

Databases Notes Summary

DBMS allows us to manage the information stored in a database. It also stores a complete definition of the databases **structure** and **constraints**:

- Hierarchical database: Fields or record are structured nodes connected by branches.
- Networking database: Hierarchical arrangement where child nodes have more than one parent with many-to-many relationship allowing for multiple pathways.
- Multidimensional database
- Object-oriented database
- Relational database – Most flexible as data is stored in **tables** related via a common data item. Simple to add to, delete and modify.

Tables represent things we are interested in storing data about. Each row represents a single occurrence of this thing and each column represents a piece of data.

Relationships store information about how these 'things' are related.

SQL is used to create, retrieve and update data in a database.

Queries are information retrieval requests made to the database.

Reports are the answers to queries.

Basics

SELECT – extracts data from one or more tables

e.g. `SELECT columns FROM tables WHERE condition;`

INSERT INTO – inserts new data into a new record

e.g. `INSERT INTO customers (CustomerID, CustomerName, Address)
VALUES (13451458, Alex, Deansgrange);`

UPDATE – updates a specified record in a table, where command specifies row

e.g. `UPDATE Customers
SET CustomerName = 'Rian'
WHERE CustomerID = 13451458;`

DELETE FROM – deletes from a table

e.g. `DELETE FROM Customers
WHERE CustomerName = 'Alex';`

CREATE DATABASE/TABLE

e.g. `CREATE DATABASE Wiggle;
CREATE TABLE Customers (
 CustomerID int(16),
 CustomerName varchar(255),
 Address varchar(255)
);
ALTER TABLE Customers
ADD OrderID int(16);
DROP TABLE Customers;`

Sorting and Restricting Data

ORDER BY – Orders records by either ascending (default) or descending

e.g.

```
SELECT *  
FROM Customers  
ORDER BY Address DESC;
```

WHERE – Restricts the rows that are returned

e.g.

```
SELECT Country, PostalCode  
FROM Customers  
WHERE PostalCode = 'Dublin';
```

BETWEEN – Displays row based on a range of values

e.g.

```
WHERE CustomerID BETWEEN 134500 AND 134600;
```

IN – Test for values in a list

e.g.

```
WHERE Address = 'Blackrock';
```

LIKE – Searches a column for a specified pattern, such as all city's beginning with 'd' or 'o' as 2nd letter

e.g.

```
WHERE City LIKE 'd%' || '_o%';
```

IS NULL – Tests to see if field is null

e.g.

```
WHERE CustomerID IS NULL;
```

AND, OR, NOT – Similar to C, returns 1 or 0 and can be used with WHERE statement

e.g.

```
WHERE CustomerID >= 134500  
AND PostalID LIKE '%Dublin%'  
NOT IN ('Dublin1', 'Dublin5', 'Dublin6');
```

Substitution Variables

Temporarily store values with '&' and '&&', used with WHERE, ORDER BY, SELECT etc.

e.g.

```
WHERE CustomerID = &customer_num; – single & prompts user for a value
```

```
WHERE job_id = '&job_title'; – use quotation marks for characters and date
```

e.g.

```
SELECT employee_id, job_id, &&column_name  
FROM employees  
ORDER BY &column_name; – Double ampersand checks if var has been previously defined
```

DEFINE/UNDEFINE – Create and assign a value to a variable then remove if necessary

e.g.

```
DEFINE employee_num = 200  
SELECT employee_id  
FROM employees  
WHERE employee_id = &employee_num;  
UNDEFINE employee_num
```

SET VERIFY – Turns on and off the display of lines that have substitutions

Functions

Two types of functions:

Single-Row - Returns one result per inputted row.

Multiple-Row - Returns one result per multiple inserted rows.

Nesting functions – Single-row functions can be nested and are evaluated from deepest level up.

e.g.

```
SELECT last_name,  
       UPPER(CONCAT(SUBSTR (LAST_NAME, 1, 8), '_US'))  
FROM employees  
WHERE department_id = 60;
```

 – Retrieves last 8 chars of last name and concatenates with '_US' then converts to upper case and stores in new column

There are 5 main single-row functions we can use:

1. **Character Functions**: Broken into Case-manipulation and Character-manipulation.

LOWER - Changes string to lower case

e.g.

```
LOWER ('Alex') => alex
```

UPPER – Changes string to upper case

e.g.

```
UPPER ('Alex') => ALEX
```

INITCAP – Capitalizes first char of each word

e.g.

```
INITCAP ('alex kiernan') => Alex Kiernan
```

CONCAT – Concatenates specified words

e.g.

```
CONCAT ('Alex', 'Kiernan') => AlexKiernan
```

SUBSTR – Picks out specified part of string

e.g.

```
SUBSTR('AlexKiernan', 1, 4) => Alex
```

LENGTH – Displays length of string

e.g.

```
LENGTH ('Alex') => 4
```

INSTR – Finds the starting location of a specified char

e.g.

```
INSTR ('AlexKiernan', 'i') => 6
```

LPAD | RPAD – pads specified part of string with characters where num is total to fill

e.g.

```
LPAD (lastName, 10, '*') => ***Kiernan
```

TRIM – removes specified char from string

e.g.

```
TRIM ('A' FROM 'Alex') => lex
```

REPLACE – Replace specified character(s) with new characters(s)

e.g.

```
REPLACE ('Alex AND Kiernan', 'K', 'T') =? Alex Tiernan
```

Example:

```
SELECT employee_id, CONCAT(first_name, last_name) NAME, job_id,
LENGTH (last_name), INSTR (last_name, 'a') "Contains 'a'?"
FROM employees
WHERE SUBSTR(last_name, -1, 1) = 'n';
```

2. Number functions

ROUND/TRUNC – Round value up or down to specified decimal place

e.g. `ROUND (20.723, 2) => 20.73`

MOD – Returns the remainder of a division

e.g. `MOD (20, 9) => 2`

COUNT – Returns the number of rows that match a criteria

e.g.

```
SELECT COUNT(*) AS NumberOfOrders
FROM Orders;
```

3. Date Functions

The default date display format is DD-MON-YY

Example:

```
SELECT last_name, hire_date
FROM employees
WHERE hire_date < '09-MAY-94';
```

SYSDATE – Function that returns current date (use table as placeholder)

e.g.

```
SELECT SYSDATE FROM employees
```

MONTHS_BETWEEN – Number of months between two dates

e.g. `MONTHS_BETWEEN(date1, date2)`

ADD_MONTHS – Add calendar months to date

NEXT_DAY – Returns first specified weekday that is greater than a date

e.g. `NEXT_DAY ('27-OCT-14', FRIDAY) => 31-OCT-14`

LAST_DAY – Returns last day of month of a specified date

4. Conversion Functions

Oracle can convert VARCHAR2 to NUMBER and DATE and vice versa, referred to as the **format model**.

Date elements:

YYYY – Full year numerically

YEAR – Year spelled out

MM – 2 digit value for month

MONTH – Full name of month

MON – Three letter abbrev. for month

DY – Three letter abbrev. for day of week

DAY – Full name of day

DD – Numeric day of month

Time elements:

9 – Represents a number

0 – Forces a zero

\$ – Places a floating dollar sign

L – Uses local currency symbol

. – Prints a decimal

, – Prints a comma to indicate thousands

Date can be formatted DD-MON-YYYY'

Time can be formatted HH24: MI : SS

Currency can be formatted '\$9,999.99

TO_CHAR (number, 'format_model') **N.B.** format_model is the INPUTTED variable

```
e.g.  SELECT TO_CHAR (salary, '$99, 999.00') SALARY
      FROM employees
      WHERE last_name = 'Kiernan';
```

TO_NUMBER (char [, 'format_model'])

```
e.g.  TO_NUMBER ('1900.83', '9999.99') => 1900.83
```

TO_DATE (char[, 'format_model'])

```
e.g.  TO_DATE ('2014/10/27', yyyy/mm/dd) => October 27, 2014
```

5. General Functions

These functions work with any data type and pertain to using nulls, data types must match:

NVL (string1, val_if_nul) – Returns x if the string is null, else returns a value specified by string1

```
e.g.  SELECT NVL (supplier_city, 'n/a')
      FROM suppliers;
```

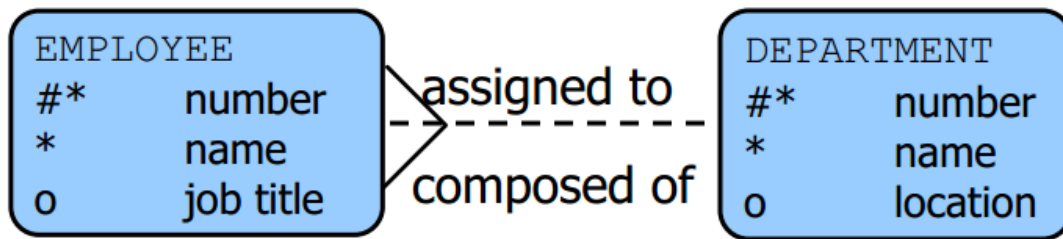
NVL2 (string1, value_if_not_null, val_if_nul) – Expands on nvl, prints Sal column if null

```
e.g.  SELECT last_name, salary, commission_pct, NVL2('Sal+Com', 'Sal')
      FROM suppliers;
```

NULLIF (expression1, expression2) – Compares both expressions and if equal returns null otherwise 1st.

COALESCE (expr1, ... expr_n) – Returns the first non-null expression in the list. If all are null returns null.

Entity Relationship Model



Scenario: "Assign one or more employees to a department"

"Some departments do not yet have assigned employees"

Table contains data that describes one entity.

Primary key uniquely identifies each row of data in a table.

Foreign key can relate data from multiple tables.

Table name: EMPLOYEES

EMPLOYEE_ID	FIRST_NAME	LAST_NAME	DEPARTMENT_ID
174	Ellen	Abel	80
142	Curtis	Davies	50
102	Lex	De Haan	90
104	Bruce	Ernst	60
202	Pat	Fay	20
206	William	Gietz	110

...

Primary key

Foreign key

Table name: DEPARTMENTS

DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
10	Administration	200	1700
20	Marketing	201	1800
50	Shipping	124	1500
60	IT	103	1400
80	Sales	149	2500
90	Executive	100	1700
110	Accounting	205	1700
190	Contracting		1700

Primary key

Rows should be identified by a primary key, i.e. *EMPLOYEE_ID*.

Foreign key is a primary key of another table that shares a common field, i.e. *DEPARTMENT_ID*.

Entity is a thing or object of interest to the model, e.g. person, place, object, event or concept.

Instance of an entity is a single occurrence of an entity such as the student. Alex Kiernan is an instance of the entity Student. There may be hundreds of instances of an entity.

Attributes are pieces of data that describe an entity such as StudentName = "Alex Kiernan".

Keys identify instances. An entity can have more than one key.

Primary key of an entity is an attribute or set of attributes that uniquely identify each instance of the entity and must have a value and one that doesn't change.

Relationships






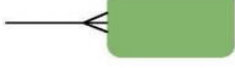
A **relationship** is a link between two entities, e.g. A Customer 'places' an Order.

Relationships are related in **quantitative terms**, so the number of instances of one to the number of instances of the second. These relationships are **bi-directional** so the **cardinality** need to be defined in both directions.

E.g. A customer places one or more orders, A student enrolls on one or more modules.

Cardinality types:

- **zero-to-one relationship**: any occurrence of the first entity is related to at maximum one occurrence of the second entity.
E.g. Each Alex is allowed a bicycle, however they are only allowed a maximum of one. They can choose to not have a bicycle for whatever insane reason.
- **one-to-one relationship**: any instance of the first entity is related to at maximum one and only one instance of the second entity.
E.g. If we look at it in reverse a bicycle cannot exist without an Alex, so each bicycle must belong to an Alex and can only belong to one Alex. So for example, the entity bikeSaddle cannot exist without the entity AlexName.
- **one-to-many relationship**: any instance of the first is related to at least one instance of the second entity but can be related to many instances.
E.g. Thankfully, all Alex's are required to have a bicycle. They must always be in possession of a bicycle. They are allowed to have more than one bicycle.
- **many to many relationship**: two entities are related to each other by one to many relationships.
E.g. Each Alex must join a cycling club and therefore each cycling club has many Alex's.

CARDINALITY INTERPRETATION	MINIMUM INSTANCES	MAXIMUM INSTANCES	GRAPHIC NOTATION
Exactly one (one and only one)	1	1	 — or — 
Zero or one	0	1	
One or more	1	many (>1)	
Zero, one, or more	0	many (>1)	
More than one	>1	>1	

A **Weak entity** is an entity that resolves the problem of where to put information related to the many to many relationship between two entities. Such as who voted what on the decision of how many members a cycling club can have? Does this **Vote** info belong in the **Alex** or **Club** entity? The primary key of the weak entity is the combination of the primary keys of the entities in the relationship it is resolving.

Identifying and non-identifying relationships:

- Identifying relationship is when the existence of an instance of an entity in a child table depends on a row in a parent table. The primary key of the parent therefore must be part of the child's primary key.

Example: Suppose a Person has one or more phone numbers. If they had one we could store it in Person but since we have multiple we store it in PhoneNumbers whose primary key includes PersonID which references the Person table.

- Non-identifying relationship: A relationship where the primary key attributes of the parent aren't shared with the primary key attributes of the child.

Example: Account(AccountID, AccountNum, AccountTypeID)

Parent

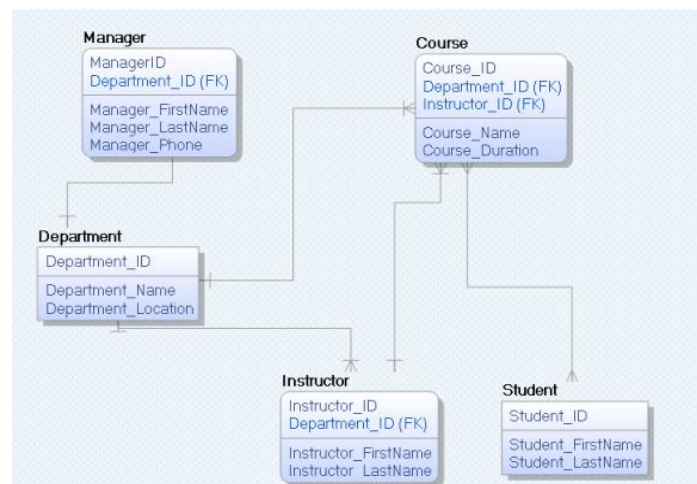
AccountType(AccountTypeID, Code, Name, Description)

Child doesn't rely on Parent

Steps for Model

Example:

- A college contains many departments.
- Each department can offer any number of courses.
- Many instructors can work in a department.
- An instructor can work only in one department.
- For each department there is a Manager.
- Each manager can manage only one department.
- Each instructor can take any number of courses.
- A course can be taken by only one instructor.
- A student can enrol for any number of courses.
- Each course can have any number of students.



Step 1: Identify the entities – Department, Course, Instructor, Student, Manager.

Step 2: Identify the keys – Use ID tags for each entity. i.e. Course_ID is the key attribute for Course.

Step 3: Identify the attributes – Department entity would have id, name and location...

Step 4: Identify the relationships:

- One department offers many courses but one particular course can only be offered by one department, hence the cardinality is a (1:N) relationship.
- One department has multiple instructors but instructor only has one department. Again a (1:N).
- One department has only one manager and one manager can only manage one department. This loving relationship is (1:1).
- One course can be enrolled by many students and one student can enroll in many courses. This is many to many (M:N).
- One course is taught by only one instructor but one instructor teaches many courses causing a many to one relationship (N:1).

Tables

CREATE TABLE - Creating a basic table involves naming the table and defining its columns and each column's data type.

e.g.

```
CREATE TABLE table_name (  
    column1 data type,  
    column2 data type,  
    column3 data type,  
    .....  
    columnN data type,  
    PRIMARY KEY( one or more columns )  
);
```

ALTER TABLE

e.g.

```
ALTER TABLE table_name  
ADD column_name datatype  
DROP COLUMN column_name  
MODIFY column_name datatype
```

Data types

- **CHAR** denotes a single character.
CHAR(n) denotes a n-long fixed length string. The remaining characters not filled are padded with blanks.
VARCHAR2(n) denotes a variable length string where n is the max. length.
Always surround CHAR values with single quotes (also DATE)!
- **NUMBER(n)** denotes a number n digits long. Can store all types of numbers such as negative, float etc.
NUMBER(n, m) where n is the total number of digits and m is the decimal.
Numeric values obviously don't need quotes.
- **DATE** denotes a **Date AND Time** field correct to a second. Dates are denoted DD-MMM-YY.
SYSDATE is a variable holding the current system date.
TIMESTAMP(n) extends on DATE by allowing fractions of a second for whatever reason to max n digits.
- **Binary:**
SQL doesn't allow Boolean data types so you can use something like CHAR(1).

Constraints

Constraints are rules that data must follow to get into the table in order to preserve the integrity of the data. They also prevent deletion of a table if there are dependencies.

Constraints include:

- Key constraints such as the **PRIMARY KEY** and the **FOREIGN KEY** with the table and primary key it references.
- Value constraints to define if **NOT NULL** values are allowed and if **UNIQUE** values are required.


Constraints can be created at the same time as the table is created or after and at the table level or column level.

Primary key constraint:

Name of constraint

1. Column level:

```
CREATE TABLE employees (  
    employee_id NUMBER(6)  
    CONSTRAINT emp_emp_id_pk PRIMARY KEY,  
    first_name VARCHAR2(20),  
);
```



2. Table Level:

```
CREATE TABLE employees (  
    employee_id NUMBER(6),  
    first_name VARCHAR2(20),  
    ...  
    job_id VARCHAR2(10) NOT NULL,  
    CONSTRAINT emp_emp_id_pk PRIMARY KEY (EMPLOYEE_ID)  
);
```

Foreign key constraint:

Note: The referenced table must be created first otherwise an error will occur.

```
CREATE TABLE employees (  
    employee_id NUMBER(6),  
    last_name VARCHAR2(25) NOT NULL,  
    email VARCHAR2(25),  
    salary NUMBER(8,2),  
    commission_pct NUMBER(2,2),  
    hire_date DATE NOT NULL,  
    ...  
    department_id NUMBER(4),  
    CONSTRAINT emp_dept_fk FOREIGN KEY (department_id) REFERENCES departments  
    (department_id),  
    CONSTRAINT emp_email_uk UNIQUE (email)  
);
```

Insert

You can do either a FULL INSERT that inserts a value for every column in a table or a PARTIAL INSERT that only inserts specified values.

	↕ COLUMN_NAME	↕ DATA_TYPE	↕ NULLABLE	DATA_DEFAULT	↕ COLUMN_ID	↕ COMMENTS
1	MNO	NUMBER (6, 0)	Yes	(null)	1	(null)
2	MCURRENT	CHAR (1 BYTE)	Yes	(null)	2	(null)
3	MNAME	VARCHAR2 (50 BYTE)	Yes	(null)	3	(null)
4	MDATEOFBIRTH	DATE	Yes	(null)	4	(null)
5	MAMOUNTOWED	NUMBER (7, 2)	Yes	(null)	5	(null)
6	MEMAIL	VARCHAR2 (50 BYTE)	Yes	(null)	6	(null)
7	MNOCHILDREN	NUMBER (2, 0)	Yes	(null)	7	(null)
8	MRENEWDATE	DATE	Yes	(null)	8	(null)

Also denoted as: ClubMember (Mno, Mcurrent, Mname, MDateOfBirth, MAmountOwed, MEMail, MNoChildren, MRenewDate);

Full insert:

```
INSERT INTO ClubMember VALUES (  
    654321,          -- MNO  
    'Y',             -- MCurrent  
    'Fred',          -- Mname  
    '23-FEB-1940',   -- MDateofbirth  
    0,               -- Mamountowed  
    'FRED@CLUB.IE', -- MEMail  
    2,               -- MNoChildren  
    '01-DEC-08'      -- MRenewalDate  
);
```

Partial Insert:

```
INSERT INTO ClubMember (Mno, Mcurrent, Mname, MDateOfBirth)  
VALUES (654322, 'Y', 'Jane', '13-Mar-88');
```

COMMIT – Don't forget to commit any data inserts so they persist, insert **COMMIT** at end of inserts.

Joins allow the selection of data from multiple data using common key relationships.

e.g.

```
SELECT table1.column, table2.column
FROM table1
JOIN table2 USING (column_name);
```

OR

```
JOIN table2
ON (table1.column_name = table2.column_name);
```

There are 2 main types of join:

- **Inner join** – Retrieves all matching fields in joined tables.
- **Outer join** – Retrieves fields in one table and matching fields in the second table only if they exist.

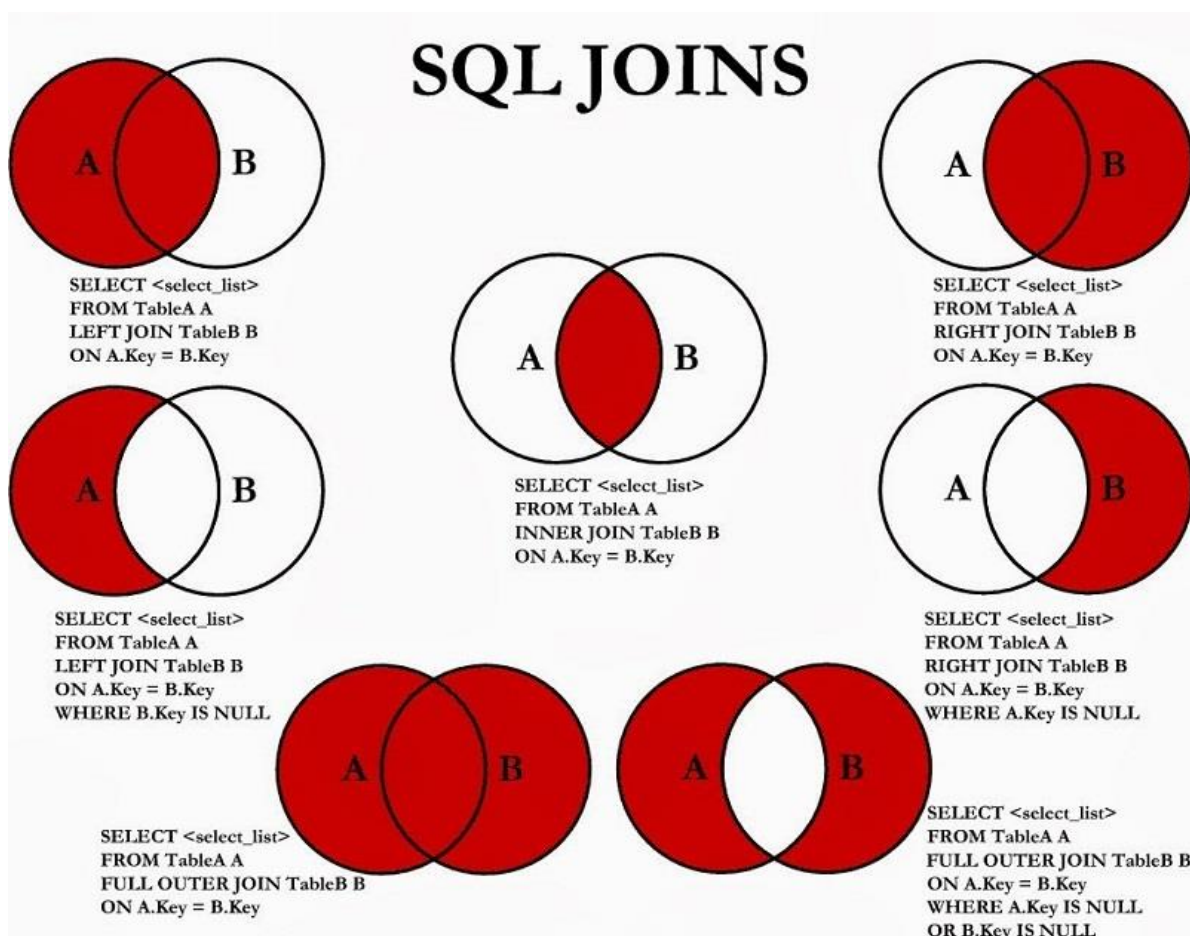
USING clause: Specifies which columns to test for equality when column names match in both tables.

ON clause: When column names don't match.

Natural join: Simple join that presumes fields have the same name, doesn't use an equality statement (avoid!).

Full outer join: Returns the results of all rows from the left and right table.

Left join: Retrieves all specified records from the LEFT table but for any specified columns on the RIGHT with no related records, inserts NULL. Same idea for **Right join**.



Sub queries are used to add even more structure to queries. They are normally used in the **WHERE** clause.

```
SELECT last_name
FROM employees
WHERE salary >
    (
        SELECT salary
        FROM employees
        WHERE last_name = 'Abel');
```

You can use a sub query in conjunction with:

Operators: =, >, <, <>

Group operators in a sub query: **SELECT** MIN(salary) FROM employees

ALL – Holds all results of a sub query and can be used to test against it.

```
SELECT employee_id, last_name, job_id, salary
FROM employees
WHERE salary < ALL
    (
        SELECT salary
        FROM employees
        WHERE job_id = 'IT_PROG')
AND job_id <> 'IT_PROG';
```

This query returns all employees who are not IT programmers with a salary less than all of the IT programmers

IN – Tests for a column in a sub query.

```
SELECT last_name "Last Name", first_name "First Name"
FROM employee
WHERE employee_id IN
    (
        SELECT employee_id
        FROM job_history
        WHERE department_id = 50);
```

INSERT – Specified columns can be inserted in other tables.

```
INSERT INTO sales_reps(id, name, salary, commission_pct)
SELECT employee_id, last_name, salary, commission_pct
FROM employees
WHERE job_id LIKE '%REP%';
```

UPDATE – Update a row based on another row.

```
UPDATE employee
SET job_id = ( SELECT job_id
               FROM employees
               WHERE employee_id = 205),
    salary = ( SELECT salary
               FROM employees
               WHERE employee_id = 205)
WHERE employee_id = 114;
```

FIN!

Databases 2

Alter Session – Changes schema to specified.

```
ALTER SESSION SET current_schema = akiernan;
```

Sequences – Auto increments a number, *nextval* function increments.

```
CREATE SEQUENCE student_seq START WITH 1;

insert into student (studentno, prog_code, stage_code)
values (student_seq.nextval, 'DT228', 3);
```

Locking – Users can be given different privileges for specified tables. Can also revoke privileges.

```
GRANT ALL ON Academic TO AHAKIM;
```

PL/SQL combines SQL with procedural languages in an Oracle database. It is designed to take and return parameters.

```
DECLARE
    V_Sno char(10);
    V_Sname varchar2(40);
BEGIN
    ...
END;
```

Data Types

- CHAR/VARCHAR2
- NUMBER/DATE
- BOOLEAN

Input/Output

Substitution variables are used for **input** and the command for 'DBMS_OUTPUT.PUT_LINE' for **output**.

```
set serveroutput on
DECLARE
    V_SNO STUDENT.STUDENTNO%TYPE;
    V_SNAME STUDENT.STUDENTNAME%TYPE;
BEGIN
    V_SNO := '&Enter_Student_Number';
    SELECT STUDENTNAME INTO V_SNAME FROM STUDENT
    WHERE STUDENTNO = V_SNO;
    DBMS_OUTPUT.PUT_LINE ('Student number ' || V_SNO || ' is called ' || V_SNAME || '.');
END;
```

We can declare an attribute to have the **same data type** as the target rather than hard coding it.

```
DECLARE
    V_SNO STUDENT.STUDENTNO%TYPE;
    V_SNAME STUDENT.STUDENTNAME%TYPE;
BEGIN
    V_SNO := '&Enter_Student_Number';
    ...
```

Exceptions – When an error occurs, each exception is checked until it finds a match you declared.

```
...
EXCEPTION
WHEN NO_DATA_FOUND THEN
    DBMS_OUTPUT.PUT_LINE ('There is no student with number' || V_SNO);
```

Locking

Transaction is the term for a sequence of operations that occur when an INSERT, UPDATE or DELETE takes place.

- The row is locked and other sessions cannot see the changes that are being made.
- The lock holds until the transaction sessions issues a **COMMIT** or **ROLLBACK**.

commit – Makes all changes since the beginning of a transaction permanent.

rollback – Undoes all changes since the beginning of a transaction.

Random Stuff

GROUP BY – Used in conjunction with an aggregate function.

```
select Booking_name from Table_Booking
where Booking_name is not null
group by Booking_name;
```

HAVING – Aggregate functions cannot be used with the WHERE keyword.

```
select Booking_name from Table_Booking
where Booking_name is not null
group by Booking_name
having count(Booking_name) > 2;
```