Applied 8 DML and Transaction Management

Applied 8 DML and Transaction Management

After completing this activity, you should be able to:

- analyse a specification in terms of the required change to the content of records in a database
- code SQL insert, update and delete statements to add or modify rows in a table
- use ORACLE's sequences to generate keys
- recognise the properties that database transactions should exhibit for proper database operation
- explain basic transaction serialisation through locking
- analyse and implement the concept of a database transaction

This activity supports unit learning outcomes 1 and 4.

A8-1 Transaction Management - Locks

Lock Exercise - Theory

Task 1

Given the following transaction sequence, complete the table by clearly indicating what locks are present at each of the indicated times (Time 0 to Time 12). Cell entries must have the form **S(Tn)** - for a shared lock by Tn, **X(Tn)** - for an exclusive lock by Tn, **Tn wait Tm** - for a wait of Tn due to Tm (where n and m are transaction numbers)

Time	Transaction	Activity	A	B	C	D
0	T1	READ A				
1	T2	READ B				
2	T3	READ C				
3	T1	READ D				
4	T4	READ A				
5	T4	READ B				
6	T4	UPDATE B				
7	T3	UPDATE C				
8	T2	READ A				
9	T1	UPDATE C				
10	T3	COMMIT				
11	T2	READ D				
12	T2	UPDATE D				

You can download the above table as a PDF by selecting Preview lesson (see the three dots top right) and then select Download PDF. You can then markup on this downloaded PDF. Alternatively you can use the Excel sheet below:

A8-1-Task1-LockExercise.xlsx

Task 2

Based on your answers in the above table, draw a wait for graph and determine if a deadlock is present

Lock Exercise - Practical

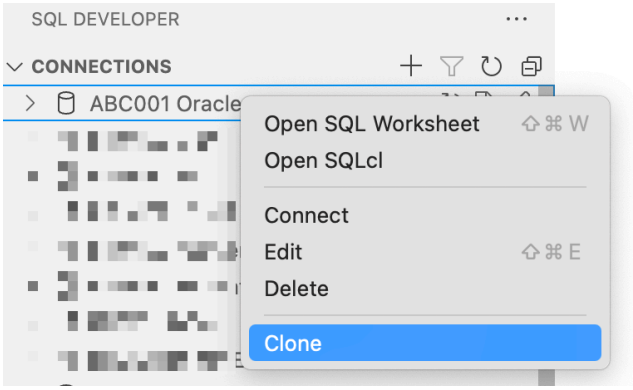
In these exercises, you will examine the issues involved in updating shared data. You will need two connections to complete this exercise.

Open your local repo and in the App08 folder create two blank .sql files (File - New text file and name as below)

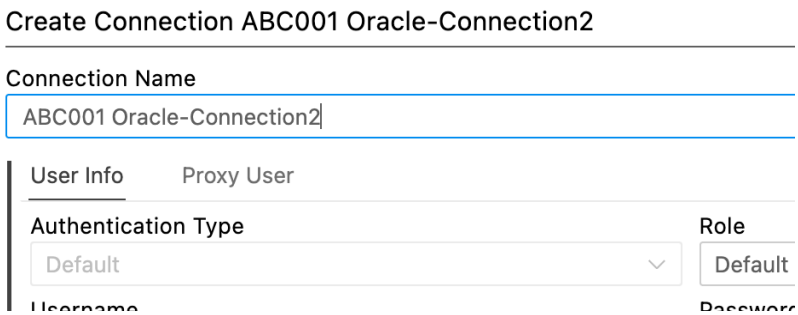
- connection-1.sql, and
- connection-2.sql

Task 1

Create a second connection to Oracle in SQL Developer by right-clicking on your current connection and selecting "Clone":

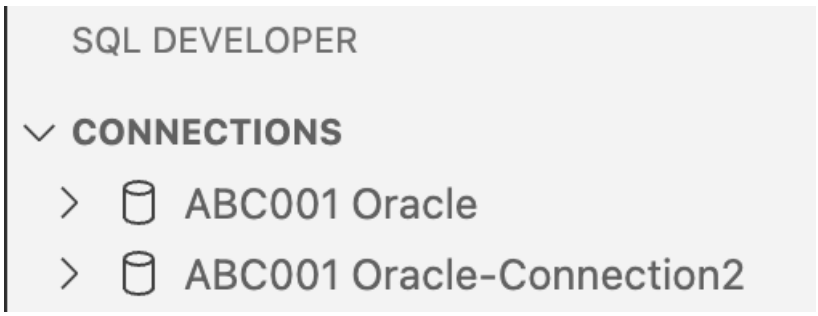


In the panel which appears give your new connection a name:



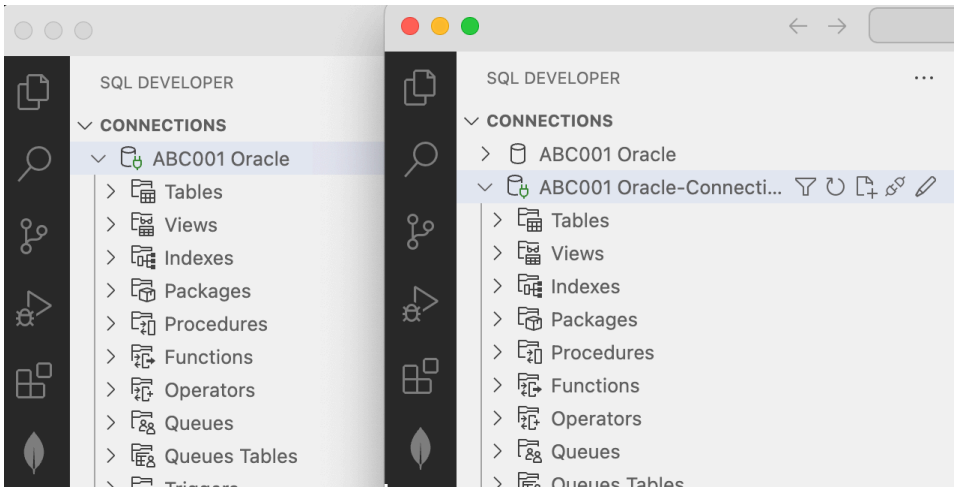
You may use whatever name you wish, provided you can recognise which is connection 2.

You will now have two Oracle Connections:



For the following work, Connection 1 refers to your original connection, and Connection 2 refers to

the new connection you just created. In Visual Studio Code, select File - New Window. You now have two windows open; in the original window, login with Connection 1, and in the new window, with Connection 2:



Open the file connection-1.sql in the first window and connection-2.sql in the connection 2 window (do not attempt to open the repo in the second widow; **if VSC prompts for this, select No**)

Task 2

Create the CUSTBALANCE table shown below using connection 1. The table will have 2 attributes, cust_id and cust_bal. Both attribute data types are NUMBER. Insert two rows of data so that the table appears as below (do not use commit):

	CUST_ID	CUST_BAL
1	1	100
2	2	200

Using connection 1 enter/run the SQL command

```
select * from CUSTBALANCE;
```

Using connection 2 enter/run the SQL command

```
select * from CUSTBALANCE;
```

What do you see and why?

What happens when you enter/run the SQL command in connection 1:

```
commit;
```

Check both connections now.

Task 3

Complete the following steps (maintain the order of the operations):

- Connection 1 updates the balance of customer 1 from 100 to 110 (without issuing a commit).
- Connection 1 run a select statement to see whether the value of customer 1 has been updated
- Connection 2 view the contents of the CUSTBALANCE table (do you see the new value? - if not, why not?)
- Connection 1 issue a commit command
- Connection 2 view the contents of the CUSTBALANCE table (do you notice any difference?)

Explain what is happening in the results of the above queries, in the context of atomic transactions.

Task 4

Complete the following steps, see what happens when two connections try concurrent updates of the table (keep the order of transactions the same as below):

- Connection 1 updates the balance of customer 2 from 200 to 150 (without issuing a commit).
- Connection 2 tries to update the balance of customer 2 to 100 (what happens?)

Explain what is happening here. What should be done to allow the Connection 2 update to proceed?

Task 5

Complete the following steps, see what happens when two connections try to update different rows in the same table (keep the order of transactions the same as below):

- Connection 1 updates the balance of customer 2 to 175 (without issuing a commit).
- Connection 2 tries to update the balance of customer 1 to 125 (what happens?)

How does this differ from the results of the transactions in Task 4? What does this tell you about the granularity of locking in Oracle? What must be done for the results of both updates to be visible to both users?

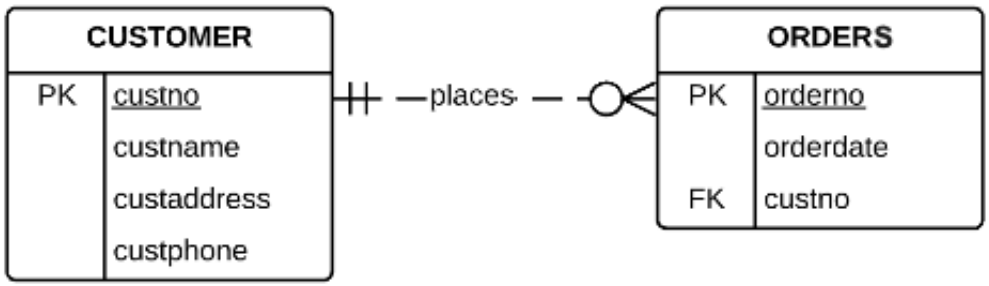
Task 6

Try to generate a deadlock between the two connections. Remember that a deadlock occurs when connection 1 holds a lock on row 1 and requests a lock on row 2, but row 2 is locked by connection 2, who is requesting a lock on row 1.

A8-2 Setting Transaction Boundaries

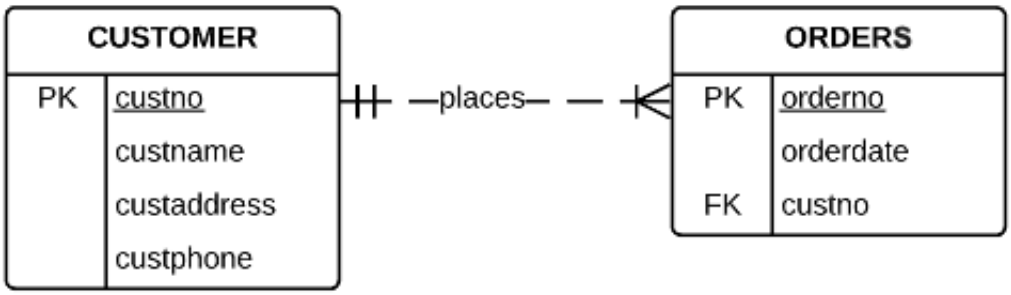
From this point forward, whenever you are working with insert, update or delete SQL commands, you must consider and define the boundaries of your transactions - ie where the transaction starts and where it ends.

In this model a customer can exist without any orders placed



When adding a new customer here, *the customer is not required to have placed an order, thus the transaction to add the new customer will contain a single insert to add the customer followed by a commit*.

Whereas in this model a customer must place at least one order:



Here when adding a new customer, *you are required to also add an order, thus the transaction must involve two inserts grouped together as a single transaction, with one commit after both inserts*.

The situation dictated by the model may be overridden where your client requests that certain actions should be grouped together in a particular situation.

A8-3 Inactive Oracle Sessions

Two important points to observe when working with insert, update and delete SQL DML commands:

- you **must not delay commit or rollback** - execute the commit/rollback command immediately after you have carried out the necessary data modification within your transaction, and
- **ensure at the end of your session you disconnect from Oracle** so as to ensure no locks are left active.

Under some circumstances, e.g., due to VPN disconnection, an Oracle session can remain connected after you have completed DML operations, sometimes with locks in place. These locks will prevent you from being able to continue to work normally with your tables.

A common trigger for this situation is where a user on a laptop drops the VPN before they disconnect from the database. In such a situation when the user reconnects to the database, they may find that they cannot complete any useful work because of the locks still in place a typical message under such a scenario is "RESOURCE BUSY AND ACQUIRE WITH NOWAIT SPECIFIED" or "The current session is failing to acquire the specified lock on the specified resource because another session holds the lock".

To solve this situation the Monash Oracle database has been configured so that you can kill your old processes/sessions. If you find that you are experiencing problems such as above, follow these steps:

1. Connect to Oracle using VSCode and run the following SQL command to view your current sessions:

```
SELECT
    username,
    sid,
    serial#,
    status,
    state,
    to_char(logon_time, 'MONdd hh24:mi') AS "Logon Time"
FROM
    v$session
WHERE
        type = 'USER'
    AND username = user
    AND upper(osuser)!= 'ORACLE'
ORDER BY
    "Logon Time";
```

This will provide details of all the sessions currently running in Oracle under your username:

	USERNAME	SID	SERIAL#	STATUS	STATE	Logon Time
☐	ABC001	1526	52369	INACTIVE	WAITED KNOWN TIME	JAN17 17:55
☐	ABC001	769	52227	INACTIVE	WAITED KNOWN TIME	JAN17 18:29
☐	ABC001	2652	56095	ACTIVE	WAITED SHORT TIME	JAN17 18:33
☐	ABC001	2284	62526	INACTIVE	WAITING	JAN17 18:33

Each session you initiate will often open several connections to the database - if there are multiples, you will need to kill off all inactive sessions that have a logon time earlier than your current session (here the last connection was at 18:33 so all the older sessions prior to 18:33 need to be killed off).

2. Kill off the 'hung' connections:

For example, in the above display, the connection with a logon time of 17:55 represents a 'hung' session - to kill such sessions, we have an Oracle procedure available called **kill_own_session**. The simplest way to call this procedure is by using the anonymous PL/SQL block shown below (code contributed by student Lauren Yim).

```
DECLARE
    sid     NUMBER;
    serial# NUMBER;
    CURSOR inactive_sessions IS
    SELECT
        sid,
        serial#
    FROM
        v$session
    WHERE
        type = 'USER'
        AND username = user
        AND upper(osuser) != 'ORACLE'
        AND status = 'INACTIVE';
BEGIN
    OPEN inactive_sessions;
    LOOP
        FETCH inactive_sessions INTO sid, serial#;
        EXIT WHEN inactive_sessions%notfound;
        kill_own_session(sid, serial#);
    END LOOP;

    CLOSE inactive_sessions;
END;
/
```

Following this, a repeat of the original select will show that the connections have now been killed:

	USERNAME	SID	SERIAL#	STATUS	STATE	Logon Time
☐	ABC001	1526	52369	KILLED	WAITED KNOWN TIME	JAN17 17:55
☐	ABC001	769	52227	KILLED	WAITED KNOWN TIME	JAN17 18:29
☐	ABC001	2652	56095	ACTIVE	WAITED SHORT TIME	JAN17 18:33
☐	ABC001	2284	62526	INACTIVE	WAITING	JAN17 18:33

Oracle will automatically clean up these killed sessions at a later time, you do not need to be

concerned about them. With the killing of inactive previous sessions, any locks associated with them will also be cleared and you will be able to proceed to work normally.

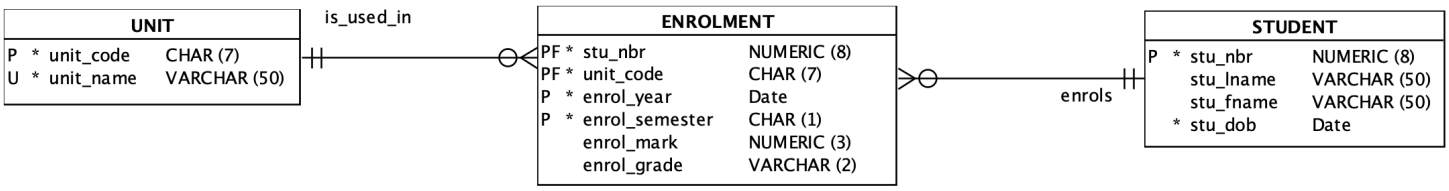
A8-4 Data Manipulation Language

SPECIAL NOTE

As a developer, trying to get DML code to work correctly, we will postpone the use of COMMIT until we are sure the command is correct. The normal approach will be to write the SQL code, use select statements to check if the DML impact was correct, if it is incorrect use ROLLBACK to undo the changes and then correct the SQL. ***Once the SQL is correct, COMMIT must be issued immediately after the transaction has been completed*** *(immediately after the relevant DML), you must not include in the transaction any select statements designed to check that the data was correctly manipulated.*

> ***Our aim here is to ensure the locks in the system are only active for the absolute minimum of time.***

First, run your applied7_schema.sql solution (or download the sample solution from the Applied 7 Sample Solutions) to recreate the student, unit and enrolment tables under your Oracle account.



Download the file below and place it into the Applied 8 folder of your local repo.

Write the required SQL statements for the tasks below in this file. Make sure that you include a semicolon ';' at the end of each SQL statement. Save the output from this run as **applied8_dml_output.txt**. To save the output, use the inbuilt Oracle SPOOL command as previously discussed.

applied8_dml.sql

A8-4.1 Basic INSERT statement

In this exercise, you will enter the data into the database using INSERT statements with the following assumptions:

- the database currently does not have any existing data, and
- the primary key is *not generated automatically* by the DBMS.

TASKS

1. Write SQL INSERT statements within the file to add the following data into the specified tables

STUDENT

stu_nbr	stu_lname	stu_fname	stu_dob
11111111	Bloggs	Fred	01-Jan-2003
11111112	Nice		10-Oct-2004
11111113	Wheat	Wendy	05-May-2005
11111114	Sheen	Cindy	25-Dec-2004

UNIT

unit_code	unit_name
FIT9999	FIT Last Unit
FIT9132	Introduction to Databases
FIT9161	Project
FIT5111	Student's Life

ENROLMENT

stu_nbr	unit_code	enrol_year	enrol_semester	enrol_mark	enrol_grade
11111111	FIT9132	2022	1	35	N
11111111	FIT9161	2022	1	61	C
11111111	FIT9132	2022	2	42	N
11111111	FIT5111	2022	2	76	D
11111111	FIT9132	2023	1		
11111112	FIT9132	2022	2	83	HD
11111112	FIT9161	2022	2	79	D
11111113	FIT9132	2023	1		
11111113	FIT5111	2023	1		
11111114	FIT9132	2023	1		
11111114	FIT5111	2023	1		

2. A file is available below:

applied8_studentdata.txt

which has this data as a starting point to build the required insert statements. Download the file, open applied8_studentdata.txt in VS Code, then copy and paste the contents to build your insert statements. In building the insert statements:

- numeric data (ie. numbers) must not be enclosed in quotation marks,
- character data must be enclosed within single quotes, and
- date data must make use of to_date to convert strings to dates

3. Ensure you make use of COMMIT to make your changes permanent. You may treat all the data that you add as a single transaction since you are setting up the initial test state for the database (i.e. add a single commit at the end of all the inserts).

4. Check that your data has inserted correctly by using the SQL command SELECT * FROM *tablename* **and** by using the VS Code GUI (select the table in the left-hand list and then select 'Show Data').

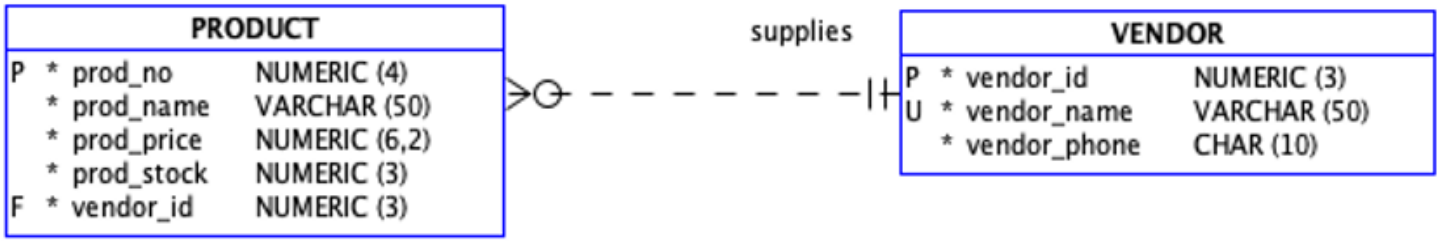
Remember that Visual Studio Code opens a new connection for each activity in the database - thus you will only see the data via the GUI if you have issued a commit.

A8-4.2 Using SEQUENCEs in an INSERT statement

In the previous exercises, you have entered the primary key value manually in the INSERT statements. In the situation where a SEQUENCE is available, you should use the sequence mechanism to generate the value of the primary key. Note that sequences may result in "gaps" in the generated primary key values. A user who obtains a sequence value via .nextval may decide to not complete the insert eg. they call a rollback. Also when the DBMS itself is restarted this may result in the available values being advanced slightly (depending on how they were cached).

Consider the following model depicting VENDOR and PRODUCT.

Assume we want to add vendors and the products they supply into a set of tables represented by:



A suitable schema would be:

```
DROP TABLE product PURGE;

DROP TABLE vendor PURGE;

DROP SEQUENCE prod_no_seq;

DROP SEQUENCE vendor_id_seq;

CREATE TABLE product (
    prod_no     NUMBER(4) NOT NULL,
    prod_name   VARCHAR2(50) NOT NULL,
    prod_price  NUMBER(6, 2) NOT NULL,
    prod_stock  NUMBER(3) NOT NULL,
    vendor_id   NUMBER(3) NOT NULL
);

COMMENT ON COLUMN product.prod_no IS
    'product number (unique for each product)';

COMMENT ON COLUMN product.prod_name IS
    'product name';

COMMENT ON COLUMN product.prod_price IS
```

```
    'product price';

COMMENT ON COLUMN product.prod_stock IS
    'product on hold (stock)';

COMMENT ON COLUMN product.vendor_id IS
    'vendor id (unique for each vendor)';

ALTER TABLE product ADD CONSTRAINT product_pk PRIMARY KEY ( prod_no );

CREATE TABLE vendor (
    vendor_id     NUMBER(3) NOT NULL,
    vendor_name   VARCHAR2(50) NOT NULL,
    vendor_phone  CHAR(10) NOT NULL
);

COMMENT ON COLUMN vendor.vendor_id IS
    'vendor id (unique for each vendor)';

COMMENT ON COLUMN vendor.vendor_name IS
    'vendor name';

COMMENT ON COLUMN vendor.vendor_phone IS
    'vendor phone';

ALTER TABLE vendor ADD CONSTRAINT vendor_pk PRIMARY KEY ( vendor_id );

ALTER TABLE vendor ADD CONSTRAINT vendor_un UNIQUE ( vendor_name );

ALTER TABLE product
    ADD CONSTRAINT vendor_product_fk FOREIGN KEY ( vendor_id )
        REFERENCES vendor ( vendor_id );

CREATE SEQUENCE prod_no_seq START WITH 1 INCREMENT BY 1;

CREATE SEQUENCE vendor_id_seq START WITH 1 INCREMENT BY 1;
```

Using the nextval and currval of the sequences to add a vendor and their related products as a single transaction.

```
-- Add Vendor 1 and the products they supply - transaction 1
 insert into vendor values (vendor_id_SEQ.nextval,
        'Western Digital', '1234567890');
 insert into product values (prod_no_SEQ.nextval,
        '2TB My Cloud Drive',195,5,vendor_id_SEQ.currval);
 insert into product values (prod_no_SEQ.nextval,
        '1TB Portable Hard Drive',76,4,vendor_id_SEQ.currval);
 insert into product values (prod_no_SEQ.nextval,
        'Live Media Player',119,2,vendor_id_SEQ.currval);

 Commit;

-- Add Vendor 2 and the products they supply - transaction 2
```

```
insert into vendor values (vendor_id_SEQ.nextval,'Seagate',
       '2468101234');
insert into product values (prod_no_SEQ.nextval,
       '2TB Desktop Drive',94,12,vendor_id_SEQ.currval);
insert into product values (prod_no_SEQ.nextval,
       '4TB 4 Bay NAS',76,4,vendor_id_SEQ.currval);
insert into product values (prod_no_SEQ.nextval,
       '2TB Central Personal Storage' ,169,5, vendor_id_SEQ.currval);
commit;
```

TASKS

1. Create a sequence for the STUDENT table called STUDENT_SEQ

- Create a sequence for the STUDENT table called STUDENT_SEQ that starts at 11111115 and increases by 1.
- Check that the sequence exists in two ways (using SQL and browsing connection objects).

2. Add a new student (*MICKEY MOUSE)* and an enrolment for this student as listed below, treat all the data that you add as a single transaction.

- Use the student sequence to insert *MICKEY MOUSE* - pick any STU_DOB you wish
- Add an enrollment for this student to the unit FIT9132 in semester 1 2023

A8-4.3 Advanced INSERT

We have learned how to add data into the database in the previous exercises through the use of INSERT statements. In those exercises, the INSERT statements were created as a single script assuming that the data is all added at the same time, such as at the beginning when the tables are created. On some occasions, new data is added after some relevant/related data already exists in the database. In this situation, we will need to use a combination of INSERT and SELECT statements.

A SELECT statement is an SQL statement that we use to retrieve data from a database. An example of a SELECT statement would be:

```
SELECT vendor_id
FROM vendor
WHERE upper(vendor_name) = upper('Seagate');
```

The above SQL statement consists of three SQL clauses SELECT, FROM and WHERE. The SELECT clause is used to declare which column(s) are to be displayed in the output. The FROM clause is used to declare from which table the data needs to be retrieved. The WHERE clause is used to declare which rows are to be retrieved. In the above SQL select, any row that has the vendor_name equal to *Seagate* will be retrieved. The SQL SELECT statement will be covered in more detail in the future module, retrieving data from the database. In our VENDOR-PRODUCT model, we can see that vendor_name is unique - note the U in the model above and the vendor_un constraint in the schema above. As a result, the above select returns a single vendor_id (this is known as a ***singleton select***). A select which returns more than one row is called a ***non-singleton select***.

In our VENDOR-PRODUCT model, to add a new product for a vendor at a subsequent time when the vendor already exists, we can use a subselect statement (as above) to obtain the vendor_id.

```
INSERT INTO product VALUES (
    prod_no_seq.NEXTVAL,
    'GoFlex Thunderbolt Adaptor',
    134,
    2,
    (
        SELECT vendor_id
        FROM vendor
        WHERE upper(vendor_name) = upper('Seagate')
    )
);

COMMIT;
```

TASKS

1. A new student has started a course. Subsequently this new student needs to enrol into the unit named *Introduction to Databases*. Enter the new student's details, then insert his/her enrollment to the database using the sequence in combination with a SELECT statement. You can make up details of the new student and when they will attempt *Introduction to Databases* - you may assume there is only one student with such a name in the system.

- You must not do a manual lookup to find the unit code of the *Introduction to Databases* and the student number.

A8-4.4 Creating a table and inserting data as a single SQL statement

A table can also be created based on an existing table, and immediately populated with contents by using a SELECT statement within the CREATE TABLE statement.

For example, to create a table called FIT9132_STUDENT which contains the enrolment details of all students who have been or are currently enrolled in *FIT9132*, we would use:

```
CREATE TABLE fit9132_student
    AS
        SELECT *
        FROM enrolment
        WHERE upper(unit_code) = upper('FIT9132');
```

Here, we use the SELECT statement to retrieve all columns (the wildcard "*" represents all columns) from the table enrollment, but only those rows with a value of the unit_code equal to *FIT9132*.

TASKS

1. Create a table called FIT5111_STUDENT. The table should contain all current enrolments for the unit *FIT5111*.

2. Check the table exists.
3. List the contents of the table.

A8-4.5 UPDATE

It is common for data to change in value across time. In a database, we use the SQL UPDATE statement to change the value of a cell or cells in a table.

The UPDATE statement consist of three main components:

- The name of the table where the data will be updated.
- The new value to replace the old value.
- The row or the set of rows where the value will be updated.

An example of an UPDATE statement using the student-enrolment-unit database would be:

```
UPDATE enrolment
SET enrol_mark = 63, enrol_grade = 'C'
WHERE stu_nbr = 11111111 AND
    upper(unit_code) = 'FIT9132' AND
    enrol_semester = '1' AND
    to_char(enrol_year,'yyyy') = '2023';

COMMIT;
```

TASKS

1. Update the unit name of *FIT9999* from 'FIT Last Unit' to 'place holder unit'.

2. Enter the mark and grade for the student with the student number of 11111113 for the *Introduction to Databases* unit that the student enrolled in semester 1 of 2023. The mark is 75 and the grade is D.

3. The university has introduced a new grade classification scale. The new classifications are:

- 0 - 44 is N
- 45 - 54 is P1
- 55 - 64 is P2
- 65 - 74 is C
- 75 - 84 is D
- 85 - 100 is HD

Change the database to reflect the new grade classification scale.

4. Due to new regulations, the Faculty of IT decided to change 'Project' unit code from *FIT9161* into *FIT5161*. Change the database to reflect this situation.

Note: you need to disable the FK constraint before you do the modification then re enable the FK to have it active again. The SQL syntax to enable/disable a constraint is:

```
ALTER TABLE table_name
       [ENABLE/DISABLE] CONSTRAINT constraint_name;
```

*Please note we are doing this here so that you experience one possible approach, such an approach would **never be applied to a live database with users active**, given the problems that the removal of the FK constraint could generate.*

A8-4.6 DELETE

The DELETE statement is used to remove data from the database.

It is important to consider the referential integrity issues when using a DELETE statement. In Oracle, a table can be created with a FOREIGN KEY ON DELETE clause which indicates what action should be taken when the parent of a child record is attempted to be deleted. The specified actions can be:

- CASCADE, or
- SET NULL

When the ON DELETE clause is not specified (the default), the action will be set to RESTRICT. RESTRICT means the delete of a row in a parent table (the table containing the PRIMARY KEY being referred to by a FOREIGN KEY) will not be permitted when there are related rows in the child table (the table with the FOREIGN KEY).

TASKS

1. A student with student number 11111114 has taken intermission in semester 1 2023, hence all the enrolments of this student for semester 1 2023 should be removed. Change the database to reflect this situation.

2. The faculty decided to remove all *Student's Life* unit's enrolments. Change the database to reflect this situation. Note: unit names are unique in the database.

3. Assume that Wendy Wheat (student number 11111113) has withdrawn from the university. Remove her details from the database.

Important

You need to get into the habit of establishing this as a standard workflow:

- before modifying any file/s, pull at the start of your working session, work on the activities you wish to/are able to complete during this session, add all(stage), commit changes and then push the changes back to the FIT GitLab server.