DBS211/311 - SQL Using Oracle

Robert Stewart

rob.stewart@senecacollege.ca

Table of Contents

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**	Transaction Control

SAVEPOINT	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
ANY()	Is used to compare a value to any applicable value in the list according to the condition.ANY must be preceded by comparison operators.	SELECT * FROM tblProduct WHERE ProductID > ANY(2,4, 5) ;

		Returns Any products that have a ProductID greater than 2.

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
ANY	Matches lists of values	N
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.

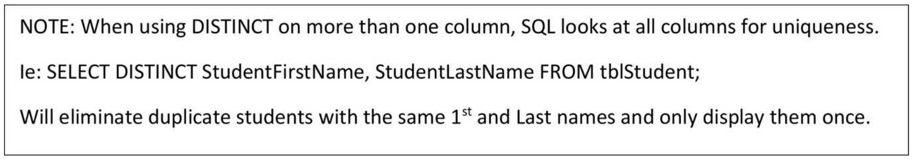
SELECT DISTINCT VendorName FROM tblProducts;

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.



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 expression1, expression2, ... expression_n, aggregate_function (aggregate_expression) FROM tables [WHERE conditions] GROUP BY expression1, expression2, ... expression_n;

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.

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.

1. is used to combine the results of two or more SELECT statements.
2. it will eliminate duplicate rows from its result set
3. the number of columns and datatype must be same in both SELECT statements

expression1, expression2, ... expression_n

The expressions that are not encapsulated within an aggregate function and must be included in the GROUP BY clause.

aggregate_function

It can be a function such as SUM, COUNT, MIN, MAX, or AVG functions.

aggregate_expression

This is the column or expression that the aggregate_function will be used on.

tables

The tables that you wish to retrieve records from. There must be at least one table listed in the FROM clause.

WHERE conditions

Optional. The conditions that must be met for the records to be selected.

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.

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

Set Operators

UNION

Example:

First Table

ID	Name
1	abhi
2	adam

Second Table

ID	Name
2	adam
3	Chester

SELECT * FROM First UNION SELECT * FROM Second;

Result:

ID	NAME
1	abhi
2	adam
3	Chester

UNION ALL

1. operation is similar to Union. But it also shows the duplicate rows

Example:

First Table

ID	Name
1	abhi
2	adam

Second Table

ID	Name
2	adam
3	Chester

SELECT * FROM First UNION ALL SELECT * FROM Second;

Result:

ID	NAME
1	abhi
2	adam
2	adam
3	Chester

INTERSECT

1. operation is used to combine two SELECT statements, but it only returns the records which are common from both SELECT statements
2. the number of columns and datatype must be same in both SELECT statements

Example:

First Table

ID	**Name**
1	abhi
2	adam

Second Table

ID	**Name**
2	adam
3	Chester

SELECT * FROM First INTERSECT SELECT * FROM Second;

Result:

ID	**NAME**
2	adam

MINUS

1. operator is used to return all rows in the first SELECT statement that are not returned by the second SELECT statement
    a. each SELECT statement will define a dataset. The MINUS operator will retrieve all records from the first dataset and then remove from the results all records from the second dataset
2. the number of columns and datatype must be same in both SELECT statements

Example:

First Table

ID	**Name**
1	abhi
2	adam

Second Table

ID	Name
2	adam
3	Chester

SELECT * FROM First MINUS SELECT * FROM Second;

Result:

ID	NAME
1	abhi

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.
B. 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:

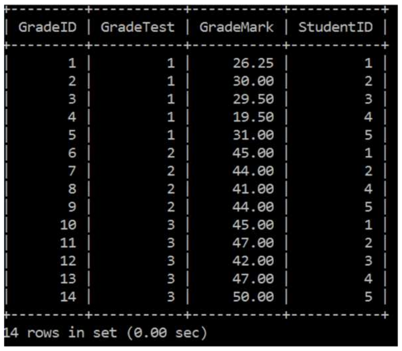
Inner joins require that a row from the first table has a match in the second table based on the join conditions.

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

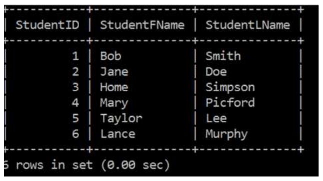
Using the 1st example above only records from tblStudent will be returned where there is a matching StudentID in both tables.

Ie: Source

tblStudent 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)
```



```
+-----------+--------------+--------------+
| 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)
```

Result:

StudentID	StudentFName	StudentLName	GradeID	GradeTest	GradeMark	Stud
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.

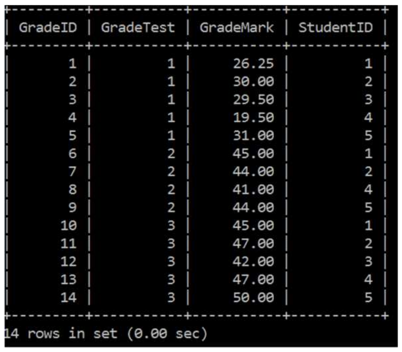
In practice there is very little or even no real purpose for using RIGHT JOIN and in majority of cases everyone just sticks to using LEFT JOIN whenever they need an outer join.

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

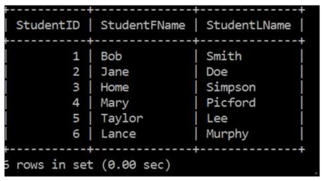
Using the 2nd example above, all records from tblStudent will be returned and matching records from tblGrade will be returned.

Source:

tblStudent 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)
```



```
+-----------+--------------+--------------+
| 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)
```

Result:

StudentID	StudentFName	StudentLName	GradeID	GradeTest	GradeMark	Stud
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	NUL

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:

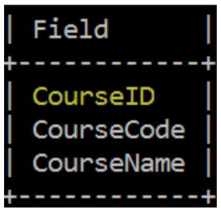
SELECT * FROM tblStudent LEFT JOIN tblGrade ON tblStudent.StudentID= tblGrade.StudentID WHERE tblGrade.StudentID IS NULL;

Query to find students who have not written test 2:

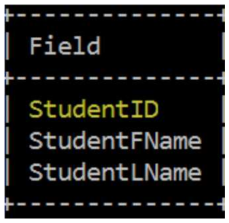
SELECT * FROM tblStudent LEFT JOIN tblGrade ON tblStudent.StudentID= tblGrade.StudentID AND tblGrade.GradeTest=2 WHERE tblGrade.StudentID IS NULL;

Creating Joins With More Than 2 Tables

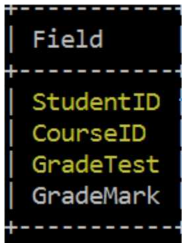
tblCourse tblGrade tblStudent



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



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



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

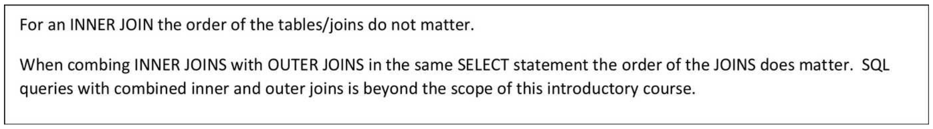
Create a query that lists all student who have written any test and the corresponding Course info and test info.

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.



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:

StudentID
StudentFName
StudentLName

tblGrade

tblGrade

GradeID
GradeTest
GradeMark
StudentID

tblStudent

tblStudent

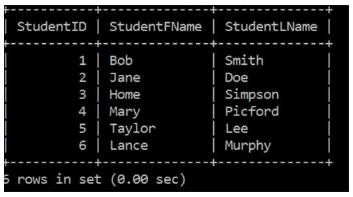
∞

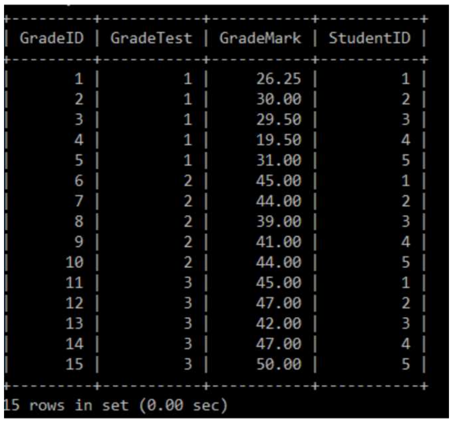
1

1

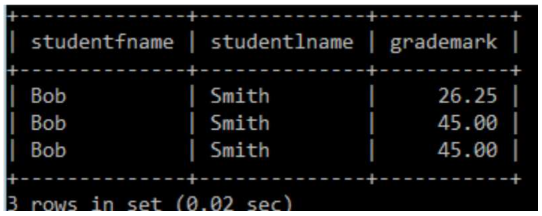
Data consists of the following:

tblStudent tblGrade


```
+-----------+--------------+--------------+
| 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)
```


```
+---------+-----------+-----------+-----------+
| 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:

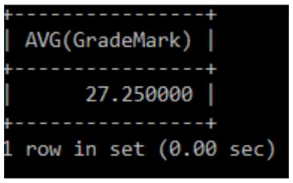
Assume you want a list of all students who are below the average for the 1st test. Display their first and last names along with their mark for the 1st test.

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)
```

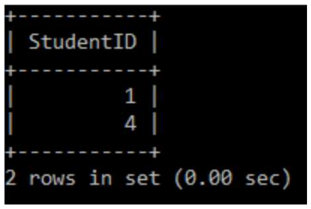
2. Now that I know how to get the average for the 1st test I would like to get all the student marks for the 1st test that are below this average. Let's use the above query in a sub query to accomplish this.

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

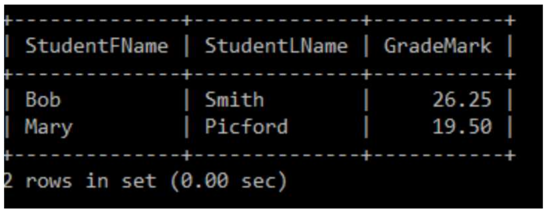
(SELECT StudentID

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.

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

Special Operators

IN()

Equal to any member in the list

ANY()

Compare value to **each** value returned by the subquery

NOTE: You must use a comparison operator with the ANY() operator

ALL()

Compare value to **EVERY** value returned by the subquery

NOTE: You must use a comparison operator with the ALL() operator

<ANY() - less than maximum

>ANY() - more than minimum

=ANY() - equivalent to IN

>ALL() - more than the maximum

<ALL() - less than the minimum

Example of IN()

Find the name of the product if all the records in the OrderDetails has Quantity either equal to 6 or 2.

SELECT ProductName FROM Products WHERE ProductID IN (SELECT ProductId FROM OrderDetails WHERE Quantity = 6 OR Quantity = 2);

Examples of ALL()

Find the name of the product if all the records in the OrderDetails has Quantity either equal to 6 or 2. Note that this can be replaced with IN()

SELECT ProductName FROM Products WHERE ProductID = ALL (SELECT ProductId FROM OrderDetails WHERE Quantity = 6 OR Quantity = 2);

Find the OrderID whose maximum Quantity among all product of that OrderID is greater than average quantity of all OrderID.

SELECT OrderID FROM OrderDetails GROUP BY OrderID HAVING max(Quantity) > ALL (SELECT avg(Quantity) FROM OrderDetails GROUP BY OrderID);

Example of ANY()

List the product names if it finds ANY records in the OrderDetails table that quantity = 10

SELECT ProductName FROM Products WHERE ProductID = ANY (SELECT ProductID FROM OrderDetails WHERE Quantity = 10);

Functions (just a few selected functions)

Here is a good list of Oracle functions that are available Oracle Functions

UPPER

converts all letters in the specified string to uppercase. If there are characters in the string that are not letters, they are unaffected by this function

select * from suppliers where UPPER(supplier_name) like ('TEST%');

or you could do this

select * from suppliers where UPPER(supplier_name) like UPPER('test%')

LOWER

converts all letters in the specified string to lowercase. If there are characters in the string that are not letters, they are unaffected by this function

select * from suppliers where LOWER(supplier_name) like ('test%');

or you could do this

select * from suppliers where LOWER(supplier_name) like LOWER('TEST%')

Database Creation

With Oracle Database, you typically have a single database that hosts multiple applications. You do not need multiple databases to run different applications. Instead, you can separate the objects that support each different application into different schemas in the same database. However, there may be situations in which you want to create multiple Oracle databases on the same host computer. When you do this, the new databases typically use the same Oracle home directory as the first database, but store database data files separately from those of the first database.

With Oracle Database, you typically have a single database that hosts multiple applications. You do not need multiple databases to run different applications. Instead, you can separate the objects that support each different application into different schemas in the same database. However, there may be situations in which you want to create multiple Oracle databases on the same host computer. When you do this, the new databases typically use the same Oracle home directory as the first database, but store database data files separately from those of the first database.

Creating Tablespaces

Before creating a new schema, you must first (1st) create a tablespace.

A tablespace is used to allocate space in the Oracle database where schema objects are stored. (Schema objects are tables, views, etc.)

There are three (3) types of tablespaces:

1. Permanent
   - contains persistent schema objects that are stored in data files
2. Temporary
   - contains schema objects that are stored in temp files that exist during a session
3. Undo
   - manage undo data if the Oracle database is being run in automatic undo management mode

Let’s create a simple Permanent tablespace that is fixed at 5 meg in size and contains a single data file:

```
CREATE TABLESPACE tbs_perm_01 DATAFILE 'tbs_perm_01.dat' SIZE 5M ONLINE;
```

Now let’s create a temporary tablespace with a single data file that is 5 meg but will grow in size if need be:

```
CREATE TEMPORARY TABLESPACE tbs_temp_01 TEMPFILE 'tbs_temp_01.dbf' SIZE 5M AUTOEXTEND
ON;
```

For now, we will not create an UNDO tablespace. This will be explained later.

Create Schemas

- essence, a schema is created in Oracle when a user is created.
- therefore use the CREATE USER statement to create a schema

- some people look at a schema and user as the same thing but a user can own objects in the schema but you can add new users that have access to the schema – these users are therefore just regular users of the schema (ie: application data)

Let’s assume we are creating an invoicing application. We could create a user named invoice as follows (which will be the schema for the application INVOICE:

```
CREATE USER invoice IDENTIFIED BY pwd1 DEFAULT TABLESPACE tbs_perm_01 TEMPORARY
TABLESPACE tbs_temp_01 QUOTA 5M on tbs_perm_01;
```

Where:

Default tablespace = you have to specify the name of tablespace where this user will create objects (tables, indexes, views, etc.). This is an optional clause, if you omit this clause then all the objects created by this user gets stored in the database default tablespace which is “users” tablespace in most cases.

Temporary tablespace = All the temporary segments created by this user get stored here. Temporary segment such as temporary tables; Similar to default tablespace you have to specify the name of the temporary tablespace here. But make sure the tablespace you are going to specify here must be temporary tablespace. This is an optional clause so if you omit this clause then all the temporary objects created by this user get stored in the database default temporary tablespace which is "temp" tablespace.

Quota = Use the QUOTA clause to specify the maximum amount of space allocated to a user in the tablespace (cannot assign a quota to a temporary tablespace). A CREATE USER statement can have multiple QUOTA clauses for multiple tablespaces. UNLIMITED lets the user allocate space in the tablespace without bound.

Next we need to GRANT privileges to the INVOICE schema

```
GRANT CREATE SESSION TO invoice;
```

- this allows connection to an Oracle database

```
GRANT CREATE TABLE, CREATE VIEW, CREATE TRIGGER, CREATE SEQUENCE TO invoice;
```

- allow creating the tables, views, triggers, and sequences for the application:

Now that the schema (called invoice) has been created with the necessary privileges, you can create objects in the schema.

- create objects such as tables using the CREATE TABLE statement you would need connect as the user INVOICE. All objects created would be created in the INVOICE schema.
- you could use the CREATE SCHEMA statement to create multiple objects in a single SQL statement.

You might think that the CREATE SCHEMA statement would create your schema, but that is not the case. The CREATE SCHEMA statement is used only to create objects (ie: tables, views, etc) in your schema in a single SQL statement, but does not actually create the schema itself.

You might think that the CREATE SCHEMA statement would create your schema, but that is not the case. The CREATE SCHEMA statement is used only to create objects (ie: tables, views, etc) in your schema in a single SQL statement, but does not actually create the schema itself.

The syntax for the CREATE SCHEMA statement is as follows:

```
CREATE SCHEMA AUTHORIZATION schema_name [create_table_statement] [create_view_statement]
[grant_statement];
```

Eg:

```
CREATE SCHEMA AUTHORIZATION invoice CREATE TABLE products ( product_id number(10) not null,
product_name varchar2(50) not null, category varchar2(50), CONSTRAINT products_pk PRIMARY KEY
(product_id) ) CREATE TABLE suppliers ( supplier_id number(10) not null, supplier_name varchar2(50) not null,
city varchar2(25), CONSTRAINT suppliers_pk PRIMARY KEY (supplier_id) );
```

To execute a CREATE SCHEMA statement, Oracle Database executes each included statement. If all statements execute successfully, then the database commits the transaction. If any statement results in an error, then the database rolls back all the statements.

To execute a CREATE SCHEMA statement, Oracle Database executes each included statement. If all statements execute successfully, then the database commits the transaction. If any statement results in an error, then the database rolls back all the statements.

DDL – Data Definition Language

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

CREATE TABLE

tblProductDetail

tblProductDetail

tblProduct

tblProduct

1

8

1

tblVendor

tblVendor

VendorID
VendorFName
VendorLName
VendorPhone
VendorContact

VendorID **8**

∞

ProductID
ProductCost

ProductID
ProductPartNo
ProductDesc
ProductSellPrice

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 defined at the time of table creation or after table creation, so that constraints can be placed later by altering the structure of the 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 defined 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.	CREATE TABLE tblProduct (ProductID NUMBER(38) PRIMARY KEY,

INLINE **Note: No name has been given to the constraint**	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**	CREATE TABLE tblProductDetail (ProductID NUMBER(38), VendorID NUMBER(38),
Note: No name given to the constraint	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. **Out-of-Line**	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

tblProductDetail

tblProductDetail

tblProduct

tblProduct

1

8

1

tblVendor

tblVendor

VendorID
VendorFName
VendorLName
VendorPhone
VendorContact

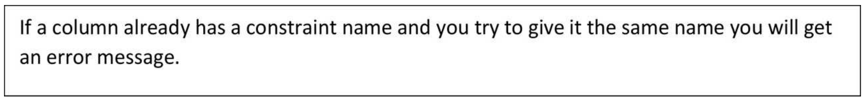
VendorID **8** ∞
ProductID
ProductCost

ProductID
ProductPartNo
ProductDesc
ProductSellPrice

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;
Add a constraint and give it a name	ALTER TABLE tblProductDetail ADD CONSTRAINT ProductDetail_PK PRIMARY KEY (ProductID, VendorID);
Removing (dropping) a column	ALTER TABLE tblVendor DROP COLUMN VendorContact;
Removing multiple columns	ALTER TABLE tblVendor DROP (VendorContact, VendorPhone);
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.



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

Dropping a Table

Use the DROP TABLE statement to move a table to the recycle bin or to remove the table and all its data from the database entirely.

When dropping a table:

- All rows from the table are dropped
- All table indexes and domain indexes are dropped, as well as any triggers defined on the table, regardless of who created them or whose schema contains them
- If the table is a base table for a view, a container or master table of a materialized view, or if it is referenced in a stored procedure, function, or package, then the database invalidates these dependent objects but does not drop them. You cannot use these objects unless you re-create the table or drop and re-create the objects so that they no longer depend on the table

Example:

```
DROP TABLE tblProduct;
```

CASCADE CONSTRAINTS

Specify CASCADE CONSTRAINTS to drop all referential integrity constraints that refer to primary and unique keys in the dropped table. If you omit this clause, and such referential integrity constraints exist, then the database returns an error and does not drop the table.

Example:

```
DROP TABLE tblProduct CASCADE CONSTRAINTS;
```

PURGE

Specify PURGE if you want to drop the table and release the space associated with it in a single step. If you specify PURGE, then the database does not place the table and its dependent objects into the recycle bin.

You cannot roll back a DROP TABLE statement with the PURGE clause, nor can you recover the table if you have dropped it with the PURGE clause.

You cannot roll back a DROP TABLE statement with the PURGE clause, nor can you recover the table if you have dropped it with the PURGE clause.

Using this clause is equivalent to first dropping the table and then purging it from the recycle bin. This clause lets you save one step in the process. It also provides enhanced security if you want to prevent sensitive material from appearing in the recycle bin.

Example:

```
DROP TABLE tblProduct PURGE;
```

Recover a Dropped Table

Use the FLASHBACK TABLE ... TO BEFORE DROP statement to recover objects from the recycle bin. You can specify either the name of the table in the recycle bin or the original table name.

Example using original table name:

```
FLASHBACK TABLE tblproduct TO BEFORE DROP;
```

Example using recycle bin name:

```
FLASHBACK TABLE "BIN$gk3lsj/3akk5hg3j2lkl5j3d==$0" TO BEFORE DROP;
```

Note the use of quotations due to the possibility of special characters appearing in the recycle bin object names.

Note the use of quotations due to the possibility of special characters appearing in the recycle bin object names.

Viewing the Contents of the Recycle Bin

```
SELECT object_name as recycle_name, original_name, type FROM recyclebin;

RECYCLE_NAME ORIGINAL_NAME TYPE

-------------------------------- ---------------------- ----------

BIN$gk3lsj/3akk5hg3j2lkl5j3d==$0 EMPLOYEE_DEMO TABLE

BIN$JKS983293M1dsab4gsz/I249==$0 I_EMP_DEMO INDEX

BIN$NR72JJN38KM1dsaM4gI348as==$0 LOB_EMP_DEMO LOB

BIN$JKJ399SLKnaslkJSLK330SIK==$0 LOB_I_EMP_DEMO LOB INDEX
```

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) ;
```

Insert Multiple Rows with a Single Insert Statement

The Oracle INSERT ALL statement is used to add multiple rows with a single INSERT statement. The rows can be inserted into one table or multiple tables using only one SQL command.

Syntax:

```
INSERT ALL

INTO mytable (column1, column2, column_n) VALUES (expr1, expr2, expr_n)

INTO mytable (column1, column2, column_n) VALUES (expr1, expr2, expr_n)

INTO mytable (column1, column2, column_n) VALUES (expr1, expr2, expr_n)

SELECT * FROM dual;
```

Example – Insert into 1 table

```
INSERT ALL

INTO suppliers (supplier_id, supplier_name) VALUES (1000, 'IBM')

INTO suppliers (supplier_id, supplier_name) VALUES (2000, 'Microsoft')

INTO suppliers (supplier_id, supplier_name) VALUES (3000, 'Google')

SELECT * FROM dual;
```

Example – insert into multiple tables

```
INSERT ALL

INTO suppliers (supplier_id, supplier_name) VALUES (1000, 'IBM')

INTO suppliers (supplier_id, supplier_name) VALUES (2000, 'Microsoft')

INTO customers (customer_id, customer_name, city) VALUES (99999, 'Jones Construction', 'New York')

SELECT * FROM dual;
```

Note: You cannot use SEQUENCES with the INSERT ALL statement since it will not increment the sequence as it is only a single INSERT statement.

Note: You cannot use SEQUENCES with the INSERT ALL statement since it will not increment the sequence as it is only a single INSERT statement.

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 separated by commas, 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 use a WHERE clause 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).
B. 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.

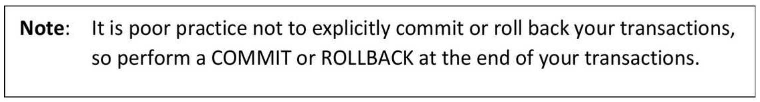
A transaction begins when one of the following events occurs:

- 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.

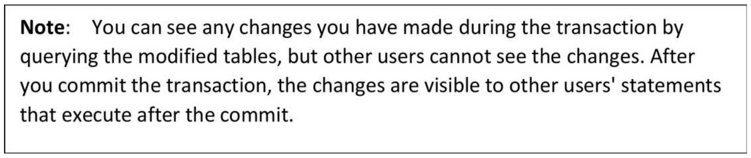
A transaction ends when one of the following events occurs:

- 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: 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.

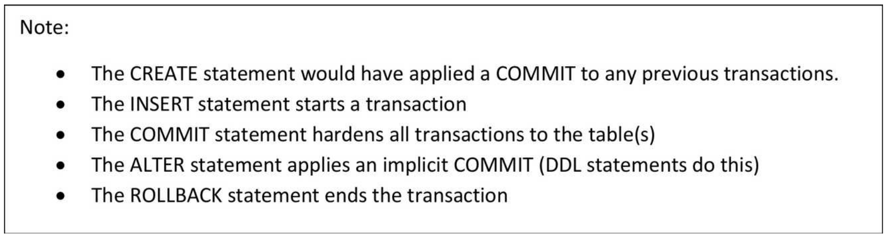

Note: You can see any changes you have made during the transaction by querying the modified tables, but other users cannot see the changes. After you commit the transaction, the changes are visible to other users' statements that execute after the commit.

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;

Note:

- The CREATE statement would have applied a COMMIT to any previous transactions.
- The INSERT statement starts a transaction
- The COMMIT statement hardens all transactions to the table(s)
- The ALTER statement applies an implicit COMMIT (DDL statements do this)
- The ROLLBACK statement ends the transaction


Note:

- The CREATE statement would have applied a COMMIT to any previous transactions.
- The INSERT statement starts a transaction
- The COMMIT statement hardens all transactions to the table(s)
- The ALTER statement applies an implicit COMMIT (DDL statements do this)
- The ROLLBACK statement ends the transaction

Result:

SELECT ID, name FROM stud;

ID	**NAME**
1	Bob
2	Sara
4	Li

Rollback to a Specific Point Using SAVEPOINT

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

Note: The ROLLBACK TO SAVEPOINT does NOT end the transaction.

Note: The ROLLBACK TO SAVEPOINT does NOT end the transaction.

Result:

SELECT ID, name FROM stud;

ID	NAME
1	Bob
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, and newer versions support Sequence
- 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.

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

Sequence_name.NEXTVAL will return the next value in the sequence

Sequence_name.CURRVAL will return the current value of the sequence

Sequence_name.NEXTVAL will return the next value in the sequence

Sequence_name.CURRVAL will return the current value of the sequence

Selected Sequence Parameters

INCREMENT BY Specify the interval between sequence numbers. This integer value can be any positive or negative integer, but it cannot be 0. This value can have 28 or fewer digits. The absolute of this value must be less than the difference of MAXVALUE and MINVALUE. If this value is negative, then the sequence descends. If the value is positive, then the sequence ascends. If you omit this clause, then the interval defaults to 1.

START WITH Specify the first sequence number to be generated. Use this clause to start an ascending sequence at a value greater than its minimum or to start a descending sequence at a value less than its maximum. For ascending sequences, the default value is the minimum value of the sequence or 1 if no minimum is supplied. For descending sequences, the default value is the maximum value of the sequence. This integer value can have 28 or fewer digits.

Note: If you wish to start the sequence at 0 then you must supply the MINVALUE parameter and set it to 0.

Note: If you wish to start the sequence at 0 then you must supply the MINVALUE parameter and set it to 0.

MAXVALUE Specify the maximum value the sequence can generate. This integer value can have 28 or fewer digits. MAXVALUE must be equal to or greater than START WITH and must be greater than MINVALUE.

NOMAXVALUE Specify NOMAXVALUE to indicate a maximum value of 1027 for an ascending sequence or -1 for a descending sequence. This is the default.

MINVALUE Specify the minimum value of the sequence. This integer value can have 28 or fewer digits. MINVALUE must be less than or equal to START WITH and must be less than MAXVALUE.

NOMINVALUE Specify NOMINVALUE to indicate a minimum value of 1 for an ascending sequence or -1026 for a descending sequence. This is the default.

CYCLE Specify CYCLE to indicate that the sequence continues to generate values after reaching either its maximum or minimum value. After an ascending sequence reaches its maximum value, it generates its minimum value. After a descending sequence reaches its minimum, it generates its maximum value.

NOCYCLE Specify NOCYCLE to indicate that the sequence cannot generate more values after reaching its maximum or minimum value. This is the default.

CACHE Specify how many values of the sequence the database preallocates and keeps in memory for faster access. This integer value can have 28 or fewer digits. The minimum value for this parameter is 2. For sequences that cycle, this value must be less than the number of values in the cycle. You cannot cache more values than will fit in a given cycle of sequence numbers.

NOCACHE Specify NOCACHE to indicate that values of the sequence are not preallocated. If you omit both CACHE and NOCACHE, then the database caches 20 sequence numbers by default.

Alter Sequence

This statement sets a new maximum value for a sequence named ‘customers_seq’:

ALTER SEQUENCE customers_seq

MAXVALUE 1500;

Drop Sequence

DROP SEQUENCE customers_seq;

Managing Users

Each Oracle database has a list of valid database users. To access a database, a user must run a database application and connect to the database instance using a valid user name defined in the database.

Creating Users

You create a database user with the CREATE USER statement. To create a user, you must have the CREATE USER system privilege. Because it is a powerful privilege, a DBA or security administrator is normally the only user who has the CREATE USER system privilege.

Example:

CREATE USER jward IDENTIFIED BY AZ7BC2 DEFAULT TABLESPACE data_ts QUOTA 500K ON data_ts;

A newly created user cannot connect to the database until granted the CREATE SESSION system privilege. (this is synonymous with the GRANT CONNECT privilege)

A newly created user cannot connect to the database until granted the `CREATE SESSION` system privilege. (this is synonymous with the GRANT CONNECT privilege)

GRANT create session TO jward;

Specifying a Name

Within each database, a user name must be unique with respect to other user names and roles. A user and role cannot have the same name. Furthermore, each user has an associated schema. Within a schema, each schema object must have a unique name.

Setting Up User Authentication

The new user has to be authenticated to use the database. In this case, the connecting user must supply the correct password to the database to connect successfully.

Assigning a Default Tablespace

Each user should have a default tablespace. When a user creates a schema object and specifies no tablespace to contain it, Oracle Database stores the object in the default user tablespace.

The default setting for the default tablespaces of all users is the SYSTEM tablespace. If a user does not create objects, and has no privileges to do so, then this default setting is fine. However, if a user is likely to create any type of object, then you should specifically assign the user a default tablespace. Using a tablespace other than SYSTEM reduces contention between data dictionary objects and user objects for the same data files. In general, it is not advisable for user data to be stored in the SYSTEM tablespace.

When you specify the default tablespace for a user, also specify a quota on that tablespace.

In our example, the default tablespace for user jward is data_ts, and his quota on that tablespace is 500K.

When you specify the default tablespace for a user, also specify a quota on that tablespace.

In our example, the default tablespace for user jward is data_ts, and his quota on that tablespace is 500K.

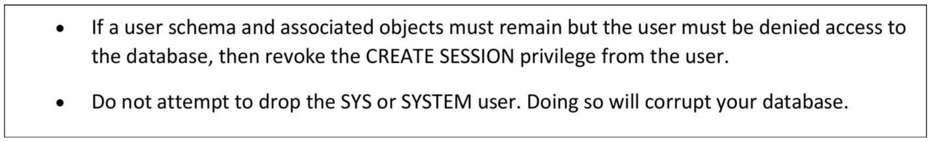
By default, a user has no quota on any tablespace in the database. If the user has the privilege to create a schema object, then you must assign a quota to allow the user to create objects. Minimally, assign users a quota for the default tablespace, and additional quotas for other tablespaces in which they can create objects.

By default, a user has no quota on any tablespace in the database. If the user has the privilege to create a schema object, then you must assign a quota to allow the user to create objects. Minimally, assign users a quota for the default tablespace, and additional quotas for other tablespaces in which they can create objects.

Dropping Users

When a user is dropped, the user and associated schema are removed from the data dictionary and all schema objects contained in the user schema, if any, are immediately dropped.

- If a user schema and associated objects must remain but the user must be denied access to the database, then revoke the CREATE SESSION privilege from the user.
- Do not attempt to drop the SYS or SYSTEM user. Doing so will corrupt your database.



- If a user schema and associated objects must remain but the user must be denied access to the database, then revoke the CREATE SESSION privilege from the user.
- Do not attempt to drop the SYS or SYSTEM user. Doing so will corrupt your database.

A. user that is currently connected to a database cannot be dropped. To drop a connected user, you must first terminate the user sessions using the SQL statement ALTER SYSTEM with the KILL SESSION clause (see next section).
B. the user's schema contains any dependent schema objects, then use the CASCADE option to drop the user and all associated objects and foreign keys that depend on the tables of the user successfully.

- If you do not specify CASCADE and the user schema contains dependent objects, then an error message is returned and the user is not dropped.
- Before dropping a user whose schema contains objects, thoroughly investigate which objects the user's schema contains and the implications of dropping them.
- Pay attention to any unknown cascading effects. For example, if you intend to drop a user who owns a table, then check whether any views or procedures depend on that particular table.

Example:

The following statement drops the user, jones and all associated objects and foreign keys that depend on the tables owned by jones.

DROP USER jones CASCADE;

Steps to Identify a Session You Want to Terminate

Killing sessions can be very destructive if you kill the wrong session, so be very careful when identifying the session to be killed. If you kill a session belonging to a background process you will cause an instance crash.

Identify the offending session using the [G]V$SESSION and [G]V$PROCESS views as follows.

SET LINESIZE 100

COLUMN spid FORMAT A10

COLUMN username FORMAT A10

COLUMN program FORMAT A45

SELECT s.inst_id,

s.sid,

s.serial#,

p.spid,

s.username,

s.program

FROM gv$session s

```
JOIN gv$process p ON p.addr = s.paddr AND p.inst_id = s.inst_id

WHERE s.type != 'BACKGROUND';
```

Result:

```
INST_ID SID SERIAL# SPID USERNAME PROGRAM

---------- ---------- ---------- ---------- ---------- -----------------------------------------

1 30 15 3859 TEST sqlplus@oel5-11gr2.localdomain (TNS V1-V3)

1 23 287 3834 SYS sqlplus@oel5-11gr2.localdomain (TNS V1-V3)

1 40 387 4663 oracle@oel5-11gr2.localdomain (J000)

1 38 125 4665 oracle@oel5-11gr2.localdomain (J001)
```

The SID and SERIAL# values of the relevant session can then be substituted into the ALTER SYSTEM KILL SESSION command:

```
SQL> ALTER SYSTEM KILL SESSION 'sid, serial#';
```

Granting Object Privileges

You also use the GRANT statement to grant object privileges to roles and users. To grant an object privilege, you must fulfill one of the following conditions:

- You own the object specified.
- You possess the GRANT ANY OBJECT PRIVILEGE system privilege that enables you to grant and revoke privileges on behalf of the object owner.
- The WITH GRANT OPTION clause was specified when you were granted the object privilege by its owner.

  System privileges and roles cannot be granted along with object privileges in the same GRANT statement.

System privileges and roles cannot be granted along with object privileges in the same GRANT statement.

You can grant users various privileges to tables. These privileges can be any combination of SELECT, INSERT, UPDATE, DELETE, REFERENCES, ALTER, INDEX, or ALL

ALTER	**Change the table definition with the ALTER TABLE statement.**
DELETE	Remove rows from the table with the DELETE statement. Note: You must grant the SELECT privilege on the table along with the DELETE privilege.
INDEX	Create an index on the table with the CREATE INDEX statement.
INSERT	Add new rows to the table with the INSERT statement.

REFERENCES	Create a constraint that refers to the table. You cannot grant this privilege to a role.
SELECT	Query the table with the SELECT statement.
UPDATE	Change data in the table with the UPDATE statement.
ALL	Specify ALL to grant all the privileges for the object that you have been granted with the GRANT OPTION. The user who owns the schema containing an object automatically has all privileges on the object with the GRANT OPTION.
	Note: You must grant the SELECT privilege on the table along with the UPDATE privilege.

Example:

The following statement grants the SELECT, INSERT, and DELETE object privileges for all columns of the emp table to the users jfee and tsmith:

GRANT SELECT, INSERT, DELETE ON emp TO jfee, tsmith;

To grant all object privileges on the salary view to the user jfee, use the ALL keyword, as shown in the following example:

GRANT ALL ON salary TO jfee;

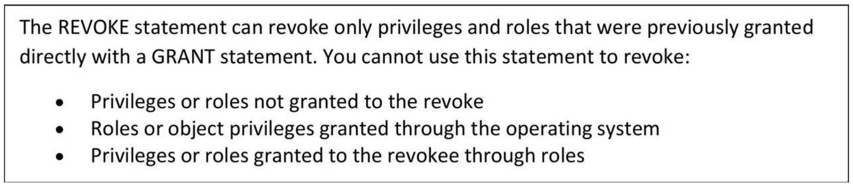
Revoking Privileges

Use the REVOKE statement to:

- Revoke system privileges from users and roles
- Revoke roles from users and roles
- Revoke object privileges for a particular object from users and roles

The REVOKE statement can revoke only privileges and roles that were previously granted directly with a GRANT statement. You cannot use this statement to revoke:

- Privileges or roles not granted to the revoke
- Roles or object privileges granted through the operating system
- Privileges or roles granted to the revokee through roles



The REVOKE statement can revoke only privileges and roles that were previously granted directly with a GRANT statement. You cannot use this statement to revoke:

- Privileges or roles not granted to the revoke
- Roles or object privileges granted through the operating system
- Privileges or roles granted to the revokee through roles

Examples:

Assume you grant DELETE, INSERT, SELECT, and UPDATE privileges on the table orders to the user hr with the following statement:

GRANT ALL

ON orders TO hr;

To revoke the DELETE privilege on orders from hr, issue the following statement:

REVOKE DELETE

ON orders FROM hr;

Revoking All Object Privileges from a User: Example To revoke the remaining privileges on orders that you granted to hr, issue the following statement:

REVOKE ALL

ON orders FROM hr;

Views

Use the CREATE VIEW statement to define a view, which is a logical table based on one or more tables or views. A view contains no data itself. The tables upon which a view is based are called base tables.

To create a view in your own schema, you must have the CREATE VIEW system privilege. To create a view in another user's schema, you must have the CREATE ANY VIEW system privilege.

There are two basic types of views:

- Simple views, which contain a subquery that retrieves from one base table

- Complex views, which contain a subquery that
  - Retrieves from multiple base tables
  - Groups rows using a GROUP BY or DISTINCT clause
  - Contains a function call

You create a view using CREATE VIEW statement, which has the following simplified syntax:

CREATE [OR REPLACE] VIEW [{FORCE | NOFORCE}] VIEW view_name

[(alias_name[, alias_name ...])] AS subquery

[WITH {CHECK OPTION | READ ONLY} CONSTRAINT constraint_name];

WHERE

- OR REPLACE means the view replaces an existing view.
- FORCE means the view is to be created even if the base tables don’t exist.
- NOFORCE means the view is not created if the base tables don’t exist. NOFORCE is the default.
- view_name is the name of the view.
- alias_name is the name of an alias for an expression in the subquery. There must be the same number of aliases as there are expressions in the subquery.
- subquery is the subquery that retrieves from the base tables. If you’ve supplied aliases, you can use those aliases in the list after the SELECT.
- WITH CHECK OPTION means that only the rows that would be retrieved by the subquery can be inserted, updated, or deleted. By default, the rows are not checked.
- constraint_name is the name of the WITH CHECK OPTION or WITH READ ONLY constraint.
- WITH READ ONLY means the rows may only read from the base tables.

Examples:

The following statement creates a simple view of the sample table employees named emp_view. The view shows the employees in department 20 and their annual salary:

```
CREATE VIEW emp_view AS

SELECT last_name, salary*12 annual_salary

FROM employees

WHERE department_id = 20;
```

The following example connects as the store user and creates a view named cheap_products_view whose subquery retrieves products only where the price is less than $15:

```
CREATE VIEW cheap_products_view AS SELECT * FROM tblProduct WHERE ProductPrice < 15;
```

The next example creates a view named employees_view whose subquery retrieves all the columns from the employees table except salary:

```
CREATE VIEW employees_view AS SELECT EmployeeID, EmployeeFirstName, EmployeeLastName,
EmployeeTitle FROM tblEmployee;
```

Performing a Query on a View

Once you've created a view, you can use it to access the base table. The following query retrieves rows from cheap_products_view:

```
SELECT ProductID, ProductName, ProductPrice FROM cheap_products_view;
```

DML Operations in a View

With some restrictions, rows can be inserted into, updated in, or deleted from a base table using a view.

Restrictions on DML operations for views use the following criteria in the order listed:

1. If a view is defined by a query that contains SET or DISTINCT operators, a GROUP BY clause, or a group function, then rows cannot be inserted into, updated in, or deleted from the base tables using the view.
2. If a view is defined with WITH CHECK OPTION, a row cannot be inserted into, or updated in, the base table (using the view), if the view cannot select the row from the base table.
3. If a NOT NULL column that does not have a DEFAULT clause is omitted from the view, then a row cannot be inserted into the base table using the view.
4. If the view was created by using an expression, such as DECODE(deptno, 10, "SALES", ...), then rows cannot be inserted into or updated in the base table using the view.

   The owner of the view (whether it is you or another user) must have been explicitly granted privileges to access all objects referenced in the view definition. The owner cannot have obtained these privileges through roles. Also, the functionality of the view depends on the privileges of the view owner. For example, if the owner of the view has only the INSERT privilege for Scott's emp table, then the view can be used only to insert new rows into the emp table, not to SELECT, UPDATE, or DELETE rows.

The owner of the view (whether it is you or another user) must have been explicitly granted privileges to access all objects referenced in the view definition. The owner cannot have obtained these privileges through roles. Also, the functionality of the view depends on the privileges of the view owner. For example, if the owner of the view has only the INSERT privilege for Scott's emp table, then the view can be used only to insert new rows into the emp table, not to SELECT, UPDATE, or DELETE rows.

Performing an INSERT Using a View

You can perform DML statements using cheap_products_view. The following example performs an INSERT using cheap_products_view:

```
INSERT INTO cheap_products_view ( ProductID, ProductTypeID, ProductName, ProductPrice ) VALUES ( 13, 1, 'Western Front', 13.50 );
```

Note: You can perform DML statements only with simple views. Complex views don't support DML.

Note: You can perform DML statements only with simple views. Complex views don't support DML.

Because cheap_products_view didn't use WITH CHECK OPTION, you can insert, update, and delete rows that aren't retrievable by the view. The following example inserts a row whose price is $16.50 (this is greater than $15 and therefore not retrievable by the view):

```
INSERT INTO cheap_products_view (

ProductID, ProductTypeID, ProductName, ProductPrice

) VALUES (

14, 1, 'Eastern Front', 16.50

);
```

A select statement would not display the above inserted record because the view is filtered on only products whose price is less than 15.

If you tried to insert a record into the employee table using the employee view from above the salary column would contain a null since the employee view from above has not given you the salary column of the base table.

Creating a View with a CHECK OPTION Constraint

You can specify that DML statements on a view must satisfy the subquery using a CHECK OPTION constraint. For example, the following statement creates a view named cheap_products_view2 that has a CHECK OPTION constraint:

```
CREATE VIEW cheap_products_view AS SELECT * FROM tblProduct WHERE ProductPrice < 15 WITH CHECK OPTION CONSTRAINT cheap_products_view_price;
```

Now if we tried to insert a record into the product table using this view and the price was greater than 15 we would get an error message and the record would not get inserted.

```
ERROR at line 1: ORA-01402: view WITH CHECK OPTION where-clause violation
```

If you use the READ ONLY constraint then no records can be added using DML.

If you use the READ ONLY constraint then no records can be added using DML.

Creating Complex Views

Complex views contain subqueries that

- Retrieve rows from multiple base tables.

- Group rows using a GROUP BY or DISTINCT clause.
- Contain a function call.

The following example creates a view named products_and_types_view whose subquery performs a full outer join on the products and product_types tables.

```
CREATE VIEW products_and_types_view AS SELECT p.ProductID, p.ProductName ‘Product Name’,
pt.ProductTypeName ‘Product Type’, p.ProductPrice FROM tblProduct p FULL OUTER JOIN tblProductType pt
USING (ProductTypeID) ORDER BY p.ProductID;
```

Change the Definition of an Existing View

You can change the definition of an existing view without dropping, re-creating, and regranting object privileges previously granted on it.

Example:

Assume that you created the sales_staff view as shown earlier, and, in addition, you granted several object privileges to roles and other users. However, now you need to redefine the sales_staff view to change the department number specified in the WHERE clause. You can replace the current version of the sales_staff view with the following statement:

```
CREATE OR REPLACE VIEW sales_staff AS

SELECT empno, ename, deptno

FROM emp

WHERE deptno = 30

WITH CHECK OPTION CONSTRAINT sales_staff_cnst;
```

You can redefine the view with a CREATE VIEW statement that contains the OR REPLACE clause. The OR REPLACE clause replaces the current definition of a view and preserves the current security authorizations.

You can redefine the view with a CREATE VIEW statement that contains the **OR REPLACE** clause. The OR REPLACE clause replaces the current definition of a view and preserves the current security authorizations.

You can alter the constraints on a view using ALTER VIEW. The following example uses ALTER VIEW to drop the cheap_products_view_price constraint from cheap_products_view:

```
ALTER VIEW cheap_products_view DROP CONSTRAINT cheap_products_view_price;
```

Dropping a View

You drop a view using DROP VIEW. The following example drops cheap_products_view:

```
DROP VIEW cheap_products_view;
```

Indexes

A database index is a data structure that improves the speed of data retrieval operations on a database table at the cost of additional writes and storage space to maintain the index data structure. Indexes are used to quickly locate data without having to search every row in a database table every time a database table is accessed. Indexes can be created using one or more columns of a database table, providing the basis for both rapid random lookups and efficient access of ordered records.

Indexes can be unique or non-unique. Unique indexes guarantee that no two rows of a table have duplicate values in the key column (or columns). Non-unique indexes do not impose this restriction on the column values.

Creating a Non-Unique Index Explicitly

You can create indexes explicitly (outside of integrity constraints) using the SQL statement CREATE INDEX. The following statement creates an index named emp_ename for the ename column of the emp table:

CREATE INDEX emp_ename ON emp(ename);

Creating a Unique Index Explicitly

CREATE UNIQUE INDEX dept_unique_index ON dept (dname)

Renaming an Index

To rename an index, issue this statement:

ALTER INDEX index_name RENAME TO new_name;

Dropping Indexes

If a table is dropped, all associated indexes are dropped automatically.

If a table is dropped, all associated indexes are dropped automatically.

If you need to modify an index you must drop it and recreate it.

If you need to modify an index you must drop it and recreate it.

How you drop an index depends on whether you created the index explicitly with a CREATE INDEX statement, or implicitly by defining a key constraint on a table.

How you drop an index depends on whether you created the index explicitly with a CREATE INDEX statement, or implicitly by defining a key constraint on a table.

If you created the index explicitly with the CREATE INDEX statement, then you can drop the index with the DROP INDEX statement. The following statement drops the emp_ename index:

DROP INDEX emp_ename;

You cannot drop only the index associated with an enabled UNIQUE key or PRIMARY KEY constraint. To drop a constraints associated index, you must disable or drop the constraint itself.

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,,…);

Oracle Data Dictionary

Introduction

One of the most important parts of an Oracle database is its data dictionary, which is a read-only set of tables that provides information about the database. A data dictionary contains:

- The definitions of all schema objects in the database (tables, views, indexes, clusters, synonyms, sequences, procedures, functions, packages, triggers, and so on)
- How much space has been allocated for, and is currently used by, the schema objects
- Default values for columns
- Integrity constraint information
- The names of Oracle users
- Privileges and roles each user has been granted
- Auditing information, such as who has accessed or updated various schema objects
- Other general database information

The data dictionary is structured in tables and views, just like other database data. All the data dictionary tables and views for a given database are stored in that database's SYSTEM tablespace.

Not only is the data dictionary central to every Oracle database, it is an important tool for all users, from end users to application designers and database administrators. Use SQL statements to access the data dictionary. Because the data dictionary is read-only, you can issue only queries (SELECT statements) against its tables and views.

How it is Used

The data dictionary has three primary uses:

- Oracle accesses the data dictionary to find information about users, schema objects, and storage structures.
- Oracle modifies the data dictionary every time that a data definition language (DDL) statement is issued.
- Any Oracle user can use the data dictionary as a read-only reference for information about the database.

Selected System Views/Tables

USER_TABLES - Information about the relational tables

Column	Returns
TABLE_NAME	Name of the table
TABLESPACE_NAME	Name of the tablespace containing the table
	Name of the cluster, if any, to which the table belongs

CLUSTER_NAME	
IOT_NAME	Name of the index-only table, if any, to which the overflow or mapping table entry belongs
PCT_FREE	Minimum percentage of free space in a block
PCT_USED	Minimum percentage of used space in a block
INI_TRANS	Initial number of transactions
MAX_TRANS	Maximum number of transactions
INITIAL_EXTENT	Size of the initial extent in bytes
NEXT_EXTENT	Size of secondary extents in bytes
MIN_EXTENTS	Minimum number of extents allowed in the segment
MAX_EXTENTS	Maximum number of extents allowed in the segment
PCT_INCREASE	Percentage increase in extent size
FREELISTS	Number of process freelists allocated in this segment
FREELIST_GROUPS	Number of freelist groups allocated in this segment
LOGGING	Logging attribute
BACKED_UP	Has table been backed up since last modification?
NUM_ROWS	The number of rows in the table
BLOCKS	The number of used blocks in the table
EMPTY_BLOCKS	The number of empty (never used) blocks in the table

AVG_SPACE	The average available free space in the table
CHAIN_CNT	The number of chained rows in the table
AVG_ROW_LEN	The average row length, including row overhead
AVG_SPACE_FREELIST_BLOCKS	The average freespace of all blocks on a freelist
NUM_FREELIST_BLOCKS	The number of blocks on the freelist
DEGREE	The number of threads per instance for scanning the table
INSTANCES	The number of instances across which the table is to be scanned
CACHE	Whether the table is to be cached in the buffer cache
TABLE_LOCK	Whether table locking is enabled or disabled
SAMPLE_SIZE	The sample size used in analyzing this table
LAST_ANALYZED	The date of the most recent time this table was analyzed
PARTITIONED	Is this table partitioned? YES or NO
IOT_TYPE	If index-only table, then IOT_TYPE is IOT or IOT_OVERFLOW or IOT_MAPPING else NULL
TEMPORARY	Can the current session only see data that it place in this object itself?
SECONDARY	Is this table object created as part of icreate for domain indexes?
NESTED	Is the table a nested table?
BUFFER_POOL	The default buffer pool to be used for table blocks

ROW_MOVEMENT	Whether partitioned row movement is enabled or disabled
GLOBAL_STATS	Are the statistics calculated without merging underlying partitions?
USER_STATS	Were the statistics entered directly by the user?
DURATION	If temporary table, then duration is sys$session or sys$transaction else NULL
SKIP_CORRUPT	Whether skip corrupt blocks is enabled or disabled
MONITORING	Should we keep track of the amount of modification?
CLUSTER_OWNER	Owner of the cluster, if any, to which the table belongs
DEPENDENCIES	Should we keep track of row level dependencies?

Ex: SELECT table_name, num_rows FROM USER_TABLES;

USER_CONS_COLUMNS – Information about the columns in each constraint

Column	**Returns**
OWNER	Owner of the constraint definition
CONSTRAINT_NAME	Name of the constraint definition
TABLE_NAME	Name of the table with constraint definition
COLUMN_NAME	Name of the column or attribute of the object type column specified in the constraint definition
POSITION	Original position of column or attribute in the definition of the object

Ex: SELECT owner, table_name FROM ALL_CONS_COLUMNS;

USER_CONSTRAINTS - describes constraint definitions on tables in the current user's schema

OWNER	**Owner of the constraint definition**

CONSTRAINT_NAME	Name of the constraint definition
CONSTRAINT_TYPE	Type of constraint definition: • C (check constraint on a table) • P (primary key) • U (unique key) • R (referential integrity) • V (with check option, on a view) • O (with read only, on a view)
TABLE_NAME	Name associated with the table (or view) with constraint definition
SEARCH_CONDITION	Text of search condition for a check constraint
R_OWNER	Owner of table referred to in a referential constraint
R_CONSTRAINT_NAME	Name of the unique constraint definition for referenced table
DELETE_RULE	Delete rule for a referential constraint (CASCADE or NO ACTION)
STATUS	Enforcement status of constraint (ENABLED or DISABLED)
DEFERRABLE	Whether the constraint is deferrable
DEFERRED	Whether the constraint was initially deferred
VALIDATED	Whether all data obeys the constraint (VALIDATED or NOT VALIDATED)
GENERATED	Whether the name of the constraint is user or system generated
BAD	A YES value indicates that this constraint specifies a century in an ambiguous manner. To avoid errors resulting from this ambiguity, rewrite the constraint using the TO_DATE function with a four-digit year.
RELY	Whether an enabled constraint is enforced or unenforced.
LAST_CHANGE	When the constraint was last enabled or disabled

INDEX_OWNER	Name of the user owning the index
INDEX_NAME	Name of the index
INVALID	Whether the constraint is invalid
VIEW_ONLY	Whether the constraint depends on a view

More on Constraint Types

Constraint	**Description**	**Example**
Check	Check constraints validate that values in a given column meet a specific criteria. For example, you could create a check constraint on a varchar2 column so it only can contain the values T or F.	CREATE TABLE emp12 (empno NUMBER PRIMARY KEY, empname VARCHAR2(20), sal NUMBER(10,2) CHECK (sal between 1000 and 20000));
Primary Key	Primary key constraints define a column or series of columns that uniquely identify a given row in a table. Defining a primary key on a table is optional and you can only define a single primary key on a table. A primary key constraint can consist of one or many columns (up to 32). Any column that is defined as a primary key column is automatically set with a NOT NULL status.	CREATE TABLE emp12 (empno NUMBER PRIMARY KEY, empname VARCHAR2(20), sal NUMBER(10,2));
Unique	Unique constraints are like alternative primary key constraints. A unique constraint defines a column, or series of columns, that must be unique in value. You can have a number of unique constraints defined and the columns can have NULL values in them, unlike a column that belongs to a primary key constraint.	ALTER TABLE tblProduct ADD CONSTRAINT tblProduct_ProductCode_uq UNIQUE (ProductCode);
	A foreign key constraint is used to enforce a relationship	

Foreign Key (Referential Integrity)	between two tables.	
Not Null	NOT NULL constraints are inline and indicate that a column can not contain NULL values.	CREATE TABLE emp12 (empno NUMBER PRIMARY KEY, empname VARCHAR2(20) NOT NULL, sal NUMBER(10,2) CHECK (sal between 1000 and 20000));

Constraint	**Constraint Type**	**Meaning**
CHECK	C	The value for a column, or group of columns, must satisfy a certain condition.
NOT NULL	C	The column cannot store a null value. This is actually enforced as a CHECK constraint.
PRIMARY KEY	P	The primary key of a table. A primary key is made up of one or more columns that uniquely identify each row in a table.
FOREIGN KEY	R	A foreign key for a table. A foreign key references a column in another table or a column in the same table (known as a self-reference).
UNIQUE	U	The column, or group of columns, can store only unique values.
CHECK OPTION	V	Changes to the table rows made through a view must pass a check first.
READ ONLY	O	The view may only be read from.

Constraint Examples

Description	Example	Meaning
Adding a CHECK Constraint	ALTER TABLE tblOrder ADD CONSTRAINT OrderStatus_status_ck CHECK (OrderStatus IN ('PLACED', 'PENDING', 'SHIPPED'));	This constraint ensures the OrderStatus column is always set to PLACED, PENDING, or SHIPPED.
Adding a FOREIGN KEY Constraint	ALTER TABLE tblProduct ADD CONSTRAINT tblProduct_Vendor_by_fk VendorID REFERENCES tblVendor(VendorID);	Adds a new column called VendorID into the table tblProduct and adds a FOREIGN KEY constraint that references the tblVendor.VendorID column:
Adding a FOREIGN KEY Constraint	ALTER TABLE tblProduct ADD CONSTRAINT tblProduct_Vendor_by_fk FOREIGN KEY(VendorID) REFERENCES tblVendor(VendorID);	Add a foreign key constraint to an EXISTING column called VendorID
DELETE CASCADE for Foreign key constraint	ALTER TABLE tblProduct ADD CONSTRAINT tblProduct_Vendor_by_fk VendorID REFERENCES tblVendor(VendorID) ON DELETE CASCADE;	Specify that when a row in the parent table is deleted, any matching rows in the child table are also deleted.
ON DELETE SET NULL for Foreign key constraint	ALTER TABLE tblProduct ADD CONSTRAINT tblProduct_Vendor_by_fk VendorID REFERENCES tblVendor(VendorID) ON DELETE SET NULL;	Specify that when a row in the parent table is deleted, the foreign key column for the row (or rows) in the child table is set to null.

Adding a UNIQUE Constraint	ALTER TABLE tblProduct ADD CONSTRAINT tblProduct_ProductCode_uq UNIQUE (ProductCode);	Adds a UNIQUE constraint to the tblProduct .Product column:
Dropping a Constraint	ALTER TABLE tblProduct DROP CONSTRAINT tblProduct_ProductCode_uq;	Remove the constraint created above.

Get information on constraints	SELECT constraint_name, constraint_type, status
	
	FROM user_constraints
	
	WHERE table_name = tblProduct';
		Displays information on the constraints in the table tblProduct

Oracle Analyze

Use the ANALYZE statement to collect statistics, for example, to:

- Collect or delete statistics about an index or index partition, table or table partition, index-organized table, cluster, or scalar object attribute.
- Validate the structure of an index or index partition, table or table partition, index-organized table, cluster, or object reference (REF).
- Identify migrated and chained rows of a table or cluster.

Oracle Database collects the following statistics for a table. Statistics marked with an asterisk are always computed exactly. Table statistics, including the status of domain indexes, appear in the data dictionary views USER_TABLES, ALL_TABLES, and DBA_TABLES in the columns shown in parentheses.

- Number of rows (NUM_ROWS)
- * Number of data blocks below the high-water mark—that is, the number of data blocks that have been formatted to receive data, regardless whether they currently contain data or are empty (BLOCKS)
- * Number of data blocks allocated to the table that have never been used (EMPTY_BLOCKS)
- Average available free space in each data block in bytes (AVG_SPACE)
- Number of chained rows (CHAIN_COUNT)
- Average row length, including the row overhead, in bytes (AVG_ROW_LEN)