Lecture 10

Structured Query Language(SQL)

Dr. Vandana Kushwaha

Department of Computer Science

Institute of Science, BHU, Varanasi

Introduction

- **SQL** stands for **Structured Query Language.**
- **SQL** lets us to **create, access** and **manipulate** the **databases.**
- **SQL** became a standard of the **American National Standards Institute (ANSI)** in **1986**, and of the **International Organization for Standardization (ISO)** in **1987.**
- **SQL** uses the terms **table**, **row**, and **column** for the formal **relational model** terms ***relation***, ***tuple*** and ***attribute***, respectively.
- **SQL Commands** can be categorized as:

1. **Data Definition Language(DDL)** – Consists of commands which are used to **define** the database.
2. **Data Manipulation Language(DML)** – Consists of commands which are used to **manipulate** the data present in the database.
3. **Data Control Language(DCL)** – Consists of commands which deal with the user **permissions** and **controls** of the database system.

SQL Commands

Data manipulation language (DML)	SELECT, INSERT, UPDATE, DELETE
Data definition language (DDL)	CREATE, ALTER, DROP, RENAME, TRUNCATE
Data control language (DCL)	GRANT, REVOKE
Transaction control	COMMIT, ROLLBACK, SAVEPOINT

SQL Constraints

- **Constraints** are the **rules** enforced on **data columns** on **a table**.
- These are **used to limit** the **type of data** that can go into a table.
- This ensures the **accuracy** and **reliability** of the **data** in the database.
- **Constraints** can either be **column level** or **table level.**
- **Column level constraints** are applied only to **one column** whereas, **table level constraints** are applied to the **entire table.**
- Some of the most commonly used **constraints** available in **SQL:**
  - **NOT NULL Constraint**: Ensures that a column cannot have a **NULL value**.
  - **DEFAULT Constraint**: Provides a **default value** for a column when none is specified.
  - **UNIQUE Constraint**: Ensures that all the values in a column are **different.**

SQL Constraints

- **PRIMARY Key**: Uniquely identifies each row/record in a database table.
- **FOREIGN Key:** Uniquely identifies a row/record in any another database table.
- **CHECK Constraint:** The **CHECK constraint** ensures that all values in a column satisfy certain conditions.
- **INDEX:** Used to create and retrieve data from the database very quickly.

The CREATE TABLE Command in SQL

- The **CREATE TABLE** command is used to specify a **new relation** by giving it a **name** and specifying its **attributes** and initial **constraints.**
- The **attributes** are specified by:
  - Giving a **name,**
  - a **data type** to specify its **domain of values**, and
  - any attribute **constraints**, such as **NOT NULL.**
- The **key, entity integrity**, and **referential integrity** constraints can be specified within the **CREATE TABLE** statement after the attributes are declared, or they can be **added later** using the **ALTER TABLE** command.
- In **SQL,** the **attributes** in a **table** are considered to be ***ordered*** *in the sequence in which they are specified* in the **CREATE TABLE** statement.
- However, **rows (tuples)** are **not considered to be ordered** within a **relation.**

The CREATE TABLE Command in SQL

- **SYNTAX:**

```
CREATE TABLE table_name
( column1   datatype ,
column2   datatype,
.....
columnN   datatype,
);
```

- **Example:**

```
CREATE TABLE EMPLOYEE
( Fname VARCHAR(15) ,
Lname VARCHAR(15) ,
Ssn CHAR(9) ,
Bdate DATE,
Address VARCHAR(30)
);
```

Basic data types in SQL

- **Numeric Data Type:**
  - **Numeric** data types include **integer numbers** of various sizes and **floating-point (real) numbers** of various precision.
  - **Formatted numbers** can be declared by using **NUMBER(*i*,*j*)**.
    - Where ***i*, the *precision***, is the **total number of decimal digits**
    - *And* ***j*, the *scale***, is the **number of digits after the decimal point**.
  - The **default for scale is zero.**

Basic data types in SQL

- **Character Data Type:**
- **Character** is like **string** data types are either
  - **Fixed length—CHAR(*n*)** where ***n*** is the **number of characters** or
  - **Varying length—VARCHAR(*n*) or VARCHAR2(*n*)** , where ***n*** is the **maximum number of characters.**
- For **fixed length strings**, a shorter string is **padded** with **blank characters** to the **right.**
- **Example**, if the value 'Smith' is for an attribute of type CHAR(10), it is padded with five blank characters to become 'Smith'.
- **Padded blanks** are generally **ignored** when strings are compared.

Basic data types in SQL

- **DATE data type**
- The **DATE data type** allows you to store point-in-time values that include both date and time with a precision of one second.
- The **standard date format** for input and output is **DD-MON-YY** e.g., **01-JAN-17.**
- The following statement returns the **current date** with the standard date format by using the **SYSDATE function:**

```
SELECT sysdate FROM dual;
```

- The result is: **31-JUL-23**

Constraints in SQL

1. NOT NULL Constraint

- As **SQL** allows **NULLs** as attribute values, a ***constraint*** **NOT NULL** may be specified if NULL is not permitted for a particular attribute.
- This is always **implicitly specified** for the **attributes** that are part of the ***primary key*** of each relation.
- But it can be specified for any other attributes whose values are required not to be **NULL.**

- **Example:**

```
CREATE TABLE EMPLOYEE
( Name Varchar2(20) NOT NULL ,
  Dno Number NOT NULL DEFAULT 1,….
);
```

Constraints in SQL

2. DEFAULT Constraint

- It is also possible to define a ***default value*** for an attribute by appending the clause **DEFAULT** <value> to an attribute definition.
- The **default value** is included in any new tuple if an explicit value is not provided for that attribute.
- An **example** of specifying a default manager for a new department and a default department for a new employee.
- If no default clause is specified, the **default** *default value* is **NULL** for attributes *that do not have* the NOT NULL constraint.
- **Example:**

```
CREATE TABLE EMPLOYEE
(……,
Dno Number NOT NULL DEFAULT 1,
Mgr_ssn CHAR(9) NOT NULL DEFAULT '888665555',……);
```

Constraints in SQL

3. CHECK Constraint

- **CHECK Constraint** can **restrict attribute** or **domain values** using the **CHECK** clause following an attribute or domain definition.
- For **example,** suppose that **department numbers** are restricted to **integer numbers between 1 and 20.**
- Then, we can change the attribute declaration of Dnumber in the DEPARTMENT table to the following:

……

Dnumber Number **NOT NULL CHECK** (Dnumber > 0 **AND** Dnumber < 21);

Key Constraints

- **PRIMARY KEY Constraint**
- The **PRIMARY KEY** clause specifies one or more attributes that make up the **primary key** of a relation.
- If a **primary key** has a ***single*** **attribute**, the clause can follow the attribute directly or it can be defined at **attribute level.**
- **Example**: the primary key of **DEPARTMENT** can be specified at **column level** as follows:

```
CREATE TABLE DEPARTMENT
( . . . ,                    Dnumber Number PRIMARY KEY;….);
```

- **Primary key** can also be defined at **table leve**l as:

```
CREATE TABLE EMPLOYEE

( . . . ,   PRIMARY KEY (Ssn),…);
```

Create Constraint Command

- **Syntax:**

```
CREATE TABLE table_name
(
  column1 datatype null/not null,
  column2 datatype null/not null,
  ...

  CONSTRAINT constraint_name PRIMARY KEY (column1, column2, ... column_n)
);
```

- **Example**

```
CREATE TABLE supplier
(
  supplier_id numeric(10) not null,
  supplier_name varchar2(50) not null,
  contact_name varchar2(50),
  CONSTRAINT supplier_pk PRIMARY KEY (supplier_id)
);
```

Create Primary Key - Using ALTER TABLE statement

- **Syntax**

```
ALTER TABLE table_name
ADD CONSTRAINT constraint_name PRIMARY KEY (column1, column2, ... column_n);
```

- **Example:**

```
ALTER TABLE supplier
ADD CONSTRAINT supplier_pk PRIMARY KEY (supplier_id);
```

Drop Primary Key

- **Syntax**

```
ALTER TABLE table_name
DROP CONSTRAINT constraint_name;
```

- **Example**

```
ALTER TABLE supplier
DROP CONSTRAINT supplier_pk;
```

Key Constraints

- **UNIQUE Constraint**
- The **UNIQUE** clause specifies **alternate (secondary) keys**, as illustrated in the **DEPARTMENT** and **PROJECT** table declarations.
- **Example:**

CREATE TABLE DEPARTMENT

(…….,

Dname VARCHAR(15) **UNIQUE NOTNULL**;…);

Or

CREATE TABLE DEPARTMENT

(…….,

Dname VARCHAR(15) NOTNULL;…);

UNIQUE (Dname),…….);

Key Constraints

- **Referential integrity using FOREIGN KEY**
- Referential integrity is specified via the **FOREIGN KEY** clause at **table level**, as shown below:

```
CREATE TABLE EMPLOYEE
(…..,
Ssn CHAR(9) NOT NULL,
Salary NUMBER(10,2),
Super_ssn CHAR(9),
Dno NUMBER NOT NULL,
PRIMARY KEY (Ssn),
FOREIGN KEY (Super_ssn) REFERENCES EMPLOYEE(Ssn),
FOREIGN KEY (Dno) REFERENCES DEPARTMENT(Dnumber) );
```

Key Constraints

- A **referential integrity constraint** can be **violated** when tuples are inserted or deleted, or when a foreign key or primary key attribute value is modified.
- The **default action** that SQL takes for an **integrity violation** is to **reject** the **update operation** that will cause a violation.
- However, the schema designer can specify an alternative action to be taken by attaching a **referential triggered action** clause to any foreign key constraint.
- The options include **SET NULL**, **CASCADE**, and **SET DEFAULT.**
- An option must be qualified with either **ON DELETE** or **ON UPDATE**.
- **Example:**
- The database designer chooses **ON DELETE SET NULL** and **ON UPDATE CASCADE** for the **foreign key Super_ssn** of **EMPLOYEE.**

Key Constraints

- What is a foreign key with **Cascade DELETE** in Oracle?
- A foreign key with **cascade delete** means that ***if a record in the parent table is deleted, then the corresponding records in the child table will automatically be deleted.***
- A **foreign key** with a **cascade/set null** can be defined in either a **CREATE TABLE** statement or an **ALTER TABLE** statement.

  CREATE TABLE EMPLOYEE

  (…..,

  PRIMARY KEY (Ssn),

  FOREIGN KEY (Super_ssn) **REFERENCES** EMPLOYEE(Ssn) **ON DELETE SET NULL**)

Key Constraints

- This means that if the tuple for a *supervising employee* is *deleted*, the value of Super_ssn is automatically set to NULL for all employee tuples that were referencing the deleted employee tuple.
- On the other hand, if the Ssn value for a supervising employee is *updated* (say, because it was entered incorrectly), the new value is ***cascaded*** to Super_ssn for all employee tuples referencing the updated employee tuple.

Example1:

CREATE TABLE EMPLOYEE
(. . . ,
Dno INT **NOT NULL DEFAULT** 1,
PRIMARY KEY (Ssn),
FOREIGN KEY (Super_ssn) **REFERENCES** EMPLOYEE(Ssn)
ON DELETE SET NULL **ON UPDATE** CASCADE,
FOREIGN KEY(Dno) **REFERENCES** DEPARTMENT(Dnumber)
ON DELETE SET DEFAULT **ON UPDATE** CASCADE);

Specifying Constraints on Tuples Using CHECK

- In addition to key and referential integrity constraints, which are specified by special keywords, other *table constraints* can be specified through additional **CHECK clauses** at the end of a **CREATE TABLE statement**.
- These can be called **tuple-based** constraints because they apply to each tuple *individually* and are checked whenever a tuple is inserted or modified.
- For **example,** suppose that the DEPARTMENT table had an additional attribute Dept_create_date, which stores the date when the department was created.
- Then we could add the following CHECK clause at the end of the CREATE TABLE statement for the DEPARTMENT table to make sure that a manager's start date is later than the department creation date.
- **Example: CHECK** (Dept_create_date <= Mgr_start_date);

Insert Command

- The **INSERT INTO** statement is **used to insert new records** in a table.
- It is possible to write the INSERT INTO statement in two ways:

Syntax 1

•Specify both the column names and the values to be inserted:

INSERT INTO *table_name* (*column1*, *column2*, *column3*, ...)

VALUES (*value1*, *value2*, *value3*, ...);

- If we are adding values for all the columns of the table, we do not need to specify the column names in the SQL query.

Syntax 2

- However, make sure the order of the values is in the same order as the columns in the table.

INSERT INTO *table_name* VALUES (*value1*, *value2*, *value3*, ...);

Insert Command

- **Example:**
- INSERT INTO Customers (CustomerName, ContactName, Address, City, PostalCode, Country) VALUES ('Cardinal', 'Tom B. Erichsen', 'Skagen 21', 'Stavanger', '4006', 'Norway');

OR

- INSERT INTO Customers VALUES ('Cardinal', 'Tom B. Erichsen', 'Skagen 21', 'Stavanger', '4006', 'Norway');

DESCRIBE Command

- We can view the structure of an already created table using the describe statement.
- ***Syntax:***
- **DESCRIBE tablename;**
- Oracle also supports the short form **DESC** of **DESCRIBE** to get description of table.
- To retrieve details about the structure of relation STUDENT, we can write DESC or DESCRIBE followed by table name:
- oracle> **DESC Employee;**

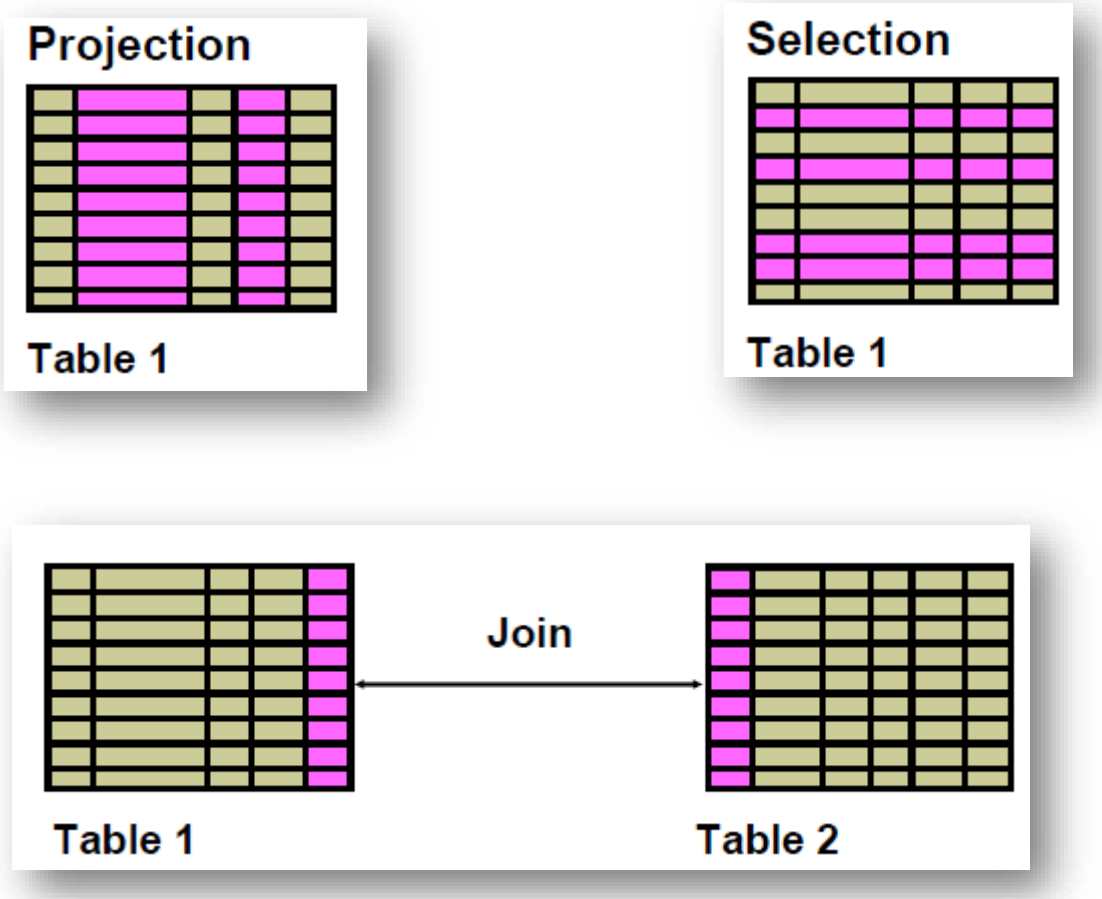
Object Type **TABLE** Object **EMPLOYEE**

Table	Column	Data Type	Length	Precision	Scale	Primary Key	Nullable	Default
EMPLOYEE	FNAME	Varchar2	15	-	-	-	-	-
	LNAME	Varchar2	15	-	-	-	✓	-
	SSN	Char	9	-	-	1	-	-
	BDATE	Date	7	-	-	-	✓	-
	ADDRESS	Varchar2	30	-	-	-	✓	-

Retrieving Data Using the SQL SELECT Statement

- A **SELECT statement** retrieves information from the database.
- With a SELECT statement, you can use the following capabilities:
- **Projection:**
  - Select the **columns** in a table that are returned by a query.
  - Select as few or as many of the columns as required.
- **Selection:**
  - Select the **rows** in a table that are returned by a query.
  - Various criteria can be used to restrict the rows that are retrieved.
- **Joining:**
  - Bring together data that is stored in different tables by specifying the link between them.

Capabilities of SQL SELECT Statements



Basic SELECT Statement

- A **SELECT** clause, specifies the columns to be displayed.
- A **FROM** clause, identifies the table containing the columns that are listed in the SELECT clause
- **Syntax:**

```
SELECT *|{[DISTINCT] column|expression [alias],...}
FROM    table;
```

  - * selects all columns
  - **DISTINCT** suppresses duplicates
  - ***column|expression*** selects the named column or the expression
  - ***alias*** gives the selected columns different headings
  - **FROM** *table* specifies the table containing the columns

Basic SELECT Statement

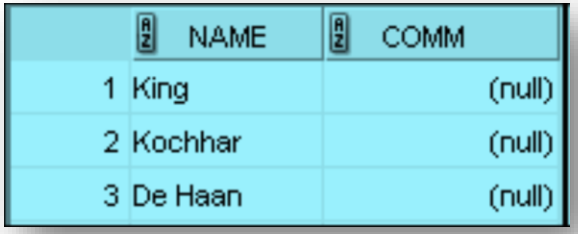
- **EXAMPLES**
- **SELECT** * FROM departments
- **SELECT** department_id, location_id FROM departments;
- **SELECT** last_name, salary, salary + 300 FROM employees;

	LAST_NAME	SALARY	SALARY+300
1	King	24000	24300
2	Kochhar	17000	17300
3	De Haan	17000	17300
4	Hunold	9000	9300
5	Ernst	6000	6300
6	Lorentz	4200	4500
7	Mourgos	5800	6100
8	Rajs	3500	3800
9	Davies	3100	3400
10	Matos	2600	2900

- If any column value in an **arithmetic expression** is **null,** the result is **null.**

Using Column Aliases

- **SELECT** last_name **AS** name, commission_pct **AS** comm **FROM** employees;



	NAME	COMM
1	King	(null)
2	Kochhar	(null)
3	De Haan	(null)

- **SELECT** last_name "Name" , salary*12 "Annual Salary“ **FROM** employees;

	Name	Annual Salary
1	King	288000
2	Kochhar	204000
3	De Haan	204000

Concatenation Operator

- We can link columns to other columns, arithmetic expressions, or constant values to create a character expression by using the **concatenation operator (||).**
- **Columns** on either side of the operator are combined to make a single output column.
- **Example: SELECT** last_name||job_id **AS** "Employees“ **FROM** employees;

	Employees
1	AbelSA_REP
2	DaviesST_CLERK
3	De HaanAD_VP
4	ErnstIT_PROG
5	FayMK_REP

- If we concatenate a **null value** with a **character string**, the result is a **character string.**
- **Example:** LAST_NAME**||** NULL results in LAST_NAME.

Using Literal Character Strings

- SELECT last_name ||' is a '||job_id **AS** "Employee Details“ **FROM** employees;

	Employee Details
1	Abel is a SA_REP
2	Davies is a ST_CLERK
3	De Haan is a AD_VP
4	Ernst is a IT_PROG
5	Fay is a MK_REP

- SELECT last_name ||': 1 Month salary = '||salary Monthly **FROM** employees;

	MONTHLY
1	King: 1 Month salary = 24000
2	Kochhar: 1 Month salary = 17000
3	De Haan: 1 Month salary = 17000
4	Hunold: 1 Month salary = 9000
5	Ernst: 1 Month salary = 6000
6	Lorentz: 1 Month salary = 4200
7	Mourgos: 1 Month salary = 5800
8	Rajs: 1 Month salary = 3500

Duplicate Rows

- The default display of queries is all rows, including duplicate rows.

```
SELECT department_id
FROM   employees;
```

1

	DEPARTMENT_ID
1	90
2	90
3	90
4	60
5	60

```
SELECT DISTINCT department_id
FROM   employees;
```

2

	DEPARTMENT_ID
1	(null)
2	90
3	20
4	110

WHERE Clause: Limiting the Rows that Are Selected

```
SELECT *|{[DISTINCT] column|expression [alias],...}
FROM   table
[WHERE condition(s)];
```

- ***condition*** is composed of column names, expressions, constants, and a comparison operator.
- A condition specifies a combination of one or more expressions and logical (Boolean) operators, and returns a value of TRUE, FALSE, or UNKNOWN.

Using the WHERE Clause

```
SELECT employee_id, last_name, job_id, department_id
FROM   employees
WHERE  department_id = 90 ;
```

```
SELECT last_name, job_id, department_id
FROM   employees
WHERE  last_name = 'Whalen' ;
```

```
SELECT last_name
FROM   employees
WHERE  hire_date = '17-FEB-96' ;
```

Comparison Operators

Operator	Meaning
=	Equal to
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to
<>	Not equal to
BETWEEN ...AND...	Between two values (inclusive)
IN(set)	Match any of a list of values
LIKE	Match a character pattern
IS NULL	Is a null value

Example

... WHERE hire_date = '01-JAN-95'

... WHERE salary >= 6000

... WHERE last_name = 'Smith'

Range Conditions Using the BETWEEN Operator

```
SELECT last_name, salary
FROM   employees
WHERE  salary BETWEEN 2500 AND 3500;
```

Lower limit Upper limit

	LAST_NAME	SALARY
1	Rajs	3500
2	Davies	3100
3	Matos	2600
4	Vargas	2500

- Values that are specified with the **BETWEEN** operator are **inclusive**.

Range Conditions Using the BETWEEN Operator

- SELECT last_name
- FROM employees
- WHERE last_name BETWEEN 'King' AND 'Smith';

	LAST_NAME
1	King
2	Kochhar
3	Lorentz
4	Matos
5	Mourgos
6	Rajs

Membership Condition Using the IN Operator

```
SELECT employee_id, last_name, salary, manager_id
FROM   employees
WHERE  manager_id IN (100, 101, 201) ;
```

	EMPLOYEE_ID	LAST_NAME	SALARY	MANAGER_ID
1	101	Kochhar	17000	100
2	102	De Haan	17000	100
3	124	Mourgos	5800	100
4	149	Zlotkey	10500	100
5	201	Hartstein	13000	100
6	200	Whalen	4400	101
7	205	Higgins	12000	101
8	202	Fay	6000	201

Membership Condition Using the IN Operator

- The **IN operator** can be used with any **data type**.
- The following example returns a **row** from the **EMPLOYEES table**, for any employee whose **last name** is included in the list of names in the **WHERE clause**:

```
SELECT employee_id, manager_id, department_id
FROM employees
WHERE last_name IN ('Hartstein', 'Vargas');
```

- If characters or dates are used in the list, they must be enclosed with single quotation marks ('').
- **Note:** The IN operator is internally evaluated by the Oracle server as a set of **OR conditions**, such as a=value1 **or** a=value2 **or** a=value3.
- Therefore, using the IN operator has **no performance benefits** and is used only for logical simplicity.

Pattern Matching Using the LIKE Operator

- Use the **LIKE operator** to perform **wildcard searches** of valid search string values.
- Search conditions can contain either literal characters or numbers:

  % denotes **zero or many characters**.

  _ denotes **one character**.

```
SELECT   first_name
FROM     employees
WHERE    first_name LIKE 'S%' ;
```

- The **SELECT** statement mentioned above returns the first name from the EMPLOYEES table for any employee whose first name begins with the letter “S.”
- Note the uppercase “S.” Consequently, names beginning with a lowercase “s” are not returned.

Combining Wildcard Characters

```
SELECT last_name
FROM   employees
WHERE  last_name LIKE '_o%' ;
```

	LAST_NAME
1	Kochhar
2	Lorentz
3	Mourgos

- **SELECT** employee_id, last_name, job_id
- **FROM** employees
- **WHERE** job_id **LIKE** '%SA_%' ESCAPE '\';

	EMPLOYEE_ID	LAST_NAME	JOB_ID
1	149	Zlotkey	SA_MAN
2	174	Abel	SA_REP
3	176	Taylor	SA_REP
4	178	Grant	SA_REP

Using the NULL Conditions

- Test for nulls with the IS **NULL operator**.
- The **NULL conditions** include the **IS NULL** condition and the **IS NOT NULL** condition.
- The **IS NULL** condition tests for nulls.
- A **null value** means that the value is unavailable, unassigned, unknown, or inapplicable.
- Therefore, you cannot test with **=,** because a **null cannot be equal or unequal to any value.**

```
SELECT last_name, manager_id
FROM   employees
WHERE  manager_id IS NULL;
```

	LAST_NAME	MANAGER_ID
1	King	(null)

Using the AND Operator

```
SELECT employee_id, last_name, job_id, salary
FROM   employees
WHERE  salary >= 10000
AND    job_id LIKE '%MAN%' ;
```

	EMPLOYEE_ID	LAST_NAME	JOB_ID	SALARY
1	149	Zlotkey	SA_MAN	10500
2	201	Hartstein	MK_MAN	13000

AND Truth Table

The following table shows the results of combining two expressions with AND:

AND	TRUE	FALSE	NULL
TRUE	TRUE	FALSE	NULL
FALSE	FALSE	FALSE	FALSE
NULL	NULL	FALSE	NULL

Using the OR Operator

```
SELECT employee_id, last_name, job_id, salary
FROM   employees
WHERE  salary >= 10000
OR     job_id LIKE '%MAN%' ;
```

	EMPLOYEE_ID	LAST_NAME	JOB_ID	SALARY
1	100	King	AD_PRES	24000
2	101	Kochhar	AD_VP	17000
3	102	De Haan	AD_VP	17000
4	124	Mourgos	ST_MAN	5800
5	149	Zlotkey	SA_MAN	10500
6	174	Abel	SA_REP	11000
7	201	Hartstein	MK_MAN	13000
8	205	Higgins	AC_MGR	12000

OR Truth Table

The following table shows the results of combining two expressions with OR:

OR	TRUE	FALSE	NULL
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	NULL
NULL	TRUE	NULL	NULL

Using the NOT Operator

```
SELECT last_name, job_id
FROM   employees
WHERE  job_id
       NOT IN ('IT_PROG', 'ST_CLERK', 'SA_REP') ;
```

	LAST_NAME	JOB_ID
1	De Haan	AD_VP
2	Fay	MK_REP
3	Gietz	AC_ACCOUNT
4	Hartstein	MK_MAN
5	Higgins	AC_MGR
6	King	AD_PRES
7	Kochhar	AD_VP
8	Mourgos	ST_MAN
9	Whalen	AD_ASST
10	Zlotkey	SA_MAN

NOT Truth Table

The following table shows the result of applying the NOT operator to a condition:

NOT	TRUE	FALSE	NULL
	FALSE	TRUE	NULL

Using the NOT Operator

- The **NOT operator** can also be used with other SQL operators, such as **BETWEEN, LIKE,** and **NULL.**
  - ... WHERE job_id **NOT IN** ('AC_ACCOUNT', 'AD_VP')
  - ... WHERE salary **NOT BETWEEN** 10000 AND 15000
  - ... WHERE last_name **NOT LIKE** '%A%'
  - ... WHERE commission_pct IS **NOT NULL**

Using the ORDER BY Clause

- Sorted rows can be retrieved with the **ORDER BY clause**:
  - **ASC:** Ascending order, default
  - **DESC:** Descending order
- The ORDER BY clause comes last in the SELECT statement:

```
SELECT   last_name, job_id, department_id, hire_date
FROM     employees
ORDER BY hire_date ;
```

	LAST_NAME	JOB_ID	DEPARTMENT_ID	HIRE_DATE
1	King	AD_PRES	90	17-JUN-87
2	Whalen	AD_ASST	10	17-SEP-87
3	Kochhar	AD_VP	90	21-SEP-89
4	Hunold	IT_PROG	60	03-JAN-90
5	Ernst	IT_PROG	60	21-MAY-91
6	De Haan	AD_VP	90	13-JAN-93

- **Null values** are displayed **last** for **ascending sequences** and **first** for **descending sequences.**

Using the ORDER BY Clause

- Sorting by using the column's numeric position:

```
SELECT   last_name, job_id, department_id, hire_date
FROM     employees
ORDER BY 3;
```

3

- Sorting by multiple columns:

```
SELECT last_name, department_id, salary
FROM   employees
ORDER BY department_id, salary DESC;
```

4

Working with Dates

```
SELECT last_name, hire_date
FROM   employees
WHERE  hire_date < '01-FEB-88';
```

	LAST_NAME	HIRE_DATE
1	King	17-JUN-87
2	Whalen	17-SEP-87

```
SELECT sysdate
FROM dual;
```

SYSDATE
26-JUL-23

Arithmetic with Dates

Operation	Result	Description
date + number	Date	Adds a number of days to a date
date – number	Date	Subtracts a number of days from a date
date – date	Number of days	Subtracts one date from another
date + number/24	Date	Adds a number of hours to a date

```
SELECT last_name, (SYSDATE-hire_date)/7 AS WEEKS
FROM   employees
WHERE  department_id = 90;
```

	LAST_NAME	WEEKS
1	King	1041.168239087301587301587301587301587302
2	Kochhar	923.025381944444444444444444444444444444
3	De Haan	750.168239087301587301587301587301587302

Date-Manipulation Functions

Function	Result
MONTHS_BETWEEN	Number of months between two dates
ADD_MONTHS	Add calendar months to date
NEXT_DAY	Next day of the date specified
LAST_DAY	Last day of the month
ROUND	Round date
TRUNC	Truncate date

Function	Result
MONTHS_BETWEEN ('01-SEP-95','11-JAN-94')	19.6774194
ADD_MONTHS ('31-JAN-96',1)	'29-FEB-96'
NEXT_DAY ('01-SEP-95','FRIDAY')	'08-SEP-95'
LAST_DAY ('01-FEB-95')	'28-FEB-95'

CASE Expression

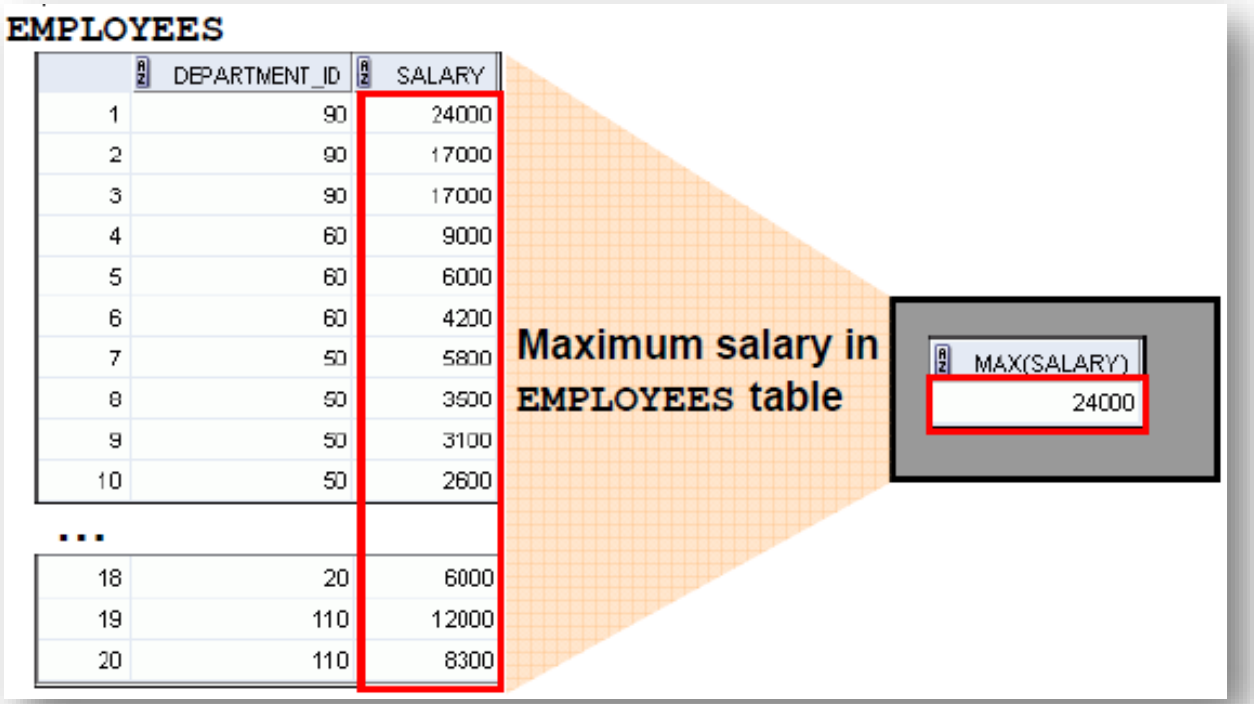
- **CASE expressions** allow you to use the **IF-THEN-ELSE logic** in **SQL statements** without having to invoke procedures.

```
SELECT last_name, job_id, salary,
       CASE job_id WHEN 'IT_PROG'  THEN  1.10*salary
                   WHEN 'ST_CLERK' THEN  1.15*salary
                   WHEN 'SA_REP'   THEN  1.20*salary
       ELSE       salary END     "REVISED_SALARY"
FROM   employees;
```

	LAST_NAME	JOB_ID	SALARY	REVISED_SALARY
...				
5	Ernst	IT_PROG	6000	6600
6	Lorentz	IT_PROG	4200	4620
7	Mourgos	ST_MAN	5800	5800
8	Rajs	ST_CLERK	3500	4025
9	Davies	ST_CLERK	3100	3565
...				
13	Abel	SA_REP	11000	13200
14	Taylor	SA_REP	8600	10320

What Are Group Functions?

- **Group functions** operate on sets of rows to give one result per group.



Types of Group Functions

Function	Description
AVG([DISTINCT|ALL]*n*)	Average value of *n*, ignoring null values
COUNT({*|[DISTINCT|ALL]*expr*})	Number of rows, where *expr* evaluates to something other than null (count all selected rows using *, including duplicates and rows with nulls)
MAX([DISTINCT|ALL]*expr*)	Maximum value of *expr*, ignoring null values
MIN([DISTINCT|ALL]*expr*)	Minimum value of *expr*, ignoring null values
STDDEV([DISTINCT|ALL]*x*)	Standard deviation of *n*, ignoring null values
SUM([DISTINCT|ALL]*n*)	Sum values of *n*, ignoring null values
VARIANCE([DISTINCT|ALL]*x*)	Variance of *n*, ignoring null values

Group Functions: Syntax

```
SELECT      group_function(column), ...
FROM        table
[WHERE      condition]
[ORDER BY   column];
```

```
SELECT AVG(salary), MAX(salary),
       MIN(salary), SUM(salary)
FROM   employees
WHERE  job_id LIKE '%REP%';
```

	AVG(SALARY)	MAX(SALARY)	MIN(SALARY)	SUM(SALARY)
1	8150	11000	6000	32600

Using the COUNT Function

- **COUNT(*)** returns the number of rows in a table:

```
SELECT COUNT(*)
FROM   employees
WHERE  department_id = 50;
```

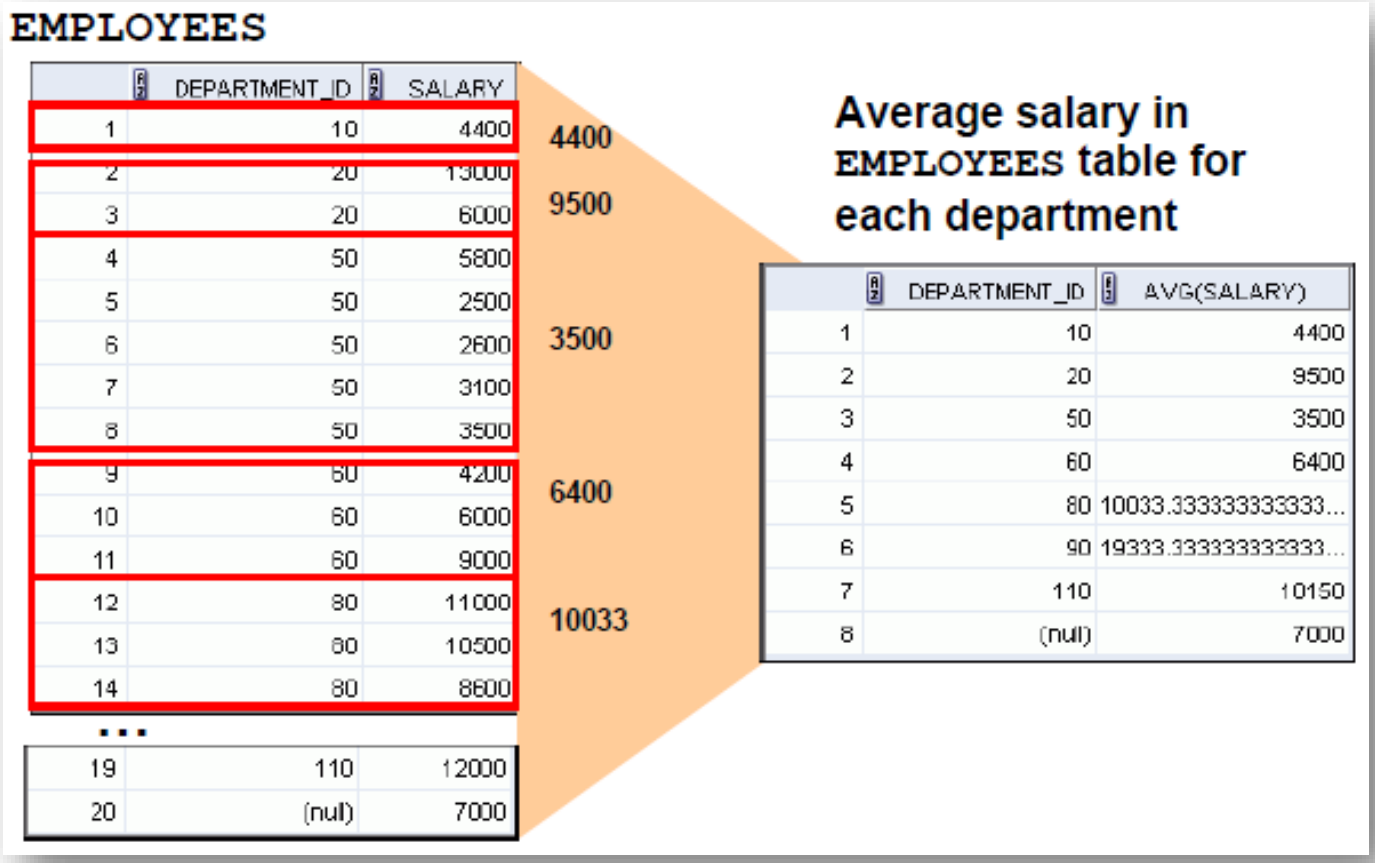
- **COUNT(*expr*)** returns the number of rows with non-null values for *expr*:

```
SELECT COUNT(commission_pct)
FROM   employees
WHERE  department_id = 80;
```

- **COUNT(DISTINCT expr)** returns the number of unique, non-null values that are in the column identified by expr.

```
SELECT COUNT(DISTINCT department_id)
FROM   employees;
```

Creating Groups of Data



GROUP BY Clause Syntax

```
SELECT     column, group_function(column)
FROM       table
[WHERE     condition]
[GROUP BY group_by_expression]
[ORDER BY column];
```

```
SELECT   department_id, AVG(salary)
FROM     employees
GROUP BY department_id ;
```

	DEPARTMENT_ID	AVG(SALARY)
1	(null)	7000
2	90	19333.33333333333...
3	20	9500
4	110	10150
5	50	3500
6	80	10033.33333333333...
7	60	6400
8	10	4400

GROUP BY Clause with Order BY

- You can also use the **group function** in the **ORDER BY clause**:

```
SELECT department_id, AVG(salary)
FROM employees
GROUP BY department_id
ORDER BY AVG(salary);
```

	DEPARTMENT_ID	AVG(SALARY)
1	50	3500
2	10	4400
3	60	6400
...		
7	110	10150
8	90	19333.33333333333333333333333333333333333

Grouping by More than One Column

EMPLOYEES

	DEPARTMENT_ID	JOB_ID	SALARY
1	10	AD_ASST	4400
2	20	MK_MAN	13000
3	20	MK_REP	6000
4	50	ST_MAN	5800
5	50	ST_CLERK	2500
6	50	ST_CLERK	2600
7	50	ST_CLERK	3100
8	50	ST_CLERK	3500
9	60	IT_PROG	4200
10	60	IT_PROG	6000
11	60	IT_PROG	9000
12	80	SA_REP	11000
13	80	SA_MAN	10500
14	80	SA_REP	8600
...			
19	110	AC_MGR	12000
20	(null)	SA_REP	7000

Add the salaries in the EMPLOYEES table for each job, grouped by department.

	DEPARTMENT_ID	JOB_ID	SUM(SALARY)
1	10	AD_ASST	4400
2	20	MK_MAN	13000
3	20	MK_REP	6000
4	50	ST_CLERK	11700
5	50	ST_MAN	5800
6	60	IT_PROG	19200
7	80	SA_MAN	10500
8	80	SA_REP	19600
9	90	AD_PRES	24000
10	90	AD_VP	34000
11	110	AC_ACCOUNT	8300
12	110	AC_MGR	12000
13	(null)	SA_REP	7000

Grouping by More than One Column

```
SELECT   department_id dept_id, job_id, SUM(salary)
FROM     employees
GROUP BY department_id, job_id
ORDER BY department_id;
```

	DEPARTMENT_ID	JOB_ID	SUM(SALARY)
1	10	AD_ASST	4400
2	20	MK_MAN	13000
3	20	MK_REP	6000
4	50	ST_CLERK	11700
5	50	ST_MAN	5800
6	60	IT_PROG	19200
7	80	SA_MAN	10500
8	80	SA_REP	19600
9	90	AD_PRES	24000
10	90	AD_VP	34000
11	110	AC_ACCOUNT	8300
12	110	AC_MGR	12000
13	(null)	SA_REP	7000

Illegal Queries Using Group Functions

```
SELECT department_id, COUNT(last_name)
FROM   employees;
```

A GROUP BY clause must be added to count the last names for each department_id.

```
SELECT department_id, count(last_name)
FROM employees
GROUP BY department_id;
```

Illegal Queries Using Group Functions

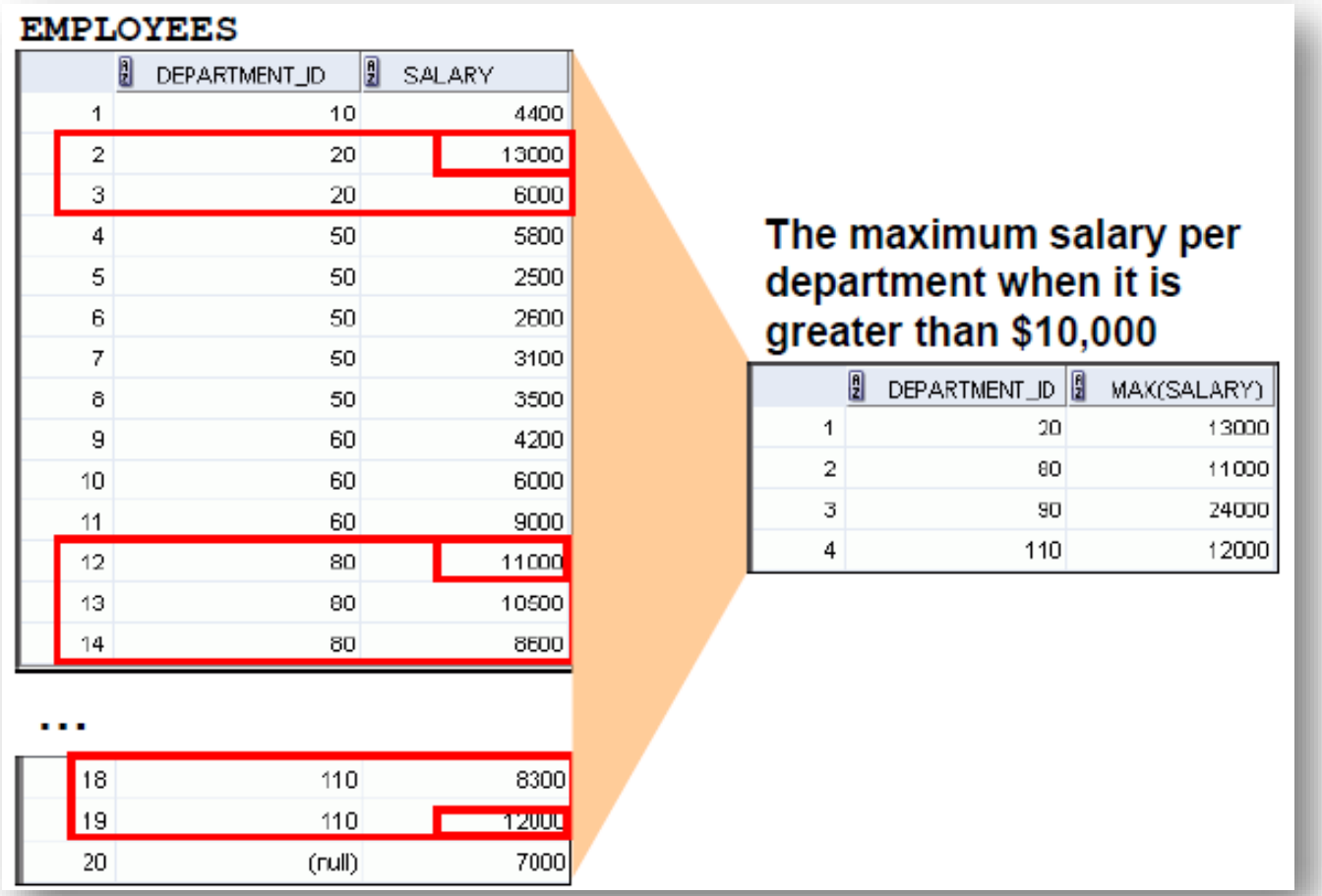
- We cannot use the **WHERE** clause to restrict groups i.e. we cannot use group functions in the **WHERE** clause.
- We use the **HAVING** clause to restrict groups.

```
SELECT    department_id, AVG(salary)
FROM      employees
WHERE     AVG(salary) > 8000
GROUP BY department_id;
```

```
SELECT department_id, AVG(salary)
FROM employees
GROUP BY department_id
HAVING AVG(salary) > 8000;
```

Restricting Group Results

- We use the HAVING clause to restrict groups in the same way that we use the WHERE clause to restrict the rows that we select.



Restricting Group Results with the HAVING Clause

- When you use the **HAVING clause**, the Oracle server restricts groups as follows:
  - 1. Rows are grouped.
  - 2. The group function is applied.
  - 3. Groups matching the **HAVING clause** are displayed.

```
SELECT     column, group_function
FROM       table
[WHERE     condition]
[GROUP BY  group_by_expression]
[HAVING    group_condition]
[ORDER BY  column];
```

Using the HAVING Clause

```
SELECT    department_id, MAX(salary)
FROM      employees
GROUP BY  department_id
HAVING    MAX(salary)>10000 ;
```

	DEPARTMENT_ID	MAX(SALARY)
1	90	24000
2	20	13000
3	110	12000
4	80	11000

Using the HAVING Clause

```
SELECT department_id, AVG(salary)
FROM employees
GROUP BY department_id
HAVING max(salary)>10000;
```

	DEPARTMENT_ID	AVG(SALARY)
1	90	19333.333333333333333...
2	20	9500
3	110	10150
4	80	10033.333333333333333...

Using the HAVING Clause

```
SELECT job_id, SUM(salary) PAYROLL
FROM employees
WHERE job_id NOT LIKE '%REP%'
GROUP BY job_id
HAVING SUM(salary) > 13000
ORDER BY SUM(salary);
```

	JOB_ID	PAYROLL
1	IT_PROG	19200
2	AD_PRES	24000
3	AD_VP	34000

Nesting Group Functions

- **Group functions** can be **nested** to a depth of **two** functions.
- The **example** the **average salary** for each **department_id** and then displays the maximum average salary.
- Note that **GROUP BY clause** is **mandatory** when nesting group functions.

```
SELECT   MAX(AVG(salary))
FROM     employees
GROUP BY department_id;
```

	MAX(AVG(SALARY))
1	19333.333333333333333333333333333333333

ALTER Table Command

- After creating a table we may realize that we need to add/remove/rename an attribute or to modify the datatype of an existing attribute or to add constraint in attribute.
- In all such cases, we need to change or alter the structure of the table by using the **alter statement.**
- **Add Column in table**
- **Syntax**

```
ALTER TABLE table_name  ADD column_name column_definition;
```

- **Example**

ALTER TABLE customers ADD customer_name varchar2(45);

ALTER Table Command

- **Add multiple columns in table**
- **Syntax**

```
ALTER TABLE table_name
 ADD (column_1 column_definition,
     column_2 column_definition,
     ...
     column_n column_definition);
```

- **Example**

```
ALTER TABLE customers
 ADD (customer_name varchar2(45),
     city varchar2(40) DEFAULT 'Seattle');
```

ALTER Table Command

- **Modify column in table**
- **Syntax**

```
ALTER TABLE table_name  MODIFY column_name column_type;
```

- **Example**

ALTER TABLE customers MODIFY customer_name varchar2(100) NOT NULL;

- **Drop column in table**
- **Syntax**

```
ALTER TABLE table_name  DROP COLUMN column_name;
```

- **Example**

ALTER TABLE customers DROP COLUMN customer_name;

ALTER Table Command

- **Rename column in table**
- **Syntax**

```
ALTER TABLE table_name  RENAME COLUMN old_name TO new_name;
```

- **Example**
- ALTER TABLE customers RENAME COLUMN customer_name TO cname;
- **Rename table**
- **Syntax**

```
ALTER TABLE table_name  RENAME TO new_table_name;
```

- **Example**
- ALTER TABLE customers RENAME TO contacts;

UPDATE Command

- In Oracle, **UPDATE statement** is used to update the existing records in a table.
- **Syntax**

```
UPDATE table_name
SET column1 = expression1,
    column2 = expression2,
     ...
    column_n = expression_n
WHERE conditions;
```

- **Example:**
- **Update single column**

```
UPDATE suppliers  SET supplier_name = 'Kingfisher'

WHERE supplier_id = 2;
```

UPDATE Command

- **Update multiple columns**
- **Example**

```
UPDATE suppliers
SET supplier_address = 'Agra',  supplier_name = 'Bata shoes'
WHERE supplier_id = 1;
```

DELETE Command

- In Oracle, **DELETE statement** is used to remove or delete a single record or multiple records from a table.
- **Syntax**

```
DELETE FROM table_name WHERE conditions;
```

- **Examples:**
- DELETE FROM customers WHERE name = 'Sohan';
- DELETE FROM customers WHERE last_name = ‘Bond' AND customer_id > 2;
- DELETE * FROM customers OR DELETE FROM customers
- **Important Note:** DELETE is a DML (Data Manipulation Language) command hence operation performed by DELETE can be **rolled back** or undone.
- We can use the “ROLLBACK” command to restore the tuple because it **does not auto-commit.**

DROP Command

- It is a Data Definition Language Command (DDL).
- It is used to drop the **whole table**.
- With the help of the “DROP” command we can drop (delete) the whole structure in one go i.e. it removes the named elements of the schema.
- **SYNTAX –**
- **DROP table <table_name>;**
- We can’t restore the table by using the “ROLLBACK” command because it **auto commits.**

TRUNCATE

- It is also a Data Definition Language Command (DDL).
- It is used to delete all the rows of a relation (table) in one go.
- With the help of the "**TRUNCATE**" command, we can't delete the single row as here WHERE clause is not used.
- By using this command the existence of all the rows of the table is lost.
- It is comparatively faster than the delete command as it deletes all the rows fastly.
- **SYNTAX**
- If we want to use truncate :
- **TRUNCATE table <table_name>;**
- **Note –** Here we can't restore the tuples of the table by using the "ROLLBACK" command.

JOIN Operation in SQL

1. CROSS JOIN
2. EQUI JOIN
3. OUTER JOIN
    a. LEFT OUTER JOIN
    b. RIGHT OUTER JOIN
    c. FULL OUTER JOIN
4. SELF JOIN

CROSS JOIN

Example:

SELECT *

FROM employee e, department d

Employee	
EMPNAME	**DEPTNO**
king	10
blake	NULL
clark	10
martin	20
turner	10
jones	NULL

Department	
DEPTNO	**DEPTNAME**
10	accounting
30	sales
20	operations

X

EMPNAME	DEPTNO	DEPTNO	DEPTNAME
king	10	10	accounting
blake	NULL	10	accounting
clark	10	10	accounting
martin	20	10	accounting
turner	10	10	accounting
jones	NULL	10	accounting
king	10	30	sales
blake	NULL	30	sales
clark	10	30	sales
martin	20	30	sales
turner	10	30	sales
jones	NULL	30	sales
king	10	20	operations
blake	NULL	20	operations
clark	10	20	operations
martin	20	20	operations
turner	10	20	operations
jones	NULL	20	operations

EQUIJOIN/INNER JOIN

- **Inner Join** is the simplest and most common type of join.
- It returns all rows from multiple tables where the join condition is met.
- **Syntax**

```
SELECT columns
FROM table1   t1
INNER JOIN table2  t2
ON t1.column = t2.column;
```

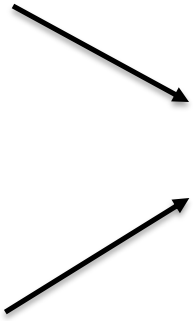
OR

```
SELECT columns
FROM table1 t1 , table2  t2
WHERE  t1.column = t2.column;
```

EQUIJOIN/INNER JOIN

EMPNAME	DEPTNO
king	10
blake	NULL
clark	10
martin	20
turner	10
jones	NULL

DEPTNO	DEPTNAME
10	accounting
30	sales
20	operations



EMPNAME	DEPTNO	DEPTNAME
king	10	accounting
clark	10	accounting
martin	20	operations
turner	10	accounting

```
SELECT e.empname, e.deptno, d.deptname
FROM employee e, department d
WHERE e.deptno = d.deptno
```

EQUIJOIN/INNER JOIN

```
SELECT e.empname, e.deptno, d.deptname
FROM employee e, department d
WHERE e.deptno = d.deptno
```

OR

```
SELECT e.empname, e.deptno ,d.deptname
FROM employee e
INNER JOIN department d ON e.deptno=d.deptno;
```

SELF JOIN

- A **SELF JOIN** is used for joining a table with itself.
- **Syntax**

```
SELECT columns
FROM table t1
INNER JOIN table t2
ON t1.column = t2.column;
```

OR

```
SELECT columns
FROM table t1 , table  t2
WHERE  t1.column = t2.column;
```

Example: SELF JOIN

- Consider the **Employee** schema:
- Each employee in the table has a **manager ID** associated with it.
- We use **SELF JOIN** for referencing the same table twice in a same query by using table alias.
- The **Join Condition** matches the employees with their managers using the manager_id and employee_id columns.

```
SELECT  e.first_name || '''s manager is '
||m.last_name || '.'
FROM employees e, employees m
WHERE e.manager_id = m.employee_id ;
```

	Column_Name
1	EMPLOYEE_ID
2	FIRST_NAME
3	LAST_NAME
4	EMAIL
5	PHONE_NUMBER
6	HIRE_DATE
7	JOB_ID
8	SALARY
9	COMMISSION_PCT
10	MANAGER_ID
11	DEPARTMENT_ID

Example: SELF JOIN

- The previous query results in:

	E.FIRST_NAME||'''S MANAGER IS '||M.LAST_NAME||'.'
1	William's manager is Cambrault.
2	Lisa's manager is Cambrault.
3	Sundita's manager is Cambrault.
4	Tayler's manager is Cambrault.
5	Harrison's manager is Cambrault.
6	Elizabeth's manager is Cambrault.
7	Alexander's manager is De Haan.
8	Clara's manager is Errazuriz.
9	Mattea's manager is Errazuriz.
10	David's manager is Errazuriz.

Left Outer Join

- **Left Outer Join** returns **all rows from the left (first) table** specified in the ON condition and only those rows from the **right (second) table** where the join condition is met.

```
SELECT columns
FROM table1  t1
LEFT JOIN table2  t2
ON t1.column = t2.column;
```

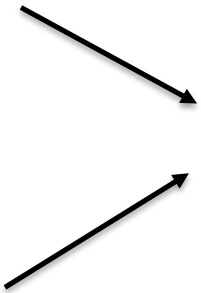
OR

```
SELECT columns
FROM table1 t1 , table2  t2
WHERE  t1.column = t2.column(+);
```

Left Outer Join

EMPNAME	DEPTNO
king	10
blake	NULL
clark	10
martin	20
turner	10
jones	NULL

DEPTNO	DEPTNAME
10	accounting
30	sales
20	operations



EMPNAME	DEPTNO	DEPTNO	DEPTNAME
turner	10	10	accounting
clark	10	10	accounting
king	10	10	accounting
martin	20	20	operations
jones	NULL	NULL	NULL
blake	NULL	NULL	NULL

```
SELECT e.empname, e.deptno, d.deptname
FROM employee e
LEFT JOIN department d
ON e.deptno=d.deptno
```

OR

```
SELECT e.empname, e.deptno, d.deptname
FROM employee e, department d
WHERE e.deptno = d.deptno(+)
```

Right Outer Join

- **Right Outer Join** returns **all rows from the right (second) table** specified in the ON condition and only those rows from the **left (first) table** where the join condition is met.

```
SELECT columns
FROM table1  t1
RIGHT JOIN table2  t2
ON t1.column = t2.column;
```

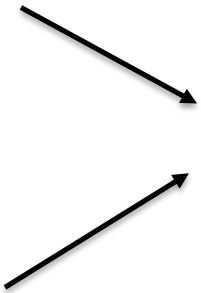
OR

```
SELECT columns
FROM table1 t1 , table2  t2
WHERE  t1.column(+) = t2.column;
```

Right Outer Join

EMPNAME	DEPTNO
king	10
blake	NULL
clark	10
martin	20
turner	10
jones	NULL

DEPTNO	DEPTNAME
10	accounting
30	sales
20	operations



EMPNAME	DEPTNO	DEPTNO	DEPTNAME
king	10	10	accounting
clark	10	10	accounting
martin	20	20	operations
turner	10	10	accounting
NULL	NULL	30	sales

```
SELECT *
FROM employee e
RIGHT JOIN department d
ON e.deptno=d.deptno
```

OR

```
SELECT *
FROM employee e, department d
WHERE e.deptno(+) = d.deptno
```

FULL Outer Join

- The **Full Outer Join** returns all rows from the left hand table and right hand table.

  It places **NULL** where the join condition is not met.

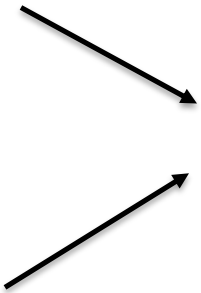
- **Syntax:**

```
SELECT columns
FROM table1  t1
FULL JOIN table2  t2
ON t1.column = t2.column;
```

FULL Outer Join

EMPNAME	DEPTNO
king	10
blake	NULL
clark	10
martin	20
turner	10
jones	NULL

DEPTNO	DEPTNAME
10	accounting
30	sales
20	operations



EMPNAME	DEPTNO	DEPTNO	DEPTNAME
turner	10	10	accounting
clark	10	10	accounting
king	10	10	accounting
martin	20	20	operations
jones	NULL	NULL	NULL
blake	NULL	NULL	NULL
NULL	NULL	30	sales

```
SELECT *
FROM employee e
FULL JOIN department d
ON e.deptno=d.deptno
```

Introduction to the Oracle View

CUSTOMER_ID	NAME	ADDRESS	WEBSITE	CREDIT_LIMIT
1	Raytheon	514 W Superior St, Kokomo, IN	http://www.raytheon.com	100
2	Plains GP Holdings	2515 Bloyd Ave, Indianapolis, IN	http://www.plainsallamerican.com	100
3	US Foods Holding	8768 N State Rd 37, Bloomington, IN	http://www.usfoods.com	100
4	AbbVie	6445 Bay Harbor Ln, Indianapolis, IN	http://www.abbvie.com	100
5	Centene	4019 W 3Rd St, Bloomington, IN	http://www.centene.com	100
6	Community Health Systems	1608 Portage Ave, South Bend, IN	http://www.chs.net	100
7	Alcoa	23943 Us Highway 33, Elkhart, IN	http://www.alcoa.com	100
8	International Paper	136 E Market St # 800, Indianapolis, IN	http://www.internationalpaper.com	100
9	Emerson Electric	1905 College St, South Bend, IN	http://www.emerson.com	100
10	Union Pacific	3512 Rockville Rd # 137C, Indianapolis, IN	http://www.up.com	200

- Let us consider a query:

  SELECT name, credit_limit FROM customers;

- The result of a query is a **derived table** as shown below:

NAME	CREDIT_LIMIT
Kimberly-Clark	400
Hartford Financial Services Group	400
Kraft Heinz	500
Fluor	500
AECOM	500
Jabil Circuit	500
CenturyLink	500
General Mills	600
Southern	600
Thermo Fisher Scientific	700

Introduction to the Oracle View

- The **derived table** consists of only **partial data** from the customers table.
- If we give this query a name, then we have a **view.**
- That is why sometimes a **view is referred** to as a **named query.**
- A **view** is a "**virtual" table** whose data is the result of a **stored query**, which is derived each time when we query against the view.
- Every **view** has columns with data types so we can execute a query against views or manage their contents (with some restrictions) using the INSERT, UPDATE, DELETE, statements.
- Unlike a table, a **view does not store any data.**

Introduction to the Oracle View

- When we query data from a **view**, Oracle uses this **stored query** to retrieve the data from the underlying tables.
- Suppose, we assign the query (SELECT name, credit_limit FROM customers) a name called customer_credits and query data from this view:
  - **SELECT * FROM customer_credits;**
- Behind the scenes, Oracle finds the stored query associated with the name customer_credits and executes the following query:

  SELECT * FROM (SELECT name, credit_limit FROM customers);
- In this example, the customers table is called the **base table.**
- The result set returned from the customer_credits view depends on the data of the underlying table, which is the customers table in this case.
- If we rename or drop one of the columns referenced by the query such as name or credit_limit, the view customer_credits will not work anymore.

Features of a View

- A view is defined by a **sub-query**, i.e, SELECT statement.
- It's a **virtual table** that can be based on one or more base tables or views.
- It contains **no physical data** as it simply fetches data from tables on which it is based.
- When we use view in our query, the sub-query gets executed and view data is fetched **dynamically.**
- There are **two types** of Views:
  - **Simple View** – Based on one base table
  - **Complex View** – Based on one or more base tables and a sub-query that might contain a join condition or aggregate function

When to use the Oracle view

- We can use views in many cases for different purposes.
- The most common uses of views are as follows:
  - *Simplifying data retrieval.*
  - *Maintaining logical data independence.*
  - *Implementing data security.*
- **Simplifying data retrieval**
  - Views help simplify data retrieval significantly.
  - First, we build a complex query, test it carefully, and encapsulate the query in a view.
  - Then, we can access the data of the underlying tables through the view instead of rewriting the whole query again and again.

View: Simplifying data retrieval

- The following query returns sales amount by the customer by year:

```
SELECT   name AS customer,   SUM( quantity * unit_price ) sales_amount,
EXTRACT( YEAR   FROM   order_date ) YEAR FROM   orders INNER JOIN
order_items  USING(order_id)  INNER JOIN customers USING(customer_id)
WHERE  status = 'Shipped'
GROUP BY   name,  EXTRACT(YEAR  FROM  order_date   );
```

CUSTOMER	SALES_AMOUNT	YEAR
International Paper	613735.06	2015
AutoNation	968134.3	2017
Becton Dickinson	484279.39	2016
Plains GP Holdings	655593.37	2016
Supervalu	394765.27	2017
Centene	417049.24	2017
CenturyLink	711307.09	2016
Abbott Laboratories	697288.63	2016
AutoNation	236531.07	2016
Jabil Circuit	508588.59	2017

View: Simplifying data retrieval

- This query is quite **complex.**
- Typing it over and over again is time-consuming and potentially cause a mistake.
- To simplify it, you can create a view based on this query:

```
CREATE OR REPLACE VIEW customer_sales AS
SELECT   name AS customer,    SUM( quantity * unit_price ) sales_amount,
EXTRACT(  YEAR    FROM   order_date ) YEAR FROM    orders INNER JOIN
order_items  USING(order_id)  INNER JOIN customers USING(customer_id)
WHERE  status = 'Shipped'
GROUP BY    name,  EXTRACT(YEAR  FROM  order_date    );
```

View: Simplifying data retrieval

- Now, we can easily retrieve the sales by the customer in 2017 with a more simple query:

```
SELECT    customer,    sales_amount FROM    customer_sales

WHERE    YEAR = 2017

ORDER BY    sales_amount DESC;
```

CUSTOMER	SALES_AMOUNT
Raytheon	2406081.53
NextEra Energy	1449154.87
Southern	1304990.28
General Mills	1081679.88
AutoNation	968134.3
Emerson Electric	952330.09
Progressive	950118.04
Aflac	698612.98
Goodyear Tire & Rubber	676068.67
AutoZone	645379.54
Jabil Circuit	508588.59

View: Maintaining logical data independence

- We can expose the data from underlying tables to the external applications via views.
- Whenever the structures of the **base tables** change, we just need to **update** the view.
- The **interface** between the **database** and the **external applications** remains intact.
- The beauty is that we don’t have to change a single line of code to keep the external applications up and running.
- For example, some reporting systems may need only customer sales data for composing strategic reports.
- Hence, you can give the application owners the **customer_sales view**.

Implementing data security

- Views allow us to implement an additional **security layer.**
- They help us hide certain columns and rows from the underlying tables and expose only needed data to the appropriate users.
- Oracle provide us the with **GRANT and REVOKE** commands on views so that we can specify which actions a user can perform against the view.
- Note that in this case, we don’t grant any privileges on the underlying tables because we may not want the user to bypass the views and access the base tables directly.

Oracle CREATE VIEW

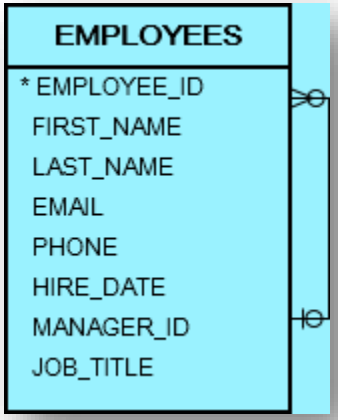
- To create a new view in a database, you use the following Oracle CREATE VIEW statement:
- **SYNTAX**

```
CREATE [OR REPLACE] VIEW view_name
[(column_aliases)] AS   defining-query
[WITH READ ONLY]
[WITH CHECK OPTION]
```

- The OR REPLACE option replaces the definition of existing view.
- It is handy if we have granted various privileges on the view.
- Because when we use the DROP VIEW and CREATE VIEW to change the view's definition, Oracle removes the view privileges, which may not be what we want.
- To avoid this, you can use the CREATE OR REPLACE clause that preserves the view privileges.

Example: Oracle CREATE VIEW

- The following statement creates a view named employee_yos based on the employees table.



- The view shows the employee id, name and years of service:

```
CREATE VIEW employee_yos AS
SELECT    employee_id,    first_name || ' ' || last_name full_name,
   FLOOR( months_between( CURRENT_DATE, hire_date )/ 12 ) yos
FROM    employees;
```

Example: Oracle CREATE VIEW

- If you don’t want to use column aliases in the query, you must define them in the CREATE VIEW clause:

```
CREATE VIEW employee_yos (employee_id, full_name, yos) AS
SELECT    employee_id,    first_name || ' ' || last_name,
    FLOOR( months_between( CURRENT_DATE, hire_date )/ 12 )
FROM    employees;
```

Example: Oracle CREATE VIEW

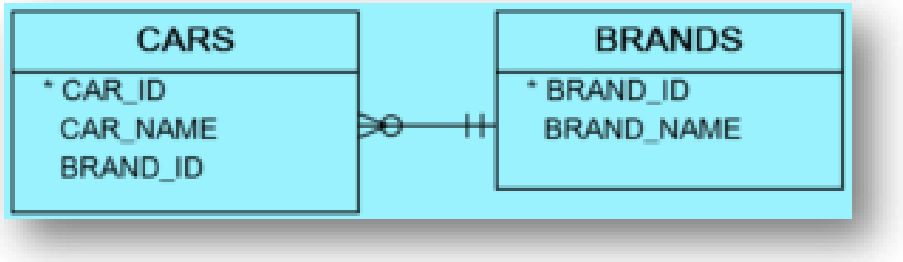
- The following query returns employees whose years of service are 15:

```
SELECT    *FROM    employee_yos
WHERE    yos = 15
ORDER BY    full_name;
```

EMPLOYEE_ID	FULL_NAME	YOS
105	Gracie Gardner	15
104	Harper Spencer	15
9	Mohammad Peterson	15
106	Rose Stephens	15
10	Ryan Gray	15
107	Summer Payne	15

Oracle Updatable View

- A view behaves like a table because you can query data from it.
- However, you cannot always manipulate data via views.
- A view is updatable if the statement against the view can be translated into the corresponding statement against the underlying table.
- Let's consider the following database tables:



- In database diagram, a car belongs to one brand while a brand has one or many cars. The relationship between brand and car is a one-to-many.

Oracle Updatable View

- The following statement creates a new view named cars_master:

  CREATE VIEW cars_master AS

  SELECT car_id, car_name FROM cars;

- It's possible to delete a row from the cars table via the cars_master view, for example:

  DELETE FROM cars_master

  WHERE car_id = 1;

- And you can update any column values exposed to the cars_master view:

  UPDATE cars_master SET car_name = 'Audi RS7 Sportback'

  WHERE car_id = 2;

Oracle Updatable View

- We cannot update a view if it contains any of the following:
  - Aggregate functions like SUM, MAX, MIN etc.
  - GROUP BY clause
  - DISTINCT clause
  - HAVING
  - Set Operators like UNION, INTERSECT etc.
  - Subquery in SELECT
  - Certain Joins