Querying Data with SQL (DML)

Session 3

Database for Big Data Analytics

Siti Mariyah, Ph.D.
Poilteknik Statistika STIS

Email: sitimariyah@stis.ac.id
Github: https://github.com/diahnuri

Session 3 – Querying Data with SQL (DML)

- Goal: Be able to work with multiple tables and more complex queries
- Content:
  - SQL Data Manipulation Language (DML)
  - Insert, select, joins, subqueries
  - Learn group by and having
  - Inner joins between events, towers, subscribers
  - Subqueries and filter data
- Hands-on:
  - Query across multiple tables (e.g., subscribers x towers)
  - Compute ‘usual environment’ (night-time tower)
  - 3-day moving average of active subscribers
  - Rank subscribers by fraction of events outside home
  - Find Top-5 itinerant subscribers per regency

INSERT

- INSERT INTO TableName [(columnList)]

  VALUES (dataValueList)
- *columnList* is optional; if omitted, SQL assumes a list of all columns in their original CREATE TABLE order.
- If specified, any columns omitted must have been declared as NULL when table was created, unless DEFAULT was specified when creating column.
- *dataValueList* must match *columnList* as follows:
  - number of items in each list must be same;
  - must be direct correspondence in position of items in two lists;
  - data type of each item in dataValueList must be compatible with data type of corresponding column.

Example of INSERT…VALUES

- Insert a new subscriber

```
INSERT INTO subscribers (subscriber_id, home_regency, home_tower_id, sim_type)
VALUES (301, 'Denpasar', 12, 'prepaid');
```

- Insert a new tourism event

```
INSERT INTO tourism_events (event_id, event_name, regency, start_date, end_date)
VALUES (101, 'Ubud Jazz Festival', 'Gianyar', '2025-09-20', '2025-09-22');
```

Example INSERT using Defaults

- Order is not significant
- Insert a new subscriber

```
INSERT INTO subscribers
VALUES (301, 'Denpasar', 12, 'prepaid’);
```

- Suppose the cdr_events.event_ts column defaults to CURRENT_TIMESTAMP.

```
INSERT INTO cdr_events (event_id, subscriber_id, tower_id, event_type, event_ts)
VALUES (25001, 100, 12, 'sms', DEFAULT);
```

SELECT Statement

```
SELECT [DISTINCT | ALL]
   { * | [columnExpression [AS newName]] [,...] }
FROM TableName [alias] [, ...]
[WHERE     condition]
[GROUP BY       columnList]  [HAVING     condition]
[ORDER BYcolumnList]
```

SELECT Statement

- FROM: Specifies relation/table(s) to be used.
- WHERE: Filters tuple/rows.
- GROUP BY: Forms groups of rows with same column value.
- HAVING: Filters groups subject to some condition.
- SELECT: Specifies which columns are to appear in output.
- ORDER BY: Specifies the order of the output.
- Order of the clauses cannot be changed.
- Only SELECT and FROM are mandatory.

Example 1: All Columns, All Rows

- *List full details of all towers.*

```
SELECT tower_id, regency, latitude, longiture
FROM towers;
```

- Can use * as an abbreviation for ‘all columns’:

```
SELECT *
FROM towers;
```

Example 2: Specific Columns, All Rows

Produce a list of subscribers, showing only subscriber ID and home regency

```
SELECT subscriber_id, home_regency
FROM subscribers;
```

Use of DISTINCT

- Use DISTINCT to eliminate duplicates:

```
SELECT DISTINCT regency
FROM towers
```

```
SELECT DISTINCT home_tower_id
FROM subscribers
```

Calculated Fields

- Show event duration in hours (derived from timestamp)

```
SELECT event_id,
        subscriber_id,
        event_ts,
        EXTRACT(HOUR FROM event_ts) AS event_hour
        FROM cdr_events
        LIMIT 10;
```

Comparison Search Condition

- Select all events after a certain date (2025-09-20)

```
SELECT event_id, subscriber_id, event_ts, event_type
FROM cdr_events
WHERE event_ts > '2025-09-20';
```

- Select all subscribers ID less than 200

```
SELECT subscriber_id, home_regency, sim_type
FROM subscribers
WHERE subscriber_id > 200;
```

Compound Comparison Search Condition

- Select towers located in a specific regency

```
SELECT tower_id, regency, latitude, longitude
FROM towers
WHERE regency = 'Denpasar';
```

- Select all subscribers either in Badung or Depansar

```
SELECT *
FROM subscribers
WHERE regency = 'Badung' OR regency = 'Denpasar';
```

Range Search Condition

- Select events between two dates

```
SELECT event_id, subscriber_id, event_ts, event_type
FROM cdr_events
WHERE event_ts BETWEEN '2025-09-10' AND '2025-09-15';
```

- Select towers latitude range

```
SELECT tower_id, regency, latitude, longitude
FROM towers
WHERE latitude BETWEEN -8.5 AND -8.2;
```

Set Membership

- Filter towers in multiple regencies

```
SELECT tower_id, regency, latitude, longitude
FROM towers
WHERE regency IN ('Denpasar', 'Badung', 'Gianyar');
```

- Filter by event types

```
SELECT event_id, subscriber_id, event_type, event_ts
FROM cdr_events
WHERE event_type IN ('call', 'sms');
```

Pattern Matching

- Find subscribers with SIM type starting with "pre”

```
SELECT subscriber_id, home_regency, sim_type
FROM subscribers
WHERE sim_type LIKE 'pre%’;
```

- Find tourism events containing "Festival" in their name

```
SELECT event_id, event_name, regency, start_date, end_date
FROM tourism_events
WHERE event_name LIKE '%Festival%';
```

NULL Search Condition

- List all Subscribers without a home tower assigned

```
SELECT subscriber_id, home_regency, home_tower_id
FROM subscribers
WHERE home_tower_id IS NULL;
```

- Tourism events with no end date

```
SELECT event_id, event_name, regency, start_date, end_date
FROM tourism_events
WHERE end_date IS NULL;
```

Single Column Ordering

- Order towers by latitude (descending)

```
SELECT tower_id, regency, latitude, longitude
FROM towers
ORDER BY latitude DESC;
```

Multiple Columns Ordering

- Order subscribers by regency and SIM type

```
SELECT subscriber_id, home_regency, sim_type
FROM subscribers
ORDER BY home_regency ASC, sim_type ASC;
```

SELECT Statement – Aggregates

- ISO standard defines five aggregate functions:

- COUNT returns number of values in specified column.
- SUM returns sum of values in specified column.
- AVG returns average of values in specified column.
- MIN returns smallest value in specified column.
- MAX returns largest value in specified column.

SELECT Statement – Aggregates

- Each operates on a single column of a table and returns a single value.
- COUNT, MIN, and MAX apply to numeric and non-numeric fields, but SUM and AVG may be used on numeric fields only.
- Apart from COUNT(*), each function eliminates nulls first and operates only on remaining non-null values.
- COUNT(*) counts all rows of a table, regardless of whether nulls or duplicate values occur.
- Can use DISTINCT before column name to eliminate duplicates.
- DISTINCT has no effect with MIN/MAX, but may have with SUM/AVG.
- Aggregate functions can be used only in SELECT list and in HAVING clause; CANNOT be used in WHERE clause!

Use of COUNT(*)

- Count all events

```
SELECT COUNT(*) AS total_events
FROM cdr_events;
```

- Count unique subscribers

```
SELECT COUNT(DISTINCT subscriber_id) AS unique_subscribers
FROM cdr_events;
```

- Total and calls only

```
SELECT COUNT(*) AS total_events,
        SUM(CASE WHEN event_type = 'call' THEN 1 ELSE 0 END) AS total_calls
FROM cdr_events;
```

Use of MIN, MAX, AVG

- Select earliest event timestamp

```
SELECT MIN(event_ts) AS first_event
FROM cdr_events;
```

- Select earliest event timestamp

```
SELECT MAX(event_ts) AS first_event
FROM cdr_events;
```

SELECT Statement – Grouping

- Use GROUP BY clause to get sub-totals.
- SELECT and GROUP BY closely integrated: *each item in SELECT list must be single-valued per group, and SELECT clause may only contain*:
  - column names
  - aggregate functions
  - constants
  - expression involving combinations of the above.
- All column names in SELECT list must appear in GROUP BY clause unless name is used only in an aggregate function.
- If WHERE is used with GROUP BY, WHERE is applied first, then groups are formed from remaining rows satisfying predicate.
- ISO considers two nulls to be equal for purposes of GROUP BY.

Use of GROUP BY

- Count events per subscribers, groups all call/SMS/data events **by subscriber**, showing how many events each produced.

```
SELECT subscriber_id,
        COUNT(*) AS total_events
FROM cdr_events
GROUP BY subscriber_id
ORDER BY total_events DESC;
```

- Count towers per regency, Groups towers **by regency**, showing how many towers exist in each area.

```
SELECT regency,
        COUNT(*) AS total_towers
FROM towers
GROUP BY regency
ORDER BY total_towers DESC;
```

Use of GROUP BY HAVING

- Subscribers with more than 100 events, groups by subscriber, then keeps only those with **more than 100 events.**

```
SELECT subscriber_id,
        COUNT(*) AS total_events
FROM cdr_events
GROUP BY subscriber_id
HAVING COUNT(*) > 100
ORDER BY total_events DESC;
```

- Regencies with at least 5 towers, groups towers by regency, then filters for regencies that have **5 or more towers**

```
SELECT regency,
        COUNT(*) AS total_towers
FROM towers
GROUP BY regency
HAVING COUNT(*) >= 5
ORDER BY total_towers DESC;
```

Subqueries

- Some SQL statements can have a SELECT embedded within them.
- A subselect can be used in SELECT, WHERE and HAVING clauses of an outer SELECT, where it is called a subquery or nested query.
- Subselects may also appear in INSERT, UPDATE, and DELETE statements.

Subquery with Equality

- Find all subscribers who had at least **one event at the busiest tower**.
  - The **inner query** finds the tower_id with the most events.
  - The **outer query** lists all subscribers who used that tower.

```
SELECT subscriber_id, tower_id
FROM cdr_events
WHERE tower_id = (
    SELECT tower_id
    FROM cdr_events
    GROUP BY tower_id
    ORDER BY COUNT(*) DESC
    LIMIT 1
);
```

Subquery with Aggregate

- Find subscribers whose total events are **above the average** across all subscribers.
  - The **inner query** computes the average number of events per subscriber.
  - The **outer query** keeps only subscribers with totals **greater than that average**.

```
SELECT subscriber_id, COUNT(*) AS total_events
FROM cdr_events
GROUP BY subscriber_id
HAVING COUNT(*) > (
    SELECT AVG(event_count)
    FROM (
        SELECT subscriber_id, COUNT(*) AS event_count
        FROM cdr_events
        GROUP BY subscriber_id
    ) sub
);
```

Subquery Rules

- ORDER BY clause may not be used in a subquery (although it may be used in outermost SELECT).
- Subquery SELECT list must consist of a single column name or expression, except for subqueries that use EXISTS.
- By default, column names in a subquery refer to table name in FROM clause of the subquery. Can refer to a table in FROM using an alias.
- When subquery is an operand in a comparison, subquery must appear on right-hand side.
- The following is invalid

```
SELECT staffNo, fName, lName, position, salary
FROM Staff
WHERE
        (SELECT AVG(salary)
         FROM Staff) < salary;
```

Nested Query

- Find Subscribers with more events than the busiest tower's *average*

```
SELECT subscriber_id, COUNT(*) AS total_events
FROM cdr_events
GROUP BY subscriber_id
HAVING COUNT(*) > (
    SELECT AVG(event_count)
    FROM (
        SELECT tower_id, COUNT(*) AS event_count
        FROM cdr_events
        GROUP BY tower_id
        ORDER BY event_count DESC
    ) tower_counts
);
```

INSERT...SELECT

- Second form of INSERT allows multiple rows to be copied from one or more tables to another:

```
INSERT INTO TableName [ (columnList) ]
SELECT ...
```

INSERT...SELECT

- Insert frequent subscribers into a special table. Suppose we create a table to track **heavy users.** Inserts all subscribers with **more than 500 events** into heavy_users.

```
CREATE TABLE heavy_users (
 subscriber_id INT,
 total_events INT
);
INSERT INTO heavy_users (subscriber_id, total_events)
SELECT subscriber_id, COUNT(*) AS total_events
FROM cdr_events
GROUP BY subscriber_id
HAVING COUNT(*) > 500;
```

INSERT...SELECT

- Insert towers involved in tourism events. Suppose we create a table for **event_towers.** Now we can populate it with towers in regencies that host events. Inserts towers that belong to regencies where tourism events are scheduled.

```
CREATE TABLE event_towers (
 tower_id INT,
regency TEXT
);

INSERT INTO event_towers (tower_id, regency)
SELECT DISTINCT t.tower_id, t.regency
FROM towers t
JOIN tourism_events e ON t.regency = e.regency;
```

Multi-Table Queries

- Can use subqueries provided result columns come from same table.
- If result columns come from more than one table must use a join.
- To perform join, include more than one table in FROM clause.
- Use comma as separator and typically include WHERE clause to specify join column(s).
- Also possible to use an alias for a table named in FROM clause.
- Alias is separated from table name with a space.
- Alias can be used to qualify column names when there is ambiguity.

Simple Join

Join subscribers with their home tower info.

```
SELECT s.subscriber_id,  s.home_regency, t.tower_id,
        t.regency AS tower_regency,  t.latitude, t.longitude
FROM subscribers s, towers t
WHERE s.home_tower_id = t.tower_id;
```

This joins the **subscribers** table with the **towers** table using the home_tower_id. The result shows each subscriber along with the **location details of their home tower**.

Alternative JOIN Constructs

- SQL provides alternative ways to specify joins (if both using the same column name):
  - FROM towers t JOIN subscriber s ON t. tower_id = s.tower_id
  - FROM towers JOIN subscriber USING tower_id
  - FROM towers NATURAL JOIN subscriber
- In each case, FROM replaces original FROM and WHERE.

```
SELECT s.subscriber_id,  s.home_regency, t.tower_id,
        t.regency AS tower_regency,  t.latitude, t.longitude
FROM subscribers s JOIN towers t
ON s.home_tower_id = t.tower_id;
```

OUTER JOIN

- If one row of a joined table is unmatched, row is omitted from result table.
- Outer join operations retain rows that do not satisfy the join condition.

Example of OUTER JOIN (LEFT)

- Find all subscribers and their home tower info (even if missing)

```
SELECT s.subscriber_id,
       s.home_regency,
       s.home_tower_id,
       t.regency AS tower_regency,
       t.latitude,
       t.longitude
FROM subscribers s
LEFT JOIN towers t
  ON s.home_tower_id = t.tower_id;
```

This is a **LEFT OUTER JOIN**.
It returns **all subscribers**.
If a subscriber's home_tower_id doesn't exist in the towers table, the tower columns
(tower_regency, latitude, longitude) will show **NULL**.

Example of OUTER JOIN (LEFT)

- List all towers and the subscribers linked to them (even empty towers)

```
SELECT t.tower_id,
       t.regency,
       s.subscriber_id
FROM towers t
LEFT JOIN subscribers s
  ON t.tower_id = s.home_tower_id;
```

This shows all towers, even those with **no subscribers assigned** (the subscriber_id will be NULL).

Example of OUTER JOIN (FULL)

- Write a query to return all subscribers **and** all towers, even if they don't match.

```
SELECT s.subscriber_id,
       s.home_regency,
       s.home_tower_id,
       t.tower_id,
       t.regency AS tower_regency
FROM subscribers s
FULL OUTER JOIN towers t
  ON s.home_tower_id = t.tower_id;
```

A **FULL OUTER JOIN** returns:
Subscribers without a matching tower (tower_id will be NULL).
Towers without a matching subscriber (subscriber_id will be NULL).
And all normal matches.

Window Function

- A **window function** is a special type of SQL function that performs a calculation **across a set of rows related to the current row**, without collapsing them into one row like GROUP BY does.
  - It gives you **extra information per row** (e.g., rank, running total, moving average).
  - The “window” is defined using the OVER (...) clause.
- **Key Differences from Aggregates**
  - **Aggregate (GROUP BY)** → reduces many rows into **one row per group**.
  - **Window function (OVER (...))** → keeps **all rows**, but adds a calculated column.
- Window functions are super useful for analytics — they let you calculate things **across rows without collapsing them**(unlike GROUP BY)
- A window function is like giving each row a “view” of its neighbours, so we can calculate across them without losing detail.

Example of Window Function

- Aggregate with GROUP BY

```
SELECT subscriber_id, COUNT(*) AS total_events
FROM cdr_events
GROUP BY subscriber_id;
```

→ One row per subscriber

- Window Function version

```
SELECT subscriber_id,
       event_id,
       COUNT(*) OVER (PARTITION BY subscriber_id) AS total_events
FROM cdr_events;
```

Example of Window Function

- Rank subscribers by activity within each regency

```
SELECT s.subscriber_id,
       s.home_regency,
       COUNT(c.event_id) AS total_events,
       RANK() OVER (PARTITION BY s.home_regency ORDER BY COUNT(c.event_id) DESC) AS regency_rank
FROM subscribers s
JOIN cdr_events c ON s.subscriber_id = c.subscriber_id
GROUP BY s.subscriber_id, s.home_regency;
```

- COUNT(c.event_id) → total events per subscriber.
- RANK() OVER (PARTITION BY s.home_regency ORDER BY COUNT(*) DESC) → assigns a **rank inside each regency**(1 = most active subscriber in that regency).

Window Function

Original Table (before window function)

event_id	subscriber_id	event_type	event_ts
1	101	call	2025-09-01 09:10
2	101	sms	2025-09-01 09:20
3	102	data	2025-09-01 10:05
4	101	call	2025-09-01 11:00
5	102	call	2025-09-01 11:10

Result (after window function)

event_id	subscriber_id	event_type	total_events
1	101	call	3
2	101	sms	3
4	101	call	3
3	102	data	2
5	102	call	2

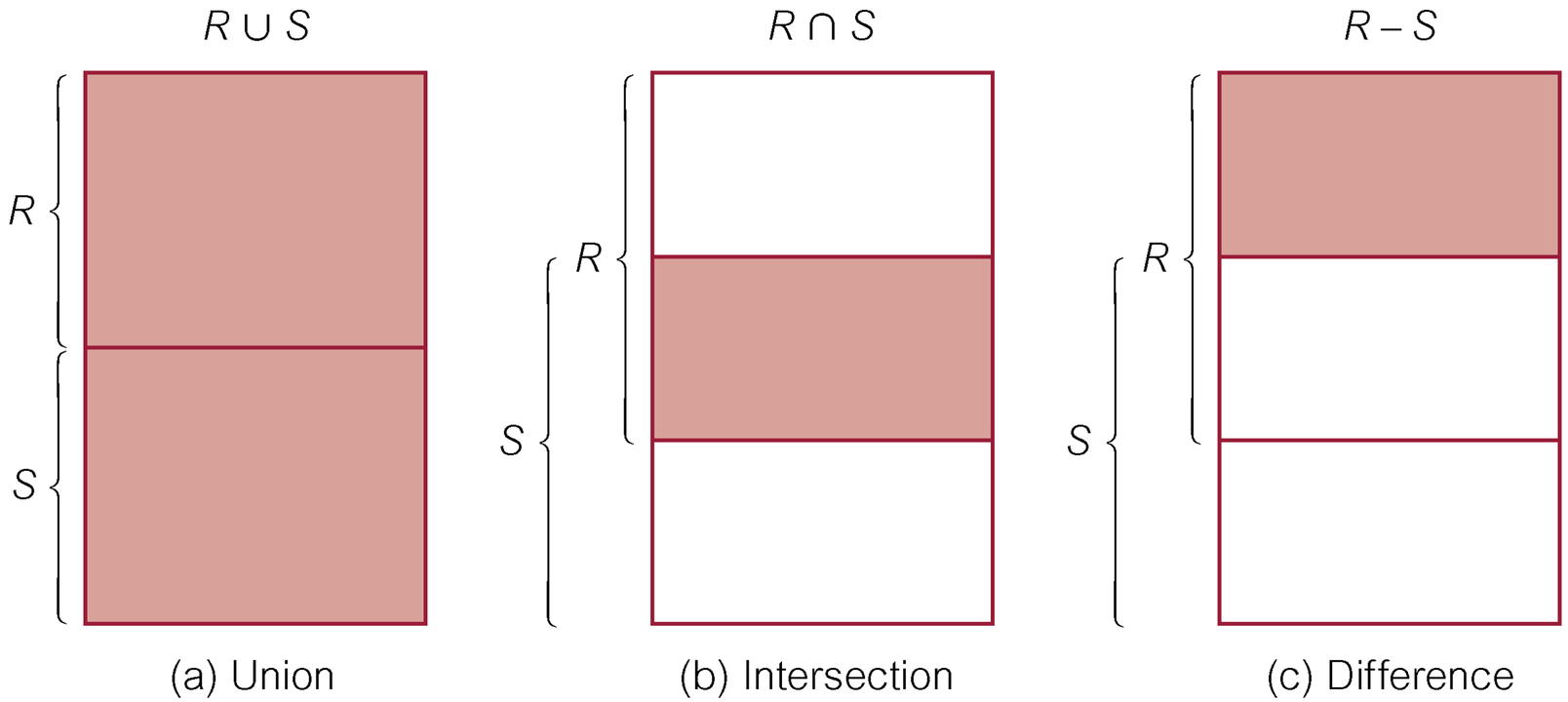
Union, Intersect, and Difference (Except)

- Can use normal set operations of Union, Intersection, and Difference to combine results of two or more queries into a single result table.
  - ***Union*** of two tables, A and B, is table containing all rows in either A or B or both.
  - ***Intersection*** is table containing all rows common to both A and B.
  - ***Difference*** is table containing all rows in A but not in B.
- Two tables must be *union compatible*; they must have the same structure
  - Containing the same number of columns and
  - Their corresponding columns have the same data type and length

Union, Intersect, and Difference (Except)

- Format of set operator clause in each case is:
  - op [ALL] [CORRESPONDING [BY {column1 [, ...]}]]
- If CORRESPONDING BY is specified, set operation performed on the named column(s).
- If CORRESPONDING specified but not BY clause, operation performed on common columns.
- If ALL specified, result can include duplicate rows.

Union, Intersect, and Difference (Except)



(a) Union (b) Intersection (c) Difference

Use of UNION

- Find all regencies that appear either in towers or in tourism_events.

```
SELECT regency
FROM towers
UNION
SELECT regency
FROM tourism_events;
```

Use of INTERSECT

- Find regencies that **have towers and also host tourism events.**

```
SELECT regency
FROM towers
INTERSECT
SELECT regency
FROM tourism_events;
```

Use of EXCEPT

- find regencies that **have towers but no tourism events.**

```
SELECT regency
FROM towers
EXCEPT
SELECT regency
FROM tourism_events;
```

UPDATE

```
UPDATE TableName
SET columnName1 = dataValue1
        [, columnName2 = dataValue2...]
[WHERE searchCondition]
```

- TableName can be name of a base table or an updatable view.
- SET clause specifies names of one or more columns that are to be updated.
- WHERE clause is optional:
  - if omitted, named columns are updated for all rows in table;
  - if specified, only those rows that satisfy searchCondition are updated.
- New dataValue(s) must be compatible with data type for corresponding column.

UPDATE All Rows and Specific Rows

- Suppose we want to standardize all subscribers' SIM types to lowercase.

```
UPDATE subscribers
SET sim_type = LOWER(sim_type);
```

- Suppose we want to update the regency of towers with missing regency info (NULL).

```
UPDATE towers
SET regency = 'Unknown'
WHERE regency IS NULL;
```

- Suppose some subscribers in **Denpasar** are incorrectly marked as "postpaid" and should be "prepaid".

```
UPDATE subscribers
SET sim_type = 'prepaid'
WHERE home_regency = 'Denpasar' AND sim_type = 'postpaid';
```

DELETE

```
DELETE FROM TableName
[WHERE searchCondition]
```

- TableName can be name of a base table or an updatable view.
- searchCondition is optional; if omitted, all rows are deleted from table. This does not delete table.
- If search_condition is specified, only those rows that satisfy condition are deleted.

Use of DELETE

- Suppose we want to delete **tourism events** that already ended before September 2025.

```
DELETE FROM tourism_events
WHERE end_date < '2025-09-01';
```

Suppose we want to clear the **cdr_events** table but keep its structure.

```
DELETE FROM cdr_events;
```

Delete subscribers who never generated any events

```
DELETE FROM subscribers
WHERE subscriber_id NOT IN (
    SELECT DISTINCT subscriber_id
    FROM cdr_events
);
```

Exercise

1. List all subscribers with subscriber_id > 100.
2. Show all towers in the regency **Denpasar**.
3. Find all cdr_events that happened **after 20 Sept 2025**.
4. Get all events where the event_type is not data.
5. List all subscribers whose subscriber_id is between **50 and 80**.
6. Find all tourism events that **ended after 2025-09-20**.
7. Find all subscribers active between Sept 12–14 in Gianyar

Answers

```
SELECT subscriber_id, home_regency,
sim_type
FROM subscribers
WHERE subscriber_id > 100;
```

```
SELECT tower_id, regency
FROM towers
WHERE regency = 'Denpasar';
```

```
SELECT event_id, subscriber_id, event_ts
FROM cdr_events
WHERE event_ts > '2025-09-20';
```

```
SELECT event_id, subscriber_id, event_type
FROM cdr_events
WHERE event_type <> 'data';
```

```
SELECT subscriber_id, home_regency
FROM subscribers
WHERE subscriber_id BETWEEN 50 AND 80;
```

```
SELECT event_id, event_name, end_date
FROM tourism_events
WHERE end_date > '2025-09-20';
```

Exercise

1. list all **subscribers** whose home_regency is **either Badung, Denpasar, or Gianyar**.
2. List all **towers** that are **not located in Denpasar or Badung**.

Answers

```
SELECT subscriber_id, home_regency, sim_type
FROM subscribers
WHERE home_regency IN ('Badung', 'Denpasar', 'Gianyar');
```

```
SELECT tower_id, regency, latitude, longitude
FROM towers
WHERE regency NOT IN ('Denpasar', 'Badung');
```

Thank You

Email: sitimariyah@stis.ac.id
Github: https://github.com/diahnuri