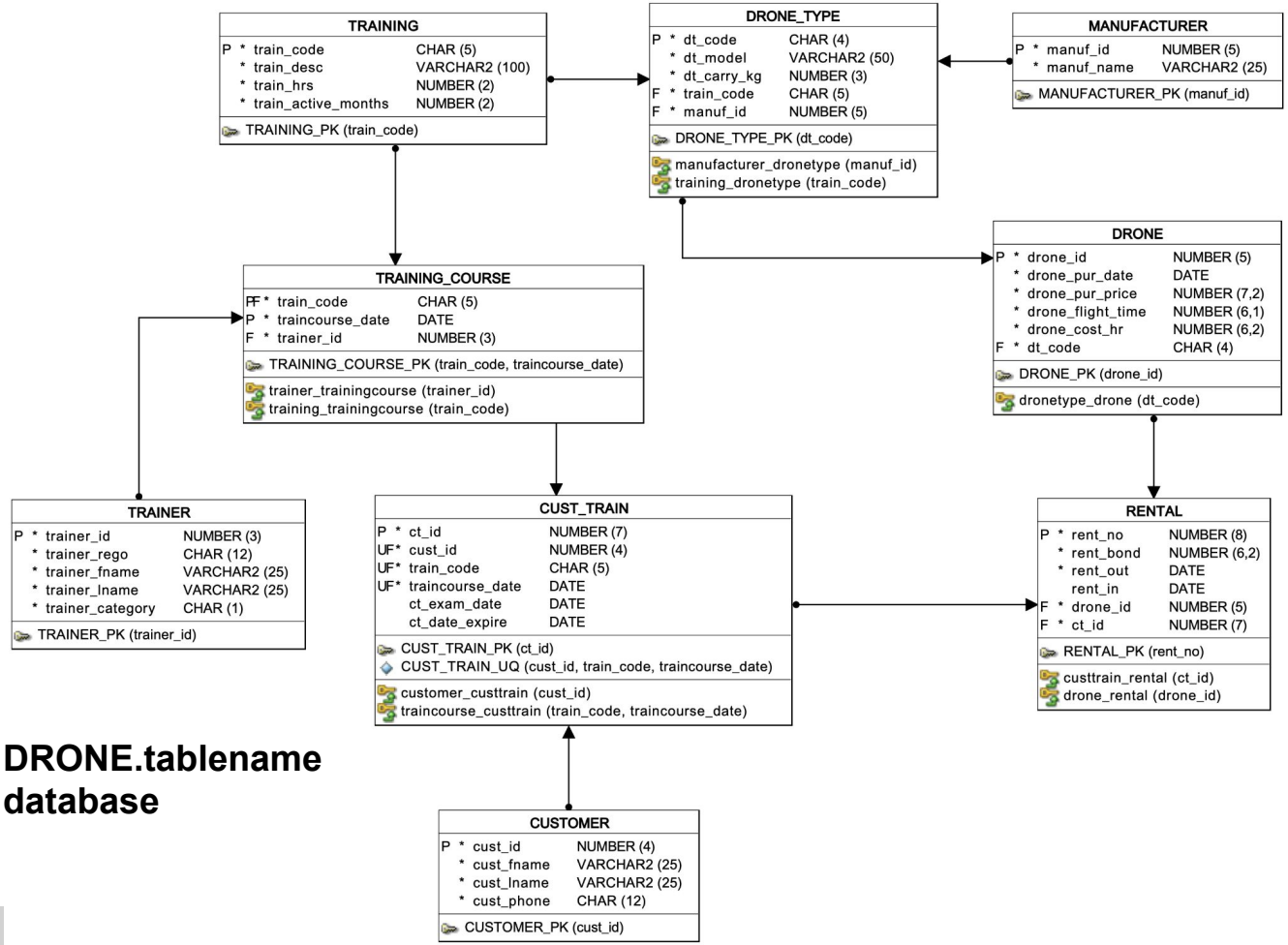
MONASH University

MONASH
INFORMATION
TECHNOLOGY

SQL Intermediate

GROUP OF EIGHT AUSTRALIA



Access tables via DRONE.tablename in Monash Oracle database

Aggregate Functions

- COUNT, MAX, MIN, SUM, AVG
- Example:

```
SELECT
   MAX(drone_flight_time)
FROM
   drone.drone;
```

```
SELECT
   MIN(drone_flight_time)
FROM
   drone.drone;
```

```
SELECT
   AVG(drone_flight_time)
FROM
   drone.drone;
```

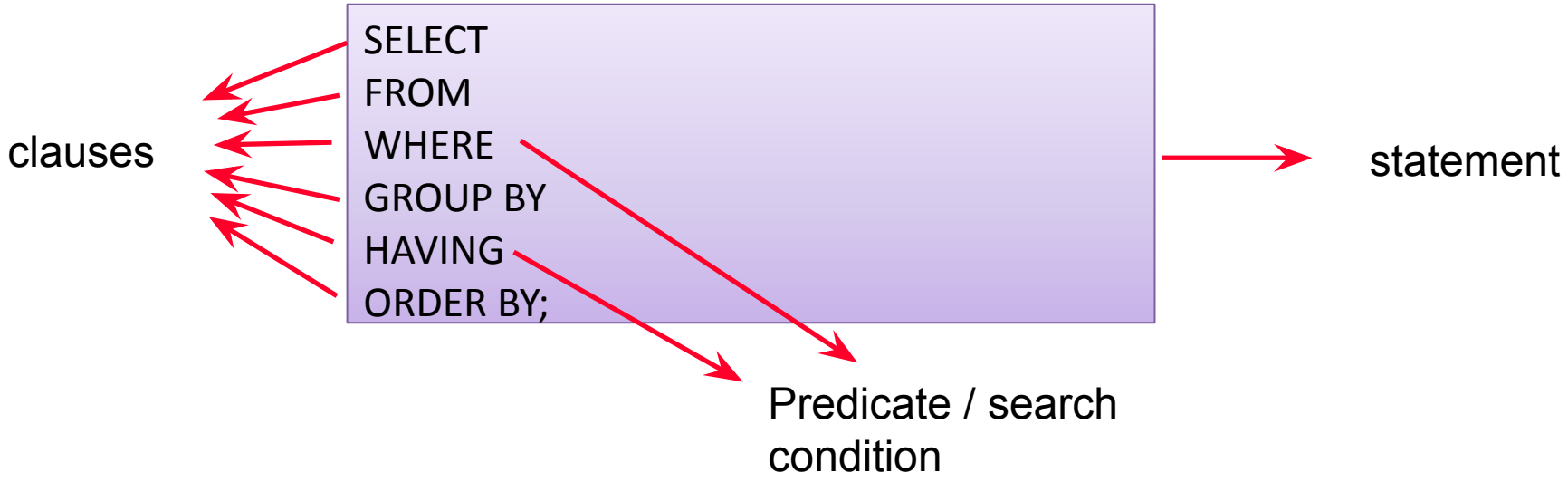
```
SELECT COUNT(*)
FROM drone.drone
WHERE drone_flight_time > 100;
```

count(*) and count(column_name)

```
SQL> SELECT count(*),
  2       count(rent_in),
  3       count(rent_out)
  4  FROM DRONE.rental;

  COUNT(*) COUNT(RENT_IN) COUNT(RENT_OUT)
---------- -------------- ---------------
        25             22              25
```

	RENT_NO	RENT_BOND	RENT_OUT	RENT_IN	DRONE_ID	CT_ID
1	1	100	20/FEB/20	20/FEB/20	100	1
2	2	100	21/FEB/20	22/FEB/20	101	2
3	3	100	22/FEB/20	23/FEB/20	102	3
4	4	100	22/FEB/20	25/FEB/20	100	4
5	5	100	25/FEB/20	25/FEB/20	101	5
6	6	200	28/FEB/20	28/MAR/20	103	6
7	7	200	01/MAR/20	02/MAR/20	103	7
8	8	200	03/MAR/20	04/MAR/20	103	8
9	9	200	06/MAR/20	10/MAR/20	103	9
10	10	100	10/MAR/20	18/MAR/20	101	1
11	11	150	26/APR/20	28/APR/20	111	10
12	12	150	26/APR/20	27/APR/20	112	11
13	13	150	28/APR/20	29/APR/20	113	12
14	14	150	28/APR/20	05/MAY/20	117	13
15	15	200	01/MAY/20	02/MAY/20	103	8
16	16	200	03/MAY/20	10/MAY/20	103	9
17	17	150	03/MAY/20	07/MAY/20	112	14
18	18	150	03/MAY/20	12/MAY/20	113	15
19	19	180	17/MAY/20	18/MAY/20	118	16
20	20	180	19/MAY/20	23/MAY/20	118	17
21	21	180	28/MAY/20	29/MAY/20	118	18
22	22	180	01/JUN/20	07/JUN/20	118	19
23	23	250	11/APR/21	(null)	119	20
24	24	150	12/APR/21	(null)	120	21
25	25	180	13/APR/21	(null)	118	18

Anatomy of an SQL Statement - Revisited



GROUP BY

- If a GROUP BY clause is used with aggregate function, the DBMS will apply the aggregate function to the different groups defined in the clause rather than all rows.

```
SELECT
 AVG(drone_flight_time)
FROM
   drone.drone;
```

```
SELECT dt_code, AVG(drone_flight_time)
FROM  drone.drone
GROUP BY  dt_code
ORDER BY dt_code;
```

```
SQL> SELECT
  2      AVG(drone_flight_time)
  3  FROM
  4      drone.drone;

AVG(DRONE_FLIGHT_TIME)
----------------------
                74.025
SQL>
SQL> SELECT
  2      dt_code,
  3      AVG(drone_flight_time)
  4  FROM
  5      drone.drone
  6  GROUP BY
  7      dt_code
  8  ORDER BY
  9      dt_code;

DT_C AVG(DRONE_FLIGHT_TIME)
---- ----------------------
DIN2             78.6666667
DMA2             53.3333333
DSPA                   45.5
PAPR                 97.625
SWPS                   56.3
```

	DRONE_ID	DRONE_PUR_DATE	DRONE_PUR_PRICE	DRONE_FLIGHT_TIME	DRONE_COST_HR	DT_CODE
1	100	13/JAN/20	1494	100	15	DMA2
2	101	13/JAN/20	1494	60	15	DMA2
3	102	13/JAN/20	872.44	45.5	9	DSPA
4	103	13/JAN/20	5300	200	55	DIN2
5	111	20/MAR/20	4200	100	45	PAPR
6	112	20/MAR/20	4200	40	45	PAPR
7	113	20/MAR/20	4200	150	45	PAPR
8	117	20/MAR/20	4200	100.5	45	PAPR
9	118	01/APR/20	1599	56.3	16	SWPS
10	119	01/APR/21	5600.8	10.2	60	DIN2
11	120	01/APR/21	5600.8	25.8	60	DIN2
12	121	17/APR/21	1610	0	16	DMA2

Q1. List all customer ids and total number of courses taken by each customer:

A. select cust_id, count(*) as no_of_courses_taken
from drone.cust_train
order by cust_id;

B. select cust_id, sum(train_code) as no_of_courses_taken
from drone.cust_train
group by cust_id
order by cust_id;

C. select cust_id, count(*) as no_of_courses_taken
from drone.cust_train
group by cust_id
order by cust_id;

D. None of the above

What output is produced?

```
SELECT count(*)
FROM drone.cust_train;

SELECT cust_id, COUNT(*) AS no_courses_taken
FROM drone.cust_train
GROUP BY cust_id
ORDER BY cust_id;

SELECT AVG(COUNT(*))
      AS average_no_courses_taken
FROM drone.cust_train
GROUP BY cust_id;
```

	CT_ID	CUST_ID	TRAIN_CODE	TRAINCOURSE_DATE
1	1	1	DJIHY	14/FEB/20
2	2	2	DJIHY	14/FEB/20
3	3	3	DJIHY	14/FEB/20
4	4	4	DJIHY	14/FEB/20
5	5	5	DJIHY	14/FEB/20
6	20	5	DJIPR	10/APR/21
7	6	6	DJIPR	18/FEB/20
8	21	6	DJIPR	10/APR/21
9	7	7	DJIPR	18/FEB/20
10	8	8	DJIPR	18/FEB/20
11	9	9	DJIPR	18/FEB/20
12	22	9	DJIPR	10/APR/21
13	13	9	PARPO	25/APR/20
14	19	9	SWELL	10/MAY/20
15	10	10	PARPO	25/APR/20
16	11	11	PARPO	25/APR/20
17	12	12	PARPO	25/APR/20
18	14	14	PARPO	25/APR/20
19	15	15	PARPO	25/APR/20
20	16	16	SWELL	10/MAY/20
21	17	17	SWELL	10/MAY/20
22	18	18	SWELL	10/MAY/20

```
SQL> SELECT count(*)
  2  FROM drone.cust_train;

  COUNT(*)
----------
        22
```

```
SQL> SELECT cust_id, COUNT(*) AS
no_courses_taken
  2  FROM drone.cust_train
  3  GROUP BY cust_id
  4  ORDER BY cust_id;

   CUST_ID NO_COURSES_TAKEN
---------- ----------------
         1                1
         2                1
         3                1
         4                1
         5                2
         6                2
         7                1
         8                1
         9                4
        10                1
        11                1
        12                1
        14                1
        15                1
        16                1
        17                1
        18                1

17 rows selected.
```

```
SQL> SELECT AVG(COUNT(*))
  2      AS average_no_courses_taken
  3  FROM drone.cust_train
  4  GROUP BY cust_id;

AVERAGE_NO_COURSES_TAKEN
------------------------
               1.29411765
```

Q2. List all customer ids, training course code and number of times each customer has taken a specific course:

A. select cust_id, train_code, count(*) as no_of_courses_taken
from drone.cust_train
group by cust_id
order by cust_id;

B. select cust_id, train_code, count(*) as no_of_courses_taken
from drone.cust_train
group by cust_id, train_code
order by cust_id, train_code;

C. select cust_id, count(*) as no_of_courses_taken
from drone.cust_train
group by train_code
order by train_code;

D. None of the above

What output is produced?

```
SELECT cust_id, train_code, count(*)
      as no_of_courses_taken
FROM drone.cust_train
GROUP BY cust_id, train_code
ORDER BY cust_id, train_code;
```

	CT_ID	CUST_ID	TRAIN_CODE	TRAINCOURSE_DATE
1	1	1	DJIHY	14/FEB/20
2	2	2	DJIHY	14/FEB/20
3	3	3	DJIHY	14/FEB/20
4	4	4	DJIHY	14/FEB/20
5	5	5	DJIHY	14/FEB/20
6	20	5	DJIPR	10/APR/21
7	6	6	DJIPR	18/FEB/20
8	21	6	DJIPR	10/APR/21
9	7	7	DJIPR	18/FEB/20
10	8	8	DJIPR	18/FEB/20
11	9	9	DJIPR	18/FEB/20
12	22	9	DJIPR	10/APR/21
13	13	9	PARPO	25/APR/20
14	19	9	SWELL	10/MAY/20
15	10	10	PARPO	25/APR/20
16	11	11	PARPO	25/APR/20
17	12	12	PARPO	25/APR/20
18	14	14	PARPO	25/APR/20
19	15	15	PARPO	25/APR/20
20	16	16	SWELL	10/MAY/20
21	17	17	SWELL	10/MAY/20
22	18	18	SWELL	10/MAY/20

```
SQL> SELECT cust_id, train_code, count(*) as no_of_courses_taken
  2  FROM drone.cust_train
  3  GROUP BY cust_id, train_code
  4  ORDER BY cust_id, train_code;

   CUST_ID TRAIN NO_OF_COURSES_TAKEN
---------- ----- -------------------
         1 DJIHY                   1
         2 DJIHY                   1
         3 DJIHY                   1
         4 DJIHY                   1
         5 DJIHY                   1
         5 DJIPR                   1
         6 DJIPR                   2
         7 DJIPR                   1
         8 DJIPR                   1
         9 DJIPR                   2
         9 PARPO                   1
         9 SWELL                   1
        10 PARPO                   1
        11 PARPO                   1
        12 PARPO                   1
        14 PARPO                   1
        15 PARPO                   1
        16 SWELL                   1
        17 SWELL                   1
        18 SWELL                   1

20 rows selected.
```

What output is produced?

```
SELECT cust_id,
       to_char(traincourse_date, 'yyyy') as year,
       count(*) as no_of_courses_taken
FROM drone.cust_train
GROUP BY cust_id, to_char(traincourse_date,
'yyyy')
ORDER BY cust_id, year;
```

Note: column alias cannot be used in group by clause

	CT_ID	CUST_ID	TRAIN_CODE	TO_CHAR(TRAINCOURSE_DATE,'YYYY')
1	1	1	DJIHY	2020
2	2	2	DJIHY	2020
3	3	3	DJIHY	2020
4	4	4	DJIHY	2020
5	5	5	DJIHY	2020
6	6	6	DJIPR	2020
7	7	7	DJIPR	2020
8	8	8	DJIPR	2020
9	9	9	DJIPR	2020
10	19	9	SWELL	2020
11	13	9	PARPO	2020
12	10	10	PARPO	2020
13	11	11	PARPO	2020
14	12	12	PARPO	2020
15	14	14	PARPO	2020
16	15	15	PARPO	2020
17	16	16	SWELL	2020
18	17	17	SWELL	2020
19	18	18	SWELL	2020
20	20	5	DJIPR	2021
21	21	6	DJIPR	2021
22	22	9	DJIPR	2021

```
SQL> SELECT cust_id, to_char(traincourse_date, 'yyyy') as year, count(*) as no_of_courses_taken
  2  FROM drone.cust_train
  3  GROUP BY cust_id, to_char(traincourse_date, 'yyyy')
  4  ORDER BY cust_id, year;

   CUST_ID YEAR NO_OF_COURSES_TAKEN
---------- ---- -------------------
         1 2020                   1
         2 2020                   1
         3 2020                   1
         4 2020                   1
         5 2020                   1
         5 2021                   1
         6 2020                   1
         6 2021                   1
         7 2020                   1
         8 2020                   1
         9 2020                   3
         9 2021                   1
        10 2020                   1
        11 2020                   1
        12 2020                   1
        14 2020                   1
        15 2020                   1
        16 2020                   1
        17 2020                   1
        18 2020                   1

20 rows selected.
```

Q3. Which rows that will be return by this select statement:

```
SELECT cust_id, train_code, count(*)
    as no_of_courses_taken
FROM drone.cust_train
GROUP BY cust_id, train_code
HAVING count(*) > 1
ORDER BY cust_id, train_code;
```

A. all rows
B. 7, 10
C. none of them
D. all rows except row 7 and 10

	CUST_ID	TRAIN_CODE	NO_OF_COURSES_TAKEN
1	1	DJIHY	1
2	2	DJIHY	1
3	3	DJIHY	1
4	4	DJIHY	1
5	5	DJIHY	1
6	5	DJIPR	1
7	6	DJIPR	2
8	7	DJIPR	1
9	8	DJIPR	1
10	9	DJIPR	2
11	9	PARPO	1
12	9	SWELL	1
13	10	PARPO	1
14	11	PARPO	1
15	12	PARPO	1
16	14	PARPO	1
17	15	PARPO	1
18	16	SWELL	1
19	17	SWELL	1
20	18	SWELL	1

HAVING clause

- It is used to put a condition or conditions on the groups defined by GROUP BY clause.

```
SELECT cust_id, train_code, count(*)
   as no_of_courses_taken
FROM drone.cust_train
GROUP BY cust_id, train_code
HAVING count(*) > 1
ORDER BY cust_id, train_code;
```

What output is produced?

```
SELECT cust_id, train_code, count(*) as no_of_courses_taken
FROM drone.cust_train
GROUP BY cust_id, train_code
HAVING count(*) > 1
ORDER BY cust_id, train_code;
```

```
SELECT dt_code, AVG(drone_flight_time) as average_drone_flight
FROM drone.drone
GROUP BY dt_code
HAVING AVG(drone_flight_time)>50
ORDER BY dt_code;
```

```
SQL> SELECT cust_id, train_code, count(*) as no_of_courses_taken
  2  FROM drone.cust_train
  3  GROUP BY cust_id, train_code
  4  HAVING count(*) > 1
  5  ORDER BY cust_id, train_code;

   CUST_ID TRAIN NO_OF_COURSES_TAKEN
---------- ----- -------------------
         6 DJIPR                   2
         9 DJIPR                   2

SQL> SELECT dt_code, AVG(drone_flight_time) as average_drone_flight
  2  FROM drone.drone
  3  GROUP BY dt_code
  4  HAVING AVG(drone_flight_time)>50
  5  ORDER BY dt_code;

DT_C AVERAGE_DRONE_FLIGHT
---- --------------------
DIN2           78.6666667
DMA2           53.3333333
PAPR               97.625
SWPS                 56.3
```

HAVING and WHERE clauses

```
SELECT dt_code, AVG(drone_flight_time) as average_drone_flight_time
FROM drone.drone
WHERE to_char(drone_pur_date,'yyyy') = '2020'
GROUP BY dt_code
HAVING AVG(drone_flight_time)>50
ORDER BY average_drone_flight_time desc;
```

- The WHERE clause is applied to ALL rows in the table.
- The HAVING clause is applied to the groups defined by the GROUP BY clause.
- The order of operations performed is FROM, WHERE, GROUP BY, HAVING and then ORDER BY.
- On the above example, the logic of the process will be:
  - All rows where drone purchase year = 2020 are retrieved. (due to the WHERE clause)
  - The retrieved rows then are grouped into different dt_code.
  - If the average flight time in a group is greater than 50, the dt_code and the average flight time is displayed. (due to the HAVING clause)

```
SQL> SELECT dt_code, AVG(drone_flight_time) as average_drone_flight_time
  2  FROM drone.drone
  3  WHERE to_char(drone_pur_date,'yyyy') = '2020'
  4  GROUP BY dt_code
  5  HAVING AVG(drone_flight_time)>50
  6  ORDER BY average_drone_flight_time desc;

DT_C AVERAGE_DRONE_FLIGHT_TIME
---- -------------------------
DIN2                       200
PAPR                    97.625
DMA2                        80
SWPS                      56.3
```

```
SELECT cust_id, train_code, count(*) as no_of_courses_taken
FROM drone.cust_train
GROUP BY cust_id
ORDER BY cust_id;
```

The above SQL generates error message

```
SQL Error: ORA-00979: not a GROUP BY expression
00979. 00000 -  "not a GROUP BY expression"
```

Why and how to fix this?

- Why? Because the grouping is based on the cust_id, whereas the display is based on cust_id and train_code. The two groups may not have the same members.
- How to fix this?
  - Include the train_code as part of the GROUP BY condition.
- Attributes that are used in the SELECT, HAVING and ORDER BY must be included in the GROUP BY clause.

Subqueries

Query within a query.

"Find all drones which flight time is higher than the average flight time of all drones"

```
SELECT *
FROM drone.drone
WHERE drone_flight_time >
    ( SELECT AVG(drone_flight_time)
      FROM drone.drone)
ORDER BY drone_id;
```

Types of Subqueries

Single-value

Multiple-row subquery (a list of values – many rows, one column)

Multiple-column subquery (many rows, many columns)

Comparison Operators for Subquery

- Operator for single value comparison.
  =, <, >
- Operator for multiple rows or a list comparison.
  - –equality
    - IN
  - –inequality
    - ALL, ANY combined with <, >

Summary

- Aggregate Functions
  - count, min, max, avg, sum
- GROUP BY and HAVING clauses.
- Subquery
  - Inner vs outer query
  - comparison operators (IN, ANY, ALL)

PART 2
PL/SQL - Triggers

Oracle Triggers

- A trigger is PL/SQL code associated with a table, which performs an action when a row in a table is inserted, updated, or deleted.
- Triggers are used to implement some types of data integrity constraints that cannot be enforced at the DBMS design and implementation levels
- A trigger is a stored procedure/code block associated with a table
- Triggers specify a condition and an action to be taken whenever that condition occurs
- The DBMS automatically executes the trigger when the condition is met ("fires")
- A Trigger can be ENABLE'd or DISABLE'd via the ALTER command
  - ALTER TRIGGER *trigger_name* ENABLE;

Oracle Triggers - general form

```
CREATE [OR REPLACE] TRIGGER <trigger_name>

        {BEFORE | AFTER | INSTEAD OF }

        {UPDATE | INSERT | DELETE}

            [OF <attribute_name>] ON <table_name>

        [FOR EACH ROW]

        [WHEN]

DECLARE

BEGIN
                …. trigger body goes here …..
END;
```

Triggering Statement

BEFORE|AFTER INSERT|UPDATE [of colname]|DELETE ON Table

- The triggering statement specifies:
  - the type of SQL statement that fires the trigger body.
  - the possible options include DELETE, INSERT, and UPDATE. One, two, or all three of these options can be included in the triggering statement specification.
  - the table associated with the trigger.
- Column List for UPDATE
  - if a triggering statement specifies UPDATE, *an optional list of columns can be included in the triggering statement*.
  - if you include a column list, the trigger is fired on an UPDATE statement only when one of the specified columns is updated.
  - if you omit a column list, the trigger is fired when any column of the associated table is updated

Trigger Body

```
BEGIN
    .....
END;
```

- is a PL/SQL block that can include SQL and PL/SQL statements. These statements are executed if the triggering statement is issued and the trigger restriction (if included) evaluates to TRUE.
- Within a trigger body of a row trigger, the PL/SQL code and SQL statements have access to the **old** and **new** column values of the current row affected by the triggering statement.
- Two correlation names exist for every column of the table being modified: **one for the old column value** and **one for the new column value**.

Correlation Names

- Oracle uses two correlation names in conjunction with every column value of the current row being affected by the triggering statement. These are denoted by:

  OLD.ColumnName & NEW.ColumnName

  - For DELETE, only OLD.ColumnName is meaningful
  - For INSERT, only NEW.ColumnName is meaningful
  - For UPDATE, both are meaningful
- A colon must precede the OLD and NEW qualifiers when they are used in a trigger's body, but a colon is not allowed when using the qualifiers in the WHEN clause.
- Old and new values are available in both BEFORE and AFTER **row triggers**.

FOR EACH ROW Option

- The FOR EACH ROW option determines whether the trigger is a row trigger or a statement trigger. If you specify FOR EACH ROW, the trigger fires once for each row of the table that is affected by the triggering statement. The absence of the FOR EACH ROW option means that the trigger fires only once for each applicable statement, but not separately for each row affected by the statement.

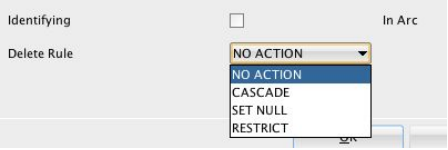
```
CREATE OR REPLACE TRIGGER display_salary_increase
AFTER UPDATE OF empmsal ON employee
FOR EACH ROW
WHEN (new.empmsal > 1000)
BEGIN
   DBMS_OUTPUT.PUT_LINE ('Employee:  '|| :new.empno ||' Old salary:  '||
    :old.empmsal || ' New salary:  '|| :new.empmsal);
END;
```

Statement Level Trigger

- Executed once for the whole table but will have to check all rows in the table.
- In many cases, it will be inefficient.
- **No access to the correlation values :new and :old**.

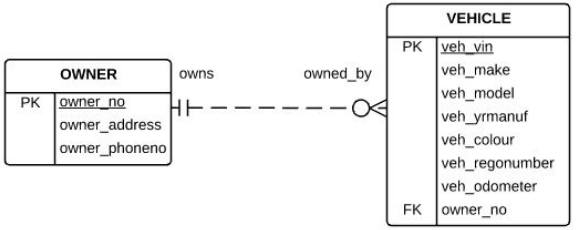
Oracle Data FK Integrity

- Oracle offers the options:
  - UPDATE
    - no action (the default - not specified)
  - DELETE
    - no action (the default - not specified)
    - cascade
    - set null



- Subtle difference between "no action" and "restrict"
  - RESTRICT - will not allow action if child records exist, checks first
  - NO ACTION - allows action and any associated triggers, *then* checks integrity
- Databases implementations vary, for example:
  - Oracle no RESTRICT
  - IBM DB2, SQLite implement both as above

Common use of triggers



- In the model above OWNER is the PARENT (PK end) and VEHICLE is the CHILD (FK end)
- What should the database do to maintain integrity if the user:
  - attempts to UPDATE the owner_no of the owner (parent)
  - attempts to DELETE an owner who still has vehicles in the vehicle table
- Oracle, by default, takes the safe approach
  - UPDATE NO ACTION (no update of PK permitted if child records)
  - DELETE NO ACTION (no delete permitted if child records)
  - what if you as the developer want UPDATE CASCADE?

Oracle Triggers

```
CREATE OR REPLACE TRIGGER Owner_Upd_Cas
BEFORE UPDATE OF owner_no ON owner
FOR EACH ROW
BEGIN
    UPDATE vehicle
    SET          owner_no = :new.owner_no
    WHERE  owner_no = :old.owner_no;
    DBMS_OUTPUT.PUT_LINE ('Corresponding owner number in the VEHICLE
  table has also been updated');
END;
/
```

Implement UPDATE CASCADE rule

OWNER 1 ---- has --- M VEHICLE

:new.owner_no – value of owner_no after update

:old.owner_no – value of owner_no before update

- SQL Window: To CREATE triggers, include the RUN command (/) after the last line of the file

Common use of triggers - data integrity

- A trigger can be used to enforce user-defined integrity by triggering on a preset condition, carrying out some kind of test and then if the test fails, the trigger can raise an error (and stop the action) via a call to **`raise_application_error`**

  The syntax for this call is:

  ```
  raise_application_error(-20000, 'Error message to display');
  ```

  the -20000 is the error number which is reported to the user, the error message is the error message the user will see. The error number can be any number less than or equal to -20000.

Common use of triggers - data integrity - example

For example: a trigger which will ensure any unit added (ie. inserted) to the UNIT table has a unit code which starts with 'FIT'. Test your trigger and ensure it works correctly and shows your error message.

```
CREATE OR REPLACE TRIGGER check_unit_code BEFORE
    INSERT ON unit
    FOR EACH ROW
BEGIN
    IF :new.unit_code NOT LIKE 'FIT%' THEN
        raise_application_error(-20000, 'Unit code must begin with FIT');
    END IF;
END;
/
-- Test Harness
-- display before value
select * from unit;

insert into unit values ('ABC0001','Test Insert',6);

-- display after value
select * from unit;
-- closes transaction
rollback;
```

Mutating Table

- A table that is currently being modified through an INSERT, DELETE or UPDATE statement SHOULD NOT be read from or written to because it is in a transition state between two stable states (before and after) where data integrity can be guaranteed.
  - Such a table is called **mutating table**.

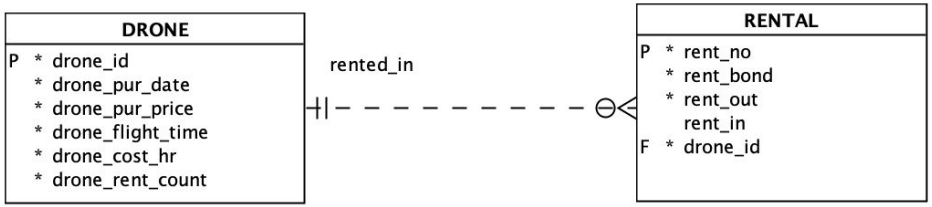
```
CREATE OR REPLACE TRIGGER Owner_Upd_Cas BEFORE
  UPDATE OF owner_no ON owner
  FOR EACH ROW

  DECLARE
    owner_count NUMBER;

  BEGIN

    SELECT COUNT(*) INTO owner_count
    FROM owner
    WHERE owner_no = :old.owner_no;

    IF owner_count = 1 THEN
      UPDATE vehicle
      SET owner_no = :NEW.owner_no
      WHERE owner_no = :OLD.owner_no;
      DBMS_OUTPUT.PUT_LINE ('Corresponding owner number in the VEHICLE table '
      || 'has also been updated');
    END IF;

  END;
```

```
update owner set owner_no = 1 where owner_no = 2
Error report -
SQL Error: ORA-04091: table LSMI1.OWNER is mutating, trigger/function may not see it
ORA-06512: at "LSMI1.OWNER_UPD_CAS", line 6
ORA-04088: error during execution of trigger 'LSMI1.OWNER_UPD_CAS'
04091. 00000 -  "table %s.%s is mutating, trigger/function may not see it"
*Cause:    A trigger (or a user defined plsql function that is referenced in
           this statement) attempted to look at (or modify) a table that was
           in the middle of being modified by the statement which fired it.
*Action:   Rewrite the trigger (or function) so it does not read that table.
```

Triggers Case Study

Case Study

Given an DRONE table and a RENTAL table with the structure:



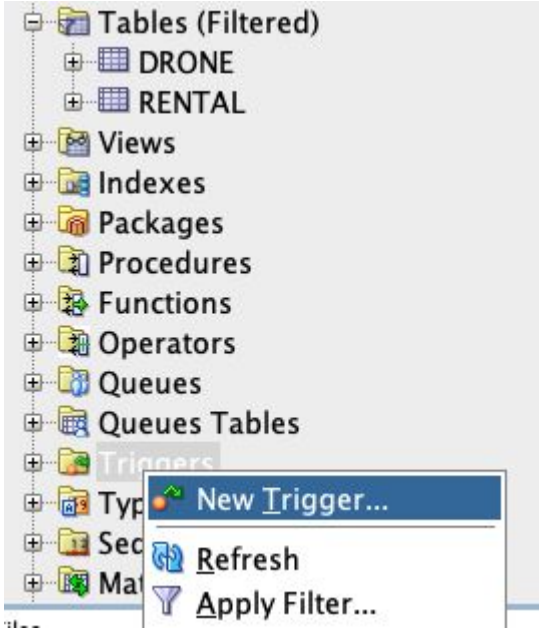
We will create a trigger, which automatically maintains this value:

- insert a rental increases the count for drone_rent_count
- delete a rental decreases the count for drone_rent_count

Using the file available from Moodle (drone_rental_schema.sql), create the tables, and create the trigger using the following steps.

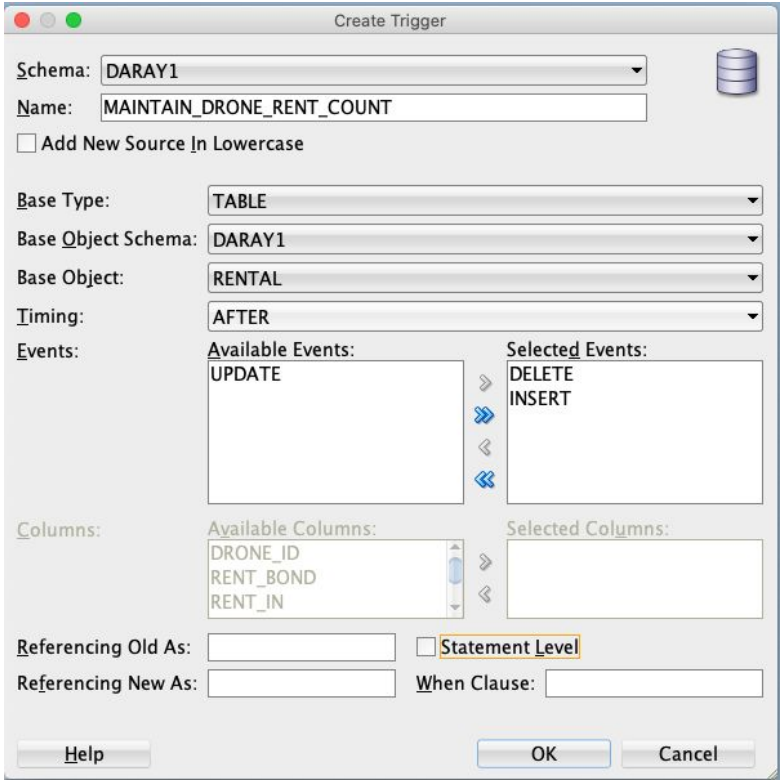
Create Trigger using GUI - step 1

Create a new trigger using the SQL Developer GUI to build the trigger framework:



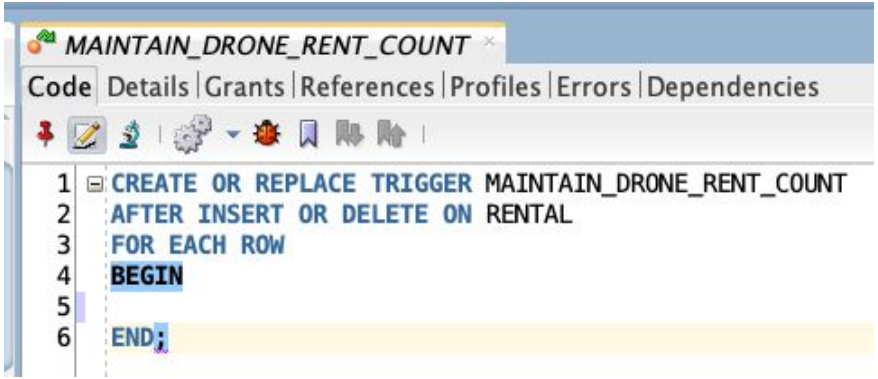
Create Trigger using GUI - step 2

Create a new trigger using the SQL Developer GUI to build the trigger framework:



Create Trigger using GUI - step 3

Select OK and you will be transferred to the PL/SQL Editor, where you can then enter the trigger BODY (the part between the begin and end). First remove the placeholder null;



Q4. What would be the logic to update the drone_rent_count attribute in the DRONE table when a new row is inserted to RENTAL?

A. UPDATE drone
SET drone_rent_count = drone_rent_count + 1
WHERE drone_id = :old.drone_id;

B. UPDATE drone
SET drone_rent_count = drone_rent_count + 1
WHERE drone_id = :new.drone_id;

C. UPDATE drone
SET drone_rent_count = drone_rent_count + 1;

D. UPDATE drone
SET drone_rent_count = drone_rent_count - 1
WHERE drone_id = :old.drone_id;

Q5. What would be the logic to update the drone_rent_count attribute in the DRONE table when a row is deleted to RENTAL?

A.
```
UPDATE drone
SET drone_rent_count = drone_rent_count + 1
WHERE drone_id = :old.drone_id;
```

B.
```
UPDATE drone
SET drone_rent_count = drone_rent_count + 1
WHERE drone_id = :new.drone_id;
```

C.
```
UPDATE drone
SET drone_rent_count = drone_rent_count - 1;
```

D.
```
UPDATE drone
SET drone_rent_count = drone_rent_count - 1
WHERE drone_id = :old.drone_id;
```

Create Trigger using GUI - step 4

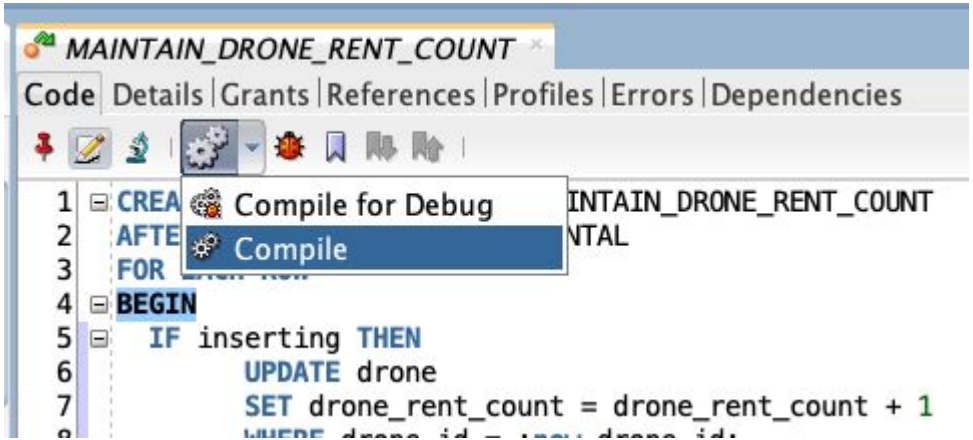
MAINTAIN_DRONE_RENT_COUNT

Code | Details | Grants | References | Profiles | Errors | Dependencies

```
CREATE OR REPLACE TRIGGER MAINTAIN_DRONE_RENT_COUNT
AFTER INSERT OR DELETE ON RENTAL
FOR EACH ROW
BEGIN
  IF inserting THEN
        UPDATE drone
        SET drone_rent_count = drone_rent_count + 1
        WHERE drone_id = :new.drone_id;
    END IF;

    IF deleting THEN
        UPDATE drone
        SET drone_rent_count = drone_rent_count - 1
        WHERE drone_id = :old.drone_id;
    END IF;
END;
```

Create Trigger using GUI - step 4

Finally compile the trigger, correcting any errors which appear:



Create Trigger using Script - Assignment 2B

An alternative way of adding a trigger is via a script or entering the code directly into the SQL worksheet window.

Note when using this approach after the end of the trigger (line 16, you must include a */* **on a line by itself in column 1** (line 17) **followed by a blank line** (line 18).

SQL Worksheet History

Worksheet Query Builder

```
1  CREATE OR REPLACE TRIGGER MAINTAIN_DRONE_RENT_COUNT
2  AFTER INSERT OR DELETE ON RENTAL
3  FOR EACH ROW
4  BEGIN
5    IF inserting THEN
6          UPDATE drone
7          SET drone_rent_count = drone_rent_count + 1
8          WHERE drone_id = :new.drone_id;
9      END IF;
10
11     IF deleting THEN
12         UPDATE drone
13         SET drone_rent_count = drone_rent_count - 1
14         WHERE drone_id = :old.drone_id;
15     END IF;
16 END;
17 /
18
```

Trigger Screen Output

- If you wish your trigger to provide output to the screen, you can include a message which will be written to the screen using DBMS_OUTPUT.PUT_LINE. You should note that **such messages are for your use as a developer**, in a production environment normal users will not see such messages. *We will also use these messages for you to demonstrate a trigger works for assessment purposes*.
- To obtain DBMS_OUTPUT in SQL Developer you need to run the command

```
SET SERVEROUTPUT ON
```

- A typical developer output message in our current example might be:

```
DBMS_OUTPUT.PUT_LINE('New rental has been added to drone ' || :new.drone_id);
```

  then when a new employee is inserted a message such as:

```
New rental has been added to drone 10
```

  would be output by DBMS Output.

Complete Trigger Command

SQL Worksheet History

Worksheet Query Builder

```
CREATE OR REPLACE TRIGGER MAINTAIN_DRONE_RENT_COUNT
AFTER INSERT OR DELETE ON RENTAL
FOR EACH ROW
BEGIN
  IF inserting THEN
        UPDATE drone
        SET drone_rent_count = drone_rent_count + 1
        WHERE drone_id = :new.drone_id;

        DBMS_OUTPUT.PUT_LINE('New rental has been added for drone '
        || :new.drone_id);

    END IF;

    IF deleting THEN
        UPDATE drone
        SET drone_rent_count = drone_rent_count - 1
        WHERE drone_id = :old.drone_id;

        DBMS_OUTPUT.PUT_LINE('A rental has been deleted for drone '
        || :old.drone_id);

    END IF;
END;
/
```

Testing Harness

- It is not sufficient to code a trigger only, a suitable test harness must be developed at the same time and used to ensure the trigger is working correctly.
- Now test the trigger that you have just created:
  - Insert a drone with a drone rent count of zero.
  - Insert an rental for that drone and observe what happens with the drone.drone_rent_count attribute.
  - Delete the rental data, again check the effect on the drone.drone_rent_count attribute.

Test the Trigger

```
--Testing Harness
--Initial data
SET SERVEROUTPUT ON;

insert into drone values (10,to_date('11/JAN/2020','dd/MON/yyyy'),1399,0,15,0);

select * from drone;
select * from rental;

--test for insert
insert into rental values (1, 100, to_date('30/JAN/2020','dd/MON/yyyy'), null, 10 );

select * from drone;
select * from rental;

--test for delete
delete from rental where rent_no = 1;

select * from drone;
select * from rental;

rollback;
--End of Testing Harness
```

Statement Level Trigger

```
create or replace
TRIGGER DELETE_STATEMENT
AFTER DELETE ON ENROLMENT
BEGIN
    INSERT INTO enrol_history VALUES (SYSDATE, USER, 'Deleted');
END;
```

Row Level Trigger

```
create or replace
TRIGGER DELETE_ENROLMENT
AFTER DELETE ON ENROLMENT
FOR EACH ROW
BEGIN
    INSERT INTO audit_trail VALUES
                  (SYSDATE, USER, :old.stu_nbr, :old.unit_code);
END;
```

Oracle Triggers

- Use triggers where:
  - a specific operation is performed, to ensure related actions are also performed
  - to enforce integrity where data has been denormalised
  - to maintain an audit trail
  - global operations should be performed, regardless of who performs the operation
  - they do NOT duplicate the functionality built into the DBMS
  - their size is reasonably small (< 50 - 60 lines of code)
- Do not create triggers where:
  - they are recursive
  - they modify or retrieve information from triggering tables