Database

SQL

Aggregate Queries

Case Study

Game Store

Requirement

Game Store Requirement

Our company, **Apasaja Pte Ltd**, has been commissioned to develop an application to manage the data of an online app store. We want to store several items of information about our customers such as their first name, last name, date of birth, e-mail, date and country of registration to our online sales service and the customer identifier that they have chosen.

We also want to manage the list of our products, games, their name, their version, and their price. The price is fixed for each version of each game. Finally, our customers buy and download games. We record which version of which game each customer has downloaded. It is not essential to keep the download date for this application.

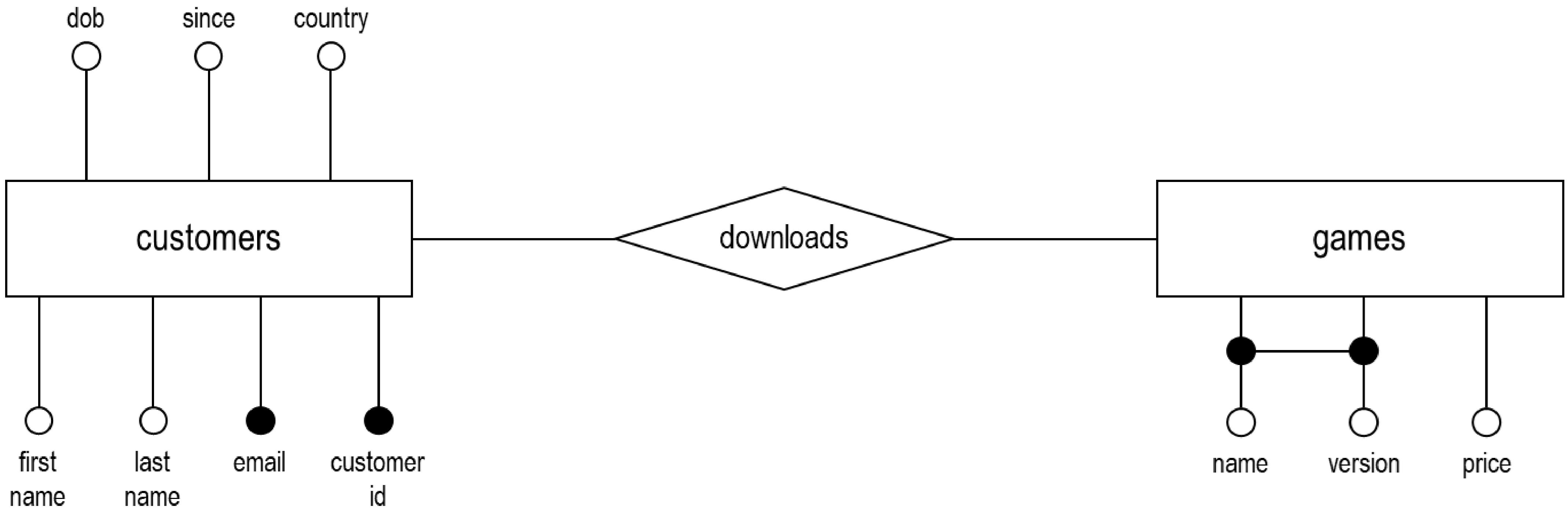
Case Study

Requirement

❱ Design

Design

Entity-Relationship Diagram



Aggregation

Functions

Basic

Aggregation Functions

An **aggregate function** is a function that takes in a set of values and returns a single value.

The values of column can be aggregated using **aggregation functions** such as `COUNT()`, `SUM()`, `MAX()`, `MIN()`, `AVG()`, `STDDEV()`, *etc.*. PostgreSQL also allows user-defined aggregate functions.

```
SELECT COUNT(*)
FROM customers c;
```

```
SELECT COUNT(c.customerid)
FROM customers c;
```

count
1000

Note

`COUNT(*)` counts the total number of rows in the table.

```
SELECT COUNT(ALL c.country)
FROM customers c;
```

Note

`ALL` is the default and often omitted. This counts duplicate entries.

Aggregation

Functions

Distinct

DISTINCT Keyword

We need to add the keyword **DISTINCT** inside the `COUNT()` aggregate function if we want to count the number of **different** countries in the column `country` of the table `customers`.

The keyword **DISTINCT** can be used in other aggregate functions similarly.

```
SELECT COUNT(DISTINCT c.country)
FROM customers c;
```

count
5

Aggregation

- **Functions**
  - Basic
  - Distinct
  - *Example*
  - WHERE
  - Behavior
  - NULL
- Grouping
- Having
- Summary

Functions

Example

Aggregate Functions Example

The following query finds the **maximum**, **minimum**, **average**, and **standard deviation** prices of our games. These aggregate functions only works on **numerical data**.

It uses the arithmetic `TRUNC()` to display **two decimal places** for average and standard deviation.

```sql
SELECT MAX(g.price), MIN(g.price),
  TRUNC(AVG(g.price), 2) AS avg,
  TRUNC(STDDEV(g.price), 2) AS std
FROM games g;
```

max	min	avg	std
12	1.99	6.97	3.96

Aggregation

Functions

WHERE

Aggregate with WHERE Clause

We can first remove the unwanted rows before aggregating by using the `WHERE` clause. The following query finds the maximum, minimum, and average price for "Aerified".

Currently, there is only one single big group to be aggregated.

```sql
SELECT MAX(g.price), MIN(g.price),
  TRUNC(AVG(g.price), 2)
FROM games g
WHERE name = 'Aerified';
```

max	min	avg
12	1.99	6.49

Aggregation

- ❱ Functions
  - *Basic*
  - *Distinct*
  - *Example*
  - *WHERE*
  - ***Behavior***
  - *NULL*
- Grouping
- Having
- Summary

Functions

Behavior

Basic Interpretation

Let R be a non-empty table with attribute A.

...	A	...
...	3	...
...	NULL	...
...	42	...
...	0	...
...	3	...

Query	Interpretation	Result
`SELECT MIN(A) FROM R;`	The smallest non-null value in A	0
`SELECT MAX(A) FROM R;`	The largest non-null value in A	42
`SELECT AVG(A) FROM R;`	Average of non-null values in A	12
`SELECT SUM(A) FROM R;`	The sum of non-null values in A	48
`SELECT COUNT(A) FROM R;`	The count of non-null values in A	4
`SELECT COUNT(*) FROM R;`	The count of all rows in A	5
`SELECT AVG(DISTINCT A) FROM R;`	Average of *distinct* non-null values in A	15
`SELECT SUM(DISTINCT A) FROM R;`	The sum of *distinct* non-null values in A	45
`SELECT COUNT(DISTINCT A) FROM R;`	The count of *distinct* non-null values in A	3

Aggregation

Functions

NULL

Basic Interpretation

Let T be empty relation. Let R be non-empty but contains only NULL.

...	A	...

...	A	...
...	NULL	...
...	NULL	...
...	NULL	...
...	NULL	...
...	NULL	...

Query	Result
`SELECT MIN(A) FROM T`	NULL
`SELECT MAX(A) FROM T`	NULL
`SELECT AVG(A) FROM T`	NULL
`SELECT SUM(A) FROM T`	NULL
`SELECT COUNT(A) FROM T`	0
`SELECT COUNT(*) FROM T`	0

Query	Result
`SELECT MIN(A) FROM R`	NULL
`SELECT MAX(A) FROM R`	NULL
`SELECT AVG(A) FROM R`	NULL
`SELECT SUM(A) FROM R`	NULL
`SELECT COUNT(A) FROM R`	0
`SELECT COUNT(*) FROM R`	5

Aggregation

Grouping

Logical

GROUP BY

The `GROUP BY` clause creates logical groups of records that have the same values for the specified fields before computing the aggregate functions.

```
GROUP BY c.country;
```

first_name	last_name	email	...	country
"Deborah"	"Ruiz"	"druiz0@drupal.org"	...	"Singapore"
"Tammy"	"Lee"	"tlee1@barnesandnobles.com"	...	"Singapore"
:	:	:	...	:
"Raymon"	"Tan"	"rtan1z@nature.com"	...	"Thailand"
"Jean"	"Ling"	"jlingpn@walmart.com"	...	"Thailand"
:	:	:	...	:

*The table above is only a potential representation.

Aggregation

Functions

❱ Grouping

Logical

Aggregation

Where

From

Select

Renaming

Group Order

Having

Summary

Grouping

Aggregation

Aggregation Function Per Group

The aggregation functions are calculated for each logical group.

```
SELECT c.country, COUNT(*)
FROM customers c
GROUP BY c.country;
```

country	count
"Singapore"	391
"Thailand"	100
...	

...	country
...	"Singapore"
...	"Singapore"
...	:
...	"Thailand"
...	"Thailand"
...	:

count
391
:
100
:

Aggregation

Grouping

Aggregation

Aggregation Function Per Group

The aggregation functions are calculated for each logical group.

```
SELECT c.country, COUNT(*)
FROM customers c
GROUP BY c.country;
```

```
SELECT c.country, COUNT(*)
FROM customers c;
/* only one group created */
```

country	count
"Singapore"	391
"Thailand"	100
...	

Error

This is actually an error as we cannot select only one value of `c.country`.

Aggregation

- Functions
- ❱ Grouping
  - *Logical*
  - *Aggregation*
  - ***Where***
  - *From*
  - *Select*
  - *Renaming*
  - *Group Order*
- Having
- Summary

Grouping

Where

After WHERE

Groups are formed *(logically)* **after** the rows have been filtered by the `WHERE` clause.

```sql
SELECT c.country, COUNT(*)
FROM customers c
WHERE c.dob >= '2006-01-01'
GROUP BY c.country;
```

country	count
"Vietnam"	4
"Singapore"	25
"Thailand"	5
"Indonesia"	15
"Malaysia"	12

Aggregation

Functions
❱ Grouping
Logical
Aggregation
Where
From
Select
Renaming
Group Order
Having
Summary

Grouping

From

After FROM

Groups are formed *(logically)* **after** the tables have been joined in the `FROM` clause.

```
SELECT c.customerid, c.first_name, c.last_name, SUM(g.price)
FROM customers c, downloads d, games g
WHERE c.customerid = d.customerid
  AND d.name = g.name and d.version = g.version
GROUP BY c.customerid, c.first_name, c.last_name;
```

Note

Find the total spending of each customer.

Grouping

Select

SELECT Clause

It is recommended *(and required per SQL standard)* to **include attributes projected** in the `SELECT` clause by the `GROUP BY` clause.

```
SELECT c.customerid, c.first_name, c.last_name, SUM(g.price)
FROM customers c, downloads d, games g
WHERE c.customerid = d.customerid
  AND d.name = g.name and d.version = g.version
GROUP BY c.customerid;
```

Bad Practice

The above query works only because `first_name` and `last_name` are guaranteed unique.

Aggregation

Functions
❯ Grouping
Logical
Aggregation
Where
From
Select
Renaming
Group Order
Having
Summary

Grouping

Select

Invalid Query

The following query **does not work** in PostgreSQL *(but works in SQLite with potentially incorrect result)*. We will run all codes on PostgreSQL for testing.

```sql
SELECT c.customerid, c.first_name, c.last_name, SUM(g.price)
FROM customers c, downloads d, games g
WHERE c.customerid = d.customerid
  AND d.name = g.name and d.version = g.version
GROUP BY c.first_name, c.last_name;
```

Issue

If there are two customers with the same first and last name, which `customerid` is selected?

Aggregation

Functions
Grouping
Logical
Aggregation
Where
From
Select
Renaming
Group Order
Having
Summary

Grouping

Renaming

Renamed Column

Renamed columns can be used in GROUP BY clause. The following query displays the number of downloads by country and year of birth *(using EXTRACT)*.

```
SELECT c.country, EXTRACT(YEAR FROM c.since) AS regyear, COUNT(*) AS total
FROM customers c, downloads d
WHERE c.customerid = d.customerid
GROUP BY c.country, regyear
ORDER BY regyear ASC, c.country ASC;
```

Aggregation

Functions
❯ Grouping
Logical
Aggregation
Where
From
Select
Renaming
Group Order
Having
Summary

Grouping

Group Order

GROUP BY Reordering

The order of columns in `GROUP BY` clause does not change the meaning of the query. The logical groups remain the same.

```
SELECT c.country, EXTRACT(YEAR FROM c.since) AS regyear, COUNT(*) AS total
FROM customers c, downloads d
WHERE c.customerid = d.customerid
GROUP BY regyear, c.country
ORDER BY regyear ASC, c.country ASC;
```

Aggregation

Having

Condition

Aggregate Condition

Aggregate functions can be used in **conditions**, but not in WHERE clause. Aggregate functions can be evaluated after groups are formed *(which is after WHERE clause)*.

```sql
SELECT c.country
FROM customers c
WHERE COUNT(*) >= 100
GROUP BY c.country;
```

HAVING Clause

We need a new clause: `HAVING` clause. This clause is performed **after** `GROUP BY` clause.

`HAVING` clause can **only use** aggregate functions, columns listed in the `GROUP BY` clause, and subqueries.

Aggregation

Having

Condition

Aggregate Condition

Aggregate functions can be used in **conditions**, but not in WHERE clause. Aggregate functions can be evaluated after groups are formed *(which is after WHERE clause)*.

```sql
SELECT c.country
FROM customers c
GROUP BY c.country
HAVING COUNT(*) >= 100;
```

Note

The query on the left finds the countries in which there are more than 100 customers.

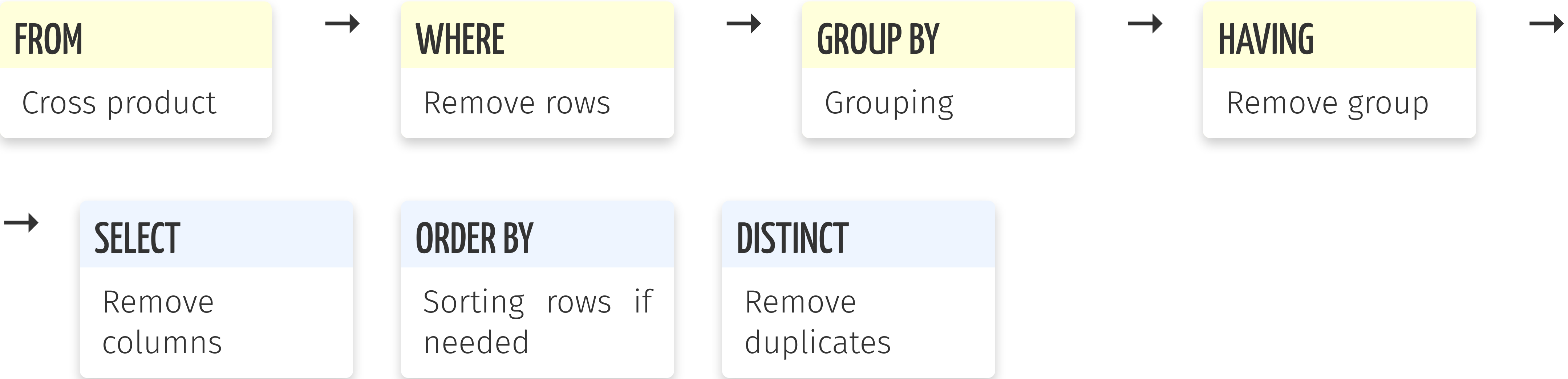
Aggregation

Summary

Evaluation

Logical Evaluation

The **logical evaluation** of SQL query up to this point is the following.



FROM
Cross product

→

WHERE
Remove rows

→

GROUP BY
Grouping

→

HAVING
Remove group

→

→

SELECT
Remove columns

ORDER BY
Sorting rows if needed

DISTINCT
Remove duplicates

Break

Joins

Inner Join

Basic

Expressiveness

While inner join is a popular construct, there is no added expressiveness or performance in `INNER JOIN`. The two queries below are equivalent.

Inner Join

```
SELECT *
FROM customers c
  INNER JOIN downloads d
    ON d.customerid = c.customerid
  INNER JOIN games g
    ON d.name = g.name
    AND d.version = g.version;
```

Cross Join

```
SELECT *
FROM customers c, downloads d,
  games g
WHERE d.customerid = c.customerid
  AND d.name = g.name
  AND d.version = g.version;
```

Joins

Inner Join

JOIN

Synonym

`JOIN` is synonymous with `INNER JOIN`. We do **NOT** recommend either as `CROSS JOIN` *(or comma)* is typically easier to read and will be optimized by DBMS.

Join

```sql
SELECT *
FROM customers c
  JOIN downloads d
    ON d.customerid = c.customerid
  JOIN games g
    ON d.name = g.name
    AND d.version = g.version;
```

Cross Join

```sql
SELECT *
FROM customers c, downloads d, games g
WHERE d.customerid = c.customerid
  AND d.name = g.name
  AND d.version = g.version;
```

Joins

❱ Inner Join
Basic
JOIN
Condition
Natural Join
Outer Join

Inner Join

Condition

Order of Condition

In JOIN, the order of condition matters. We cannot easily reorder the conditions unlike in CROSS JOIN. That is why we recommend simply using comma.

Join

```sql
SELECT *
FROM customers c
  JOIN downloads d
    ON d.name = g.name  -- what is g?
    AND d.version = g.version
  JOIN games g
    ON d.customerid = c.customerid;
```

Cross Join

```sql
SELECT *
FROM customers c, downloads d, games g
WHERE d.customerid = c.customerid
  AND d.name = g.name
  AND d.version = g.version;
```

Joins

Inner Join
- Natural Join
  - *What is?*
  - *Design*

Outer Join

Natural Join

What is Natural?

Automatic Equality

If we managed to give the same name to columns with the same meaning across the tables, we can use natural join. `NATURAL JOIN` joins rows that have the same values for columns with the same name. It also prints only one of the two columns.

Natural Join

```
SELECT *
FROM customers c
  NATURAL JOIN downloads d
  NATURAL JOIN games g;
```

Question

Can you write the equivalent of the query on the left using `CROSS JOIN`?

Joins

Natural Join

Design Impact

Universal Relation

If we want to use NATURAL JOIN more easily, we need to ensure that columns with the same should have the same meaning. Otherwise, we cannot use NATURAL JOIN and we have to use CROSS JOIN or INNER JOIN. This condition is called universal relation.

Natural Join

```
SELECT *
FROM customers c
  NATURAL JOIN downloads d
  NATURAL JOIN games g;
```

Issue

The SQL query on the left does not work if the attributes are different such as the following (e.g., we change `customerid` to `id` in `customers` table only).

Outer Join

Basic

What is Outer?

The **outer join** keeps the columns of the rows in the left *(left outer join)*, the right *(right outer join)*, or in both *(full outer join)* tables that **do not match** anything in the other table according to the join condition *(i.e., dangling rows)*. The remaining values are **padded** with `NULL` values.

Warning

It is better to **avoid outer joins** whenever possible as they introduce `NULL` values. They can sometimes be justified for efficiency reasons. However, this course does not care about efficiency as long as the query can finish within reasonable time.

Note

There are also the **natural** variant of outer joins. For instance, `NATURAL LEFT OUTER JOIN` is a natural version of **left join**.

The meaning is the **combination** of natural join *(i.e., automatic equality)* and left join *(i.e., keeps unmatched column, padded with `NULL`)*.

Joins

Inner Join
Natural Join
❯ Outer Join
Basic
Example
Condition
Anti Join
Closing

Outer Join

Basic

Difficulty of Dangle

We cannot easily obtain dangling rows by using `INNER JOIN`. A row is included in the result of an inner join if the condition is true and the condition can only look at the current row.

```
SELECT *
FROM customers c, downloads d
WHERE c.customerid <> d.customerid;
```

Result

This finds all customer and the games downloaded by other customers.

What we need is a way to say that `c.customerid` is not equal to ALL other `d.customerid`.

This is not possible without nested queries or outer join.

Joins

Outer Join

Example

Left Outer Join

In the example below, the customers --*from the left table*-- who never downloaded a game are combined with NULL values to replace missing values for the columns of the **downloads** table. Columns from the right table are padded with NULL values.

```
SELECT c.customerid, c.email, d.customerid, d.name, d.version
FROM customers c LEFT OUTER JOIN downloads d ON c.customerid = d.customerid;
```

c.customerid	c.email	d.customerid	d.name	d.version
...				
"Willie90"	"wlongjj@moonfruit.com"	"Willie90"	"Ronstring"	"1.1"
"Willie90"	"wlongjj@moonfruit.com"	"Willie90"	"Veribet"	"2.1"
"Al8"	"ahansenp3@webnode.com"	NULL	NULL	NULL
"Johnny1997"	"jstevensb0@un.org"	NULL	NULL	NULL

Joins

Inner Join
Natural Join
❱ Outer Join
Basic
Example
Condition
Anti Join
Closing

Outer Join

Example

Right Outer Join

In the example below, the games --*from the right table*-- that have never been downloade are combined with NULL values to replace missing values for the columns of the `downloads` table. Columns from the left table are padded with NULL values.

```sql
SELECT *
FROM downloads d RIGHT OUTER JOIN games g ON g.name = d.name AND g.version = d.version;
```

Full Outer Join

A **full outer join** pads missing values with NULL for both the tables on the left and on the right.

Joins

Outer Join

Condition

Condition Matters

Dangling rows is defined only with respect to the condition on the ON clause. Moving the condition to WHERE clause will result in different output.

```
SELECT *
FROM downloads d RIGHT OUTER JOIN games g
ON g.name = d.name
AND g.version = d.version;
```

```
SELECT *
FROM downloads d RIGHT OUTER JOIN games g
ON g.name = d.name
WHERE g.version = d.version;
```

d.customerid	d.name	d.version	g.name	g.version	g.price
Adam1983	Biodex	1.0	Biodex	1.0	2.99
Adam1983	Domainer	2.1	Domainer	2.1	2.99
:	:	:	:	:	:
NULL	NULL	NULL	Overhold	2.0	12
NULL	NULL	NULL	Andalax	1.0	12

d.customerid	d.name	d.version	g.name	g.version	g.price
Adam1983	Biodex	1.0	Biodex	1.0	2.99
Adam1983	Domainer	2.1	Domainer	2.1	2.99
:	:	:	:	:	:
Willie90	Ronstring	1.1	Ronstring	1.1	3.99
Willie90	Veribet	2.1	Veribet	2.1	2.99

Joins

Inner Join
Natural Join
❱ Outer Join
Basic
Example
Condition
Anti Join
Closing

Outer Join

Anti Join

Only Missing Value

Find customers who never downloaded a game.

Further Restriction

Further restriction should be on `WHERE` clause and not `ON` clause.

```sql
SELECT c.customerid
FROM customers c
  LEFT OUTER JOIN downloads d
    ON c.customerid = d.customerid
WHERE d.customerid IS NULL;
```

```sql
SELECT c.customerid
FROM customers c
  LEFT OUTER JOIN downloads d
    ON c.customerid = d.customerid
WHERE d.customerid IS NULL
  AND c.country = 'Singapore';
-- try moving the AND above
```

Outer Join

Closing

Warning

Outer joins are not easy to write as conditions in the `ON` clause are not equivalent to conditions in the `WHERE` clause *(as it was the case with `INNER JOIN`)*. Conditions in the `ON` clause determines which rows are **dangling**.

Synonyms

- `LEFT JOIN` is synonym for `LEFT OUTER JOIN`
- `RIGHT JOIN` is synonym for `RIGHT OUTER JOIN`
- `FULL JOIN` is synonym for `FULL OUTER JOIN`

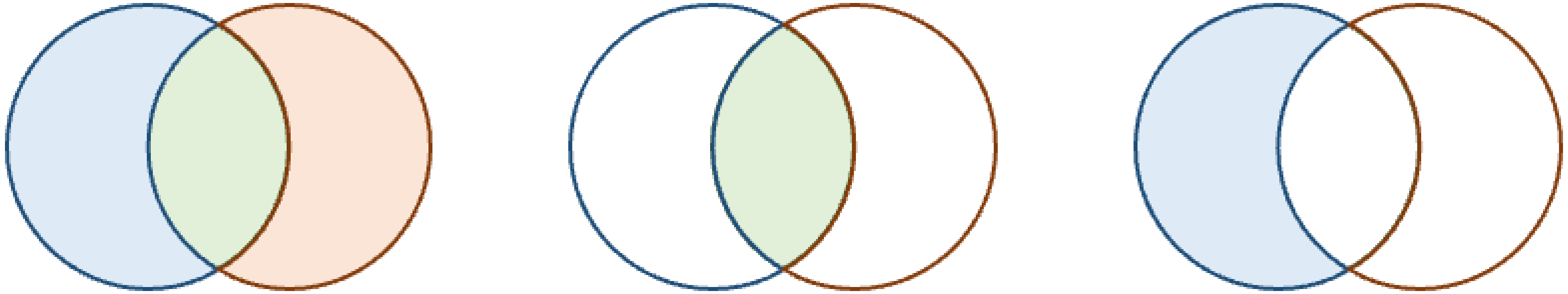
Set Operations

Set

Basic

Operators

The set operators `UNION`, `INTERSECT`, and `EXCEPT` return the union, intersection, and non-symmetric difference of the results of two queries respectively.



Deduplication

Union, intersection, and non-symmetric difference eliminate duplicates unless annotated with the keyword `ALL`.

Set Operations

Set

Compatible

Union-Compatible

Two queries must be **union-compatible** to be used with `UNION`, `INTERSECT`, or `EXCEPT`. They must return the same number of columns with the same domain in the same order.

Compatible

student.name (VARCHAR(32))

department.department (VARCHAR(32))

Incompatible

student.year (DATE)

department.department (VARCHAR(32))

Note

Just because they are **union-compatible** does not mean it is **meaningful** to perform set operations on the two tables.

Set Operations

Set
❯ Examples
Union
Intersection
Difference

Examples

Union

Question

Find the `customerid` of all the **customers** who **downloaded** version 1.0 or 2.0 of the game Aerified.

```
SELECT d.customerid
FROM downloads d
WHERE d.name = 'Aerified' AND d.version = '1.0'
UNION
SELECT d.customerid
FROM downloads d
WHERE d.name = 'Aerified' AND d.version = '2.0';
```

Set Operations

Examples

Union

Question

Find the name and versions of all the games after GST is applied. GST of 9% is applied if it is more than 30 cents.

```
SELECT g.name || ' ' || g.version AS game, ROUND(g.price * 1.09) AS price
FROM games g
WHERE g.price * 0.09 >= 0.3
UNION
SELECT g.name || ' ' || g.version AS game, g.price
FROM games g
WHERE g.price * 0.09 < 0.3;
```

Set Operations

Set
❱ Examples
Union
Intersection
Difference

Examples

Intersection

Question

Find the customerid of all the **customers** who **downloaded** both version 1.0 and 2.0 of the game Aerified.

```
SELECT d.customerid
FROM downloads d
WHERE d.name = 'Aerified' AND d.version = '1.0'
INTERSECT
SELECT d.customerid
FROM downloads d
WHERE d.name = 'Aerified' AND d.version = '2.0';
```

Set Operations

Examples

Difference

Question

Find the `customerid` of the **customers** who **downloaded** version 1.0 but not version 2.0 of the game Aerified.

```sql
SELECT d.customerid
FROM downloads d
WHERE d.name = 'Aerified' AND d.version = '1.0'
EXCEPT
SELECT d.customerid
FROM downloads d
WHERE d.name = 'Aerified' AND d.version = '2.0';
```

Conclusion

Reading

What Does This Query Find?

```sql
SELECT c.email, SUM(g.price)
FROM customers c, downloads d, games g
WHERE c.customerid = d.customerid AND g.name = d.name
  AND g.version = d.version AND c.country = 'Indonesia' AND g.name=  'Fixflex'
GROUP BY c.email
UNION
SELECT c.email, 0
FROM customers c LEFT JOIN
  (downloads d INNER JOIN games g
    ON g.name = d.name AND g.version = d.version AND g.name = 'Fixflex')
  ON c.customerid = d.customerid
WHERE c.country = 'Indonesia' AND d.name IS NULL;
```

```
postgres=# exit
Press any key to continue . . .
```