DBS301 - Database Design II and SQL Using Oracle

Robert Stewart

rob.stewart@senecacollege.ca

Table of Contents

Installing, Configuring and Connecting to an Oracle Instance

Connecting from a Seneca Computer (ie: from inside the school)

1. Go to the DOS shell (start/run/cmd)
2. Type:

   sqlplus yourUsernameHere/yourPasswordHere@neptune

Connecting to an Oracle instance at Seneca but remotely (ie: connecting to Seneca from a home computer)

1. Install either the Oracle 11g client or Oracle 11g Express (see next section for installing Oracle Express)
2. Look for a file called tnsnames.ora (it should be in the directory where you installed your Oracle client or Oracle Express.)

   Look in

   ..\ app\oracle\product\11.2.0\server\network\ADMIN

   or something like this.

3. Add the following to the end of the file or if the file does not exist then create it and add the following:

```
neptune =
(DESCRIPTION =
(ADDRESS_LIST =
(ADDRESS = (PROTOCOL = TCP)(HOST = zenit.senecac.on.ca)(PORT = 1523))
(ADDRESS = (PROTOCOL = TCP)(HOST = zenit.senecac.on.ca)(PORT = 1524))
(LOAD_BALANCE = yes))
(CONNECT_DATA =
(SERVER = DEDICATED)
(SERVICE_NAME = neptune)
(FAILOVER_MODE =
(TYPE = SELECT)
(METHOD = BASIC)
(RETRIES = 180)
(DELAY = 5))))
```

4. Go to the DOS shell (start/run/cmd)
5. Type:

   sqlplus yourUsernameHere/yourPasswordHere@neptune

Installing Oracle Express on your Home Computer

1. **IMPORTANT**: When installing Oracle Express 11g you will be asked for a password (DO NOT forget this password, you will need it)
2. After installation you will have an Oracle Express 11g option available in your Windows start menu.
   a. The 'Start Database' option will start the Oracle Express services that are needed to run Oracle.
   b. The 'Stop Database' option will stop the Oracle Express services and remove them from memory.
3. To log into Oracle
   i. Start the Oracle Services (from the Windows Start menu as above)
   ii. Select the 'Run SQL Command Line' option (from the Windows Start menu as above)
   iii. Type:

      connect system;

   iv. Enter your password that you set when installing Oracle Express

Oracle 11g Configuration on Your Home Computer if the Above does not Work

1. CONNECT / as sysdba;
2. CREATE USER yourUserName IDENTIFIED BY yourPassword;
3. GRANT dba TO yourUserName WITH ADMIN OPTION;
4. You may now Login with the username and password you created above
   - Connect yourUserName
   - Enter yourPassword when prompted

How to Access Oracle Express if You Have Forgotten Your Password (this may or may not work)

1. Go to the DOS shell (start/run/cmd)
2. Type:

   sqlplus /nolog

3. Type:

   connect / as sysdba;

4. Type:

   ALTER USER system IDENTIFIED BY enter_a_new_password_here;

5. Now connect as the system user
6. Type:

   connect system;

7. Enter your password

Loading an External SQL Script File

1. From the Oracle command line type the following (replacing path and filename with the appropriate parameters)

```
@path\filename;
```

Useful System Views

View the Basic Structure of a Table

Type:

```
DESCRIBE yourTableName
```

USER_TABLES View

To get more information on the tables you can query the USER_TABLES view

Column	Returns
table_name	Name of the table
tablespace_name	Name of the tablespace in which the table is stored. A tablespace is an area used by the database to store objects such as tables.
temporary	Whether the table is temporary. This is set to Y if temporary or N if not temporary

Example:

```
SELECT table_name, tablespace_name, temporary
FROM user_tables
WHERE table_name = 'TBLVENDOR';
```

USER_TAB_COLUMNS View

To get more information on columns in a table you can query the USER_TAB_COLUMNS view

Column	Returns
table_name	Name of the table
column_name	Name of the column
data_type	Data type of the column
data_length	Length of the data
data_precision	Precision of a numeric column if a precision was specified for the column.
data_scale	Scale of a numeric column

Example: (notice the use of the COLUMN statement to change the formatting of the output)

```
COLUMN column_name FORMAT a15;
COLUMN data_type FORMAT a10;
SELECT column_name, data_type, data_length, data_precision, data_scale
FROM user_tab_columns
WHERE table_name = 'TBLPRODUCT';
```

USER_CONS_COLUMNS View

To get information on the constraints for a column you can query the USER_CONS_COLUMNS view.

Column	**Returns**
owner	Owner of the constraint
constraint_name	Name of the constraint
constraint_type	Constraint type (P, R, C, U, V, or O).
table_name	Name of the table on which the constraint is defined.
status	Constraint status (ENABLED or DISABLED).
deferrable	Whether the constraint is deferrable (DEFERRABLE or NOT DEFERRABLE).
deferred	Whether the constraint is enforced immediately or deferred (IMMEDIATE or DEFERRED).

Example:

```
COLUMN column_name FORMAT a15;
SELECT constraint_name, column_name
FROM user_cons_columns
WHERE table_name = 'TBLPRODUCT'
ORDER BY constraint_name;
```

Database Design

Database Terminology

Database: A shared collection of logically related data stored in a structured format.

Table (Entity): A two-dimensional arrangement of rows and columns that contains data.

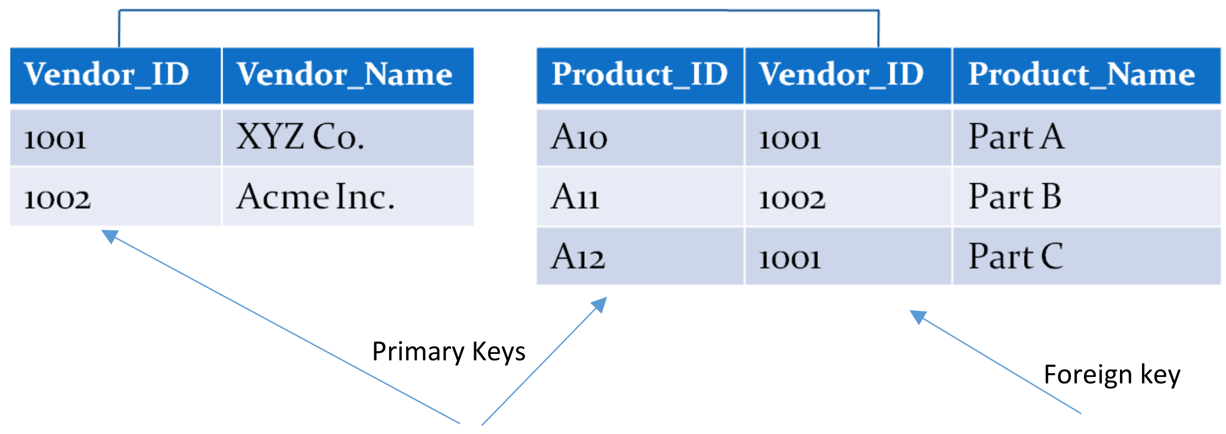
Student_ID	Student_lName	Student_fName	Student_Phone
1001	Smith	Jane	416-555-7656
1002	Jones	Bill	905-555-4736
1003	Jackson	Mary	416-555-2134

Record (Row or Tuple): Each record contains all the information about a single member or item.

Field (Column or Attribute): A record comprises of fields where each field is a single piece of information relating to the record for example your STUDENT record may contain individual fields for first name, last name, student number, date of birth, etc. All records in a table have the same structure and so contain the same fields but obviously each field has its own unique data.

Primary Key: A field or combination of fields that uniquely identifies a record in a table and has a constraint of NOT NULL. A primary key is the minimal fields needed to uniquely identify a record in a table.

Foreign Key: A field that relates to a primary key in an associated table or to a Unique column in another table. A foreign key can sometimes contain NULL values depending upon the constraints/integrity set on the field. A PK and FK are used to “link” tables together.



Vendor_ID	Vendor_Name
1001	XYZ Co.
1002	Acme Inc.

Product_ID	Vendor_ID	Product_Name
A10	1001	Part A
A11	1002	Part B
A12	1001	Part C

Composite Primary Key (compound or concatenated key): A key that consists of 2 or more fields that uniquely identifies a record.

Candidate Key: A field or combination of fields which can uniquely identify a record in a table. A table may have one or more candidate keys and one of them can be used as a Primary Key of the table.

For Example, In an Employee Table, we may have columns like Employee ID, Employee Name, and Employee SSN. We can consider either Employee ID or Employee SSN as Candidate Key's.

Database Integrity

Entity Integrity: Refers to the fact that there are no duplicate rows in a table.

In an Employee table we would not want to store the same information for one employee more than once.

Domain Integrity: Ensures valid entries in a column by restricting the type, range or format of the data values possible. Ie: Setting constraints and data types.

In a Student table we would ensure a student is designated either M or F for male/female respectively. We would therefore set the data type for the field to Char(1), NOT NULL, and constraints of M or F only.

Referential Integrity: Where row data is shared between more than one record this ensures that the data cannot be deleted while there are still dependent records.

In an Invoicing database we will not allow the system to delete a Customer record if that customer has any Invoices.

User-Defined Integrity: You may have your own business rules that do not fall into the other data integrity categories, these can be enforced by "User-Defined Integrity"

These would be based on the business rules defined and the function of the database.

Relationship Types

Definition: A relationship describes an association among tables.

Types: There are 3 types of relationships. 1:1, 1:M, M:N

1. One-to-One (1:1): A retail company may require that each of its stores be managed by a single employee. In turn, each store manager, who is an employee, manages only a single store. The relationship between EMPLOYEE manages STORE is 1:1

2. Many-to-Many (M:N): A student can take many classes and each class can have many students. The relationship between STUDENT takes CLASS is M:N.

3. One-to-Many (1:M): A painter paints many paintings and each painting is painted by one painter. The relationship between PAINTER paints PAINTING is 1:M

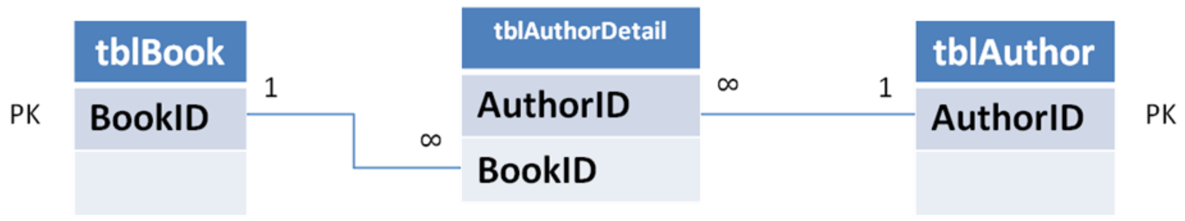
A proper database design will ALWAYS remove the M:N relationship and convert it to two 1:M relationships by the use of a bridge table. Note: new software will allow a M:N but then the design will not be portable and it is very difficult to manage.

Business Rules

Definition: A brief and precise description of a policy, procedure, or principle within the organization.

Why: Business rules are used to define entities (tables), attributes (fields), relationships, and constraints.

How many: Each set of related tables need at least two rules to define their relationship type.



1. Each Author can write many Books (1:M)
2. Each Book can be written by many Authors (1:M)

Note: With the 1st business rule the Book table has a Many relationship. With the 2nd business rule the Author table also has a Many relationship. We therefore have a many to many (M:N) relationship between the tables Book and Author. We remove the M:N relationship by inserting a bridge table between the two tables.

Bridge tables are used to eliminate M:N relationships. The minimum number of fields in any bridge table is the two Primary Keys from the related tables.

How to determine them: The main source of business rules come from company managers, policy makers, department managers, written documentation, and direct interviews with end users (end user sources are less reliable).

Naming Conventions

When designing your database, like any programming, you should get used to using consistant naming conventions. The following are some naming conventions in common use – some are required and some are optional.

1. Precede Objects with prefixes that describe the object. (**optional**)
    i. Tables – tblName
    ii. Forms – frmName
    iii. Queries – qryName
    iv. Reports – rptName
    v. Controls - intName, binName, cboName, etc.

2. Precede field names with the table name minus the prefix (**optional**)
    i. StudentFName (table name is tblStudent)
    ii. EmployeeSIN (table name is tblEmployee)

3. Do not use reserved words. ie: Date, Integer (**required**)

4. Do not use spaces – use underscore. Ie: Client_fName or ClientfName (**optional**)

5. Only use alphanumeric characters (**optional**)

6. Always start an object name with a letter NOT a number (**required**)

Normalization

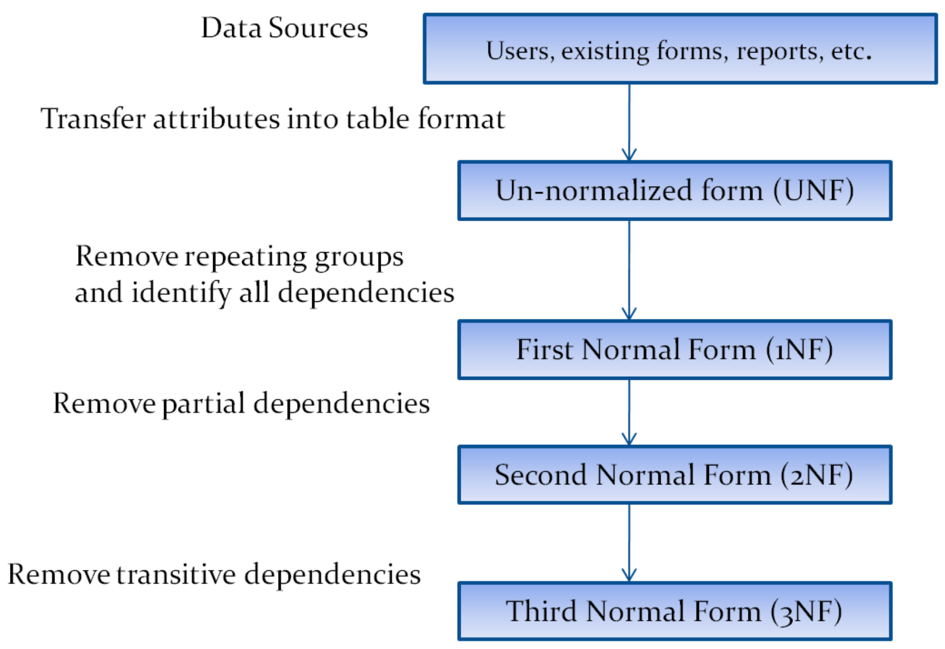
Definition: is a process for evaluating and correcting table structures to minimize data redundancies, thereby reducing the likelihood of data anomalies.

Goal: is to reduce redundancies, generate fewer data anomalies, and improve efficiency.

Concepts:

- Normalization works through a series of stages called normal forms.
- The first 3 stages are known as first normal form (1NF), second normal form (2NF), and third normal form (3NF).
- Generally each successive normal form produces more tables in your schema thus producing more joins.
- The more relational joins in a schema the more resources are required by the database system to respond to end-user queries.
    - Thus a good design must also consider end-user demand for fast performance.
- Reducing a normal form to a previous normal form is called denormalization. (The price you pay for increased performance (by denormalization) is greater data redundancy thereby more data anomalies.)

Illustration of the Process


Data Sources
Users, existing forms, reports, etc.
Transfer attributes into table format
Un-normalized form (UNF)
Remove repeating groups
and identify all dependencies
First Normal Form (1NF)
Remove partial dependencies
Second Normal Form (2NF)
Remove transitive dependencies
Third Normal Form (3NF)

A Properly Normalized Database Will Result In:

- Each table represents a single subject
- No data item will be unnecessarily stored in more than one table. (Ensures data is updated in only one place)
- All attributes in a table are dependent on the primary key and only the primary key
- All attributes are atomized

The Normalization Process

UNF

Process:

1. Create column headings (ignoring any calculated fields)
2. Enter sample data into table
3. Identify a key for table (and underline it)
4. Remove duplicate data

1NF

Rule: Remove any repeating attributes to a new table

Process:

1. Identify repeating attributes
2. Remove repeating attributes to a new table together with a copy of the key from the UNF table
3. Assign a key to the new table (and underline it). The key from the unnormalised table always becomes part of the key of the new table. A compound key is created. The value for this key must be unique for each entity occurrence.

2NF

Rule: Remove any non-key attributes that only depend on part of the table key to a new table

Ignore tables with a) a simple key or b) with no non-key attributes (these go straight to 2NF with no conversion)

Process:

1. Take each non-key attribute in turn and ask the question
   - is this attribute dependent on one part of the key?
2. If yes, remove attribute to new table with a **copy** of the **part** of the key it is dependent upon. The key it is dependent upon becomes the key in the new table. Underline the key in this new table.
3. If no, check against other part of the key and repeat above process.
4. If still no, ie not dependent on either part of key, keep attribute in current table.

3NF

Rule: Remove to a new table any non-key attributes that are more dependent on other non-key attributes than the table key

Ignore tables with zero or only one non-key attribute (these go straight to 3NF with no conversion).

Process:

- If a non-key attribute is more dependent on another non-key attribute than the table key
- Move the **dependent** attribute, together with a **copy** of the non-key attribute upon which it is dependent, to a new table
- Make the non-key attribute, upon which it is dependent, the key in the new table. Underline the key in this new table.
- **Leave** the non-key attribute, upon which it is dependent, in the original table and mark it as a **foreign key** (*).

Example:

Needs to be Added

Basic SQL Statements

The following section will cover the basic SQL Statements

The basic SQL statements can be categorized into sections.

SELECT
INSERT
UPDATE
DELETE
	Data Manipulation Language (DML)
	
	Retrieves data from the database, enters new rows, changes existing data, removes unwanted rows, inserts or changes rows based on a conditional match.

CREATE ALTER DROP RENAME	**Data Definition Language (DDL)** Creates, changes, and removes data structures.
COMMIT ROLLBACK SAVEPOINT	**Transaction Control** Manages the changes to data made by the DML statements. Changes to data can be grouped together into logical transactions.

Double Quotes vs Single Quotes

- Single-quotes are used to enclose string literals (and, in recent versions, DATE literals).
- Double-quotes are used to enclose identifiers (like table name, column names and alias names)
  - Double Quotes are only needed if the names have spaces in them.

SELECT Statement

A SELECT statement retrieves zero or more rows from one or more database tables or database views. The result from running a SELECT statement is know as a Return Set.

Basic SELECT Syntax

Anything enclosed in [] is optional in the SELECT statement (NOTE: when writing the SELECT statement DO NOT include the [])

SELECT *column list*
FROM *table(s)*
[WHERE clause]
[GROUP BY clause]
[HAVING clause]
[ORDER BY clause];

Important Aspect of SQL: The logical order in which the various SELECT query clauses are evaluated is different than the keyed-in order. The keyed-in order of a query's clauses is:

1. SELECT
2. FROM
3. WHERE
4. GROUP BY
5. HAVING
6. ORDER BY

But the logical query processing order is:

1. FROM
2. WHERE
3. GROUP BY
4. HAVING
5. SELECT -- column aliases created here
6. ORDER BY

This is why a column alias cannot be referenced in a WHERE clause but can be referenced in the ORDER BY clause.

Simple SELECT Statement

The following SELECT statement displays all rows and all columns from the tblProduct table.

SELECT * FROM tblProduct;

The following example displays all rows but only the product description and cost column from the tblProduct table.

SELECT ProductDesc, ProductCost FROM tblProduct;

WHERE Clause

You can specify which rows to return using a WHERE clause.

The WHERE clause is always placed after the FROM clause.

The following example returns only the row from the tblProduct table where the ProductID = 5

SELECT * FROM tblProduct WHERE ProductID = 5;

The following example returns only the product description and cost from the rows where the cost of the product is >25.

SELECT ProductDesc, ProductCost FROM tblProduct WHERE ProductCost>25;

Comparison Operators

Operator	Description	Example
=	Equal	SELECT * FROM tblProduct WHERE ProductID = 2;
<> Or !=	Not equal	SELECT * FROM tblProduct WHERE ProductID <> 2;
<	Less than	SELECT * FROM tblProduct WHERE ProductCost < 10;
>	Greater than	SELECT * FROM tblProduct WHERE ProductCost > 10;
<=	Less than equal to	SELECT * FROM tblProduct WHERE ProductCost <= 10;
>=	Greater than equal to	SELECT * FROM tblProduct WHERE ProductCost >=10;
IN()	Returns 1 if expr is equal to any of the values in the IN list, else returns 0.	SELECT * FROM tblProduct WHERE ProductID IN(2,4, 5) ; Returns 1 if any ProductID = 2,4, or 5 Returns 0 if no ProductID = 2,4, or 5

SQL Operators

The SQL operators allow you to limit rows based on pattern matching of strings, lists of values, ranges of values, and null values. The SQL operators are listed below:

Operator	Description	Opposite Operator
LIKE	Matches patterns in strings	NOT LIKE
IN	Matches lists of values	NOT IN
BETWEEN	Matches a range of values (inclusive)	NOT BETWEEN
IS NULL	Matches null values	IS NOT NULL

LIKE Operator and Wildcards

You use the LIKE operator in a WHERE clause to search a string for a pattern. You specify patterns using a combination of normal characters and the following two wildcard characters:

- Underscore (_) Matches one character in a specified position
- Percent (%) Matches any number of characters beginning at the specified position

Examples:

The following query uses the LIKE operator to return all students whose last name starts with the letter ‘S’.

SELECT * FROM tblStudent WHERE StudentLName LIKE ‘S%’;

The following example uses the NOT LIKE operator to return all students not returned from the previous query.

SELECT * FROM tblStudent WHERE StudentLName NOT LIKE ‘S%’;

The following query combines the underscore and percent wildcard to return students who have the letter ‘a’ as the 2nd character in their last name.

SELECT * FROM tblStudent WHERE StudentLName LIKE ‘_a%’;

ESCAPE with the LIKE Clause

Note: If you need to search for actual underscore or percent characters in a string, you can use the ESCAPE option to identify those characters.

For example, suppose we had some data as follows:

```
PromotionDesc
------------------------------
10% off Samsung 75 inch OLED TV
$100 off Panasonic 65 Inch LED TV
20% off Sharp LED 42 inch LED TV
80% off all Apple TV’s
$250 off BOSE Sound System
```

Now what if we wanted to return all the promotions that are offering a % off the product and not a fixed dollar value.

If we tried

SELECT PromotionDesc FROM tblPromotion WHERE PromotionDesc LIKE '%%%';

SQL would just treat all the % as wildcards and return all rows.

What we really want is to somehow indicate that we are LOOKING for an actual % symbol in the Promotion Decription.

This is where the ESCAPE option with the LIKE clause comes into play.

SELECT PromotionDesc FROM tblPromotion WHERE PromotionDesc LIKE '%\%%' ESCAPE '\';

- The character after the word ESCAPE tells the database how to differentiate between characters to search for and wildcards, and in the example the backslash character (\) is used.
- The first % is treated as a wildcard and matches any number of characters
- The second % is treated as an actual character to search for
- The third % is treated as a wildcard and matches any number of characters.

Example: The above query returns any rows that contain a % character.

```
PromotionDesc
------------------------------
10% off Samsung 75 inch OLED TV
20% off Sharp LED 42 inch LED TV
80% off all Apple TV’s
```

BETWEEN Operator

Use the BETWEEN operator in a WHERE clause to retrieve the rows whose column value is in a specified range. *The range is inclusive*.

Example: The following query returns all rows where the product selling price is between 20 and 25 dollars (including the 20 and 25).

SELECT ProductName, ProductSell FROM tblProduct WHERE ProductSell BETWEEN 20 AND 25;

LOGICAL Operators

Operator	**Description**
x AND y	Returns true when both x and y are true
x OR y	Returns true when either x or y is true
NOT x	Returns true if x is false, and returns false if x is true
Operator Precedence If you combine AND and OR in the same expression, the AND operator takes precedence over the OR operator ("takes precedence over" means it's executed first). The comparison operators take precedence over AND. You can override the default precedence by using parentheses to indicate the order in which you want to execute the expressions.	

Example: The following query returns the rows from the student table where either of the following two conditions is true:

- The DOB column is greater than January 1, 1970.
- The StudentID column is less than 2 and the phone column has 1211 at the end.

SELECT * FROM tblStudent
WHERE StudentDOB > '01-JAN-1970' OR StudentID < 2 AND StudentPhone LIKE '%1211';

```
StudentID     StudentFName          StudentLName         StudentDOB     StudentPhone
_____________ _____________________ ____________________ ______________ _______________
1              Homer                Simpson              01-JAN-65      800-555-1211
3             John                  Carter               16-MAR-71      800-555-1213
5             Lizzie                Borden               20-MAY-70
```

- **AND** takes precedence over **OR**, so you can think of the WHERE clause in the previous query as follows:

  StudentDOB > '01-JAN-1970' OR (StudentID < 2 AND StudentPhone LIKE '%1211')

ORDER BY Clause

You use the ORDER BY clause to sort the rows retrieved by a query. The ORDER BY clause may specify one or more columns on which to sort the data; also, the ORDER BY clause must follow the FROM clause or the WHERE clause (if a WHERE clause is supplied).

The following query would list all rows from the student table and sort them by the StudentLName column in descending order (Z-A).

SELECT * FROM tblStudent ORDER BY StudentLName DESC;

Use ASC for Ascending order or DESC for Descending order (ASC is the default):

Working with Null Values

You can check for Null values in a query by using the IS NULL operator.

SELECT CustomerID, CustomerFName, CustomerLName, CustomerDOB FROM tblCustomer WHERE CustomerDOB IS NULL;

This would display all rows where the customer's DOB has no value.

But how do you tell if a column has a Null or just a blank string? You would use either NVL or COALESCE.

SELECT CustomerID, CustomerFName, CustomerLName, COALESCE (CustomerDOB, 'No value entered') FROM tblCustomer;

Or

SELECT CustomerID, CustomerFName, CustomerLName, NVL (CustomerDOB, 'No value entered') FROM tblCustomer;

Either function will return column's value (if it has something other than `NULL`) otherwise the second parameter is returned.

Displaying Distinct Rows

Suppose you wanted to get a list all vendor names that supply us products.

```
SELECT VendorName FROM tblProducts;
```

The result may be like so:

```
VendorName
__________
Microsoft Corp.
Dell
Cisco
Microsoft Corp
Microsoft Corp
Cisco
```

Notice Microsoft Corp. supplies us 3 products and Cisco supplies us 2 products. But suppose we wanted to eliminate the duplicates and just display each vendor name once if they supply a product. You could use the DISTINCT key word for this purpose.

```
SELECT DISTINCT VendorName FROM tblProducts;
```

```
VendorName
__________
Microsoft Corp.
Dell
Cisco
```

NOTE: When using DISTINCT on more than one column, SQL looks at all columns for uniqueness.

Ie: SELECT DISTINCT StudentFirstName, StudentLastName FROM tblStudent;

Will eliminate duplicate students with the same 1st and Last names and only display them once.

GROUP BY

The GROUP BY statement is used in conjunction with the aggregate functions (sum, avg, count, etc.) to group the result-set by one or more columns.

Syntax:

```
SELECT column_name, aggregate_function(column_name)
FROM table_name
WHERE …
GROUP BY column_name
HAVING …
```

Note: All column names in SELECT list must appear in GROUP BY clause unless the column name is used only in an aggregate function.

Certain RDBMS do not enforce this rule. Oracle enforces this rule.

Example: We have the following "Orders" table:

O_Id	**OrderDate**	**OrderPrice**	**Customer**
1	2008/11/12	1000	Hansen
2	2008/10/23	1600	Nilsen
3	2008/09/02	700	Hansen
4	2008/09/03	300	Hansen
5	2008/08/30	2000	Jensen
6	2008/10/04	100	Nilsen

Now we want to find the total sum (total order) of each customer.

We will have to use the GROUP BY statement to group the customers.

We use the following SQL statement:

```
SELECT Customer,SUM(OrderPrice) FROM Orders
GROUP BY Customer
```

Customer	**SUM(OrderPrice)**
Hansen	2000
Nilsen	1700
Jensen	2000

Result with the GROUP BY statement

Customer	**SUM(OrderPrice)**
Hansen	5700
Nilsen	5700
Hansen	5700
Hansen	5700
Jensen	5700
Nilsen	5700

Result if we omit the GROUP BY statement

If we added another column to our SELECT list we would have to add another column in our GROUP BY clause as follows:

```
SELECT O_Id, Customer, SUM(OrderPrice) FROM Orders
GROUP BY O_Id, Customer
```

HAVING Clause

The HAVING clause was added to SQL because the WHERE keyword could not be used with aggregate functions. Use HAVING instead of WHERE only when you need to filter on an Aggregate function.

Using the previous table. We want to find if any of the customers have a total order of less than 2000.

```
SELECT Customer,SUM(OrderPrice) FROM Orders
GROUP BY Customer
HAVING SUM(OrderPrice)<2000
```

The result will look like this:

Customer	SUM(OrderPrice)
Nilsen	1700

If we want to filter our data with a WHERE clause we would do the following:

Find if the customers "Hansen" or "Jensen" have a total order of more than 1500.

```
SELECT Customer,SUM(OrderPrice) FROM Orders
WHERE Customer='Hansen' OR Customer='Jensen'
GROUP BY Customer
HAVING SUM(OrderPrice)>1500
```

Column Aliases

You use a column alias to change the column heading in the output that is generated by your SELECT statement.

The following example sets the heading for the product description to ‘Description’ and the product cost to ‘Cost’.

SELECT ProductDesc Description, ProductCost Cost FROM tblProduct;

If you wish to use spaces in your column heading you would enclose the alias in quotes. The following example sets the headings to *Product Description* and *Product Cost*. Note the use of DOUBLE QUOTES.

SELECT ProductDesc “Product Description”, ProductCost “Product Cost” FROM tblProduct;

Combining Column Output Using Concatenation

You can combine the column values retrieved by a query using concatenation.

CONCAT('a', 'b')
Result: 'ab'

Assume you have a customer table with two fields called CustomerFName and CustomerLName. A query without concatenation would be like so:

SELECT CustomerFName, CustomerLName FROM tblCustomer;

The output would be like so:

```
CustomerFName          CustomerLName
_______________        ________________
Bob                    Smith
Jane                   Doe
Homer                  Simpson
```

But you can concatenate the first name and last name and use a column alias of ‘Customer Name’ by using the CONCAT function: (Note: Oracle only supports 2 values to Concatenate – if you wish to concantenate more than 2 values you must nest the CONCAT statements as follows)

SELECT CONCAT(CONCAT(CustomerFName, ‘ ‘), CustomerLName) AS “Customer Name”
FROM tblCustomer;

The output would be like so:

```
Customer Name
_______________
Bob Smith
Jane Doe
Homer Simpson
```

Joins

A join allows a SELECT statement to combine records (rows) from two or more tables into a single set which can be either returned as is or used in another join.

We will look at the following types of joins:

- Inner Join
- Left Outer Join (Left Join)
- Right Outer Join (Right Join)
- Full Outer Join (Full Join)

All SELECT statements from now on will follow the SQL92 format.

When structuring your SQL query you should always (not required though) use the primary/foreign key to create the joins between tables (You need two columns of the same type, one in each table, to JOIN on).

In order to perform the operation a join has to define the relationship between records in either table, as well as the way it will evaluate the relationship. The relationship itself is created through a set of conditions that are part of the join and usually are put inside the ON clause. The rest is determined through a join type, which can either be an inner join or an outer join.

The SQL clauses that set the respective join type in a query are *[INNER] JOIN* and *{LEFT | RIGHT} [OUTER] JOIN*. As you can see the actual keywords *INNER* and *OUTER* are optional and can be omitted, however outer joins require specifying the direction – either left or right.

Examples of SQL SELECT statements using joins (note: the words INNER and OUTER are optional):

SELECT * FROM tblStudent INNER JOIN tblGrade ON tblStudent.StudentID= tblGrade.StudentID;

SELECT * FROM tblStudent LEFT OUTER JOIN tblGrade ON tblStudent.StudentID= tblGrade.StudentID;

Inner Join:

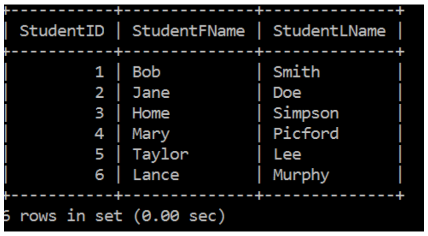
Inner joins require that a row from the first table **has a match** in the second table based on the join conditions.

SELECT * FROM tblStudent INNER JOIN tblGrade ON tblStudent.StudentID= tblGrade.StudentID;

Using the 1st example above only records from tblStudent will be returned where there is a matching StudentID in both tables.

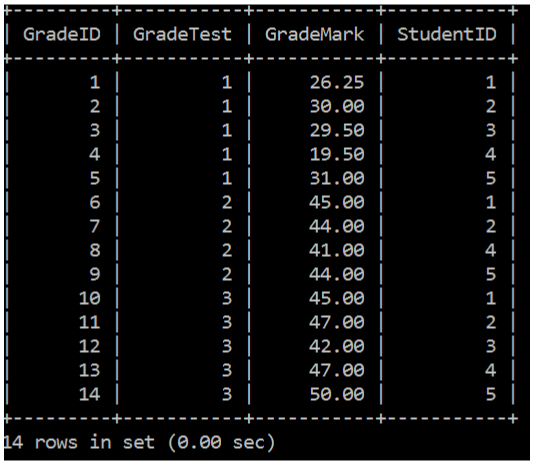
Ie: Source

tblStudent



```
+-----------+----------------+--------------+
| StudentID | StudentFName   | StudentLName |
+-----------+----------------+--------------+
|         1 | Bob            | Smith        |
|         2 | Jane           | Doe          |
|         3 | Home           | Simpson      |
|         4 | Mary           | Picford      |
|         5 | Taylor         | Lee          |
|         6 | Lance          | Murphy       |
+-----------+----------------+--------------+
 rows in set (0.00 sec)
```

tblGrade



```
+---------+-----------+-----------+-----------+
| GradeID | GradeTest | GradeMark | StudentID |
+---------+-----------+-----------+-----------+
|       1 |         1 |     26.25 |         1 |
|       2 |         1 |     30.00 |         2 |
|       3 |         1 |     29.50 |         3 |
|       4 |         1 |     19.50 |         4 |
|       5 |         1 |     31.00 |         5 |
|       6 |         2 |     45.00 |         1 |
|       7 |         2 |     44.00 |         2 |
|       8 |         2 |     41.00 |         4 |
|       9 |         2 |     44.00 |         5 |
|      10 |         3 |     45.00 |         1 |
|      11 |         3 |     47.00 |         2 |
|      12 |         3 |     42.00 |         3 |
|      13 |         3 |     47.00 |         4 |
|      14 |         3 |     50.00 |         5 |
+---------+-----------+-----------+-----------+
14 rows in set (0.00 sec)
```

Result:

StudentID	StudentFName	StudentLName	GradeID	GradeTest	GradeMark	StudentID
1	Bob	Smith	1	1	26.25	1
1	Bob	Smith	6	2	45	1
1	Bob	Smith	10	3	45	1
2	Jane	Doe	2	1	30	2
2	Jane	Doe	7	2	44	2
2	Jane	Doe	11	3	47	2
3	Homer	Simpson	3	1	29.5	3
3	Homer	Simpson	12	3	42	3
4	Mary	Picford	4	1	19.5	4
4	Mary	Picford	8	2	41	4
4	Mary	Picford	13	3	47	4
5	Taylor	Lee	5	1	31	5
5	Taylor	Lee	9	2	44	5
5	Taylor	Lee	14	3	50	5

Outer Join

Outer joins, on the other hand, consider a join successful **even if no records from the second table meet the join conditions** (i.e. whether there are any matches or not). In such case outer join sets all values in the missing columns to *NULL*.

Unlike inner joins, outer joins require that the join direction is specified.

Left Outer Join

tblA LEFT JOIN tblB returns all records from tblA and matching records from tblB based on the join condition.

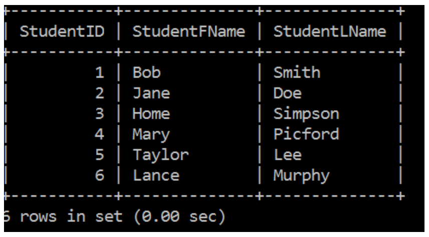
In practice there is very little or even no real purpose for using *RIGHT JOIN* and in majority of cases everyone just sticks to using *LEFT JOIN* whenever they need an outer join.

SELECT * FROM tblStudent LEFT OUTER JOIN tblGrade ON tblStudent.StudentID= tblGrade.StudentID;

Using the 2nd example above, all records from tblStudent will be returned and matching records from tblGrade will be returned.

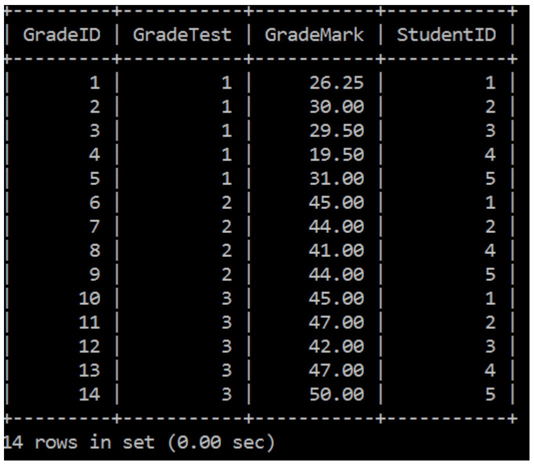
Source:

tblStudent



```
+-----------+--------------+--------------+
| StudentID | StudentFName | StudentLName |
+-----------+--------------+--------------+
|         1 | Bob          | Smith        |
|         2 | Jane         | Doe          |
|         3 | Home         | Simpson      |
|         4 | Mary         | Picford      |
|         5 | Taylor       | Lee          |
|         6 | Lance        | Murphy       |
+-----------+--------------+--------------+
6 rows in set (0.00 sec)
```

tblGrade



```
+---------+-----------+-----------+-----------+
| GradeID | GradeTest | GradeMark | StudentID |
+---------+-----------+-----------+-----------+
|       1 |         1 |     26.25 |         1 |
|       2 |         1 |     30.00 |         2 |
|       3 |         1 |     29.50 |         3 |
|       4 |         1 |     19.50 |         4 |
|       5 |         1 |     31.00 |         5 |
|       6 |         2 |     45.00 |         1 |
|       7 |         2 |     44.00 |         2 |
|       8 |         2 |     41.00 |         4 |
|       9 |         2 |     44.00 |         5 |
|      10 |         3 |     45.00 |         1 |
|      11 |         3 |     47.00 |         2 |
|      12 |         3 |     42.00 |         3 |
|      13 |         3 |     47.00 |         4 |
|      14 |         3 |     50.00 |         5 |
+---------+-----------+-----------+-----------+
14 rows in set (0.00 sec)
```

Result:

StudentID	StudentFName	StudentLName	GradeID	GradeTest	GradeMark	StudentID
1	Bob	Smith	1	1	26.25	1
1	Bob	Smith	6	2	45	1
1	Bob	Smith	10	3	45	1
2	Jane	Doe	2	1	30	2
2	Jane	Doe	7	2	44	2
2	Jane	Doe	11	3	47	2
3	Homer	Simpson	3	1	29.5	3
3	Homer	Simpson	12	3	42	3
4	Mary	Picford	4	1	19.5	4
4	Mary	Picford	8	2	41	4
4	Mary	Picford	13	3	47	4
5	Taylor	Lee	5	1	31	5
5	Taylor	Lee	9	2	44	5
5	Taylor	Lee	14	3	50	5
6	Lance	Murphy	NULL	NULL	NULL	NULL

When does the join type matter?

Choosing the appropriate type depends on the logic you are trying to implement.

You have to use **inner join** when mandatory pieces of information are located in both tables and partial information is considered incomplete or even useless.

For instance the 1st query above

SELECT *
FROM tblStudent INNER JOIN tblGrade
ON tblStudent.StudentID= tblGrade.StudentID;

Is actually displaying the marks for students who have actually written tests.

While the 2nd query can be used to find students who have not written any tests or have not written a specific test.

Query to find students who have not written any test:

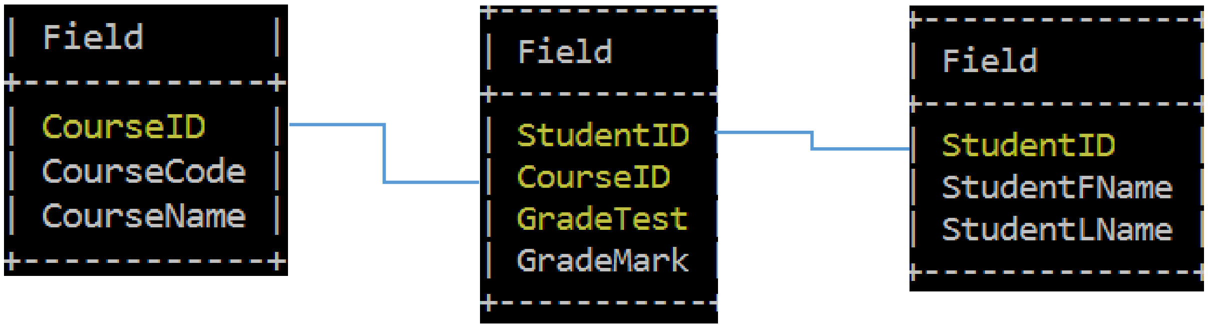
```
SELECT *
FROM tblStudent LEFT JOIN tblGrade
ON tblStudent.StudentID= tblGrade.StudentID
WHERE tblGrade.StudentID IS NULL;
```

Query to find students who have not written test 2:

```
SELECT *
FROM tblStudent LEFT JOIN tblGrade
ON tblStudent.StudentID= tblGrade.StudentID
AND tblGrade.GradeTest=2
WHERE tblGrade.StudentID IS NULL;
```

Creating Joins With More Than 2 Tables

tblCourse tblGrade tblStudent



```
| Field      |
+------------+
| CourseID   |
| CourseCode |
| CourseName |
+------------+
```

```
+-----------
| Field
+-----------
| StudentID
| CourseID
| GradeTest
| GradeMark
+-----------
```

```
+---------------
| Field
+---------------
| StudentID
| StudentFName
| StudentLName
+---------------
```

Create a query that lists all student who have written any test and the corresponding Course info and test info.

This requires an INNER query between all tables.

```
SELECT CONCAT(StudentFName, " ",StudentLName) "Student", CourseCode "Course", GradeTest "Test #", GradeMark "Mark"
 FROM  tblCourse t1
    INNER JOIN tblGrade t2
            ON t1.CourseID = t2. CourseID
    INNER JOIN tblStudent t3
            ON t2.StudentID = t3.StudentID
;
```

For an INNER JOIN the order of the tables/joins do not matter.

When combing INNER JOINS with OUTER JOINS in the same SELECT statement the order of the JOINS does matter. SQL queries with combined inner and outer joins is beyond the scope of this introductory course.

Sub Queries

Overview:

- A subquery is a query within another query. The outer query is called as main query and inner query is called as subquery.

- The subquery is always executed first and the result is passed to the main query. The main query is executed by taking the result from the subquery.

- The **IN** operator plays a very important role in subqueries as generally subqueries generate a list of values and the main query is used to compare a single value against the list of values supplied by the subquery. **

- A main query can receive values from more than one subquery.

- It is also possible to **nest** subqueries. It is also possible for a subquery to depend on another subquery and that subquery on another and so on.

- A subquery can return multiple columns. These multiple columns must be compared with multiple values.

Rules:

- Subqueries must be enclosed within parentheses.

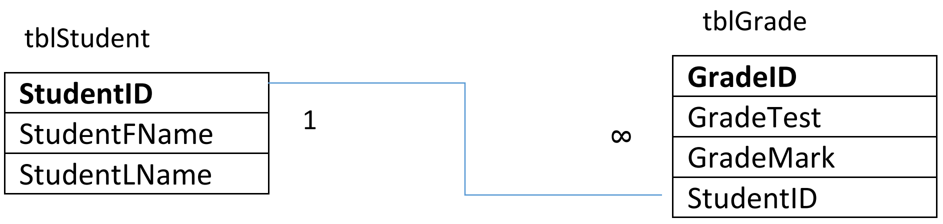
- A subquery can have only one column in the SELECT clause, unless multiple columns are in the main query for the subquery to compare its selected columns.

- Subqueries that return more than one row can only be used with multiple value operators, such as the IN operator.

- The SELECT list cannot include any references to values that evaluate to a BLOB, ARRAY, CLOB, or NCLOB.

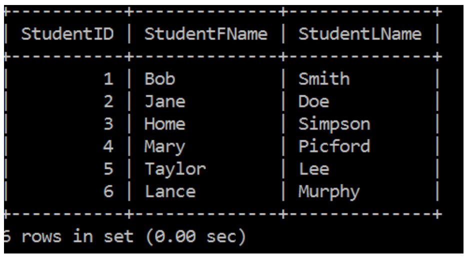
Example:

Assume we have a simple schema that has 5 students and some marks associated with each student as follows:



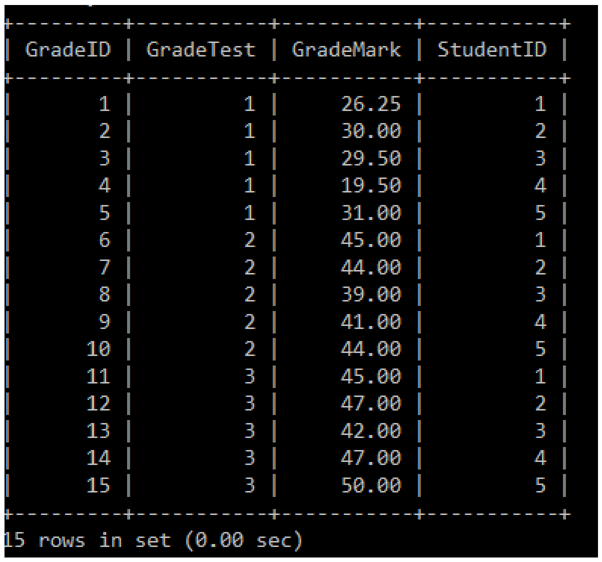
Data consists of the following:

tblStudent



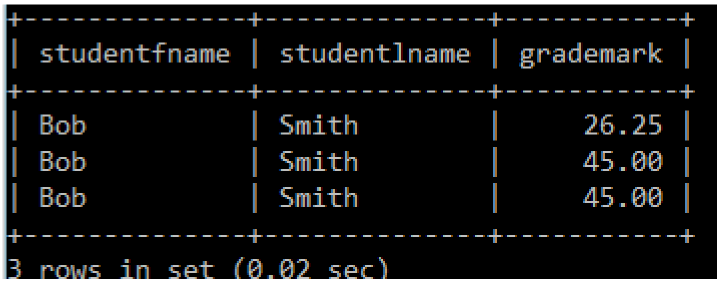
```
+-----------+--------------+--------------+
| StudentID | StudentFName | StudentLName |
+-----------+--------------+--------------+
|         1 | Bob          | Smith        |
|         2 | Jane         | Doe          |
|         3 | Home         | Simpson      |
|         4 | Mary         | Picford      |
|         5 | Taylor       | Lee          |
|         6 | Lance        | Murphy       |
+-----------+--------------+--------------+
5 rows in set (0.00 sec)
```

tblGrade



```
+---------+-----------+-----------+-----------+
| GradeID | GradeTest | GradeMark | StudentID |
+---------+-----------+-----------+-----------+
|       1 |         1 |     26.25 |         1 |
|       2 |         1 |     30.00 |         2 |
|       3 |         1 |     29.50 |         3 |
|       4 |         1 |     19.50 |         4 |
|       5 |         1 |     31.00 |         5 |
|       6 |         2 |     45.00 |         1 |
|       7 |         2 |     44.00 |         2 |
|       8 |         2 |     39.00 |         3 |
|       9 |         2 |     41.00 |         4 |
|      10 |         2 |     44.00 |         5 |
|      11 |         3 |     45.00 |         1 |
|      12 |         3 |     47.00 |         2 |
|      13 |         3 |     42.00 |         3 |
|      14 |         3 |     47.00 |         4 |
|      15 |         3 |     50.00 |         5 |
+---------+-----------+-----------+-----------+
15 rows in set (0.00 sec)
```

Let’s assume we want to return all the marks for the person with the StudentID = 1



```
+--------------+-------------+-----------+
| studentfname | studentlname | grademark |
+--------------+-------------+-----------+
| Bob          | Smith       |     26.25 |
| Bob          | Smith       |     45.00 |
| Bob          | Smith       |     45.00 |
+--------------+-------------+-----------+
3 rows in set (0.02 sec)
```

Without a sub query:

```
SELECT t1.StudentFName, t1.StudentLName, t2.GradeMark
 FROM tblStudent t1
   INNER JOIN
     tblGrade t2
       ON t1.StudentID = t2.StudentID
          WHERE t1.StudentID=1;
```

With a sub query:

```
SELECT t1.StudentFName, t1.StudentLName, t2.GradeMark
 FROM tblStudent t1
   INNER JOIN
     tblGrade t2
       ON t1.StudentID = t2.StudentID
         WHERE t1.StudentID IN
           (
              SELECT StudentID
                FROM tblGrade
                   WHERE StudentID=1
           );
```

Now let’s do a more realistic example:

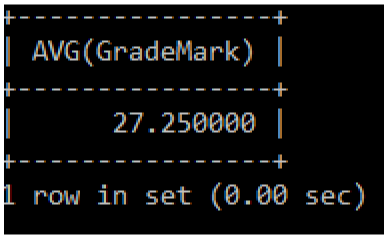
Assume you want a list of all students who are below the average for the 1st test. Display their first and last names along with their mark for the 1st test.

How would you do this?

Try breaking the solution out into steps to make it easier to understand.

1. I would try creating a query that averages out every student mark for the 1st test. We obviously will use one of the built in aggregate functions called AVG which returns the average of a set of values.

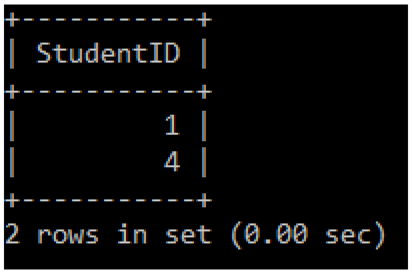
   SELECT AVG(GradeMark) FROM tblGrade WHERE GradeTest = 1;



```
+----------------+
| AVG(GradeMark) |
+----------------+
|      27.250000 |
+----------------+
1 row in set (0.00 sec)
```

2. Now that I know how to get the for the 1st test I would like to get all the student marks for the 1st test that are below this average. Let’s use the above query in a sub query to accomplish this.

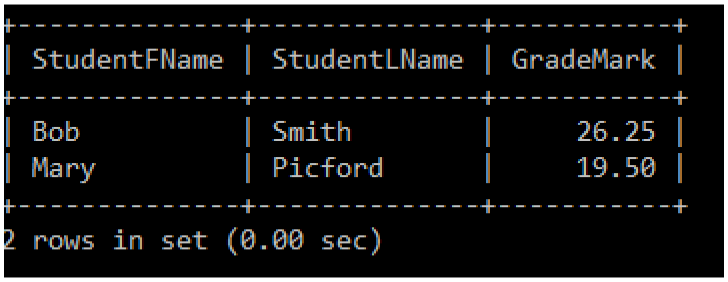
```
SELECT StudentID
  FROM tblGrade
   WHERE GradeTest=1 AND GradeMark <
     (SELECT AVG(GradeMark) FROM tblGrade WHERE GradeTest = 1);
```



```
+-----------+
| StudentID |
+-----------+
|         1 |
|         4 |
+-----------+
2 rows in set (0.00 sec)
```

3. Now let’s combine it all to create the display that was asked for.

```
SELECT StudentFName, StudentLName, GradeMark
 FROM tblStudent t1
  INNER JOIN
   tblGrade t2
    ON t1.StudentID = t2.StudentID
      WHERE t2.GradeTest=1 AND t1.StudentID IN
        (SELECT StudentID
          FROM tblGrade
           WHERE GradeTest=1 AND GradeMark <
             (SELECT AVG(GradeMark) FROM tblGrade WHERE GradeTest = 1)
        );
```



```
+--------------+--------------+-----------+
| StudentFName | StudentLName | GradeMark |
+--------------+--------------+-----------+
| Bob          | Smith        |     26.25 |
| Mary         | Picford      |     19.50 |
+--------------+--------------+-----------+
2 rows in set (0.00 sec)
```

Subqueries can be used in several places within a query:

- In the WHERE clause

```
SELECT a.studentid, a.name, b.total_marks
  FROM student a, marks b
    WHERE a.studentid = b.studentid AND b.total_marks >
 (SELECT total_marks
     FROM marks
        WHERE studentid =  'V002'
);
```

- In the FROM clause

```
SELECT AVG(sum_column1)
 FROM
     (SELECT SUM(column1) AS sum_column1
      FROM t1
        GROUP BY column1
      ) AS t1;
```

Note: In certain RDBMS Sub queries within the FROM clause need to be given an ALIAS name. Oracle does NOT require the alias name.

The results of the Sub Query are used as the source data for the outer query.

- In the SELECT column list

```
SELECT customer.customer_num,
        (SELECT SUM(ship_charge)
            FROM orders
                WHERE customer.customer_num = orders.customer_num
         ) AS total_ship_chg
FROM customer
```

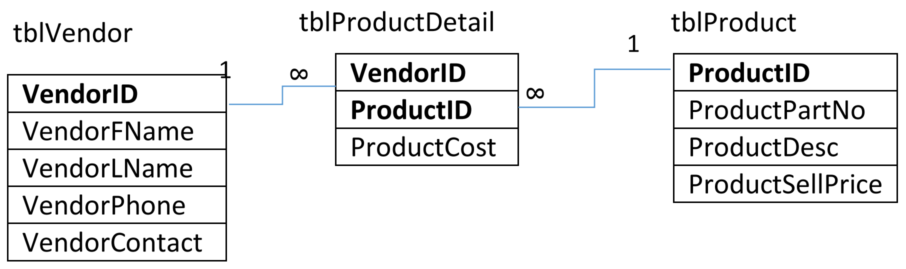
- In the HAVING clause
- In the GROUP BY clause

WEEK 8

DDL – Data Definition Language

CREATE ALTER DROP RENAME	**Data Definition Language (DDL)** Creates, changes, and removes data structures.

CREATE TABLE



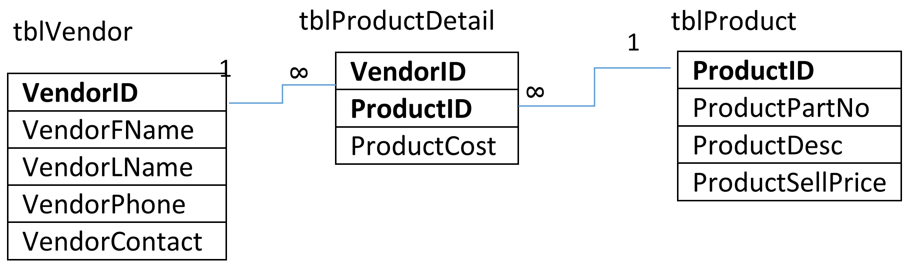
The Inline constraints are column level constraints and can only be defined at the time of table creation.

The Outline/Out-of-Line constraints can be define at the time of table creation or after table creation, so that constraints can be placed later by altering the structure of the table.

Create a table - And give the primary key constraint a name. **INLINE** Note: a name has been given to the constraint	**CREATE TABLE** *tblProduct* **(** *ProductID* **NUMBER(***38***) CONSTRAINT** tblProduct_pk **PRIMARY KEY,** *ProductPartNo* **VARCHAR2(***15***),** *ProductDesc* **VARCHAR2(***35***),** *ProductSellPrice* **Number(***5,2***)** **);**
Create a table – Same as above but do not give the PK constraint a name. **INLINE** Note: No name has been given to the constraint	**CREATE TABLE** *tblProduct* **(** *ProductID* **NUMBER(***38***) PRIMARY KEY,** *ProductPartNo* **VARCHAR2(***15***),** *ProductDesc* **VARCHAR2(***35***),** *ProductSellPrice* **Number(***5,2***)** **);**
Create a table with a composite primary key. **OUT-OF-LINE**	**CREATE TABLE** *tblProductDetail* **(** *ProductID* **NUMBER(***38***),** *VendorID* **NUMBER(***38***),** *ProductCost* **Number(***5,2***),** **PRIMARY KEY (***ProductID, VendorID***)** **);**

Create a table – Same as above but now add in the foreign key constraints. **OUT-OF-LINE** Note: No name given to the constraint	**CREATE TABLE** *tblProductDetail* **(** *ProductID* **NUMBER(***38***),** *VendorID* **NUMBER(***38***),** *ProductCost* **Number(***5,2***),** **PRIMARY KEY (***ProductID, VendorID***),** **FOREIGN KEY (***ProductID***) REFERENCES** *tblProduct***(***ProductID***),** **FOREIGN KEY (***VendorID***) REFERENCES** *tblVendor***(***VendorID***)** **);**
Same as above but give the Constraints names **OUT-OF-LINE** Note: Names given to the constraint	**CREATE TABLE** *tblProductDetail* **(** *ProductID* **NUMBER(***38***),** *VendorID* **NUMBER(***38***),** *ProductCost* **Number(***5,2***),** **CONSTRAINT** tblProductDetail_PK **PRIMARY KEY (***ProductID, VendorID***),** **CONSTRAINT** tblProductDetail_ProductID_FK **FOREIGN KEY (***ProductID***)** **REFERENCES** *tblProduct***(***ProductID***),** **CONSTRAINT** tblProductDetail_VendorID_FK **FOREIGN KEY (***VendorID***)** **REFERENCES** *tblVendor***(***VendorID***)** **);**
Create a table and set the ProductSellPrice to a default value of 0. **INLINE**	**CREATE TABLE** *tblProduct* **(** *ProductID* **NUMBER(***38***) PRIMARY KEY,** *ProductPartNo* **VARCHAR2(***15***),** *ProductDesc* **VARCHAR2(***35***),** *ProductSellPrice* **Number(***5,2***) DEFAULT** *0* **);**
Create a table with a check constraint on the SellPrice field – this check constraint ensures the selling price is between 0 and 100 dollars. **INLINE**	**CREATE TABLE** *tblProduct* **(** *ProductID* **NUMBER(***38***) PRIMARY KEY,** *ProductPartNo* **VARCHAR2(***15***),** *ProductDesc* **VARCHAR2(***35***),** *ProductSellPrice* **Number(***5,2***) DEFAULT 0,** **CONSTRAINT** *check_SellPrice* **CHECK (***ProductSellPrice* **Between** *0* **AND** *100***)** **);**

ALTER TABLE



Add a column to an existing table	**ALTER TABLE** *tblVendor* **ADD** *VendorCity* **VARCHAR2(***20***);**
Add more than one column to an existing table	**ALTER TABLE** *tblVendor* **ADD** *VendorCity* **VARCHAR2(***20***),** **ADD** *VendorProvince* **VARCHAR2(***25***);**
Change the size of an existing column or change the data type	**ALTER TABLE** *tblProduct* **MODIFY** *ProductDesc* **VARCHAR2(***55***);**
Change more than one column with a single ALTER statement	**ALTER TABLE** *tblProduct* **MODIFY** **(** *ProductDesc* **VARCHAR2(***55***),** ProductSellPrice **CONSTRAINT** tblProduct_SellPrice_Check **CHECK** (*ProductSellPrice* **Between** *0* **AND** *200)* **);**
Changing the default value of a column.	**ALTER TABLE** *tblProductDetail* **MODIFY** *ProductCost* **DEFAULT** *0*;
Removing (dropping) a column	**ALTER TABLE** *tblVendor* **DROP COLUMN** *VendorContact*;
Removing (dropping) a constraint	**ALTER TABLE** *tblProduct* **DROP CONSTRAINT** *tblProduct_SellPrice_Check*;
Drop a UNIQUE constraint if you do not know the constraint name (just list the columns that are included in the constraint)	**ALTER TABLE table_name DROP UNIQUE (column1,column2,,…);**
Rename a column	**ALTER TABLE** *tblProduct* **RENAME COLUMN** *ProductDesc* **TO** *ProdDescription;*

If a column already has a constraint name and you try to give it the same name you will get an error message.

Data Manipulation Language

Insert Statement

Use the INSERT statement to add rows to a table.

You can specify the following information in an INSERT statement:

- The table into which the row is to be inserted
- A list of columns for which you want to specify column values
- A list of values to store in the specified columns

The following INSERT statement adds a row to a customer table.

Note: That the order of values in the VALUES clause matches the order in which the columns are specified in the column list. Also notice that the statement has three parts: the table name, the column list, and the values to be added:

```
INSERT INTO tblCustomers (
CustomerID, CustomerFName, CustomerLName, CustomerDOB, CustomerPhone
)
VALUES (
6, 'Fred', 'Brown', '01-JAN-1970', '800-555-1215'
);
```

You may omit the column list when supplying values for every column, as in this example:

```
INSERT INTO tblCustomers
VALUES (6, 'Fred', 'Brown', '01-JAN-1970', '800-555-1215');
```

You can specify a null value for a column using the NULL keyword. For example, the following INSERT specifies a null value for the DOB and phone columns:

```
INSERT INTO tblCustomers (
CustomerID, CustomerFName, CustomerLName, CustomerDOB, CustomerPhone
)
VALUES (
6, 'Fred', 'Brown', NULL, NULL
);
```

You can include a single and double quote in a column value. For example, the following INSERT specifies a last name of O'Malley for a new customer; notice the use of two single quotes in the last name after the letter O:

```
INSERT INTO tblCustomers
VALUES (9, 'Kyle', 'O''Malley', NULL, NULL);
```

The next example specifies the name The "Great" Gatsby for a new product:

```
INSERT INTO tblProducts (
ProductID,ProductTypeID,ProductName, ProductDescription, ProductPrice
) VALUES (
13, 1, 'The "Great" Gatsby', NULL, 12.99
);
```

Copying Rows from One Table to Another

You can copy rows from one table to another using a query in the place of the column values in the INSERT statement. The number of columns and the column types in the source and destination must match. The following example uses a SELECT to retrieve the FirstName and LastName columns for prospect #1 and supplies those columns to an INSERT statement:

```
INSERT INTO tblCustomer (CustomerID,CustomerFName, CustomerLName)
SELECT 20, ProspectFName,ProspectLName
FROM tblProspect
WHERE ProspectID = 1;
```

Note: The CustomerID for the new row is set to 20.

UPDATE Statement

Use the UPDATE statement to modify rows in a table. When you use the UPDATE statement, you typically specify the following information:

- The table name
- A WHERE clause that specifies the rows to be changed
- A list of column names, along with their new values, specified using the SET clause

Change the data in a single row (note the WHERE clause)

```
UPDATE tblCustomer
SET CustomerLName = 'Orange'
WHERE CustomerID = 2;
```

Change the data in multiple rows (note the WHERE clause)

```
UPDATE tblProducts
SET price = price * 1.20
WHERE
price >= 20;
```

DELETE Statement

Use the DELETE statement to remove rows from a table. Generally, you should specify a WHERE clause that limits the rows that you wish to delete; if you don't, all the rows will be deleted.

```
DELETE FROM tblCustomer
WHERE CustomerID = 10;
```

Note: When deleting a row you should always try to delete the row based on the primary key value to avoid deleting more than one row inadvertently.

Database Transactions

A database transaction is a group of SQL statements that perform a logical unit of work. You can think of a transaction as an inseparable set of SQL statements whose results should be made permanent in the database as a whole (or undone as a whole).

An example of a database transaction might be a transfer of money from one bank account to another. You would need one UPDATE statement to subtract from the total amount of money from one account, and another UPDATE would add money to the other account. Both the subtraction and the addition must be permanently recorded in the database; otherwise, money will be lost. If there is a problem with the money transfer, then the subtraction and addition must both be undone.

This simple example uses only two UPDATE statements, but a transaction may consist of many INSERT, UPDATE, and DELETE statements.

Committing and Rolling Back a Transaction

To permanently record the results made by SQL statements in a transaction, you perform a commit, using the SQL COMMIT statement. If you need to undo the results, you perform a rollback, using the SQL ROLLBACK statement, which resets all the rows back to what they were originally.

The following example adds a row to the customers table and then makes the change permanent by performing a COMMIT:

```
INSERT INTO customers VALUES (6, 'Fred', 'Green', '01-JAN-1970', '800-555-1215');

1 row created.

COMMIT;

Commit complete.
```

The following example updates customer #1 and then undoes the change by performing a ROLLBACK:

```
UPDATE customers SET first_name = 'Edward' WHERE customer_id = 1;

1 row updated.

ROLLBACK;

Rollback complete.
```

Starting and Ending a Transaction

A transaction is a logical unit of work that enables you to split up your SQL statements. A transaction has a beginning and an end.

A transaction begins when one of the following events occurs:

- You connect to the database and perform a DML statement (an INSERT, UPDATE, or DELETE).
- A previous transaction ends (see below) and you enter another DML statement.

A transaction ends when one of the following events occurs:

- You perform a COMMIT or a ROLLBACK.
- You perform a DDL statement, such as a CREATE TABLE, ALTER, TRUNCATE, etc. statement, in which case a COMMIT is automatically performed.
- You perform a DCL statement, such as a GRANT statement, in which case a COMMIT is automatically performed.
- You disconnect from the database. If you exit SQL*Plus normally, by entering the EXIT command, a COMMIT is automatically performed for you. If SQL*Plus terminates abnormally—for example, if the computer on which SQL*Plus was running were to crash—a ROLLBACK is automatically performed. This applies to any program that accesses a database. For example, if you wrote a Java program that accessed a database and your program crashed, a ROLLBACK would be automatically performed.
- You perform a DML statement that fails, in which case a ROLLBACK is automatically performed for that individual DML statement.

Note: It is poor practice not to explicitly commit or roll back your transactions, so perform a COMMIT or ROLLBACK at the end of your transactions.

Note: You can see any changes you have made during the transaction by querying the modified tables, but other users cannot see the changes. After you commit the transaction, the changes are visible to other users' statements that execute after the commit.

Example:

1. CREATE TABLE stud(ID INT,name VARCHAR2(20));
2. INSERT INTO stud(ID, name) VALUES(1,’Bob’);
3. INSERT INTO stud(ID,name) VALUES(2,’Sara’);
4. COMMIT;
5. INSERT INTO stud(ID,name) VALUES(3,’Roberto’);
6. ROLLBACK;
7. INSERT INTO stud(ID,name) VALUES(4,’Li’);
8. ALTER TABLE stud ADD CONSTRAINT stud_pk PRIMARY KEY(ID);
9. INSERT INTO stud(ID,name) VALUES(5,’Maria’);
10. ROLLBACK;
11. COMMIT;

Result:

SELECT ID, name FROM stud;

ID	NAME
1	Bob
2	Sara
4	Li

Sequences

Many times, you will need a column (field) whose data type is numeric and you want it to automatically increment by a pre-defined amount each time a row (record) is added to a table.

A common use is for Primary Keys

- MS Access – Autonumber
- MySQL – Auto-increment
- MS SQL – Identity
- **Oracle – Sequence**

Example

1. Create a table

```
CREATE TABLE tblProduct
(
ProductID  INTEGER CONSTRAINT tblProduct_pk PRIMARY KEY,
ProdDesc VARCHAR2(30)
);
```

2. Create the Sequence

```
CREATE SEQUENCE seq_autoID_tblProduct
START WITH 1
INCREMENT BY 1
NOMAXVALUE;
```

3. Increment the Sequence

```
INSERT INTO tblProduct
(ProductID, ProductDesc)
VALUES (seq_autoID_tblProduct.NEXTVAL, ‘Kensington Optical Mouse’);
```

An alternative for step 3 would be to create a trigger for when a record gets inserted.