



# *Structured Query Language (SQL)*

## CONTENTS

- Fetching or retrieving data using SQL SELECT Statement
- Restricting the data being fetched
- Sorting the data after fetching it
- Single-row functions to transform the data
- Conversion functions and conditional expressions

## SQL Command Groups

- Data Query Language (DQL) statement
  - Used for querying database
  - SELECT - used to retrieve the data present in the database
- Data Definition Language (DDL) statements
  - Used to define the database structure or schema

DDL	Description
CREATE	Create objects in the database
ALTER	Alters the structure of the database
DROP	Delete objects from the database
TRUNCATE	Remove all records from a table permanently
COMMENT	Add comments to the data dictionary
RENAME	Rename an object

## SQL Command Groups (Contd ...)

- Data Manipulation Language (DML) statements
  - Used for managing data within objects

DML	Description
INSERT	insert data into a table
UPDATE	updates existing data within a table
DELETE	deletes unwanted/all records from a table
MERGE	UPSERT operation (insert or update)

- Data Control Language (DCL) statements
  - Used for granting and denying access privileges to users

DCL	Description
GRANT	gives user's access privileges to database
REVOKE	withdraw access privileges given with the GRANT command

## SQL Command Groups (Contd ...)

- Transaction Control (TCL):
  - Used to manage the changes made by DML statements
  - Allows statements to be grouped together into logical transactions

TCL	Description
COMMIT	Save work done
ROLLBACK	Restore database to original state since the last COMMIT
SAVEPOINT	Identify a point in a transaction to which you can later roll back

## Primitive Data Types

Data Type	Description
Numeric	<ul style="list-style-type: none"> <li>Stores numeric data</li> <li>Some types: INT, TINYINT, SMALLINT, MEDIUMINT, BIGINT, FLOAT(M, D), DOUBLE(M, D) where M &amp; D are display length and no. of decimals, etc.</li> </ul>
String	<ul style="list-style-type: none"> <li>Stores character strings</li> <li>Some types: CHAR(M), VARCHAR(M), BLOB or TEXT, etc.</li> <li>Value of M can be 1 to 255. For CHAR columns, it pads rest of the places by spaces; whereas for VARCHAR that actual length is M itself.</li> </ul>
Date	<ul style="list-style-type: none"> <li>Stores data and time data types</li> <li>DATE (YYYY-MM-DD format), DATETIME (DATE (YYYY-MM-DD format), TIMESTAMP (YYYYMMDDHHMMSS format), TIME (HH:MM:SS format), YEAR(M) – Year in 2-digit or 4-digit format</li> </ul>

## RETRIEVING DATA USING THE SQL SELECT STATEMENT

- SELECT statement to retrieve the data from the database
- Retrieves the data stored in relational tables
- Never modifies or alters the data in the database
- Provides a read-only method of extracting the data



## TYPES OF SELECT STATEMENTS

### Projection

Restricts the number of columns retrieved from the table

### Selection

Restricts the number of rows retrieved from the table

### Join

Obtains data from multiple tables by specifying the relationship between them

## BASIC SELECT STATEMENT

- In its simplest form, a SELECT statement should have:

- A SELECT clause, which specifies the columns to be projected
  - A FROM clause, which specifies the table/tables from which the data is to be retrieved

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

- SELECTing all columns

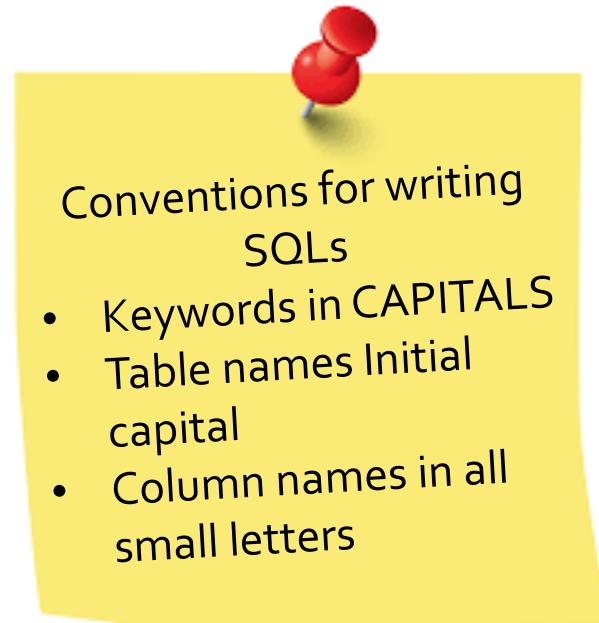
```
SELECT * FROM Employees;
```

- SELECTing specific set of columns (PROJECTION)

```
SELECT employee_id, first_name, last_name  
FROM Employees;
```

## GUIDELINES FOR SQL STATEMENTS

- SQL statements are not case sensitive
- Can be entered on one or more lines
- Put the clauses (like SELECT, FROM, WHERE etc.) on separate lines (for easy editing and code maintenance)
- Use indents are enhance readability
- Terminate every SQL statement with a semicolon



## ARITHMETIC EXPRESSION IN SELECT

- Calculations like – addition(+), subtraction(-), multiplication( \*) and division ( / ) can be performed on the data retrieved from the tables

- For example:

```
SELECT last_name, salary, salary + 300  
      FROM Employees;
```

- The column **salary + 300** is a computed column
- Annual compensation of salary is obtained by multiplying the salaries of each employee by 12 and providing a one time bonus of \$500

```
SELECT first_name, salary*12 + 500  
      FROM Employees;
```

- Get the employee first name, last name and the annual compensation of employees, by providing a monthly bonus of \$100 (Check operator precedence)

```
SELECT first_name, last_name, (salary+100)*12  
      FROM Employees;
```

## COLUMN ALIASES

- SQL uses the name of the selected column as the column heading
- The entire computation “(salary + salary \* commission\_pct)” would be displayed as column header, which is not descriptive
- A meaningful name to the column header can be given using Column Alias

```
SELECT first_name, last_name,  
(salary + salary * commission_pct) AS monthly_gross  
FROM Employees;
```

```
SELECT first_name, last_name,  
(salary + salary * commission_pct)*12 "Annual gross"  
FROM Employees;
```

- Keyword “AS” is optional
- Use double-quotes if a space or any special character is required

## COLUMNS WITH NULL IN SELECT

- If a row or a tuple does not have a value its said to be NULL
- NULL indicates that the value is unavailable, unassigned, unknown or not applicable
- NULL is not zero or a blank space, it's indicates absence of a value
- The COMMISSION\_PCT column in EMPLOYEES table has values only for sales people. For other employees it is NULL as they are not eligible for it
- This query would result in NULL values for employees who do not earn commission

```
SELECT last_name, salary + (salary*commission_pct)  
FROM Employees;
```

- If the value of any column in an expression is NULL, the whole expression becomes NULL

## CONCATENATION OPERATOR

- Useful to link columns or character strings to other columns
- Columns are combined to make a single column using CONCAT:

```
SELECT CONCAT(first_name, last_name) AS fullname  
FROM Employees;
```

- With a space between names:

```
SELECT CONCAT(first_name, ' ', last_name) AS  
fullname FROM Employees;
```

- It can be used to provide more meaning to the output:

```
SELECT CONCAT(first_name, ' works as ', job_id, ' in  
dept.', department_id) AS employee_desc  
FROM Employees;
```

## ELIMINATING DUPLICATE ROWS

- SELECT by default, displays all the rows in the table inclusive of duplicate rows
- To eliminate duplicate rows in the result, include the DISTINCT keyword in SELECT
- Get unique list of departments where employees are working:

```
SELECT DISTINCT department_id  
FROM Employees;
```
- Get unique list of departments and jobs:

```
SELECT DISTINCT department_id, job_id  
FROM Employees;
```
- Note that the above query might show duplicate department ids as there can be employees in it with more than one job id

## DESCRIBE COMMAND

- Describe command on a table displays:
  - Names of the columns in the table
  - Whether a column is NOT NULL
  - Datatype of the column
  - Note that is is not an SQL statement though
- DESCRIBE Employees;

Field	Type	Null	Key	Default	Extra
employee_id	int(11) unsigned	NO	PRI	NULL	
first_name	varchar(20)	YES		NULL	
last_name	varchar(25)	NO		NULL	
email	varchar(25)	NO		NULL	
phone_number	varchar(20)	YES		NULL	
hire_date	date	NO		NULL	
job_id	varchar(10)	NO	MUL	NULL	
salary	decimal(8,2)	NO		NULL	
commission_pct	decimal(2,2)	YES		NULL	
manager_id	int(11) unsigned	YES	MUL	NULL	
department_id	int(11) unsigned	YES	MUL	NULL	

## SELECTING IMMEDIATE DATA

- Examples:

SELECT 10+20; ... outputs 30.

- Today's date:

SELECT CURRENT\_DATE();

- Current time:

SELECT CURRENT\_TIME();

- Today's date & time:

SELECT NOW();



## *Restricting the SELECTed data*

## WHERE CLAUSE

- Rows returned from SELECT can be restricted using a WHERE clause which follows the FROM clause
- The WHERE clause contains a condition that must be satisfied (evaluated to true) to restrict the rows

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

- Example:

```
SELECT employee_id, last_name, job_id  
FROM   Employees  
WHERE  department_id = 90 ;
```
- Displays details of employees working in department 90.

## WHERE CLAUSE

- Specifies the condition which needs to be evaluated
- Character and date values need to be enclosed within single-quotes

```
SELECT last_name, hire_date, salary  
FROM Employees  
WHERE hire_date > '2000-01-01';
```

```
SELECT last_name, salary * 12 AS annual_sal  
FROM Employees  
WHERE last_name = 'Livingston';
```

- Note that the string enclosed within single-quote is case sensitive

## WHERE CLAUSE

- A column alias cannot be used in the conditions
- Example:

```
SELECT last_name, salary * 12 AS annual_sal  
FROM Employees  
WHERE annual_sal > 10000; -- Error
```

- Because aliasing of columns is done at the end, after applying WHERE conditions
- Hence the above query should be written as:

```
SELECT last_name, salary * 12 AS annual_sal  
FROM Employees  
WHERE salary * 12 > 150000;
```

## WHERE CLAUSE – LOGICAL OPERATORS

- Query: Retrieve last name and salary of all employees who have salary greater than 50,000, working in department 90.
- To retrieve the requested data:
  - Restrict the data using the conditions: salary > 50,000 AND department\_id = 90

```
SELECT last_name, salary  
FROM Employees  
WHERE salary > 10000  
AND department_id = 90;
```

- Restrict the data using the conditions: salary < 50,000 OR hire\_date > '01-JAN-2008'

```
SELECT last_name, hire_date, salary  
FROM Employees  
WHERE salary < 5000  
OR Hire_date > '2000-01-01';
```

## WHERE CLAUSE – LOGICAL OPERATORS PRECEDENCE

- Logical operators' precedence in that order is: NOT, AND, OR
- Query below tries to list employees drawing salary > 10000 from department 80 or 90:

```
SELECT last_name, department_id, salary  
FROM Employees  
WHERE salary > 10000  
AND department_id = 80  
OR department_id = 90;           // Improper query
```

- Above query retrieves all employees from dept. 90 and only those employees from dept. 80 who draw salary > 10000.
- Parenthesize the query appropriately to get the expected result

```
SELECT last_name, department_id, salary  
FROM Employees  
WHERE salary > 10000  
AND (department_id = 80  
OR department_id = 90);
```

## WHERE CLAUSE – BETWEEN OPERATOR

- Rows can be retrieved based on a range of values. The range specifies the upper limit and the lower limit (both inclusive)
- Below query displays the last name and salary of all employees whose salary is greater than or equal to 10,000 and less than or equal to 25,000

```
SELECT last_name, salary  
FROM Employees  
WHERE salary >= 10000 AND  
      salary <= 25000;
```

- Alternatively, the above can be achieved by using the BETWEEN operator

```
SELECT last_name, salary  
FROM Employees  
WHERE salary BETWEEN 10000 AND 25000 ;
```

## WHERE CLAUSE – IN OPERATOR

- To test for values in a specified set of values, IN operator is used in the WHERE clause which tests the membership condition
- Below query retrieves last name, salary and manager\_id of employees whose manager\_id is 100,101 or 201

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

- The above query can be achieved using a combination of OR conditions

```
SELECT last_name, salary, manager_id  
FROM Employees  
WHERE manager_id = 100 OR manager_id = 101  
OR manager_id = 201;
```

- The set of values to the IN operator can be specified in any order

## WHERE CLAUSE – LIKE OPERATOR

- The exact value to be searched (to restrict the rows) might not be known while executing the query
- Rows can be selected that match a character pattern by using the LIKE operator in the WHERE clause
- These patterns are denoted by:
  - % - zero or many characters
  - \_ - one character

```
SELECT first_name, last_name  
FROM Employees  
WHERE first_name LIKE 'S%';
```

```
SELECT first_name, last_name  
FROM Employees  
WHERE last_name LIKE '_O%'; -- Last name 2nd char 'O'
```

## WHERE CLAUSE – IS NULL

- Conditional operators cannot be used to check for NULL values
- IS NULL operator is used to check for the presence NULL values
- Below query retrieves all employees who does not have a manager

```
SELECT first_name, last_name,  
      FROM Employees  
     WHERE manager_id IS NULL;
```

- IS NOT operator is used to check for the absence of NULL values

```
SELECT first_name, last_name,  
      FROM Employees  
     WHERE COMMISSION_PCT IS NOT NULL;
```

## WHERE CLAUSE – OPERATOR PRECEDENCE

- Rules of precedence determine the order in which the expressions are evaluated
- Use of parenthesis around the expressions would override the default precedence

Operator	Meaning
1	Arithmetic operators
2	Concatenation operator
3	Comparison conditions
4	IS [NOT] NULL, LIKE, [NOT] IN
5	[NOT] BETWEEN
6	Not equal to
7	NOT logical condition
8	AND logical condition
9	OR logical condition

## ORDER BY CLAUSE

- The order of rows returned in a query result is undefined.
- Use ORDER BY clause to sort the retrieved rows : ASC: Ascending order
- The ORDER BY is the last clause in the SELECT statement
- Columns not in the SELECT list can also be used for sorting

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

```
SELECT      last_name, job_id, department_id, hire_date  
FROM        Employees  
ORDER BY    hire_date DESC ;
```

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

## ORDER BY CLAUSE

- Alias columns can be used for sorting purpose:

```
SELECT employee_id, last_name, salary*12 annsal  
FROM Employees  
ORDER BY annsal ;
```

- Sorting the data based on the numeric position of a column:

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

- Sorts by ascending order of dept id and then within each department sort in the descending order of the salary:

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

## CASE EXPRESSION

- CASE expressions allow to use the IF-THEN-ELSE logic in SQL statements without the need 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 AS "REVISED_SALARY"  
  FROM Employees;
```

- Both expressions, after WHEN and THEN, must be of the same data type
- All of the return values also must be of the same data type

## GROUP FUNCTIONS

- Group functions operate on sets of rows and outputs one result per group

EMPLOYEES

	DEPARTMENT_ID	SALARY
1	10	4400
2	20	13000
3	20	6000
4	110	12000
5	110	8300
6	90	24000
7	90	17000
8	90	17000
9	60	9000
10	60	6000
...		
18	80	11000
19	80	8600
20	(null)	7000

Maximum salary in  
EMPLOYEES table

MAX(SALARY)
24000



## USING 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 = 50;
```

- COUNT(DISTINCT expr) returns the number of distinct non-null values of expr. This query displays the number of distinct departments in the EMPLOYEES table

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

## USING GROUP FUNCTIONS

- All group functions ignore null values. Functions like IF, IFNULL, etc. can be used to make group functions consider them
- Query below displays the average, highest, lowest, and sum of monthly salaries for all sales representatives

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

- Below query shows the earliest and latest join dates of employees
- ```
SELECT MIN(hire_date), MAX(hire_date)  
FROM Employees;
```

## GROUP FUNCTIONS AND NULL VALUES

- Group functions ignore null values in the column  

```
SELECT AVG(commission_pct)
FROM Employees;
```
- COUNT(expr) returns the number of rows with non-null values for expr  

```
SELECT AVG(NVL(commission_pct, 0))
FROM Employees;
```
- MEDIAN(expr) returns the median value of set of data  

```
SELECT MEDIAN(salary)
FROM Employees;
```

## GROUPING ROWS

- GROUP BY clause divides the rows into groups
- Summary information of each group can be obtained

**EMPLOYEES**

|     | DEPARTMENT_ID | SALARY |
|-----|---------------|--------|
| 1   | 10            | 4400   |
| 2   | 20            | 13000  |
| 3   | 20            | 6000   |
| 4   | 50            | 2500   |
| 5   | 50            | 2600   |
| 6   | 50            | 3100   |
| 7   | 50            | 3500   |
| 8   | 50            | 5800   |
| 9   | 60            | 9000   |
| 10  | 60            | 6000   |
| 11  | 60            | 4200   |
| 12  | 80            | 11000  |
| 13  | 80            | 8600   |
| ... |               |        |
| 18  | 110           | 8300   |
| 19  | 110           | 12000  |
| 20  | (null)        | 7000   |

4400  
9500  
3500  
6400  
10033

Average salary in the  
EMPLOYEES table for each

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

## GROUPING ROWS – GROUP BY CLAUSE FUNCTIONS

- WHERE clause can be used to restrict rows before dividing them into groups

- Cannot use a column alias in the GROUP BY clause

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

- Grouping can be done on multiple columns:

```
SELECT department_id, job_id, SUM(salary)  
FROM Employees  
GROUP BY department_id, job_id;
```

## ILLEGAL GROUP FUNCTIONS

- Any column or expression in the SELECT list that is not an aggregate function must be in the GROUP BY clause
- Query below gives error. Department\_id should be included in the GROUP BY clause:

```
SELECT department_id, COUNT(last_name)
FROM Employees;
```
- Column job\_id is missing in GROUP BY

```
SELECT department_id, job_id,
       COUNT(last_name)
FROM Employees
GROUP BY department_id;
```
- Similarly, group functions cannot be included in the WHERE clause

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

## RESTRICTING GROUP ROWS – HAVING CLAUSE

- HAVING clause is used to restrict groups in the same way that you use the WHERE clause to restrict the rows that you select
- List those departments where the maximum salary > 1000

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

- Job ids which more than 10 employees have:

```
SELECT job_id, COUNT(employee_id)
FROM Employees
GROUP BY job_id
HAVING COUNT(job_id) > 10;
```

# *Displaying Data from Multiple Tables using Joins*

This file is meant for personal use by amitjain000@gmail.com only.

Proprietary content ©Great Learning. All Rights Reserved. Unauthorized use or distribution prohibited.

## JOINS

- Used to query data from two or more tables
- Based on a relationship between certain columns in these tables
- Tables in a database are often related to each other with common columns (For e.g. dept\_no in employee and dept tables)
- Two (or more) tables are joined based on the common columns among each pair of tables
- Names of common columns may not be same in tables
- If N tables are being joined, there would be N-1 joining statements

## JOINING TABLES - SYNTAX

- ANSI SQL 99 Syntax to retrieve data from multiple tables

```
SELECT table1.column, table2.column  
FROM table1  
[NATURAL JOIN table2] |  
[JOIN table2 USING (column_name)] |  
[INNER JOIN table2  
    ON (table1.column_name = table2.column_name)] |  
[LEFT|RIGHT|FULL OUTER JOIN table2  
    ON (table1.column_name = table2.column_name)] |  
[CROSS JOIN table2];
```

## CARTESIAN PRODUCT JOIN

- Each row in the first table is paired with all the rows in the second table
- This happens when there is no relationship defined between the two tables
- This is the default join between tables if no joining condition is specified
  
- E.g.: Assume Emp and Dept tables contain 100 rows and 10 rows respectively.
- Their Cartesian join will return 1000 rows.

```
SELECT Emp.emp_id, Dept.dept_id  
FROM Emp, Dept;
```
- This query can also be written as:

```
SELECT Emp.emp_id, Dept.dept_id  
FROM Emp CROSS JOIN Dept;
```
  
- One reason to use a Cartesian join is to generate a large amount of rows to use for performance testing

## DIFFERENT SQL JOINS

- INNER JOIN:
  - Returns rows that match on common key values, compared from both tables
- LEFT JOIN:
  - Returns matched rows from both tables
  - And Unmatched rows from the left table
- RIGHT JOIN:
  - Returns matched rows from both tables
  - And Unmatched rows from the Right table

## INNER JOIN

### ➤ Syntax:

```
SELECT <column_name(s)>
FROM <table_name1>
INNER JOIN <table_name2>
    ON <table_name1.column_name> = <table_name2.column_name>
[INNER JOIN <table_name3>
    ON <table_name2.column_name> = <table_name3.column_name>
... and so on.]
[WHERE <condition(s)> ]
```

## INNER JOIN

- Display details of employee and department
  - Department\_id is a common column between Employees & Departments tables
  - 'e' and 'd' are alias for the table names

### SYNTAX 1 (ANSI Std)

```
SELECT employee_id, first_name, last_name,  
       department_name  
  FROM Employees e INNER JOIN Departments d  
    ON e.department_id = d.department_id  
 WHERE employee_id < 111  
 ORDER BY department_name, employee_id;
```

### SYNTAX 2

```
SELECT employee_id, first_name, last_name,  
       department_name  
  FROM Employees e, Departments d  
 WHERE employee_id < 111  
   AND e.department_id = d.department_id  
 ORDER BY department_name, employee_id;
```

## INNER JOIN

- More than two tables can be joined together
- Two tables should have common columns between them
- Names of columns need not be same; however their data types should match
- Following query extends previous query to include location & country details

```
SELECT employee_id, first_name, last_name,      department_name,  
j.job_id, j.job_title, city  
FROM Employees e, Departments d, Jobs j, Locations l  
WHERE employee_id < 111  
AND e.department_id = d.department_id  
AND e.job_id = j.job_id  
AND d.location_id = l.location_id  
ORDER BY 4, 5, 1;
```

## LEFT OUTER JOIN

- The LEFT JOIN keyword returns all rows from the left table (table\_name1) , even if there are no matches in the right table (table\_name1)
- Retrieves rows from right table (table\_name2), that match based on common key
- For non-matching rows, only left table columns are retrieved, with right table columns as NULL

### Syntax:

```
SELECT <column_name(s)>
FROM  <table_name1>
LEFT [OUTER] JOIN <table_name2>
    ON table_name1.column_name = table_name2.column_name
[LEFT [OUTER] JOIN <table_name3>
    ON table_name2.column_name = table_name3.column_name...]
[WHERE <condition(s)>]
```

## Example: LEFT OUTER JOIN

- List all departments those with and without employees
- Employee names will be NULL for those departments in which there are no employees

```
SELECT employee_id, first_name, last_name, department_name  
FROM Departments d LEFT OUTER JOIN Employees e  
ON d.department_id = e.department_id  
ORDER BY 4, 1;
```

## Result: LEFT OUTER JOIN

| EMPLOYEE_ID | FIRST_NAME  | LAST_NAME | DEPARTMENT_NAME  |
|-------------|-------------|-----------|------------------|
| 205         | Shelley     | Higgins   | Accounting       |
| 206         | William     | Gietz     | Accounting       |
| 200         | Jennifer    | Whalen    | Administration   |
|             |             |           | Benefits         |
|             |             |           | Construction     |
|             |             |           | Corporate Tax    |
| 100         | Steven      | King      | Executive        |
| 101         | Neena       | Kochhar   | Executive        |
| 102         | Lex         | De Haan   | Executive        |
| 108         | Nancy       | Greenberg | Finance          |
| 109         | Daniel      | Faviet    | Finance          |
| 112         | Jose Manuel | Urman     | Finance          |
| 113         | Luis        | Popp      | Finance          |
|             |             |           | Government Sales |
| 203         | Susan       | Mavris    | Human Resources  |
| 103         | Alexander   | Hunold    | IT               |
| 104         | Bruce       | Ernst     | IT               |

## RIGHT OUTER JOIN

- Returns all rows from the right table (table\_name2), even if there are no matches in the left table (table\_name1)
- Retrieves rows from Left table (table\_name2) that match based on common key
- For non-matching rows, only right table columns are retrieved, with left table columns as NULL

### Syntax:

```
SELECT <column_name(s)>
FROM  <table_name1> RIGHT [OUTER] JOIN <table_name2>
      ON table_name1.column_name = table_name2.column_name
[RIGHT [OUTER] JOIN <table_name3>
      ON table_name2.column_name = table_name3.column_name...]
[WHERE <condition(s)>]
```

## Example: RIGHT OUTER JOIN

- List all cities that are outside of US and departments names in them
- For those cities where there are no departments, their names will be shown as NULL

```
SELECT department_id, department_name, city, country_id  
FROM Departments d RIGHT OUTER JOIN Locations l  
ON d.location_id = l.location_id  
WHERE country_id <> 'US';
```

## Result: RIGHT OUTER JOIN

| DEPARTMENT_ID | DEPARTMENT_NAME  | CITY        | COUNTRY_ID |
|---------------|------------------|-------------|------------|
|               |                  | Roma        | IT         |
|               |                  | Venice      | IT         |
|               |                  | Tokyo       | JP         |
|               |                  | Hiroshima   | JP         |
| 20            | Marketing        | Toronto     | CA         |
|               |                  | Whitehorse  | CA         |
|               |                  | Beijing     | CN         |
| 120           | Treasury         | Bombay      | IN         |
|               |                  | Sydney      | AU         |
|               |                  | Singapore   | SG         |
| 40            | Human Resources  | London      | UK         |
| 80            | Sales            | Oxford      | UK         |
|               |                  | Stretford   | UK         |
| 70            | Public Relations | Munich      | DE         |
|               |                  | Sao Paulo   | BR         |
|               |                  | Geneva      | CH         |
|               |                  | Utrecht     | NL         |
|               |                  | Mexico City | MX         |

This file is meant for personal use by amitjain000@gmail.com only.

## FULL OUTER JOIN

- Combination of LEFT and RIGHT OUTER JOINS
- Includes unmatched rows from both sides

```
SELECT <column_name(s)>
FROM  <table_name1> FULL [OUTER] JOIN <table_name2>
ON    table_name1.column_name = table_name2.column_name
[WHERE <condition(s)>]
```

```
SELECT e.employee_id, e.first_name, e.last_name, d.department_name
FROM Employees e FULL OUTER JOIN Departments d
ON e.employee_id = d.manager_id;
```

- Note: Employees and Departments tables are joined on manager\_id column

## Result: FULL OUTER JOIN

| EMPLOYEE_ID | FIRST_NAME | LAST_NAME | DEPARTMENT_NAME  |
|-------------|------------|-----------|------------------|
| 100         | Steven     | King      | Executive        |
| 101         | Neena      | Kochhar   |                  |
| 102         | Lex        | De Haan   |                  |
| 103         | Alexander  | Hunold    | IT               |
| 104         | Bruce      | Ernst     |                  |
| 108         | Nancy      | Greenberg | Finance          |
| 196         | Alana      | Walsh     |                  |
| 197         | Kevin      | Feeney    | Human Resources  |
| 198         | Donald     | OConnell  |                  |
| 200         | Jennifer   | Whalen    | Administration   |
| 202         | Pat        | Fay       |                  |
| 203         | Susan      | Mavris    |                  |
| 204         | Hermann    | Baer      | Public Relations |
| 205         | Shelley    | Higgins   | Accounting       |
| 206         | William    | Gietz     |                  |
|             |            |           | Payroll          |
|             |            |           | Recruiting       |
|             |            |           | Retail Sales     |
|             |            |           | Government Sales |
|             |            |           | IT Helpdesk      |

This file is meant for personal use by amitjain000@gmail.com only.

## SELF JOIN

- Joining the table with itself
- Example: Show ids and names of employees and their managers

```
SELECT emp.employee_id, emp.first_name, mgr.employee_id,  
mgr.first_name  
FROM Employees emp, Employees mgr  
WHERE emp.manager_id = mgr.employee_id  
ORDER BY mgr.employee_id, emp.employee_id;
```

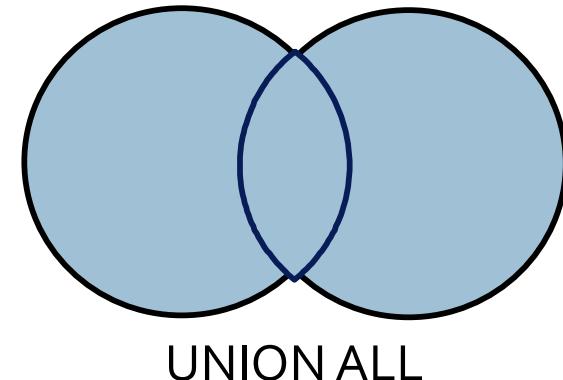
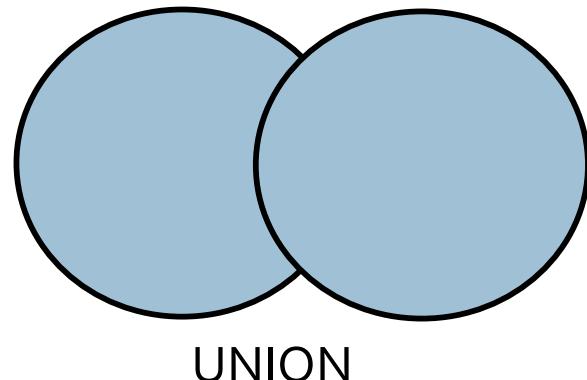
- List peers of 'John Chen' (Join Employees table with itself based on job\_id)

```
SELECT peer.employee_id, peer.first_name  
FROM Employees emp, Employees peer  
WHERE emp.job_id = 'FI_ACCOUNT'  
AND emp.first_name = 'John'  
AND peer.first_name != 'John'  
AND emp.job_id = peer.job_id  
ORDER BY peer.employee_id, emp.employee_id;
```

# *Set Operations*

## UNION

- Used to combine the result-set of two or more SELECT statements removing duplicates\*
- Each SELECT statement within the UNION **must** have the same number of columns
- The selected columns **must** be of similar data types and **must** be in the same order in each SELECT statement
- More than two queries can be clubbed using more than one UNION statement



## UNION

- Query to get a UNION of all types of managers

```
SELECT e.employee_id, e.first_name, j.job_title, d.department_name  
FROM Employees e, Jobs j, Departments d  
WHERE (j.job_id LIKE '%_MAN' OR j.job_id LIKE '%_MGR')  
AND e.job_id = j.job_id  
AND e.department_id = d.department_id  
UNION  
SELECT e.employee_id, e.first_name, j.job_title, d.department_name  
FROM Employees e, Departments d, Jobs j  
WHERE e.employee_id = d.manager_id  
AND e.job_id = j.job_id;
```

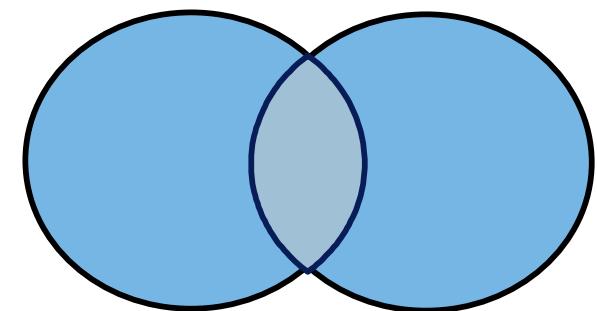
## INTERSECT

- Used to select common result-set from two or more SELECT statements
- All conditions that apply to UNION apply here also
- List all dept. managers whose role is also ending with either 'MAN' or 'MGR'

```
SELECT e.employee_id, e.first_name, j.job_title, d.department_name  
FROM Employees e, Jobs j, Departments d  
WHERE (j.job_id LIKE '%_MAN' OR j.job_id LIKE '%_MGR')  
AND e.job_id = j.job_id  
AND e.department_id = d.department_id
```

INTERSECT

```
SELECT e.employee_id, e.first_name, j.job_title, d.department_name  
FROM Employees e, Departments d, Jobs j  
WHERE e.employee_id = d.manager_id  
AND e.job_id = j.job_id;
```



INTERSECT



# *Using Subqueries*

## SUBQUERY

- Sub query is a query within a query
- Also called as Inner query or Nested query
- Usually added in the WHERE clause of the SQL, but can be in other clauses also
- Used when it is known how to search for a value using a SELECT statement, but do not know the exact value
- Sub queries are an alternate way of returning data from multiple tables

Main query:



Which employees have salaries greater than Smith's salary?

Subquery:



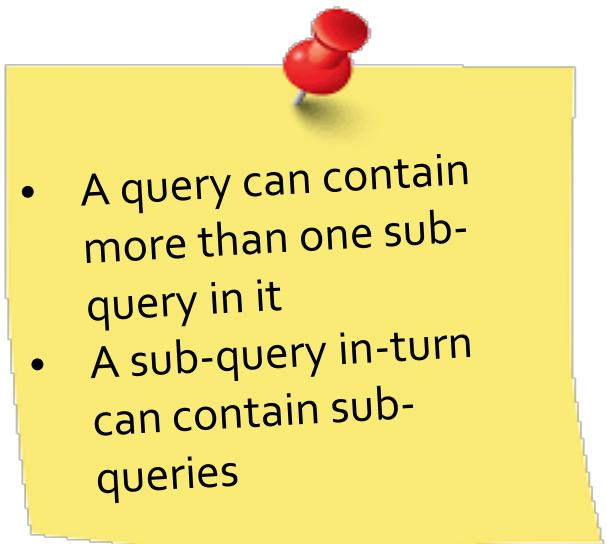
What is Smith's salary?



## SUBQUERY

- The subquery (inner query) executes before the main query (outer query)
- The result of the subquery is used by the main query to restrict the rows
- Comparison operator can be any of >, =, >=, <, <>, <= (Single row operators) and IN, ANY, ALL, EXISTS (Multiple row operators)
- The subquery generally executes first, and its output is used to complete the query condition for the main (or outer) query
- List salaries of those employees who draw higher salary than 'Abel':

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



## SUBQUERY: EXAMPLE

- List employees who have same job id say Nayer but draw higher salary than her:

```
SELECT last_name, job_id, salary  
FROM Employees  
WHERE job_id = (SELECT job_id FROM Employees  
                 WHERE last_name = 'Nayer')  
AND salary > (SELECT salary FROM Employees  
                 WHERE last_name = 'Nayer');
```

- This query can also be written using equivalent join version:

```
SELECT e2.last_name, e2.job_id, e2.salary  
FROM Employees e1, Employees e2  
WHERE e1.last_name = 'Nayer'  
AND e2.job_id = e1.job_id  
AND e2.salary > e1.salary;
```

Not all sub-queries can be converted into equivalent JOIN statements

## GROUP FUNCTIONS IN SUBQUERY

- The database executes the subqueries first
- It returns results into the HAVING clause of the main query

```
SELECT department_id, MIN(salary)
FROM Employees
GROUP BY department_id
HAVING MIN(salary) > (SELECT MIN(salary)
                           FROM Employees
                           WHERE department_id = 50);
```

List departments whose minimum salary is more than the min. salary of dept 50.

- Following query results in error as the inner query returns multiple rows:

```
SELECT employee_id, last_name
FROM Employees
WHERE salary = (SELECT MIN(salary)
                  FROM Employees
                  GROUP BY department_id);
```

## SUBQUERY: TYPES

### Single-row sub query

- Returns one row
- Scalar sub query returns a single row with one column
- Comparison operator is any of the following:
  - = equal
  - > greater than
  - >= greater than or equal
  - < less than
  - <= less than or equal
  - <> not equal n

### Multiple-row sub query

- Return multiple rows
- Commonly used to generate result sets which are passed to a DML or SELECT statement for further processing
- Comparison operator is any of the following:
  - IN: equal to any member in the list
  - NOT IN: not equal to any member in the list
  - ANY: returns rows that match with any of the values in the list
  - ALL: returns rows that match with all the values in the list

## SUBQUERIES WITHIN SUBQUERY

- List employees who work in countries other than US & UK:

```
SELECT employee_id, first_name, last_name  
FROM Employees  
WHERE department_id IN  
    (SELECT department_id FROM Departments  
     WHERE location_id IN  
         (SELECT location_id FROM Locations  
          WHERE country_id NOT IN ('US','UK')));
```

## SUBQUERY: USING ANY CLAUSE

- Compares a value to each value in a list or a query output
- Must be preceded by =, !=, >, <, <= or >=.
- Can be followed by any expression or sub-query that returns one or more values
- Evaluates to FALSE if the query returns no rows
- List employees and their job id & salary whose salary is less than any of the IT\_PROG's salary

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

## SUBQUERY: USING ALL CLAUSE

- Compares a value to every value in a list or returned by a query
- Must be preceded by =, !=, >, <, <= or >=
- Can be followed by any expression or sub-query that returns one or more values
- Evaluates to TRUE if the query returns no rows
- List employees and their job id & salary whose salary is less than all of the IT\_PROG's salary:

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

## SUBQUERY IN FROM CLAUSE

- A sub query can also be found in the FROM clause
- These are called *inline views*, 'dept' in this query example:

```
SELECT employee_id, first_name, last_name, department_name  
FROM Employees e,  
(SELECT * FROM Departments WHERE location_id <> 1700) dept  
WHERE e.department_id = dept.department_id;
```

## MULTIPLE-COLUMNS IN SUBQUERY

- Rows in the sub query results are evaluated in the main query in pair-wise comparison.
- That is, column-to-column comparison and row-to-row comparison
- Example: List peers of employee id 123

```
SELECT employee_id, first_name, last_name, job_id  
FROM Employees e  
WHERE employee_id != 123  
AND (job_id, department_id) IN  
(SELECT job_id, department_id FROM Employees  
WHERE employee_id = 123);
```

## NULL VALUES IN SUBQUERY

- Display all the departments where there are no employees working:  

```
SELECT department_id FROM Departments  
WHERE department_id NOT IN  
(SELECT department_id FROM Employees);
```
- The above query will not return any rows as the department\_id of an employee is NULL
- As there is a NULL value in the subquery, it fails
- The corrected query will be:  

```
SELECT department_id FROM Departments  
WHERE department_id NOT IN  
(SELECT department_id FROM Employees  
WHERE department_id IS NOT NULL);
```

## CORRELATED SUBQUERY

- Both inner and the outer query are interdependent
- For every row processed by the inner query, the outer query is processed as well
- The inner query depends on the outer query before it can be processed
- Example: List details of employees who draw more salary than the average salaries of their respective departments and job ids:

```
SELECT employee_id, first_name, department_id, job_id, salary  
FROM Employees e1  
WHERE salary > (SELECT AVG(salary) FROM Employees e2  
WHERE e1.department_id = e2.department_id  
AND e1.job_id = e2.job_id);
```

## EXISTS OPERATOR

- The EXISTS operator is used in queries where the query result depends on whether or not certain rows exist in a table

- It evaluates to TRUE if the subquery returns at least one row

```
SELECT employee_id, salary, last_name  
FROM employees m  
WHERE EXISTS (SELECT employee_id FROM employees e  
WHERE (e.manager_id = m.employee_id)  
AND e.salary > 10000);
```

- NOT EXISTS evaluates to TRUE if the subquery returns no rows

```
SELECT * FROM departments d  
WHERE NOT EXISTS  
(SELECT * FROM employees e  
WHERE e.department_id=d.department_id);
```

exists / not exists are usually co-related sub-queries; whereas, in / not in are normally independent sub-queries



# *Manipulating Data*

## DML STATEMENT

- Statements for manipulating the data in the database
  - INSERT, UPDATE, DELETE, MERGE, ...
- Should use COMMIT to make the changes permanent
- Use ROLLBACK to undo the changes
- ROLLBACK will undo till the last commit
- Any DDL statement executed after DML statements will implicitly COMMIT the changes
- Hence exercise caution while using DML and DDL statements together to prevent unwanted changes getting COMMITted

## INSERT INTO STATEMENT

- To insert a new row into a table (one row at a time)
- List of column names is optional
- If it is omitted, values for all columns should be passed in the same order as in the table
- To insert rows with values in specific columns, column names must be mentioned in the query
- Values for all non-nullable columns must be passed in the INSERT statements
- Syntax:

```
INSERT INTO      <table_name> [(<column_name/s>)]  
VALUES          (<value 1>, <value 2>, ...);
```

## EXAMPLE: INSERT INTO

- Inserting values in all columns:

```
INSERT INTO Employees  
VALUES(301, 'Suma', 'Sinha','ssinha', '515.124.4579',  
SYSDATE, 'AC_MGR', 12500, NULL, 101, 110);
```

- Inserting specific columns:

```
INSERT INTO Locations (location_id, city, country_id)  
VALUES(3300, 'Bengaluru', 'IN');
```

- NULL will be inserted to rest of the columns in the last case
- Note that all non-nullable columns should be specified in the column list

## EXAMPLE: INSERT INTO

- Inserting using subquery:

```
INSERT INTO Departments (department_id, department_name)  
VALUES ((SELECT MAX(department_id)+10 FROM Departments),  
'Training');
```

- Inserting using SELECT:

```
INSERT INTO Temp_Emp  
SELECT * FROM Employees WHERE department_id = 50;
```

- Temp\_Emp should have the same structure as Employees table

## UPDATE STATEMENT

- Updates columns of a table as per the condition specified

- Syntax:

```
UPDATE <table name>
SET   <column_name> = <value> [, <column_name> = <value>, ...]
[WHERE condition(s)];
```

- Updating All Rows:

```
UPDATE Employees
SET   commission_pct = NULL;
```

- Updating Specific rows:

```
UPDATE Departments
SET   manager_id = 150
WHERE department_id IS NULL
AND location_id <> 1700;
```

## UPDATE STATEMENT

### ➤ Updating Multiple Columns

UPDATE Locations

```
SET street_address = '#24, 4th Main, 7th Sector',  
postal_code = '570 100',  
state_province = 'KA'  
WHERE city = 'Bengaluru';
```

### ➤ Updating using subqueries (needed to access other tables):

UPDATE Departments

```
SET manager_id = (SELECT employee_id FROM Employees  
WHERE job_id = 'ST_MAN' AND hire_date = '01-MAY-95')  
WHERE manager_id IS NULL  
AND department_name LIKE 'IT%';
```

## DELETE FROM STATEMENT

- To delete rows from a table as per the condition specified
- DELETE command can delete only row(s) and not column(s) from a table
- Syntax:  
`DELETE FROM <table name>  
[WHERE <condition>];`
- Deleting All Rows  
`DELETE FROM Jobs;`
- Important note: Parent record will not get deleted if it has child records i.e. if there are employees in a given job\_id, then that job\_id row can not be deleted unless all its child records are deleted

## DELETE FROM STATEMENT

- Deleting Specific Rows:

```
DELETE FROM Employees  
WHERE employee_id = 199;
```

- Deleting rows based on conditions that are based on other tables

- Deleting departments where there are no employees working:

```
DELETE FROM Departments d  
WHERE NOT EXISTS  
(SELECT 1 FROM Employees e  
WHERE e.department_id = d.department_id);
```

- Note: Since the EXISTS clause subquery checks for existence of a row in the subquery, it doesn't matter which column gets selected in the subquery

## NOTE: DML STATEMENTS

- COMMIT statement makes the changes made by the DML statements, permanent
- A ROLLBACK will rollback or undo all the changes (but not after COMMIT)
- ROLLBACK to <savepoint> will undo the changes till the save point

- INSERT INTO Books ...
- UPDATE Books SET ...
- `SAVEPOINT sp1;`
- INSERT INTO Book\_Sales ...
- DELETE FROM Books ...
- ROLLBACK TO `sp1;`
- COMMIT;

Example:

- ROLLBACK will rollback the last 2 INSERT & DELETE statements
- COMMIT will commit first INSERT & UPDATE statements

## TRUNCATE STATEMENT

- Removes all rows from a table, leaving the table empty and the table structure intact
- Data definition language (DDL) statement rather than a DML statement; cannot easily be undone
- Since it's a DDL statement, it's auto-committed; hence it cannot be rolled back
- DELETE statement without a WHERE clause also deletes all rows; however, it can be rolled back

```
TRUNCATE TABLE temp_emp;
```

## ADVANTAGES OF COMMIT AND ROLLBACK STATEMENTS

- With COMMIT and ROLLBACK statements, you can:
  - Ensure data consistency
  - Preview data changes before making changes permanent
  - Group logically related operations



# *DDL to Create and Manage Tables*

## DDL STATEMENTS

- CREATE <object>
- ALTER <object>
- DROP <object>
- TRUNCATE <object>
- COMMENT <object>
- RENAME <object>
  
- <object> can be TABLE, VIEW, INDEX, SEQUENCE, etc.

## CREATE TABLE EXAMPLE

| Column Name   | Data type     | Constraints                  |
|---------------|---------------|------------------------------|
| Cust_Id       | Int(8)        | Primary key of the table     |
| Cust_Fname    | Varchar(20)   | First name - Not Null column |
| Cust_LName    | Varchar (20)  | Last name                    |
| Account_No    | Bigint(18)    | Not null, Unique column      |
| Branch_Code   | Varchar(10)   | Not Null column              |
| Cust_Email    | Varchar (50)  | Unique column                |
| Cust_Phone_no | Varchar(20)   |                              |
| Curr_Balance  | Decimal(20,2) | Current balance - Not null   |

### Saving\_Account\_Details

## SQL - CREATE TABLE

- PRIMARY KEY ,NOT NULL and UNIQUE

```
CREATE TABLE Savings_Account_Details
(
    Cust_Id INT(8) PRIMARY KEY,
    Cust_FName VARCHAR(20) NOT NULL,
    Cust_Lname VARCHAR(20),
    Account_No BIGINT(18) NOT NULL UNIQUE,
    Branch_code VARCHAR(10) NOT NULL,
    Cust_Email VARCHAR(50) UNIQUE,
    Cust_Phone_No VARCHAR(20),
    Curr_Balance DECIMAL (20, 2) NOT NULL
);
```

## TYPES OF CONSTRAINTS

- Primary Key
- Foreign Key
- Not Null \*
- Check
- Unique

```
CREATE TABLE Subject_Info  
( subject_name  Varchar(20) NOT NULL,  
  subject_id    Int(10) PRIMARY KEY,  
  semester      Int(2),  
  branch_id     Varchar(20),  
  UNIQUE (branch_id, semester));
```

\*All constraints except NOT NULL can specify more than one column

This file is meant for personal use by amitjain000@gmail.com only.

Proprietary content ©Great Learning. All Rights Reserved. Unauthorized use or distribution prohibited

## CREATE TABLE (CONTD...)

| Column Name   | Data type     | Constraints                                                                                      |
|---------------|---------------|--------------------------------------------------------------------------------------------------|
| Loan_AccNo    | Bigint(18)    | Primary key                                                                                      |
| Cust_Id       | Int(8)        | Can take only those values which are present in the Cust_Id of the Savings_Account_Details Table |
| Loan_Amount   | Decimal(20,2) |                                                                                                  |
| Loan_Term     | Int(2)        | In months                                                                                        |
| Loan_Int_Type | Character(1)  | Loan interest type:<br>'F' for 'Fixed' or 'V' for Variable                                       |

### Loan Account

#### Primary Key and Foreign Key Constraints

## CREATE TABLE (CONTD...)

- Foreign Key:

```
CREATE TABLE Loan_Account (
    Loan_AccNo BIGINT(18) PRIMARY KEY,
    Cust_Id INT(5) REFERENCES
Savings_Account_Details(Cust_Id),
    Loan_Amount DECIMAL(18, 2),
    Loan_Term INT(2),
    Loan_Int_Type CHAR(1)
        CHECK(Loan_Int_Type = 'F' OR Loan_Int_Type = 'V')
) ;
```

## CREATE TABLE (CONTD...)

| Column Name   | Data type    | Constraints                                                        |
|---------------|--------------|--------------------------------------------------------------------|
| Employee_No   | Int(6)       | Primary key of the table                                           |
| First_Name    | Varchar(20)  |                                                                    |
| Last_Name     | Varchar (20) |                                                                    |
| Email_Id      | Varchar (30) |                                                                    |
| Dept_No       | Int(2)       | Default 10 (Say, 'TRAINING')                                       |
| Manager_EmpNo | Int(6)       | Can take only those values which are present in Employee_No column |

## Employee

### Implementation of Self Referencing Foreign key

This file is meant for personal use by amitjain000@gmail.com only.

## CREATE TABLE (CONTD...)

- Self Referential Foreign Key and DEFAULT:

```
CREATE TABLE Employee
(
    Employee_No INT(6) PRIMARY KEY,
    First_Name    VARCHAR(20),
    Last_Name     VARCHAR(20),
    Email_Id      VARCHAR (30),
    Dept_No        INT(2)      DEFAULT 10,
    Manager_Emp_No INT(6) REFERENCES Employee (Employee_No)
);
```

## CREATE TABLE (CONTD...)

| Column Name   | Data type     | Constraints                                                                                      |
|---------------|---------------|--------------------------------------------------------------------------------------------------|
| FD_Acc_No     | Bitint(14)    | Primary key                                                                                      |
| Cust_Id       | Int(8)        | Can take only those values which are present in the Cust_Id of the Savings_Account_Details Table |
| FD_Amount     | Decimal(20,2) |                                                                                                  |
| Interest_Rate | Decimal(5,2)  | Can take values only between 2.5 to 12.0                                                         |

### FD\_Account

#### Implementation of Check Constraints

This file is meant for personal use by amitjain000@gmail.com only.

## CREATE TABLE (CONTD...)

- CHECK constraint:

```
CREATE TABLE FD_Account (
    FD_Acc_No BIGINT(14) PRIMARY KEY,
    Cust_Id INT(8) REFERENCES
Savings_Account_Details(Cust_Id),
    FD_Amount DECIMAL(20,2),
    Interest_Rate DECIMAL(5,2)
    CHECK (Interest_Rate >=2.5 AND Interest_Rate <=12.0)
);
```

## CREATE TABLE (CONTD...)

| Column Name   | Data type     | Constraints                                                                                     |
|---------------|---------------|-------------------------------------------------------------------------------------------------|
| Cust_Id       | Int(8)        | Can take only those values which are present in Cust_Id column of Savings_Account_Details Table |
| Txn_Date      | Date          | (Cust_Id, Transaction_Date) are together forming the Primary Key of the table.                  |
| Txn_Type      | Char(2)       | 'CR' or 'DB'                                                                                    |
| Txn_Amt       | Decimal(20,2) |                                                                                                 |
| Available_Bal | Decimal(24,2) |                                                                                                 |

### Transaction\_Table

#### Implementation of Composite Primary Key Constraints

This file is meant for personal use by amitjain000@gmail.com only.

## CREATE TABLE (CONTD...)

- Composite Primary key constraint:

```
CREATE TABLE Transaction_Table(
    Cust_Id INT(8) REFERENCES
Savings_Account_Details(Cust_Id),
    Acct_No      BIGINT(14),
    Txn_Date     TIMESTAMP(6),
    Txn_Type     CHAR(2) CHECK (txn_type='CR' OR
txn_type='DB'),
    Txn_Amt      DECIMAL(20,2),
    Available_Bal DECIMAL(24,2),
    PRIMARY KEY(Acct_No, Txn_Date)
) ;
```

## CREATE TABLE (CONTD...)

- Composite Foreign key constraint

```
CREATE TABLE Journal_Update(
    Journal_ID      BIGINT(16),
    Journal_Type    VARCHAR(20),
    JU_Cust_Id      INT(8),
    JU_Txn_Date     TIMESTAMP(6),
    Txn_Amt         DECIMAL(20,2),
    FOREIGN KEY(JU_Txn_date, JU_Cust_Id) REFERENCES
Transaction_Table(Txn_date,Cust_Id)
);
```

## CREATE TABLE (CONTD...)

- From a SELECT statement
  - New table gets created with the columns and data from the SELECT statement

```
CREATE TABLE <table name> AS  
    <SELECT statement>
```

```
CREATE TABLE Emp_Info AS  
    SELECT employee_id, first_name, last_name, job_id,  
          d.department_id, department_name  
    FROM Employees e, Departments d  
   WHERE e.department_id = d.department_id;
```

If the selected columns have NOT NULL constraint, it will be copied onto the new table.

## ALTER TABLE

- Add/Drop/Modify /Rename Column

- Syntax:

```
ALTER TABLE tablename (ADD/MODIFY/RENAME/DROP column_name)
```

- Examples:

```
ALTER TABLE Emp_Info  
    ADD contact_no VARCHAR(20);
```

```
ALTER TABLE Emp_Info  
    MODIFY contact_no INT(10);
```

```
ALTER TABLE Emp_Info  
    DROP COLUMN contact_no;
```

```
ALTER TABLE Emp_Info  
    RENAME COLUMN job_id TO job_code;
```

## ALTER TABLE (CONTD...)

### ADD/DROP/MODIFY Column

- Used to modify the structure of a table by adding and / or removing columns
- MODIFY option cannot be used to change the name of a column or table
- Column to be modified should be empty to decrease the column length or to change its data type
- If the table has only one column, the ALTER TABLE statement cannot be used to drop that column because that would make the table definition invalid

## ALTER TABLE (CONTD...)

### ➤ Add/Drop Constraint

```
ALTER TABLE Emp_Info  
ADD CONSTRAINT PK_Emp_Info PRIMARY KEY (employee_id);
```

```
ALTER TABLE Book_Info  
DROP PRIMARY KEY;
```

## ALTER TABLE (CONTD ...)

- ADD/DROP Foreign Key Constraint

```
ALTER TABLE Emp_Info  
    ADD CONSTRAINT FK_dept_id FOREIGN KEY (department_id)  
REFERENCES Departments(department_id);
```

```
ALTER TABLE Emp_Info  
    DROP FOREIGN KEY FK_dept_id;
```

- A table can have one or more Foreign keys
- Adding a foreign key constraint using ALTER TABLE command will result in an error if the existing data in master or child table does not support the foreign key restriction

## ALTER TABLE (CONTD ...)

- ALTER TABLE statement can be used to Add or Drop:
  - Primary key
  - Foreign key
  - Unique constraint
  - Check constraint
- If a table already has a primary key, then adding a primary key using the ALTER TABLE statement results in an error
- RDBMS will not allow a PRIMARY KEY constraint on column(s) if the column(s) have NULL or duplicate values

## TRUNCATE TABLE

- Deleting All Rows of a table:

```
TRUNCATE TABLE Emp_Info;
```

- Similar to DELETE, without a WHERE clause
- It is very fast on both large and small tables
- Cannot undo TRUNCATE (using ROLLBACK) unlike DELETE
- TRUNCATE is DDL and, like all DDL, performs an implicit commit
- Hence, any uncommitted DML changes will also be committed with the TRUNCATE

## SQL - DROP TABLE

- **DROP TABLE**
  - Deletes table structure
  - Cannot be recovered
  - Should be used with caution

```
DROP TABLE Emp_Info;
```

- **NOTE:** Cannot drop the table with unique or primary keys referenced by foreign keys in another table.

# THANK YOU