

Database

SQL

Simple Queries

Case Study

Game Store Requirement Design

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.

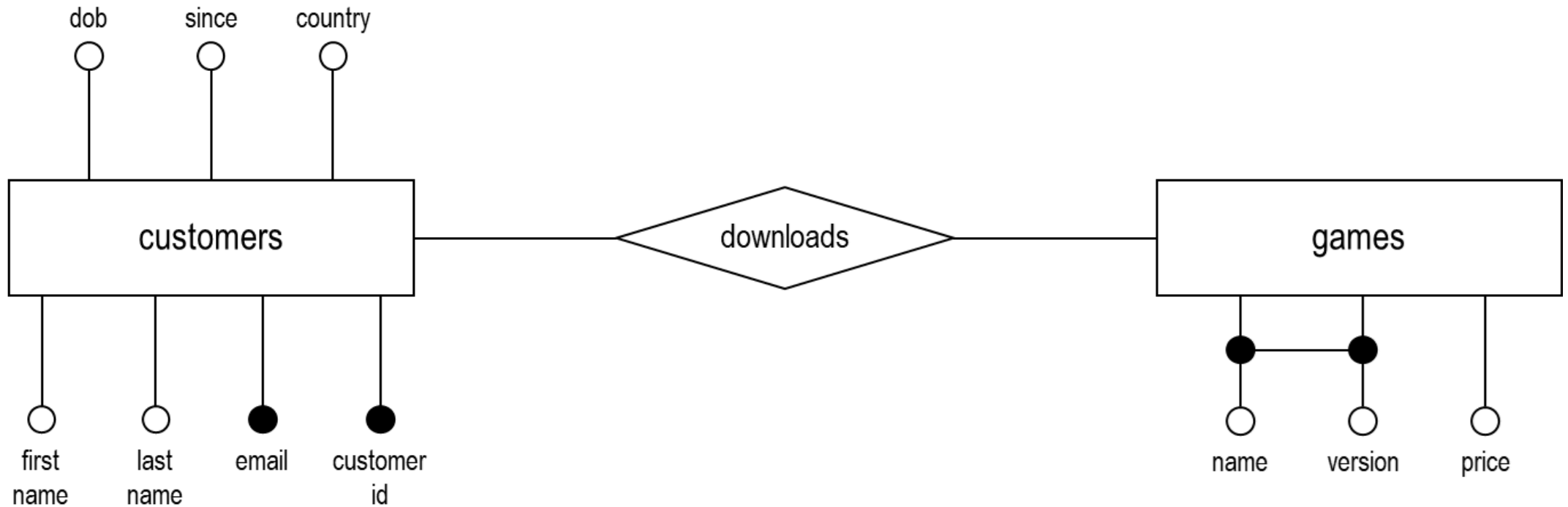
We also want to manage the list of our products, **games**, their **name**, their **version**, and their **price**. The price is fixed for each version of each game. Finally, our customers buy and **download** games. We record which version of which game each customer has downloaded. It is not essential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

Case Study

Requirement
» Design

Design

Entity-Relationship Diagram



Single Table

Basic
Simple Queries
Wildcards
Duplicate
Operation

Basic

Simple Queries

SELECT-FROM-WHERE

A **simple** SQL query includes **SELECT** clause that indicates the **columns** to be output, a **FROM** clause that indicates the **table(s)** to be queried, and possibly a **WHERE** clause that indicates a possible **condition** on the records to be printed.

Code

```
SELECT first_name, last_name
FROM customers
WHERE country = 'Singapore';
```

Table

first_name	last_name
"Deborah"	"Ruiz"
"Tammy"	"Lee"
"Walter"	"Leong"
...	

Single Table

Basic

Simple Queries

Wildcards

Duplicate
Operation

Basic

Wildcards

Asterisk

The **asterisk** (i.e., *★ symbol*) is a shorthand indicating that **all the column names**, in order of the **CREATE TABLE** declaration, should be considered in the **SELECT** clause.

Full Listing

```
SELECT first_name, last_name,  
       email, dob, since,  
       customerid, country  
FROM customers;
```

Using Wildcard

```
SELECT *  
FROM customers;
```

Single Table

Basic
» Duplicate
Ordering
Distinct
Composition
Operation

Duplicate Ordering

Projection

Selecting a **subset of the columns** (*i.e., projection*) of a table may result in **duplicate** rows even if the original table has a primary key. This can be made clearer if we order the result (*i.e., on the right*). Always use **ASC** and **DESC** for **good practice**.

No Ordering

```
SELECT name, version  
FROM downloads;
```

ORDER BY

```
SELECT name, version  
FROM downloads  
ORDER BY name, version;
```

Note

By default, order in **ASC** and not **DESC**.

Single Table

Basic
» Duplicate
Ordering
Distinct
Composition
Operation

Duplicate Ordering

Unordered

If we only **order partially** (*i.e., not using all columns*), the columns not mentioned are in **no particular order**. In particular, we **should not rely on specific ordering** as it should work for any data.

Partial Ordering

```
SELECT name, version  
FROM downloads  
ORDER BY name ASC;
```

Note

We only order by **name**. As such, **version** may appear in any order.

Single Table

Basic
» Duplicate
Ordering
Distinct
Composition
Operation

Duplicate Ordering

Unmentioned

We can order the result according to a **column that is not printed**. But this may **interfere** with other constructs. Only use this when you are sure that it will work.

Partial Ordering

```
SELECT name, version  
FROM games  
ORDER BY price ASC;
```

Note

`price` is used in `ORDER BY` but not present in the `SELECT` clause.

Single Table

Basic
» Duplicate
Ordering
Distinct
Composition
Operation

Duplicate Distinct

Deduplication

The keyword **DISTINCT** eliminates eventual **duplicates** in the result of the query. It requests that the results contains **distinct rows** (*it does not apply to columns individually*).

DISTINCT

```
SELECT DISTINCT name, version  
FROM downloads
```

Note

The query with **DISTINCT** displays **422** records. The query without **DISTINCT** displays **4214** records.

It often gives a sorted result but this is **not guaranteed**.

Single Table

Basic
» Duplicate
Ordering
Distinct
Composition
Operation

Duplicate Composition

Combining Constructs

You can **combine** different constructs **but not always**. Not all combinations make sense. Both **DISTINCT** and **ORDER BY** conceptually involve sorting and **ORDER BY** is applied before **DISTINCT**. SQL does not know which row to remove.

Compatible

```
SELECT DISTINCT name  
FROM games  
ORDER BY name ASC;
```

Incompatible

```
SELECT DISTINCT name, version  
FROM games  
ORDER BY price ASC;
```

Error

price does not appear in **DISTINCT**.

Single Table

Basic
Duplicate
» Operation
Condition
Operators

Operation Condition

WHERE Clause

The **WHERE** clause is used to **filter rows** on a **Boolean** condition. The Boolean condition uses Boolean operators such as **AND**, **OR**, and **NOT** as well as various **comparison** operators such as **>**, **<**, **>=**, **<=**, **<>**, **IN**, **LIKE**, and **BETWEEN .. AND**.

```
SELECT first_name, last_name
FROM customers
WHERE country IN ('Singapore', 'Indonesia')
    AND (dob BETWEEN '2000-01-01' AND '2000-12-01' OR since >= '2016-12-01')
    AND last_name LIKE 'B%';
```

*See: **BETWEEN .. AND** and **LIKE** documentations.

Single Table

Basic
Duplicate
» Operation
Condition
Operators

Operation Operators

Operators in PostgreSQL

Arithmetic and other functions can be used in **SELECT** and **WHERE** clauses (*as well as in the **ORDER BY**, **GROUP BY**, and **HAVING** clauses*).

The operator **AS** is a **renaming** operator.

In SELECT Clause

```
SELECT DISTINCT price * 1.09 AS gst    -- add GST of 9%
FROM games
ORDER BY gst ASC;
```

*More info on [operators](#).

Single Table

Basic
Duplicate
» Operation
Condition
Operators

Operation Operators

Operators in PostgreSQL

Arithmetic and other functions can be used in **SELECT** and **WHERE** clauses (*as well as in the **ORDER BY**, **GROUP BY**, and **HAVING** clauses*).

The operator `||` is a **concatenation** operator.

In WHERE Clauses

```
SELECT
  name || ' ' || version AS game,
  ROUND(price * 1.09, 2) AS price
FROM games
WHERE price * 0.09 >= 0.3;
```

```
SELECT
  name || ' ' || version AS game,
  price
FROM games
WHERE price * 0.09 < 0.3;
```

*We perform operations on **price** and rename the result as **price** again.

Single Table

Basic
Duplicate
» Operation
Condition
Operators

Operation Operators

Case Analysis

We can do this but not preferable.

CASE-WHEN-THEN-ELSE is like **if-else**. We perform the **first** condition that is **satisfied** with checking done from **top to bottom**.

```
SELECT name || ' ' || version,  
       CASE  
         WHEN price * 0.09 >= 0.3 THEN ROUND(price * 1.09, 2)  
         ELSE price  
       END AS price  
FROM games;
```

*The syntax of operations and functions can be specific to DBMS, see [PostgreSQL documentation](#).

Logic and NULL

► Reasoning
Differences?
De Morgan's Law
Boolean
Operations

Reasoning

Differences?

```
SELECT first_name, last_name FROM customers
WHERE (country = 'Singapore' OR country = 'Indonesia')
AND ((dob >= '2000-01-01' AND dob <= '2000-12-01') OR since >= '2016-12-01')
AND last_name LIKE 'B%';
```

```
SELECT first_name, last_name FROM customers
WHERE country IN ('Indonesia', 'Singapore')
AND last_name LIKE 'B%'
AND (since >= '2016-12-01' OR NOT (dob < '2000-01-01' OR dob > '2000-12-01'));
```

```
SELECT first_name, last_name FROM customers
WHERE (country = 'Singapore' OR country = 'Indonesia')
AND (dob BETWEEN '2000-01-01' AND '2000-12-01' OR since >= '2016-12-01')
AND last_name LIKE 'B%';
```

Logic and NULL

Reasoning
» De Morgan's Law
Boolean
Operations

De Morgan's Law Equivalence

Negation of Conjunction/Disjunction

$$\neg(A \wedge B) \equiv \neg A \vee \neg B$$

$$\neg(A \vee B) \equiv \neg A \wedge \neg B$$

[Wikipedia](#)

```
SELECT name FROM games
WHERE (version = '1.0' OR version = '1.1');
```

```
SELECT name FROM games
WHERE version IN ('1.0', '1.1');
```

```
SELECT name FROM games
WHERE NOT (version <> '1.0' AND version <> '1.1');
```


Logic and NULL

Reasoning
De Morgan's Law
» Boolean
Truth Table
NULL Value
Three Valued
Operations

Boolean

Truth Table

WHERE Clause

"SELECT FROM WHERE <condition>" returns results when the <condition> is **True** (*later on, you will see that it does not return result when the <condition> is **False** or **NULL**.*

P	Q	P AND Q	P OR Q	NOT P
True	True	True	True	False
True	False	False	True	False
False	True	False	True	True
False	False	False	False	True

Logic and NULL

Reasoning
De Morgan's Law
» Boolean

Truth Table
NULL Value
Three Valued

Operations

Boolean

NULL Value

Meaning of NULL

Every **domain** has the additional value: the **NULL** value. In SQL it generally (*but not always*) has the meaning of "unknown".

Examples

- "something = NULL" is unknown (*even if "something" is NULL*).
- "something <> NULL" is unknown (*even if "something" is NULL*).
- "something > NULL" is unknown.
- "something < NULL" is unknown.

etc

- "10 + NULL" is null.
- "0 * NULL" is null!

Logic and NULL

Reasoning
De Morgan's Law
» Boolean

Truth Table
NULL Value
Three Valued
Operations

Boolean

NULL Value

```
CREATE TABLE example (  
  column1 VARCHAR(32),  
  column2 NUMERIC  
);
```

```
SELECT *  
FROM example;
```

```
INSERT INTO example VALUES  
  ('abc', 1),  
  ('def', NULL),  
  ('ghi', NULL);
```

column1	column2
"abc"	1
"def"	
"ghi"	

Logic and NULL

Reasoning
De Morgan's Law
» Boolean
Truth Table
NULL Value
Three Valued
Operations

Boolean

NULL Value

```
SELECT column1, column2
FROM example
WHERE column2 > 0;
```

column1	column2
"abc"	1

```
SELECT column1, column2
FROM example
WHERE column2 <= 0;
```

column1	column2

Note

NULL <= 0 and NULL > 0 evaluates to NULL.

Logic and NULL

Reasoning
De Morgan's Law
Boolean
Truth Table
NULL Value
Three Valued
Operations

Boolean NULL Value

```
SELECT column1,  
       column2 * 0 AS newcolumn2  
FROM example;
```

column1	newcolumn2
"abc"	0
"def"	
"ghi"	

```
SELECT column1, column2  
FROM example  
WHERE column2 <> NULL;
```

column1	column2

Note
NULL <> NULL evaluates to NULL.

Logic and NULL

Reasoning
De Morgan's Law
» Boolean
Truth Table
NULL Value
Three Valued
Operations

Boolean Three Valued

P	Q	P AND Q	P OR Q	NOT P
True	True	True	True	False
True	False	False	True	False
True	Unknown	Unknown	True	False
False	True	False	True	True
False	False	False	False	True
False	Unknown	False	Unknown	True
Unknown	True	Unknown	True	Unknown
Unknown	False	False	Unknown	Unknown
Unknown	Unknown	Unknown	Unknown	Unknown

Logic and NULL

Reasoning
De Morgan's Law
Boolean
» Operations
NULL Operators
Counting

Operations

NULL Operators

Operators Involving NULL Values

- IS NULL and IS NOT NULL checks for NULL values.
 - "NULL IS NULL" is **True**.
 - "1 IS NULL" is **False**.
 - "NULL IS NOT NULL" is **False**.
 - "1 IS NOT NULL" is **True**.
- COALESCE(x1, x2, x3, ...) returns the first non-NULL (*from left-to-right*) of its argument.
 - "COALESCE(NULL, 1, NULL, 2)" is 1.
 - "COALESCE(NULL, NULL, NULL)" is NULL.

Logic and NULL

Reasoning
De Morgan's Law
Boolean
» Operations
NULL Operators
Counting

Operations

NULL Operators

```
SELECT column1, column2,  
       COALESCE(column2, 0) AS col2  
FROM example  
WHERE column2 = NULL;
```

column1	column2	col2

Note

NULL = NULL evaluates to NULL.

```
SELECT column1, column2,  
       COALESCE(column2, 0) AS col2  
FROM example  
WHERE column2 IS NULL;
```

column1	column2	col2
"def"		0
"ghi"		0

Logic and NULL

Reasoning
De Morgan's Law
Boolean
» Operations
NULL Operators
Counting

Operations

NULL Operators

```
SELECT column1, column2,  
  (  
    CASE WHEN column2 is NULL THEN 0  
    ELSE column2 END  
  ) AS col2  
FROM example;
```

Note

We need not use COALESCE but CASE-WHEN is harder to read in general.

column1	column2	col2
"abc"	1	1
"def"		0
"ghi"		0

Logic and NULL

Reasoning
De Morgan's Law
Boolean
» Operations
NULL Operators
Counting

Operations Counting

Counting with NULL

"COUNT(*)" counts NULL values.
"COUNT(att)", "AVG(att)", "MAX(att)", "MIN(att)" eliminate NULL values.

```
SELECT COUNT(*) AS ctx  
FROM example;
```

ctx
3

```
SELECT COUNT(column2) AS ctx  
FROM example;
```

ctx
1

Logic and NULL



Dear MR Adi Yoga Sidi Prabawa null,

Thank you for registering Singtel OnePass.

Multiple Table

» Cross Product
Primary/Foreign
Understanding

Cross Product Syntax

Cartesian Product

Cartesian product of tables is the table containing **all the columns** and all **possible combinations of the rows** of the three tables. The comma in the **FROM** clause is a **convenient shorthand** for the keyword **CROSS JOIN**.

Comma

```
SELECT *  
FROM customers, downloads, games;
```

Note

Number of **columns** is the **sum** of columns.
Number of **rows** is the **product** of rows.

CROSS JOIN

```
SELECT *  
FROM customers  
    CROSS JOIN downloads  
    CROSS JOIN games;
```

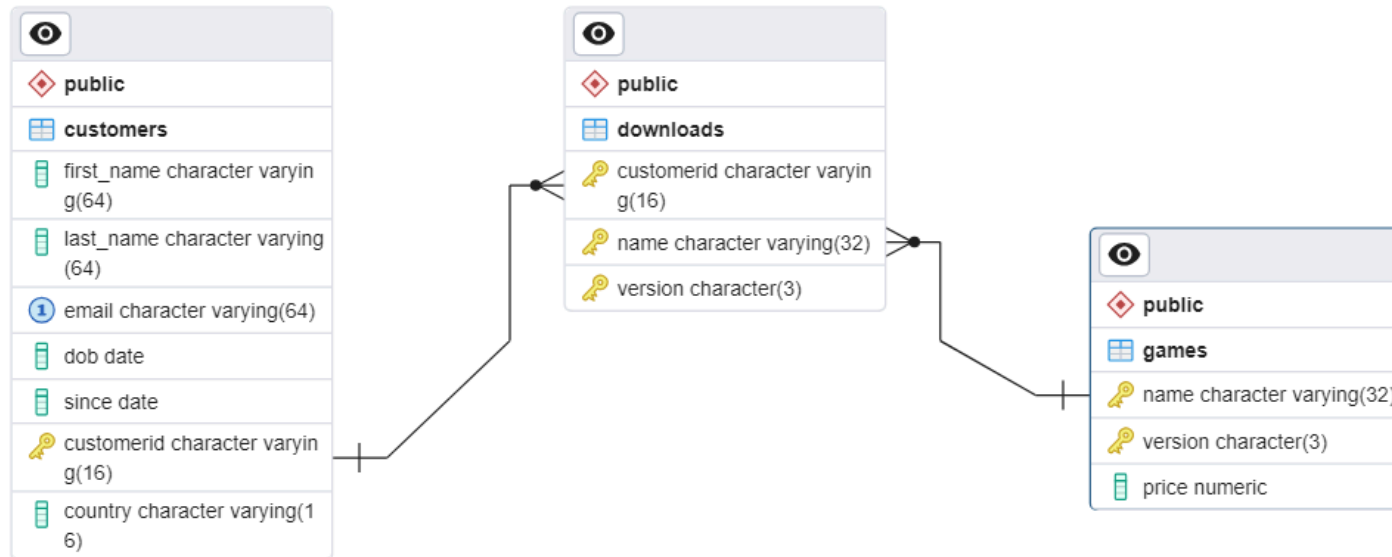
Multiple Table

Cross Product
► Primary/Foreign
Joining Tables
Meaningful Join
Understanding

Primary/Foreign Joining Tables

When to Join

We should join the tables according to their **primary** and **foreign keys** to make sense of the data. In particular, we add conditions that **referencing attributes** be **equal** to **referenced attributes** in the **WHERE** clause.



Note

This is a graphical representation of **logical schema** called **logical diagram** and not ER diagram.

Multiple Table

Cross Product
» Primary/Foreign
Joining Tables
Meaningful Join
Understanding

Primary/Foreign Meaningful Join

Conditional Join

To be more precise, we add the condition that **foreign key columns** are **equal** to the corresponding **primary key columns**. The condition below meaningfully corresponds to the "**customers who download a game**".

Code

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

Note

It is required from now on in this course to systematically define **table variables** for each table in the FROM clause and to use the **dot** notation to avoid ambiguity even when there is none.

Multiple Table

Cross Product
» Primary/Foreign
Joining Tables
Meaningful Join
Understanding

Primary/Foreign Meaningful Join

Conditional Join

To be more precise, we add the condition that **foreign key columns** are **equal** to the corresponding **primary key columns**. The condition below meaningfully corresponds to the "**customers who download a game**".

Code

```
SELECT *  
FROM customers, downloads, games  
WHERE customerid = customerid  
      AND name = name  
      AND version = version;
```

Error

Without **table variables**, the system cannot understand which column is which.

In this case, we have two tables containing the same attribute **name** and **version**.

Multiple Table

Cross Product
Primary/Foreign
» Understanding

Understanding Reading Query

Question

Consider the query below. What does this query find (*in "everyday" English*)?

```
SELECT c.email, g.version
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';
```



```
postgres=# exit
```

```
Press any key to continue . . .
```