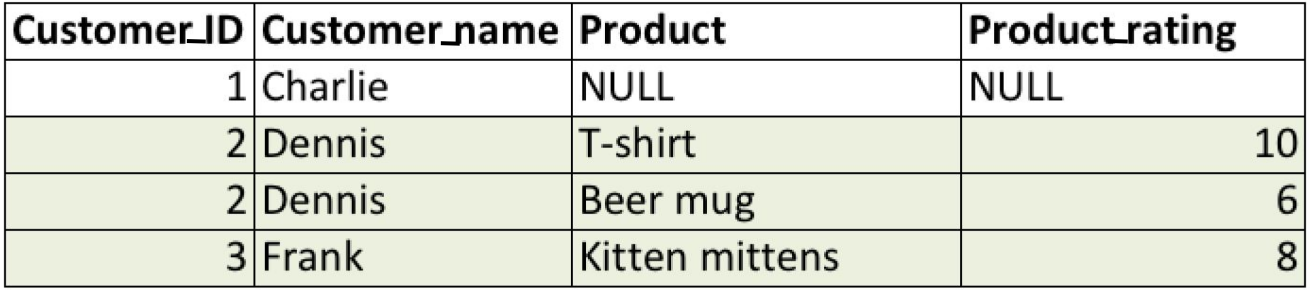
Product ratings



Customer_ID	Product	Product_rating
2	T-shirt	10
2	Beer mug	6
3	Kitten mittens	8
4	Greenman suit	9

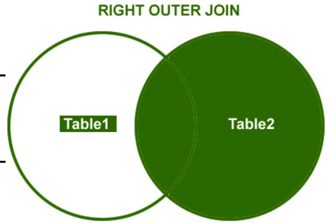
```
cc.select customer_id
, cc.customer_name
, pr.product
, pr.product_rating

from current_customers
cc

left join
product_ratings pr on
pr.customer_id =
cc.customer_id
```

Left join on Customer ID result:

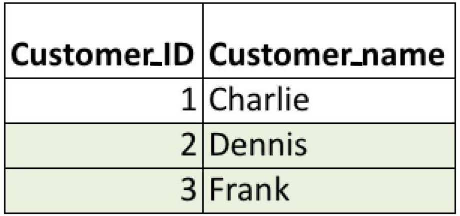


Customer_ID	Customer_name	Product	Product_rating
1	Charlie	NULL	NULL
2	Dennis	T-shirt	10
2	Dennis	Beer mug	6
3	Frank	Kitten mittens	8

EXAMPLE: RIGHT JOIN

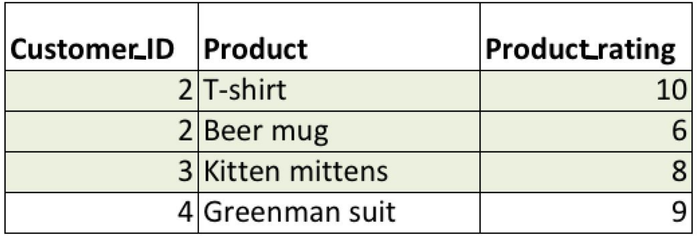


Current customers



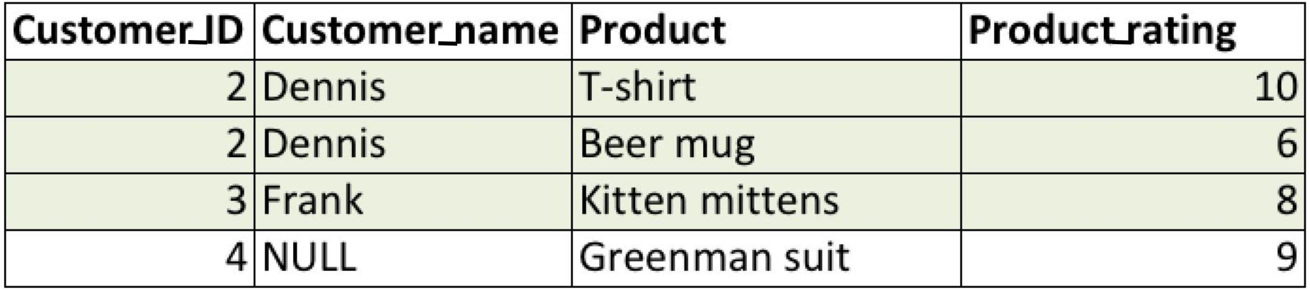
Customer_ID	Customer_name
1	Charlie
2	Dennis
3	Frank

Product ratings



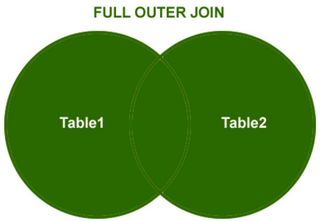
Customer_ID	Product	Product_rating
2	T-shirt	10
2	Beer mug	6
3	Kitten mittens	8
4	Greenman suit	9

Right join on Customer ID result:



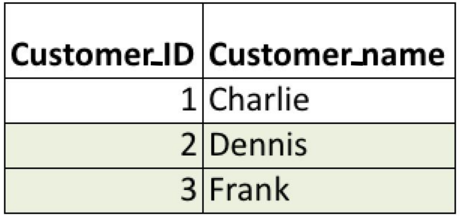
Customer_ID	Customer_name	Product	Product_rating
2	Dennis	T-shirt	10
2	Dennis	Beer mug	6
3	Frank	Kitten mittens	8
4	NULL	Greenman suit	9

```
cc.select customer_id
, cc.customer_name
, pr.product
, pr.product_rating

from current_customers
cc

right join
product_ratings pr on
pr.customer_id =
cc.customer_id
```



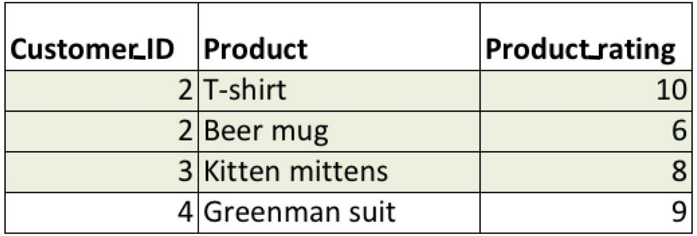
EXAMPLE: FULL OUTER JOIN

Current customers



Customer_ID	Customer_name
1	Charlie
2	Dennis
3	Frank

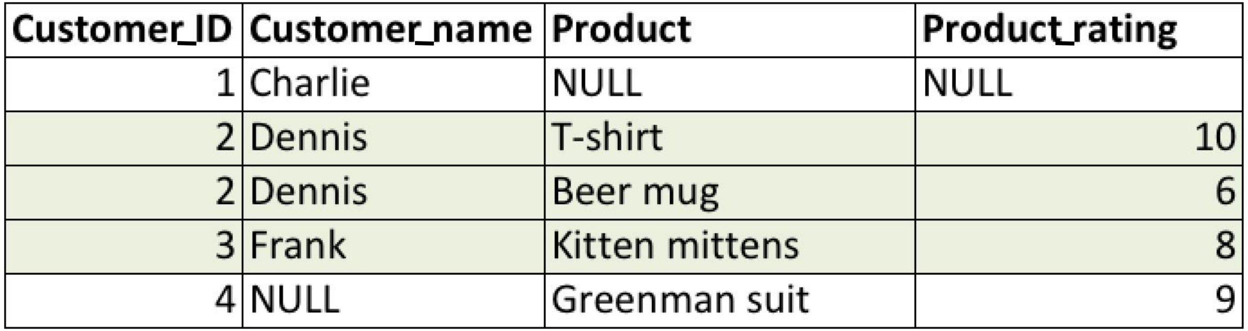
Product ratings



Customer_ID	Product	Product_rating
2	T-shirt	10
2	Beer mug	6
3	Kitten mittens	8
4	Greenman suit	9

```
cc.select customer_id
, cc.customer_name
, pr.product
, pr.product_rating

from current_customers
cc

full outer join
product_ratings pr on
pr.customer_id =
cc.customer_id
```

Full outer join on Customer ID result:



Customer_ID	Customer_name	Product	Product_rating
1	Charlie	NULL	NULL
2	Dennis	T-shirt	10
2	Dennis	Beer mug	6
3	Frank	Kitten mittens	8
4	NULL	Greenman suit	9

OUTER, EXCEPTION, AND CARTESIAN JOINS

Consider this example:

- Your `LEFT` table holds **store numbers** and **sales**.
- Your `RIGHT` table holds **store numbers** and **opening dates**.

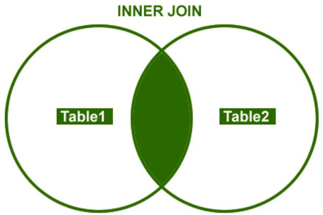
Your `RIGHT` table includes stores that haven’t opened yet and therefore do not have sales.

- An `INNER JOIN`, which is an exact match, would end up *filtering* those new stores out of your results.
- If you want to see store numbers *whether or not they have sales*, you need to do a `RIGHT OUTER JOIN`.

OUTER, EXCEPTION, AND CARTESIAN JOINS

On the next several slides, we'll define each type of `JOIN` with examples using the following tables:

- `EMPLOYEES` **table**: Contains information on employee names and has one row per employee.
- `SALARIES` **table**: Contains information on employee salaries.



OUTER, EXCEPTION, AND CARTESIAN JOINS

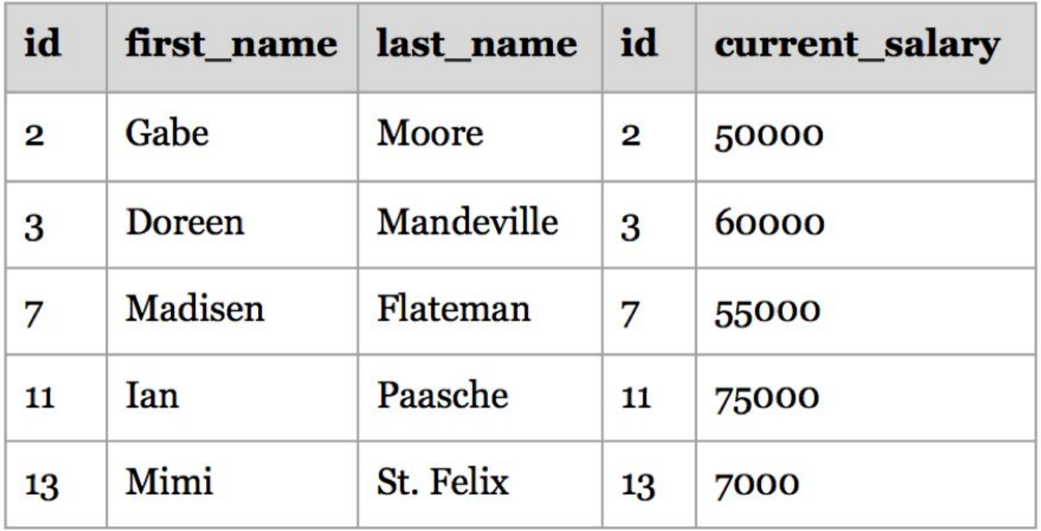
An INNER JOIN displays only the rows that have matches in both joined tables.

```
SELECT * FROM employees
    INNER JOIN salaries
    ON employees.ID = salaries.ID;
```

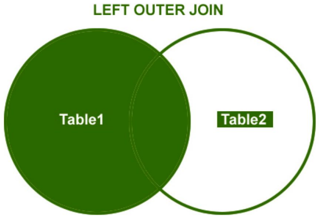
Note that the * gets you everything across all tables. You to get just one column from the SALARIES table, you can write:

```
 employees.*,
salaries.current_salary
```

An INNER JOIN would yield the table on the right.



id	first_name	last_name	id	current_salary
2	Gabe	Moore	2	50000
3	Doreen	Mandeville	3	60000
7	Madisen	Flateman	7	55000
11	Ian	Paasche	11	75000
13	Mimi	St. Felix	13	7000



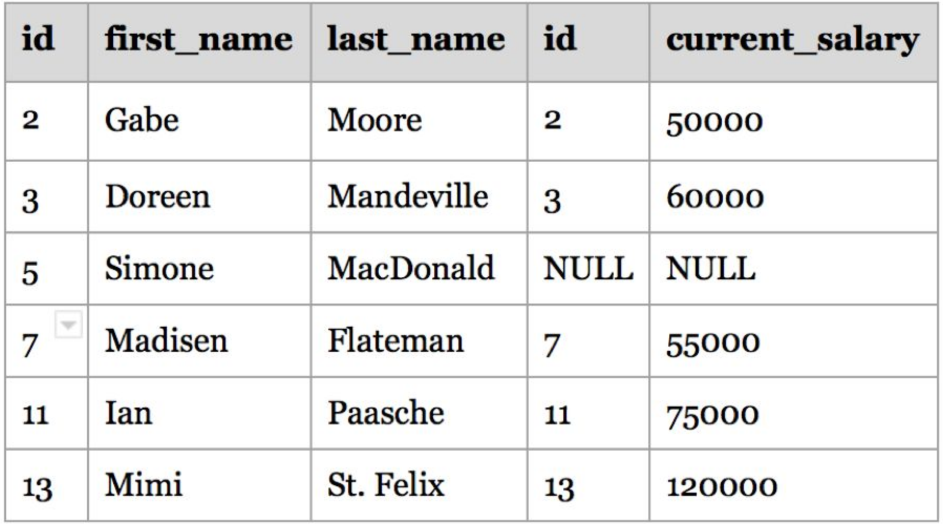
OUTER, EXCEPTION, AND CARTESIAN JOINS

A LEFT JOIN yields:

- Data that **both tables** have in common.
- Data from the **primary** table selected that doesn't have matching data to join to in the secondary table.

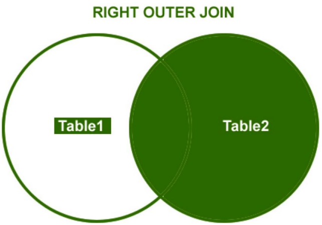
A LEFT JOIN would yield the table shown here on the right.

```
SELECT * FROM employees
    LEFT JOIN salaries
    ON employees.ID = salaries.ID;
```



id	first_name	last_name	id	current_salary
2	Gabe	Moore	2	50000
3	Doreen	Mandeville	3	60000
5	Simone	MacDonald	NULL	NULL
7	Madisen	Flateman	7	55000
11	Ian	Paasche	11	75000
13	Mimi	St. Felix	13	120000

OUTER, EXCEPTION, AND CARTESIAN JOINS

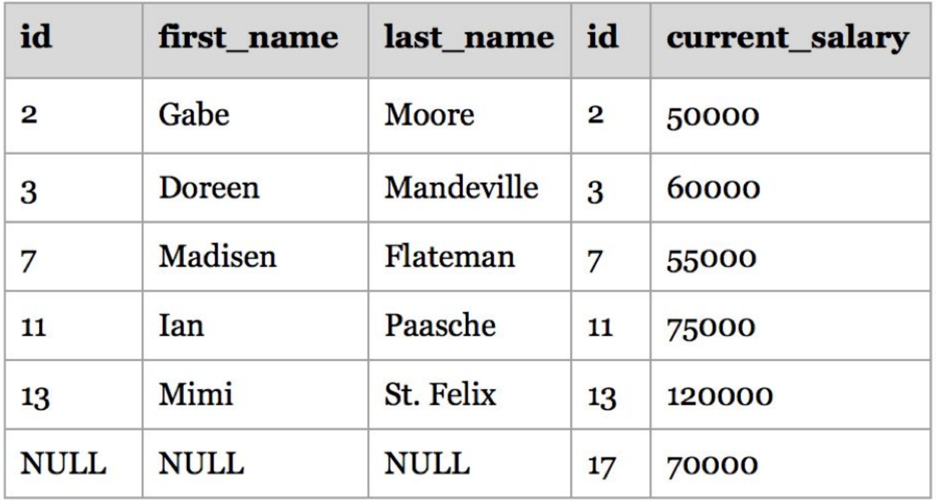


A RIGHT JOIN yields:

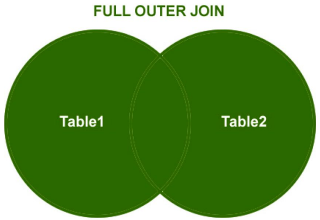
- Data that two tables have in common.
- Data from the **secondary** table selected that doesn't have matching data to join to in the primary table.

A RIGHT JOIN would yield the table shown here.

```
SELECT * FROM employees
     RIGHT JOIN salaries
     ON employees.ID = salaries.ID;
```



id	first_name	last_name	id	current_salary
2	Gabe	Moore	2	50000
3	Doreen	Mandeville	3	60000
7	Madisen	Flateman	7	55000
11	Ian	Paasche	11	75000
13	Mimi	St. Felix	13	120000
NULL	NULL	NULL	17	70000

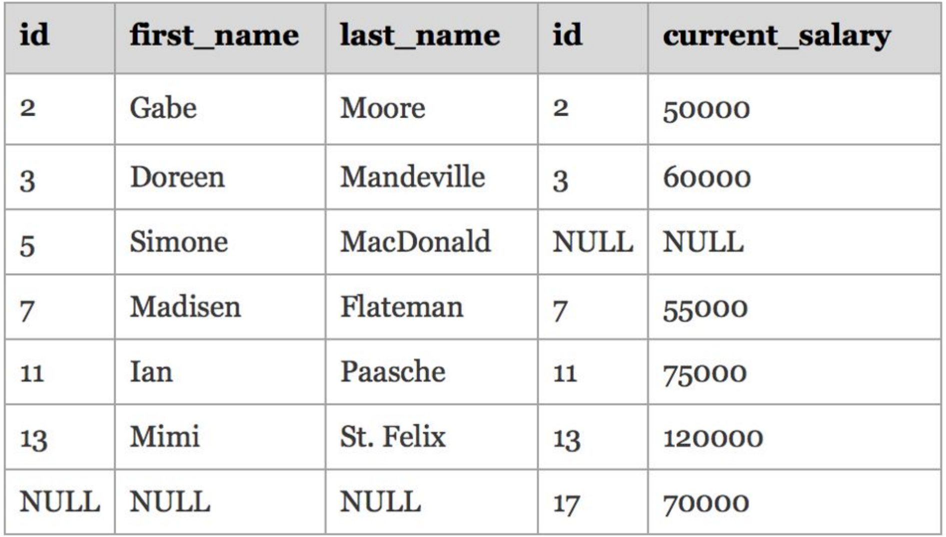


OUTER, EXCEPTION, AND CARTESIAN JOINS

A FULL JOIN returns **all** data from each table, **regardless** of whether they have matching data in the other table.

A FULL JOIN yields the table shown here.

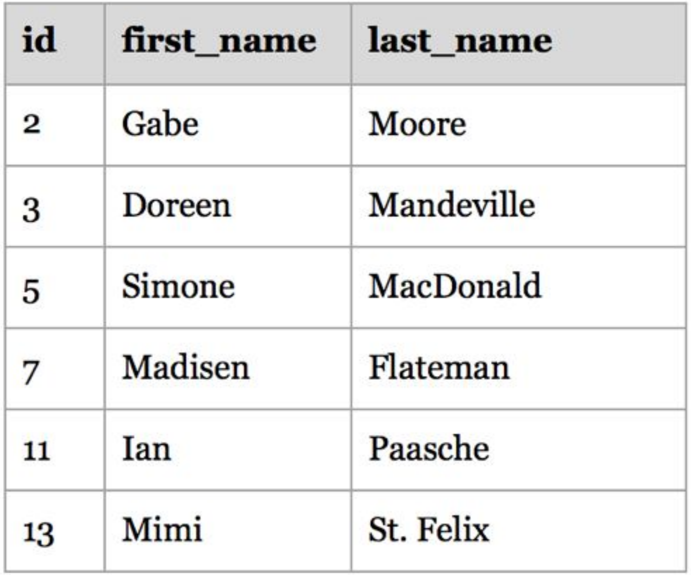
```
SELECT * FROM employees
    FULL JOIN salaries
    ON employees.ID = salaries.ID;
```



id	first_name	last_name	id	current_salary
2	Gabe	Moore	2	50000
3	Doreen	Mandeville	3	60000
5	Simone	MacDonald	NULL	NULL
7	Madisen	Flateman	7	55000
11	Ian	Paasche	11	75000
13	Mimi	St. Felix	13	120000
NULL	NULL	NULL	17	70000

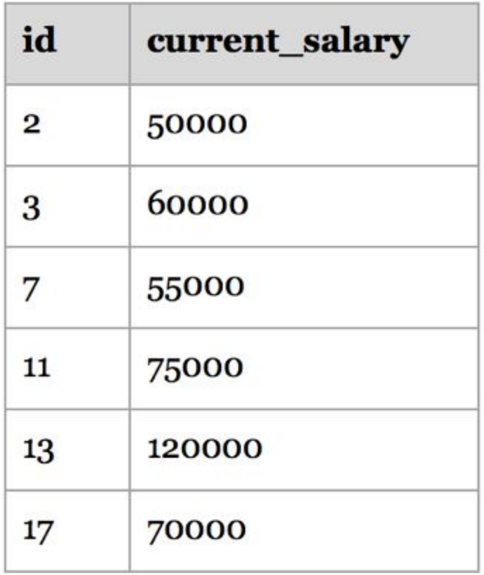
OUTER, EXCEPTION, AND CARTESIAN JOINS

EMPLOYEES



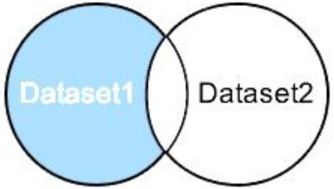
id	first_name	last_name
2	Gabe	Moore
3	Doreen	Mandeville
5	Simone	MacDonald
7	Madisen	Flateman
11	Ian	Paasche
13	Mimi	St. Felix

SALARIES



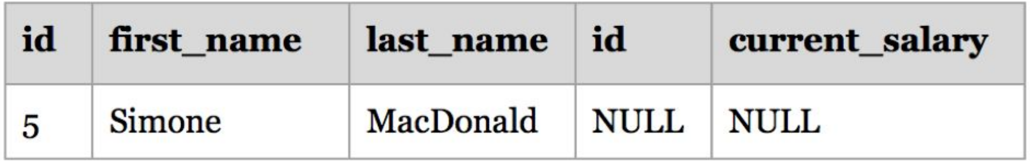
id	current_salary
2	50000
3	60000
7	55000
11	75000
13	120000
17	70000

OUTER, EXCEPTION, AND CARTESIAN JOINS



An EXCEPTION JOIN returns **only** the data from the **primary** - or first - table selected that **doesn't have matching data** to join to in the secondary table.

An EXCEPTION JOIN with the **EMPLOYEES** table first would yield this result:

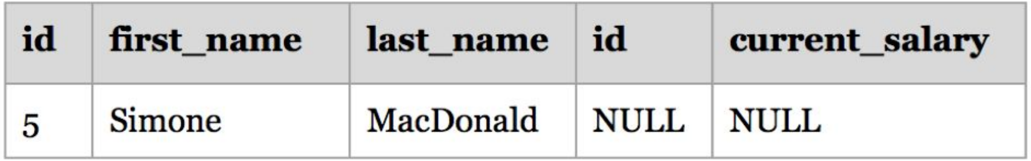


id	first_name	last_name	id	current_salary
5	Simone	MacDonald	NULL	NULL

Pro Tip: pgAdmin includes the EXCEPT function for extracting exceptions.

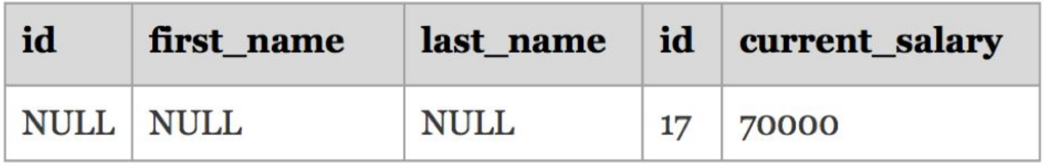
OUTER, EXCEPTION, AND CARTESIAN JOINS

```
SELECT id FROM employees
EXCEPT
SELECT id FROM salaries
```



id	first_name	last_name	id	current_salary
5	Simone	MacDonald	NULL	NULL

```
SELECT id FROM salaries
EXCEPT
SELECT id FROM employees
```



id	first_name	last_name	id	current_salary
NULL	NULL	NULL	17	70000

OUTER, EXCEPTION, AND CARTESIAN JOINS

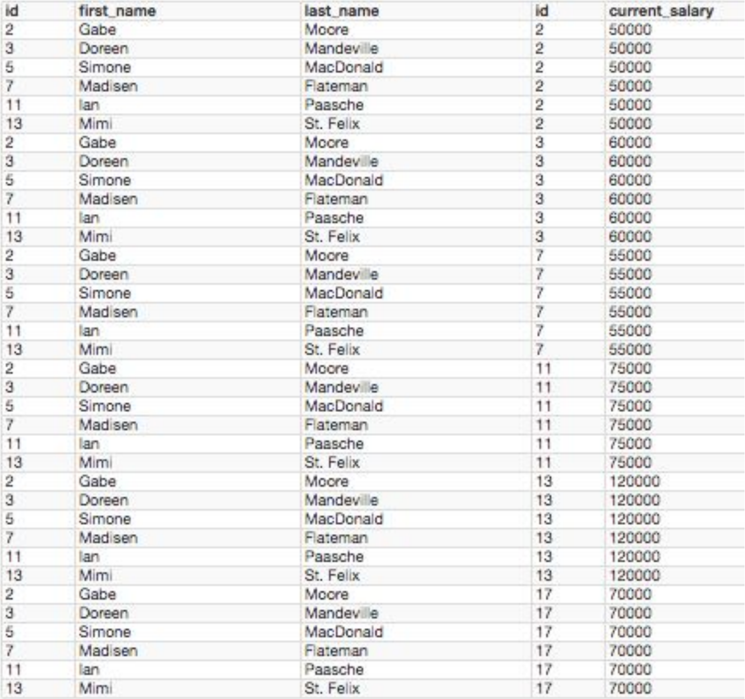
A `CROSS JOIN` matches every row of the primary table with every row of the secondary table.

This type of `JOIN` results in a Cartesian product of the tables.

- This is generally detrimental to fast performance and not desired.

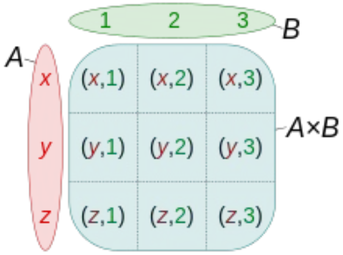
A `CROSS JOIN` would yield the table on the right.

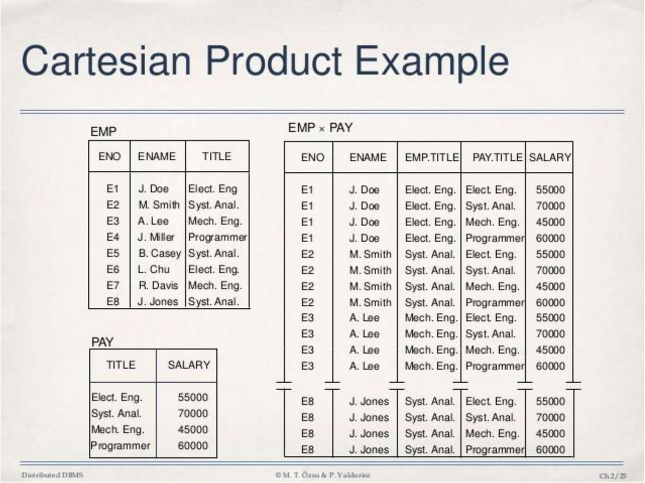
```
SELECT * FROM employees
      CROSS JOIN salaries;
```



id	first_name	last_name	id	current_salary
2	Gabe	Moore	2	50000
3	Doreen	Mandeville	2	50000
5	Simone	MacDonald	2	50000
7	Madisen	Flateman	2	50000
11	Ian	Paasche	2	50000
13	Mimi	St. Felix	2	50000
2	Gabe	Moore	3	60000
3	Doreen	Mandeville	3	60000
5	Simone	MacDonald	3	60000
7	Madisen	Flateman	3	60000
11	Ian	Paasche	3	60000
13	Mimi	St. Felix	3	60000
2	Gabe	Moore	7	55000
3	Doreen	Mandeville	7	55000
5	Simone	MacDonald	7	55000
7	Madisen	Flateman	7	55000
11	Ian	Paasche	7	55000
13	Mimi	St. Felix	7	55000
2	Gabe	Moore	11	75000
3	Doreen	Mandeville	11	75000
5	Simone	MacDonald	11	75000
7	Madisen	Flateman	11	75000
11	Ian	Paasche	11	75000
13	Mimi	St. Felix	11	75000
2	Gabe	Moore	13	120000
3	Doreen	Mandeville	13	120000
5	Simone	MacDonald	13	120000
7	Madisen	Flateman	13	120000
11	Ian	Paasche	13	120000
13	Mimi	St. Felix	13	120000
2	Gabe	Moore	17	70000
3	Doreen	Mandeville	17	70000
5	Simone	MacDonald	17	70000
7	Madisen	Flateman	17	70000
11	Ian	Paasche	17	70000
13	Mimi	St. Felix	17	70000

CARTESIAN JOINS EXAMPLES





Cartesian Product Example

EMP

ENO	ENAME	TITLE
E1	J. Doe	Elect. Eng
E2	M. Smith	Syst. Anal.
E3	A. Lee	Mech. Eng.
E4	J. Miller	Programmer
E5	B. Casey	Syst. Anal.
E6	L. Chu	Elect. Eng.
E7	R. Davis	Mech. Eng.
E8	J. Jones	Syst. Anal.

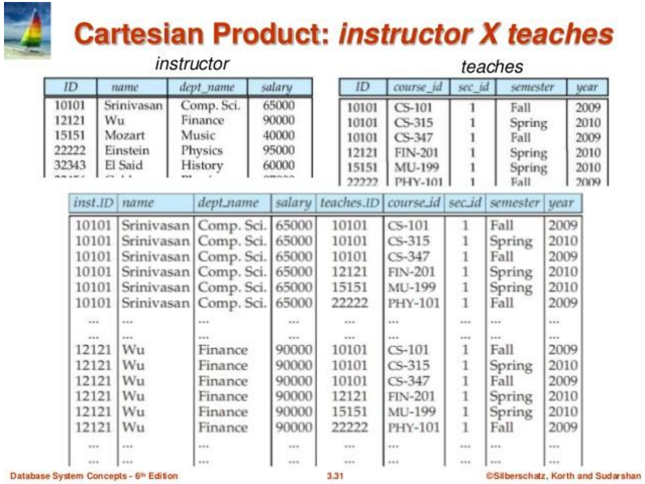
PAY

TITLE	SALARY
Elect. Eng.	55000
Syst. Anal.	70000
Mech. Eng.	45000
Programmer	60000

EMP × PAY

ENO	ENAME	EMP.TITLE	PAY.TITLE	SALARY
E1	J. Doe	Elect. Eng.	Elect. Eng.	55000
E1	J. Doe	Elect. Eng.	Syst. Anal.	70000
E1	J. Doe	Elect. Eng.	Mech. Eng.	45000
E1	J. Doe	Elect. Eng.	Programmer	60000
E2	M. Smith	Syst. Anal.	Elect. Eng.	55000
E2	M. Smith	Syst. Anal.	Syst. Anal.	70000
E2	M. Smith	Syst. Anal.	Mech. Eng.	45000
E2	M. Smith	Syst. Anal.	Programmer	60000
E3	A. Lee	Mech. Eng.	Elect. Eng.	55000
E3	A. Lee	Mech. Eng.	Syst. Anal.	70000
E3	A. Lee	Mech. Eng.	Mech. Eng.	45000
E3	A. Lee	Mech. Eng.	Programmer	60000
E8	J. Jones	Syst. Anal.	Elect. Eng.	55000
E8	J. Jones	Syst. Anal.	Syst. Anal.	70000
E8	J. Jones	Syst. Anal.	Mech. Eng.	45000
E8	J. Jones	Syst. Anal.	Programmer	60000

Distributed DBMS © M. T. Özsu & P. Valduriez Ch.2/25



Cartesian Product: *instructor X teaches*

instructor

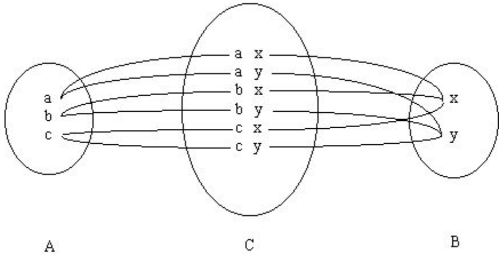
ID	*name*	*dept_name*	*salary*
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000

teaches

ID	*course_id*	*sec_id*	*semester*	*year*
10101	CS-101	1	Fall	2009
10101	CS-315	1	Spring	2010
10101	CS-347	1	Fall	2009
12121	FIN-201	1	Spring	2010
15151	MU-199	1	Spring	2010
22222	PHY-101	1	Fall	2009

inst.ID	*name*	*dept_name*	*salary*	*teaches.ID*	*course_id*	*sec_id*	*semester*	*year*
10101	Srinivasan	Comp. Sci.	65000	10101	CS-101	1	Fall	2009
10101	Srinivasan	Comp. Sci.	65000	10101	CS-315	1	Spring	2010
10101	Srinivasan	Comp. Sci.	65000	10101	CS-347	1	Fall	2009
10101	Srinivasan	Comp. Sci.	65000	12121	FIN-201	1	Spring	2010
10101	Srinivasan	Comp. Sci.	65000	15151	MU-199	1	Spring	2010
10101	Srinivasan	Comp. Sci.	65000	22222	PHY-101	1	Fall	2009
...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...
12121	Wu	Finance	90000	10101	CS-101	1	Fall	2009
12121	Wu	Finance	90000	10101	CS-315	1	Spring	2010
12121	Wu	Finance	90000	10101	CS-347	1	Fall	2009
12121	Wu	Finance	90000	12121	FIN-201	1	Spring	2010
12121	Wu	Finance	90000	15151	MU-199	1	Spring	2010
12121	Wu	Finance	90000	22222	PHY-101	1	Fall	2009
...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...

Database System Concepts - 6th Edition 3.31 ©Silberschatz, Korth and Sudarshan



INNER, OUTER, EXCEPTION, AND CROSS JOINS

What type of `JOIN` should you use in each of the following scenarios?

Table 1 represents the primary table:

`Table 1 => Primary => Left`

Table 2 represents the secondary table:

`Table 2 => Secondary => Right`

INNER, OUTER, EXCEPTION, AND CROSS JOINS

Table 1 has pending deleted items. Table 2 has item **sales**. You need to find all of the items that are pending deletion and have no sales data.

- Your `JOIN` solution choice is:

Table 1 has a list of **vendors**. Table 2 has a list of **addresses** for vendors. Table 2 is missing some data. For your purposes, you want to see all of the vendor information. You also want to see all of the address information you can find about these vendors, even if some of it is missing.

- Your `JOIN` solution choice is:

INNER, OUTER, EXCEPTION, AND CROSS JOINS

Table 1 has a list of **members**. Table 2 has a list of **items purchased**. You are running a ground beef recall and need to get an exact match of members who have purchased this item.

- Your `JOIN` solution choice is:

Table 1 has **retail location** information. Table 2 has **product** information. You want to match all products with all locations to create a list of all possible combinations.

- Your `JOIN` solution choice is:

INNER, OUTER, EXCEPTION, AND CROSS JOINS

Table 1 has a list of **item descriptions**. Table 2 has a list of **item sales**. Table 1 is missing some data. For your purposes, You want to see all of the item descriptions for these sales, even if some descriptions are missing.

- Your `JOIN` solution choice is:

Table 1 has **item sales**. Table 2 has pending **deleted** items. You need to find all items that are pending deletion and have **no sales data**.

- Your `JOIN` solution choice is:

MULTIPLE RELATIONSHIPS IN SQL

GUIDED PRACTICE: JOIN SYNTAX

SYNTAX WALK THROUGH

Let’s look at some sample syntax for JOIN statements.

RIGHT OUTER JOIN:

```
    SELECT b.location, b.address, b.status, a.location, a.sales
    FROM table1 a
    RIGHT JOIN table2 b
    ON a.location = b.location;
```

LEFT OUTER JOIN:

```
    SELECT a.location, a.sales, b.location, b.address, b.status
    FROM table1 a
    LEFT JOIN table2 b
    ON a.location = b.location;
```

MULTIPLE RELATIONSHIPS IN SQL

INDEPENDENT PRACTICE: OUTER + FULL OUTER JOINS

ACTIVITY: OUTER AND FULL OUTER JOINS

EXERCISE

DIRECTIONS

Let’s field this request for information:

“We want to see all of the information we can get on inactive stores (if there are any) for sales, as well as their addresses.”

DELIVERABLE

Construct a query to provide the necessary information.

ACTIVITY: OUTER AND FULL OUTER JOINS

EXERCISE

DIRECTIONS

Let’s field this request for information:

“We want to see all of the information we can get on inactive stores (if there are any) for sales, as well as their addresses.”

SAMPLE SOLUTION: 548 rows

```
SELECT *
FROM stores
LEFT JOIN sales
USING (store)
WHERE stores.store_status = 'I’ OR stores.store_status IS NULL;
```

Note: Because inactive stores have no recorded sales, what do you see in the total column?

ACTIVITY: A MORE EFFICIENT QUERY!

EXERCISE

DIRECTIONS

Let’s field this request for information:

“We want to see all of the information we can get on inactive stores (if there are any) for sales, as well as their addresses.”

BETTER SOLUTION: 548 rows

```
SELECT *
FROM stores s
LEFT JOIN sales ss
USING (store)
WHERE s.store_status = 'I’ OR s.store_status IS NULL;
```

Note: Because inactive stores have no recorded sales, what do you see in the total column?

ACTIVITY: A MORE EFFICIENT QUERY!

EXERCISE

```
SELECT *
FROM stores s
LEFT JOIN sales ss on ss.store = s.store
WHERE s.store_status = 'I’ OR s.store_status IS NULL;
```

Observations:

- Notice how we use an **alias** to call each column name from the **Stores** table.
- This saves us from typing “store.store,” “store.store_status,” etc. and makes writing the code more efficient.
- While we can assign `USING (store)` as the `JOIN` key, most databases will not have the same column name across different databases.
- Database admins tend to have a **primary key** in the **Stores** table and use a foreign key in other tables that call for a *store_id*.
    - It could look like *pk_store* in the **Stores** table and *fk_stores* in the Sales table.

MULTIPLE RELATIONSHIPS IN SQL

INTRODUCTION: OPTIMIZING JOINS

OPTIMIZING JOINS

- Take advantage of `WHERE` clauses in `JOINs` with large tables (such as the Iowa Liquor Sales table).

Recommended practice for slow queries:

- Filter one or both of the tables with a `WHERE` clause in the same query as the `JOIN`.
- This will filter the table **before** the `JOIN` occurs. Depending on the server environment, it may save time.
- When testing `JOINs`, use `LIMIT` to control query sizes.

OPTIMIZING JOINS

Optional syntax when the key field name is identical:

- USING(store) is shorthand for ON a.store = b.store

Controlling DISTINCT by using DISTINCT ON():

```
SELECT DISTINCT ON(location) location, time, report
    FROM weather_reports
    ORDER BY location, time DESC
```

OPTIMIZING JOINS – PREVIEW NULLs

PostgreSQL’s supported list of comparison methods:

Evaluating for NULL?

- IS NULL
- IS NOT NULL
- ISNULL
- NOTNULL

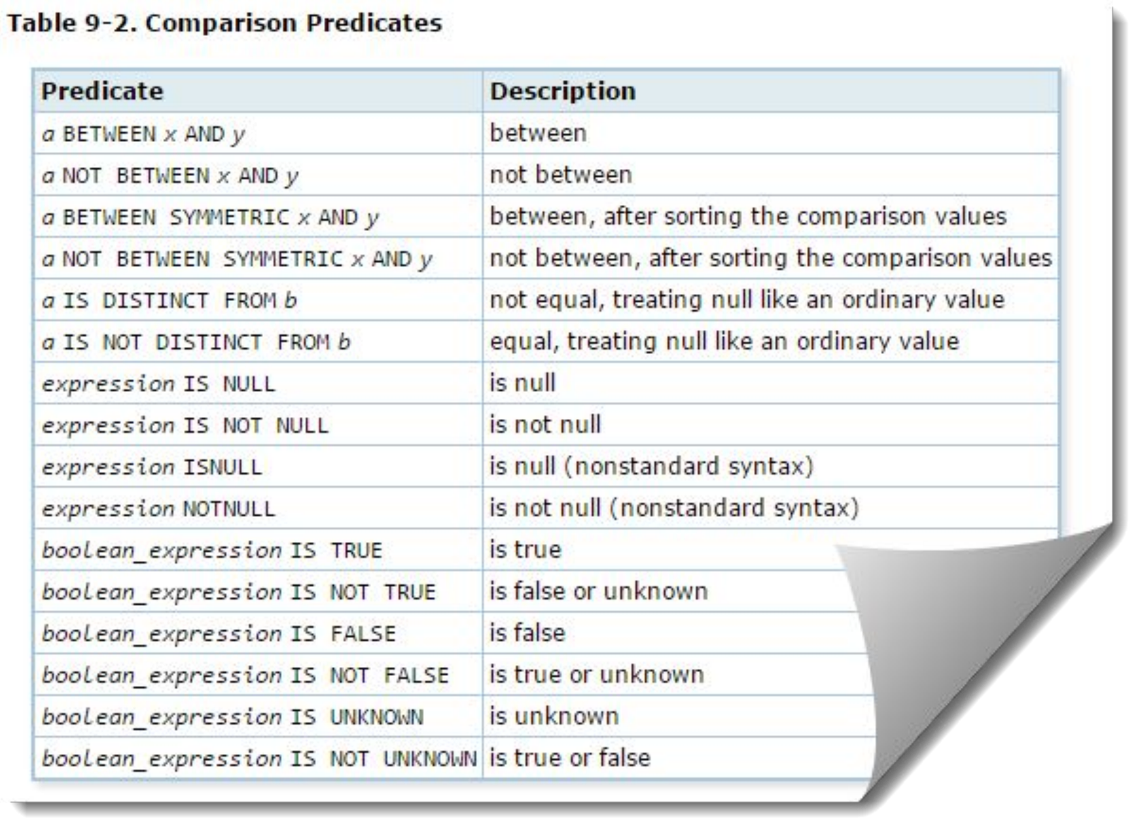


Table 9-2. Comparison Predicates

Predicate	Description
a BETWEEN *x* AND *y*	between
a NOT BETWEEN *x* AND *y*	not between
a BETWEEN SYMMETRIC *x* AND *y*	between, after sorting the comparison values
a NOT BETWEEN SYMMETRIC *x* AND *y*	not between, after sorting the comparison values
a IS DISTINCT FROM *b*	not equal, treating null like an ordinary value
a IS NOT DISTINCT FROM *b*	equal, treating null like an ordinary value
expression IS NULL	is null
expression IS NOT NULL	is not null
expression ISNULL	is null (nonstandard syntax)
expression NOTNULL	is not null (nonstandard syntax)
boolean_expression IS TRUE	is true
boolean_expression IS NOT TRUE	is false or unknown
boolean_expression IS FALSE	is false
boolean_expression IS NOT FALSE	is true or unknown
boolean_expression IS UNKNOWN	is unknown
boolean_expression IS NOT UNKNOWN	is true or false

OPTIMIZING JOINS; PREVIEW NULLS

`NULL` values can be produced with all `JOINs` except `INNER`. `NULL` is helpful, as it shows that there wasn’t a match. But what if we wanted to specify a default value?

1. `COALESCE(field1,‘value’)` is a way to take a `NULL` and replace it with a more useful value. It returns the first non-null column argument.

2. `NULLIF(field1,alternative)` where the “alternative” could be zero (for math equations) or a text string (like “none”).

3. `CASE(IF,THEN,ELSEIF)` is a logic structure frequently used to refine or clean data.

MULTIPLE RELATIONSHIPS IN SQL

INDEPENDENT PRACTICE: JOINS AND NULLS

ACTIVITY: JOINS AND NULLS

EXERCISE

DIRECTIONS

Your Deloitte boss has some more questions. Write queries to answer the following questions:

1. Show the sales in the database completed at an active store.
   a. Try connecting the keys with “USING.”
   b. Limit to 1,000 rows.
   c. Experiment with grouping and order.
2. Which sales included tequila products?
3. Which tequila products were not sold?
4. Which distinct products were sold in Mason City, IA?
5. Which Scotch whiskies were sold in Mason City, IA?
6. Which unique products, other than whiskies, were sold in Mason City, IA?
7. As a check for data consistency, were there any sales of products that are not listed in the Products table?
8. As another check for data consistency, were there any sales at a store that doesn’t exist?

JOINS AND NULLS: EXAMPLE SOLUTIONS

1. Show the sales in the database completed at an active store. Try connecting the keys with “USING.” Limit to 1,000 rows. Experiment with grouping and order:

```
select * from sales a inner join stores b using (store) where
b.store_status = ‘A’;
```

2. Which sales included tequila products?
   - 134,504 sale entries

```
select * from sales b where b.category_name like '%TEQUILA%';
```

JOINS AND NULLS: EXAMPLE SOLUTIONS

3. Which tequila products were not sold?
   - 335 items were not sold (distinct and distinct on don’t change the row count).

```
select item_description from products a left join sales b on
a.item_no = b.item where a.category_name like '%TEQUILA%' and
b.store is null;
```

4. Which distinct products were sold in Mason City, IA?
   - 1475 unique products.

```
select distinct on(a.description) a.description,
b.store_address from sales a
inner join stores b using (store)
where b.store_address like '%Mason City%';
```

JOINS AND NULLS: EXAMPLE SOLUTIONS

5. Which Scotch whiskies were sold in Mason City, IA?
    - 40 unique products.

```
select distinct description, category_name from sales a inner join
stores b using (store) where store_address like '%Mason City%' and
category_name like '%SCOTCH WHISKIES%';
```

6. Which unique products, other than whiskies, were sold in Mason City, IA?
    - 1191 unique products.

```
select distinct a.description, a.category_name from sales a inner join
stores b using (store) where a.category_name not like '%WHISKIES%' and
store_address like '%Mason City%';
```

JOINS AND NULLS: EXAMPLE SOLUTIONS

7. As a check for data consistency, were there any sales of products that are not listed in the Products table?

   - 177 rows.

```
SELECT a.description FROM sales a EXCEPT SELECT
b.item_description FROM products b;
```

8. As another check for data consistency, were there any sales at a store that does not exist?

   - 31 rows.

```
SELECT store FROM sales EXCEPT SELECT store FROM stores;
```

MULTIPLE RELATIONSHIPS IN SQL

DIGGING DEEPER

pgAdmin EXCEPT FUNCTION

The syntax for the **EXCEPT** operator in PostgreSQL is:

```
SELECT expression1, expression2, ... expression_n
FROM tables [WHERE conditions]
EXCEPT
SELECT expression1, expression2, ... expression_n
FROM tables [WHERE conditions];
```

Expressions

- Expressions are the columns or calculations that you wish to compare between the two `SELECT` statements.
- They do not have to be the same fields in each of the `SELECT` statements, but the corresponding columns must be similar data types.

pgAdmin EXCEPT FUNCTION

Next, let's look at an example of an EXCEPT query in PostgreSQL that returns more than one column:

```
SELECT contact_id, last_name, first_name
FROM contacts
WHERE last_name = 'Anderson'
EXCEPT
SELECT customer_id, last_name, first_name
FROM customers
WHERE customer_id < 99;
```

In this EXCEPT example, the query will return the records in the contacts table with any values from contact_id, last_name, and first_name value that ***do not match*** the customer_id, last_name, and first_name values in the customers table.

MULTIPLE RELATIONSHIPS IN SQL

CONCLUSION

REVIEW: MULTIPLE RELATIONSHIPS IN SQL

In this lesson, we learned how to:

1. Create relationships between tables using:
   a. `INNER`, `RIGHT`, and `LEFT JOINS`.
   b. `FULL OUTER JOINS`.
   c. `EXCEPTION JOINS`.
   d. `CROSS JOINS`.
2. Practice the concepts and syntax for each `JOIN`.
3. Apply strategies for optimizing queries using `WHERE`, `LIMIT`, and `COALESCE`.

MULTIPLE RELATIONSHIPS IN SQL

Q&A