FIT2094-FIT3171 2019 S1 -- Week 7 eBook

Credits and Copyrights:
Authors: FIT3171 2018 S2 UG Databases
FIT Database Teaching Team
Maria Indrawan, Manoj Kathpalia, Lindsay Smith, et al

Change Log:

- Formatted and Imported from Alexandria APRIL 2019.

7.0. SQL Data Definition Language (DDL)

7.0.1. Introduction to Drop Tables

When creating schema files, you should always also create a drop file or add the drop commands to the top of your schema file. You should drop the tables using the:

```
drop table tablename purge;
```

...syntax.

The drop table statements should list tables in the **reverse order of your create table order** so that FK relationships will be able to be removed successfully. Should a syntax error occur while testing your schema, you simply need to run the drop commands to remove any tables which may have been created.

IMPORTANT: always remember that DROP is a destructive command - use with care, and always double check the table names and syntax!

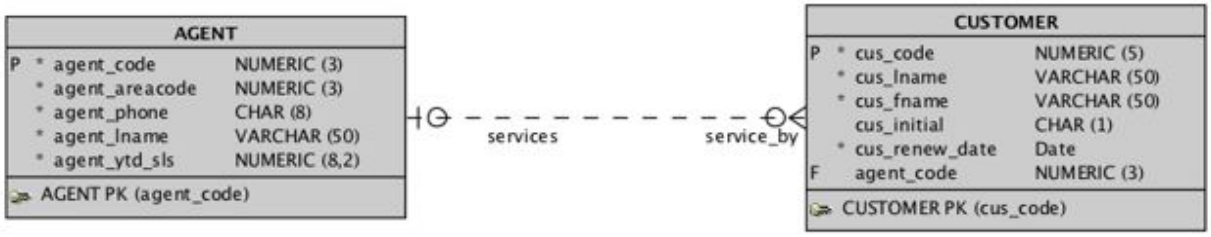
7.0.2. Reminder of Data Types

An excellent summary of the Oracle data types and version restrictions is available from:

https://www.techonthenet.com/oracle/datatypes.php

For this unit, we make use of CHAR, VARCHAR2 (or VARCHAR), NUMBER (or NUMERIC) and DATE.

7.0.3. Case Study: Figure 3.3 (Coronel & Morris)



The data model above represents Figure 3.3 from Coronel & Morris. There are two different ways of coding this model as a set of create table statements.

Using table constraints

SQL constraints are classified as column or table constraints; depending on which item they are attached to:

```
create table agent
  (
    agent_code      number (3) constraint agent_pk primary key,
    agent_areacode number (3) not null ,
    agent_phone    char (8) not null ,
    agent_lname    varchar2 (50) not null ,
    agent_ytd_sls  number (8,2) not null
  ) ;
```

This is a declaration of the primary key as a column constraint (in **bold**).

```
create table agent
  (
    agent_code      number (3) not null ,
    agent_areacode  number (3) not null ,
    agent_phone     char (8) not null ,
```

```
    agent_lname     varchar2 (50) not null ,
    agent_ytd_sls   number (8,2) not null ,
    constraint agent_pk primary key ( agent_code )
  ) ;
```

Here the primary key has been declared as a table constraint, at the end of the table after all column declarations have been completed (in **bold**).

In some circumstances, for example, a composite primary key, you must use a table constraint since a column constraint refers only to a single column.

The create table statements for the two tables in Figure 3.3 (Coronel & Morris) would be:

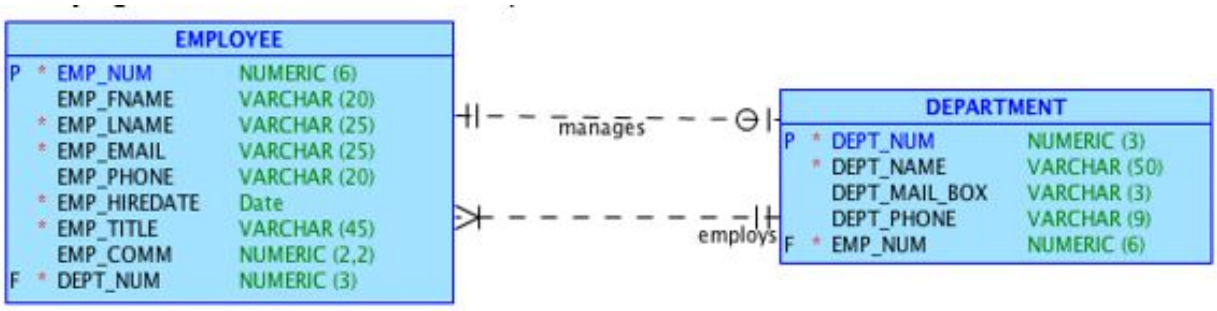
```
create table agent
  (
    agent_code      number (3) not null ,
    agent_areacode  number (3) not null ,
    agent_phone     char (8) not null ,
    agent_lname     varchar2 (50) not null ,
    agent_ytd_sls   number (8,2) not null,
    constraint agent_pk primary key ( agent_code )
  ) ;

create table customer
  (
    cus_code       number (5) not null ,
    cus_lname      varchar2 (50) not null ,
    cus_fname      varchar2 (50) not null ,
    cus_initial    char (1) ,
    cus_renew_date date not null ,
    agent_code     number (3),
    constraint customer_pk primary key ( cus_code ),
```

```
    constraint customer_agent_fk foreign key ( agent_code)
      references agent ( agent_code ) on delete set null
  ) ;
```

The inclusion of the referential integrity rule **`on delete set null`** in the above **`create table`** statement is appropriate in this scenario – when an agent leaves a reasonable approach would be to set the foreign key for that agent's customers to **null**. The default **`on delete restrict`** (which you do not specify, simply omit an **`on delete`** clause) would also be an alternative approach.

Using `on delete cascade` would not: since this would cause the customers of the agent who left to also be deleted. When coding a foreign key definition you **must always consider what is a suitable on delete approach (RESTRICT, CASCADE, NULLIFY) given the scenario you are working with**.

In some circumstances, this approach of defining the foreign keys as part of the table definitions cannot be used. Can you see what the issue is with trying to create the two tables depicted below?



In such a situation an alternative approach to declaring constraints needs to be adopted.

7.0.4. Using ALTER table commands

In this approach, the tables are declared without constraints and then the constraints are applied via the **`ALTER TABLE`** command (see section 7.5 of Coronel & Morris).

```
create table agent
  (
    agent_code     number (3) not null ,
    agent_areacode number (3) not null ,
    agent_phone    char (8) not null ,
    agent_lname    varchar2 (50) not null ,
    agent_ytd_sls  number (8,2) not null
  ) ;


alter table agent add constraint agent_pk primary key
  ( agent_code ) ;

create table customer
  (
    cus_code        number (5) not null ,
    cus_lname       varchar2 (50) not null ,
    cus_fname       varchar2 (50) not null ,
    cus_initial     char (1) ,
    cus_renew_date  date not null ,
    agent_code      number (3)
  ) ;

alter table customer add constraint customer_pk primary key
  ( cus_code ) ;

alter table customer add constraint customer_agent_fk foreign key
  ( agent_code ) references agent ( agent_code )
  on delete set null;
```

Remember, from above, when coding a foreign key definition you **must always consider what is a suitable on delete approach (RESTRICT, CASCADE, NULLIFY) given the scenario you are working with**.

After creating the tables we need to insert the data, for **AGENT** the insert will have the form:

```
insert into agent values (501,713,'228-1249','Alby',132735.75);
```

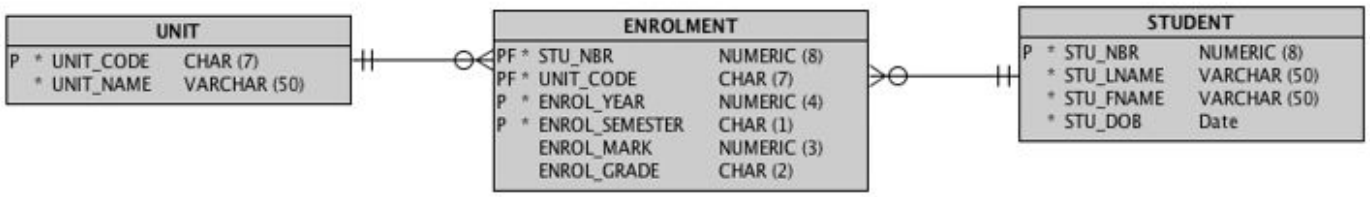
… and for **CUSTOMER**:

```
insert into customer values(10010,'Ramas','Alfred','A',
     '05-Apr-2014',501);
```

It is important to note that for the insert into customer we are using the default Oracle date format of `'dd-mon-yyyy'` – in the near future we will correct this and allow any date format via the Oracle function **`todate`**.

7.0.5. Lab Tasks: STUDENT, UNIT, ENROLMENT

Using the model from the DDL lecture for student, unit and enrolment:



Creating tables from scratch.

- Code a schema file to create these three tables, noting the following ***extra*** constraints:
    1. `stu_nbr > 10000000`
    2. `unit_name` is unique in the **UNIT** table
    3. `enrol_semester` can only contain the value of **1** or **2** or **3**.
- In implementing these constraints you will need to make use of `CHECK` clauses (see Coronel & Morris section 7.2.6).
- Ensure your script file has appropriate comments in the header, includes the required `drop` commands and includes `echo on` and `echo off` commands.
- Run your script and create the three required tables.
- Save the output from this run.

As an alternative to using echo on/off and having to save the output, a simpler approach is through the use of the inbuilt Oracle `SPOOL` command.

To use `SPOOL`, place as the top line in your schema file:

```
spool ./myoutput.txt
```

...and as the last line in your script file

```
spool off
```

This will produce a file, in the same folder that your script is saved in, called `myoutput.txt` which contains the full run of your SQL script.

7.1. Exercises: INSERTing data into the database

7.1.1. 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.
- the primary key is not generated automatically by the DBMS.

TASKS
Insert the following data into the tables specified using the SQL INSERT statement:

STUDENT

stu_nbr	stu_lname	stu_fname	stu_dob
11111111	Bloggs	Fred	01-Jan-1990
11111112	Nice	Nick	10-Oct-1994
11111113	Wheat	Wendy	05-May-1990
11111114	Sheen	Cindy	25-Dec-1996

UNIT

unit_code	unit_name
FIT9999	FIT Last Unit
FIT5132	Introduction to Databases
FIT5016	Project
FIT5111	Student's Life

ENROLMENT

stu_nbr	unit_code	enrol_year	enrol_semester	enrol_mark	enrol_grade
11111111	FIT5132	2013	1	35	N
11111111	FIT5016	2013	1	61	C
11111111	FIT5132	2013	2	42	N
11111111	FIT5111	2013	2	76	D
11111111	FIT5132	2014	2		
11111112	FIT5132	2013	2	83	HD
11111112	FIT5111	2013	2	79	D
11111113	FIT5132	2014	2		
11111113	FIT5111	2014	2		
11111114	FIT5111	2014	2		

- Ensure you make use of **COMMIT** to make your changes permanent.
- Check that your data has inserted correctly by using the SQL command `SELECT * FROM tablename` **and** by using the SQL GUI (select the table in the right-hand list and then select the Data tab).

7.1.2. Using SEQUENCEs in an INSERT statement.

In the previous exercises, you have entered the primary key value manually in the **INSERT** statements. In the case where a **SEQUENCE** is available, you should use the sequence mechanism to generate the value of the primary key.

TASKS

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 your SQL Developer connection objects).

Add a new student ('MICKEY MOUSE')

- Use the student sequence – pick any **STU_DOB** you wish.
- Check that your insert worked.
- Add an enrolment for this student to the unit FIT5132 in semester 2 2016.

7.1.3. 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 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 data already exists in the database. In this situation, it is a good idea 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:

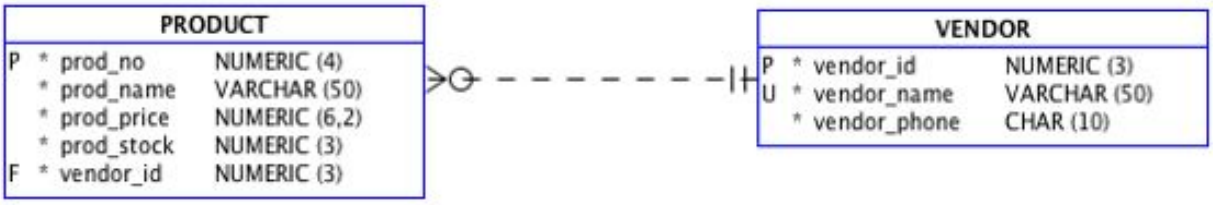
```
SELECT vendor_id

FROM vendor

WHERE vendor_name = '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.

For our exercise on using the advanced **INSERT** statement, 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 PRODUCT_prod_no_SEQ;
DROP SEQUENCE VENDOR_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_vendor_id  NUMBER (3) NOT NULL
  ) ;

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
  ) ;

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 PRODUCT_VENDOR_FK FOREIGN KEY (
VENDOR_vendor_id ) REFERENCES VENDOR ( vendor_id ) ON DELETE CASCADE ;

CREATE SEQUENCE PRODUCT_prod_no_SEQ START WITH 1 INCREMENT BY 1;
CREATE SEQUENCE VENDOR_vendor_id_SEQ START WITH 1 INCREMENT BY 1;
```

There are two ways in which we can perform the INSERT.

1. Use the **`nextval`** and **`currval`** of the sequences.

```
-- Add Vendor 1 and the products they supply
 insert into vendor values (VENDOR_vendor_id_SEQ.nextval,
      'Western Digital', '1234567890');
 insert into product values (PRODUCT_prod_no_SEQ.nextval,
      '2TB My Cloud Drive',195,5,VENDOR_vendor_id_SEQ.currval);
 insert into product values (PRODUCT_prod_no_SEQ.nextval,
      '1TB Portable Hard Drive',76,4,VENDOR_vendor_id_SEQ.currval);
 insert into product values (PRODUCT_prod_no_SEQ.nextval,
      'Live Media Player',119,2,VENDOR_vendor_id_SEQ.currval);
 commit;
 -- Add Vendor 2 and the products they supply
 insert into vendor values (VENDOR_vendor_id_SEQ.nextval,'Seagate',
      '2468101234');
 insert into product values (PRODUCT_prod_no_SEQ.nextval,
      '2TB Desktop Drive',94,12,VENDOR_vendor_id_SEQ.currval);
 insert into product values (PRODUCT_prod_no_SEQ.nextval,
      '4TB 4 Bay NAS',76,4,VENDOR_vendor_id_SEQ.currval);
 insert into product values (PRODUCT_prod_no_SEQ.nextval,
```

```
      '2TB Central Personal Storage' ,169,5,
      VENDOR_vendor_id_SEQ.currval);
 commit;
```

2. Use the **nextval** in combination with the **SELECT** statement.

```
-- Add a new product for a vendor at a subsequent time (vendor names will
be unique – note the U in the model above and the vendor_un constraint in
the schema)
insert into product values (PRODUCT_prod_no_SEQ.nextval,
      'GoFlex Thunderbolt Adaptor',134,2,
      (select vendor_id from vendor where vendor_name = 'Seagate'));
```

In subsequent weeks you will see that the same concept can be used with other data manipulation statements such as UPDATE and DELETE.

TASKS

- A new student has started a course and needs to enrol into “Introduction to databases”. Enter the new student’s details and his/her enrolment to the database using the nextval in combination with a **SELECT** statement. You can make up details of the new student and when they will attempt “Introduction to databases”.
    - You must not do a manual lookup to find the unit code of the “Introduction to databases”.

7.1.4. 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 **FIT5132_STUDENT** which contains the enrolment details of all students who have been or are currently enrolled in FIT5132, we would use:

```
create table FIT5132_STUDENT

as select *

from enrolment

where unit_code = 'FIT5132';
```

Here, we use the **SELECT** statement to retrieve all columns (the wildcard "*****" represents all columns) from the table enrolment, but only those rows with a value of the unit_code equal to FIT5132.

TASKS

- Create a table called **FIT5111_STUDENT**. The table should contain all enrolments for the unit FIT51111.
- Check the table exists.
- List the contents of the table.

7.1.5. Changing a table’s structure.

TASKS

- Add a new column to the UNIT table which will represent credit points for the unit (hint use the ALTER command). The default value for the new column should be 6 points.
- Insert a new unit after you have added the new column. You can make up the details of the new unit.
- Check that the new insert has worked correctly.

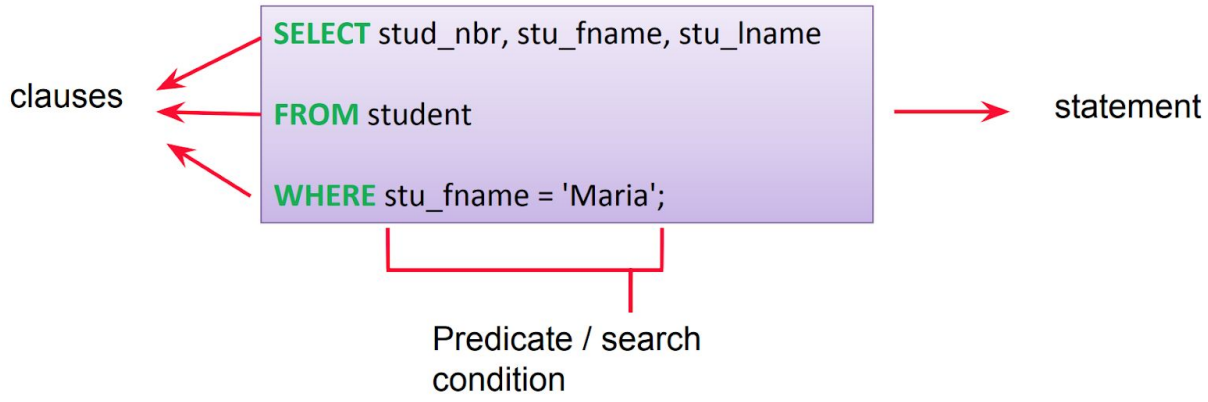
7.2. Pre-Lecture Notes

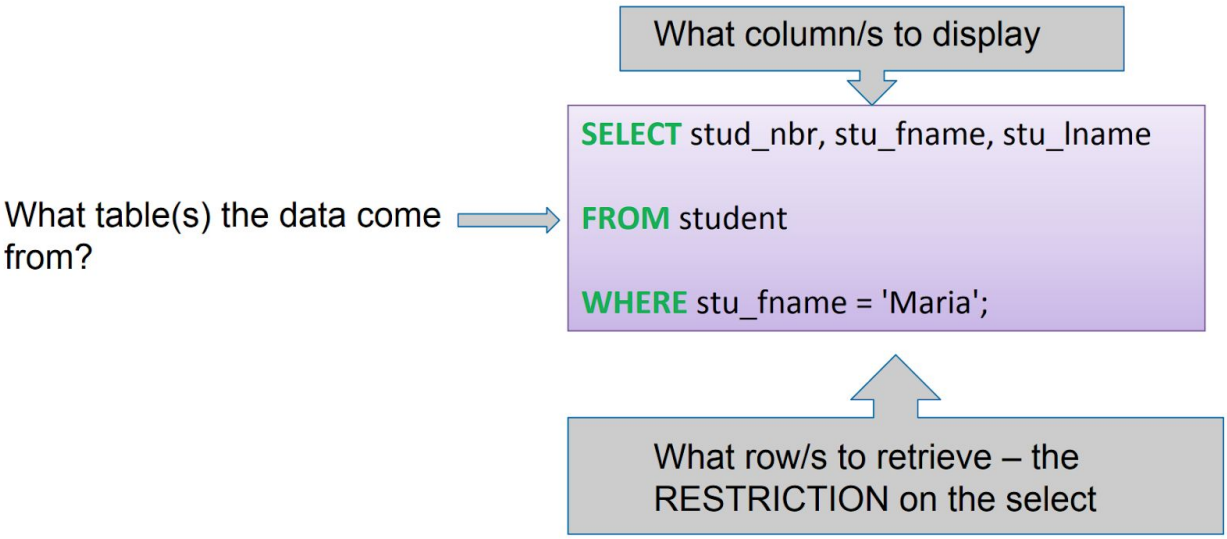
These are notes you may find useful to read through before the lecture, adapted from Lindsay Smith's lecture material. NOTE: THESE ARE NOT THE FINAL LECTURE SLIDES.

Read up the pre-lecture notes below.

IMPORTANT: AS THE LECTURE FOCUSES ON THE 'FLIPPED CLASSROOM' APPROACH - I.E. MORE INTERACTIVE DISCUSSION AND LESS ON DISCOVERING NEW MATERIAL - THE THEORY SLIDES BELOW ARE PROVIDED FOR YOUR READING CONVENIENCE BEFORE THE LECTURE.

Anatomy of an SQL SELECT Statement





SQL Predicates or Search Conditions

- The search conditions are applied on each row, and the row is returned if the search conditions are evaluated to be TRUE for that row.
- **Comparison**
  - Compare the value of one expression to the value of another expression.
  - Operators:
    - =, < >, <, >, !=, <=, >=
  - Example: salary > 5000
- **Range**
  - Test whether the value of an expression falls within a specified range of values.
  - Operators:
    - BETWEEN
  - Example: salary BETWEEN 1000 AND 3000 (both are inclusive)

- **Set Membership**
  - To test whether the value of expression equals one of a set of values.
  - Operator:
    - IN
  - Example : city IN ('Melbourne', 'Sydney')
- **Pattern Match**
  - To test whether a string (text) matches a specified pattern.
  - Operator:
    - LIKE
  - Patterns:
    - % character represents any sequence of zero or more character.
    - _ character represents any single character.
  - Example:
    - WHERE city LIKE 'M%'
    - WHERE unit_code LIKE 'FIT20__'

- **NULL**
  - To test whether a column has a NULL (unknown) value.
  - Example: WHERE grade IS NULL.
- Use in subquery (to be discussed in the future)
  - ANY, ALL
  - EXISTS

What row will be retrieved?

- Predicate evaluation is done using three-valued logic.
  - **TRUE**, **FALSE** and **UNKNOWN**
- DBMS will evaluate the predicate against each row.
- Row that is evaluated to be **TRUE** will be retrieved.
- NULL is considered to be UNKNOWN.

Combining Predicates

- Logical operators
  - AND, OR, NOT
- Rules:
  - An expression is evaluated LEFT to RIGHT.
  - Sub-expression in brackets are evaluated first.
  - NOTs are evaluated before AND and OR
  - ANDs are evaluated before OR.

Truth Table

- **AND** is evaluated to be TRUE if and only if **both** conditions are TRUE
- **OR** is evaluated to be TRUE if and only if at least one of the conditions is TRUE

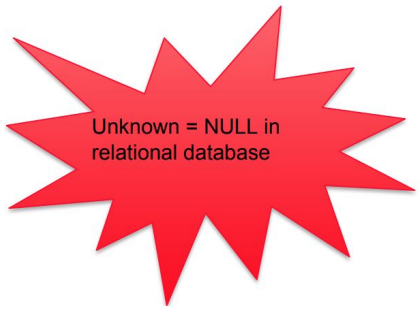
AND

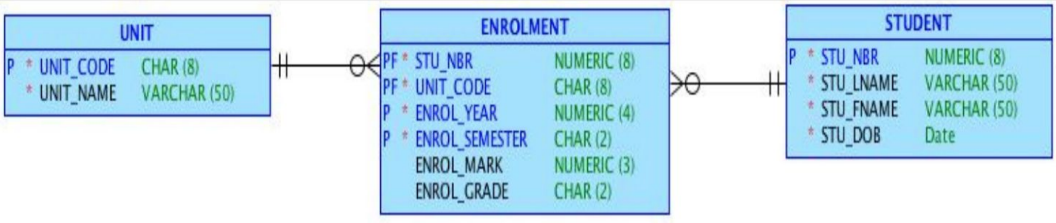
A \ B	T	U	F
T	T	U	F
U	U	U	F
F	F	F	F

OR

A \ B	T	U	F
T	T	T	T
U	T	U	U
F	T	U	F

T = TRUE
F = FALSE
U = Unknown






```
SELECT * FROM UNIT;
```

Script Output | Query Result | SQL | All Rows Fetched: 3 in 0.015 seconds

	UNIT_CODE	UNIT_NAME
1	FIT1001	Computer Systems
2	FIT1002	Computer Programming
3	FIT1004	Database

```
SELECT * FROM STUDENT;
```

Script Output | Query Result | SQL | All Rows Fetched: 4 in 0.022 seconds

	STU_NBR	STU_LNAME	STU_FNAME	STU_DOB
1	11111111	Bloggs	Fred	01/JAN/90
2	11111112	Nice	Nick	10/OCT/94
3	11111113	Wheat	Wendy	05/MAY/90
4	11111114	Sheen	Cindy	25/DEC/96

```
SELECT * FROM ENROLMENT;
```

Script Output | Query Result | SQL | All Rows Fetched: 8 in 0.016 seconds

	STU_NBR	UNIT_CODE	ENROL_YEAR	ENROL_SEMESTER	ENROL_MARK	ENROL_GRADE
1	11111111	FIT1001	2012	1	78	D
2	11111111	FIT1002	2013	1	(null)	(null)
3	11111111	FIT1004	2013	1	(null)	(null)
4	11111112	FIT1001	2012	1	35	N
5	11111112	FIT1001	2013	1	(null)	(null)
6	11111113	FIT1001	2012	2	65	C
7	11111113	FIT1004	2013	1	(null)	(null)

MONASH

Arithmetic Operations

- Can be performed in SQL.
- For example:

```
SELECT stu_nbr, enrol_mark/10
FROM enrolment;
```

	STU_NBR	ENROL_MARK/10
1	11111111	7.8
2	11111111	(null)
3	11111111	(null)
4	11111112	3.5
5	11111112	(null)

Oracle NVL function

- It is used to replace a NULL with a value.

```
SELECT stu_nbr,
  NVL(enrol_mark,0),
  NVL(enrol_grade,'WH')
FROM enrolment;
```

	STU_NBR	NVL(ENROL_MARK,0)	NVL(ENROL_GRADE,'WH')
1	11111111	78	D
2	11111111	0	WH
3	11111111	0	WH
4	11111112	35	N
5	11111112	0	WH
6	11111113	65	C
7	11111113	0	WH
8	11111114	0	WH

Renaming Column

- Note column headings on slide 16
- Use the word "AS"
  - New column name in " " to maintain case or spacing
- Example

```
SELECT stu_nbr, enrol_mark/10 AS new_mark
FROM enrolment;

SELECT stu_nbr, enrol_mark/10 AS "New Mark"
FROM enrolment;
```

Sorting Query Result

- "ORDER BY" clause – *tuples have no order*
  - Must be used if more than one row may be returned
- Order can be ASCending or DESCending. The default is ASCending.
  - NULL values can be explicitly placed first/last using "NULLS LAST" or "NULLS FIRST" command
- Sorting can be done for multiple columns.
  - order of the sorting is specified for each column.
- Example:

```
SELECT stu_nbr, enrol_mark
FROM enrolment
ORDER BY enrol_mark DESC
```

	STU_NBR	ENROL_MARK
1	11111111	(null)
2	11111111	(null)
3	11111114	(null)
4	11111112	(null)
5	11111113	(null)
6	11111111	78
7	11111113	65
8	11111112	35

Removing Duplicate Rows in the Query Result

- Use "DISTINCT" as part of SELECT clause.

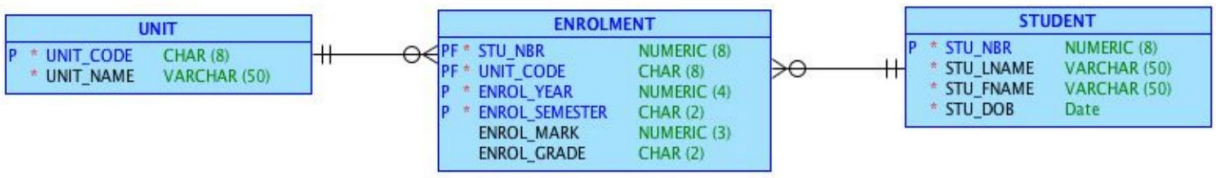
```
SELECT DISTINCT stu_nbr
FROM enrolment
WHERE enrol_mark IS NULL;
```

	STU_NBR
1	11111114
2	11111111
3	11111112
4	11111113

JOIN-ing Multiple Tables

Pair the PK and FK in the JOIN condition

Note table aliasing e.g. unit u in FROM clause




```
SELECT s.stu_nbr, s.stu_lname, u.unit_name
FROM ((unit u JOIN enrolment e ON u.unit_code=e.unit_code)
        JOIN student s ON e.stu_nbr=s.stu_nbr)
ORDER BY s.stu_nbr, u.unit_name;
```

Summary

- SQL statement, clause, predicate.
- Writing SQL predicates.
  - Comparison, range, set membership, pattern matching, is NULL
  - Combining predicates using logic operators (AND, OR, NOT)
- Arithmetic operation.
  - NVL function
- Column alias.
- Ordering (Sorting) result.
- Removing duplicate rows.
- JOIN-ing tables

- Dates are stored differently from the SQL standard
  - standard uses two different types: date and time
  - Oracle uses one type: DATE
    - Stored in internal format contains date and time
    - Output is controlled by formatting
      - select **to_char**(sysdate,'dd-Mon-yyyy') from dual;

        » 14-Apr-2018
      - select **to_char**(sysdate,'dd-Mon-yyyy hh:mi:ss PM') from dual;

        » 14-Apr-2018 02:51:24 PM

- DATE data type should be formatted with **TO_CHAR** when selecting for **display**.
- Text representing date **must be formatted** with **TO_DATE** when **comparing** or **inserting/updating.**
- Example:

```
select studid,
   studfname || ' ' || studlname as StudentName,
   to_char(studdob,'dd-Mon-yyyy') as StudentDOB
from uni.student
where studdob >  to_date('01-Apr-1991','dd-Mon-yyyy')
order by studdob;
```

Current Date

- Current date can be queried from the DUAL table using the **SYSDATE** attribute.
  - SELECT **sysdate** FROM dual;
- Oracle internal attributes include:
  - **sysdate**: current date/time
  - **systimestamp**: current date/time as a timestamp
  - **user**: current logged in user

EOF.