SQL: Schema Definition, Constraints, and Queries and Views

SQL

- SQL is a comprehensive database language: It has statements for data definitions, queries, and updates.
- It is both a DDL and a DML.
- SQL stands for Structured Query Language
- SQL lets you access and manipulate databases
- SQL is an ANSI (American National Standards Institute) standard

What Can SQL do?

- SQL can execute queries against a database
- SQL can retrieve data from a database
- SQL can insert records in a database
- SQL can update records in a database
- SQL can delete records from a database
- SQL can create new databases
- SQL can create new tables in a database
- SQL can create stored procedures in a database
- SQL can create views in a database
- SQL can set permissions on tables, procedures, and views

Using SQL in Your Web Site

- To build a web site that shows data from a database, you will need:
  - An RDBMS database program (i.e. MS Access, SQL Server, MySQL)
  - To use a server-side scripting language, like PHP or ASP
  - To use SQL to get the data you want
  - To use HTML / CSS

Data Definition Language

- Data definition language, which is usually part of a database management system, is used to define and manage all attributes and properties of a database, including row layouts, column definitions, key columns, file locations, and storage strategy
- A DDL statement supports the definition or declaration of database objects such as databases, tables, and views

SQL Data Definition and Data Types

- SQL uses the terms **table, row** and **column** for the formal relational model terms relation, tuple, and attribute, respectively.

- The main SQL command for data definition is the CREATE statement, which can be used to create schemas, tables (relations), and domains (as well as other constructs such as views, and triggers).

Data Definition Language

Most DDL statements take the following form:

- CREATE *object_name*
- ALTER *object_name*
- DROP *object_name*

Create Command

- CREATE DATABASE DatabaseName;

- CREATE TABLE *table_name*
  (
  column_name1 data_type(*size*),
  column_name2 data_type(*size*),
  column_name3 data_type(*size*),
  ....
  );

- To list all the databases that are present in the SQL serve

```
SELECT * FROM sys.databases;
```

- To show all the databases that are present in the SQL server using the stored procedure

```
EXEC sp_databases;
```

CREATE TABLE

```
USE Northwind
CREATE TABLE Importers
(
  CompanyID int NOT NULL,
  CompanyName varchar(40) NOT NULL,
  Contact varchar(40) NOT NULL
)
```

ALTER TABLE

- The ALTER TABLE statement enables you to modify a table definition by altering, adding, or dropping columns and constraints or by disabling or enabling constraints and triggers.
- To add a column in a table, use the following syntax:

```
ALTER TABLE table_name
ADD column_name datatype
```

ALTER TABLE

- To delete a column in a table, use the following syntax (notice that some database systems don't allow deleting a column):

ALTER TABLE table_name
DROP COLUMN column_name

- To change the data type of a column in a table, use the following syntax:(SQL Server / MS Access)

ALTER TABLE table_name
ALTER COLUMN column_name datatype

OR (My SQL / Oracle)

ALTER TABLE table_name
MODIFY COLUMN column_name datatype

ALTER TABLE

- The following statement will alter the Importers table in the Northwind database by adding a column named ContactTitle to the table.

```
USE Northwind
ALTER TABLE Importers
ADD ContactTitle varchar(20) NULL
```

Implementing Integrity Constraints

- A constraint is a property assigned to a table or a column within a table that prevents invalid data values from being placed in specified column(s).

- Examples include UNIQUE, constraints, PRIMARY KEY constraints and FOREIGN KEY constraints

Creating PRIMARY KEY Constraints

You can create a PRIMARY KEY onstraint by using one of the following methods:

- Creating the constraint when the table is created (as part of the table definition)
- Adding the constraint to an existing table, provided that no other PRIMARY KEY constraint already exists.

PRIMARY KEY -EXAMPLE

```
CREATE TABLE Table1
(
  col1 int PRIMARY KEY,
  col2 varchar(30)
)
```

PRIMARY KEY – table level constraint

```
CREATE TABLE Table1
(
  col1 int ,
  col2 varchar(30),
  CONSTRAINT table_pk PRIMARY KEY (col1)
)
```

Composite Key

```
CREATE TABLE FactoryProcess
  (
  EventType INT,
  EventTime DATETIME,
  EventSite CHAR(50),
  EventDesc CHAR(1024),
  CONSTRAINT event_key PRIMARY KEY
  (EventType, EventTime)
  )
```

Adding a Primary to an Existing Table

```
ALTER TABLE Table1
ADD CONSTRAINT table_pk PRIMARY KEY (col1)
```

UNIQUE CONSTRAINTS

- You can use UNIQUE constraints to ensure that no duplicate values are entered in specific columns that do not participate in a primary key.

- Although both a UNIQUE constraint and a PRIMARY KEY constraint enforce uniqueness, you should use a UNIQUE constraint instead of a PRIMARY KEY constraint in the following situations:

UNIQUE CONSTRAINTS

- **If a column (or combination of columns) is not the primary key.** Multiple UNIQUE constraints can be defined on a table, whereas only one PRIMARY KEY constraint can be defined on a table.

UNIQUE CONSTRAINTS

- **If a column allows null values.** UNIQUE constraints can be defined for columns that allow null values, whereas PRIMARY KEY constraints can be defined only on columns that do not allow null values.
- A UNIQUE constraint can also be referenced by a FOREIGN KEY constraint.

Creating UNIQUE Constraints

- You can create a UNIQUE constraint in the same way that you create a PRIMARY KEY constraint:
- By creating the constraint when the table is created (as part of the table definition)
- By adding the constraint to an existing table, provided that the column or combination of columns comprising the UNIQUE constraint contains only unique or NULL values. A table can contain multiple UNIQUE constraints.

Creating UNIQUE Constraints

- You can use the same Transact-SQL statements to create a UNIQUE constraint that you used to create a PRIMARY KEY constraint.
- Simply replace the words PRIMARY KEY with the word UNIQUE.

FOREIGN KEY Constraints

- A foreign key is a column or combination of columns used to establish and enforce a link between the data in two tables.
- Create a link between two tables by adding a column (or columns) to one of the tables and defining those columns with a FOREIGN KEY constraint.
- The columns will hold the primary key values from the second table. A table can contain multiple FOREIGN KEY constraints.

SQL FOREIGN KEY Constraint

- A FOREIGN KEY in one table points to a PRIMARY KEY in another table.
- Let's illustrate the foreign key with an example. Look at the following two tables:

- The "Persons" table:

The "Persons" table

P_Id	LastName	FirstName	Address	City
1	Hansen	Ola	Timoteivn 10	Sandnes
2	Svendson	Tove	Borgvn 23	Sandnes
3	Pettersen	Kari	Storgt 20	Stavanger

The "Orders" table:

O_Id	OrderNo	P_Id
1	77895	3
2	44678	3
3	22456	2
4	24562	1

- Note that the "P_Id" column in the "Orders" table points to the "P_Id" column in the "Persons" table.
- The "P_Id" column in the "Persons" table is the PRIMARY KEY in the "Persons" table.
- The "P_Id" column in the "Orders" table is a FOREIGN KEY in the "Orders" table.
- The FOREIGN KEY constraint is used to prevent actions that would destroy links between tables.
- The FOREIGN KEY constraint also prevents that invalid data form being inserted into the foreign key column, because it has to be one of the values contained in the table it points to.

SQL FOREIGN KEY Constraint on CREATE TABLE

```
CREATE TABLE Orders
(
O_Id int NOT NULL,
OrderNo int NOT NULL,
P_Id int,
PRIMARY KEY (O_Id),
FOREIGN KEY (P_Id) REFERENCES
Persons(P_Id)
)
```

To allow naming of a FOREIGN KEY constraint, and for defining a FOREIGN KEY constraint on multiple columns, use the following SQL syntax:

```
CREATE TABLE Orders
(
O_Id int NOT NULL,
OrderNo int NOT NULL,
P_Id int,
PRIMARY KEY (O_Id),
CONSTRAINT fk_PerOrders FOREIGN KEY (P_Id)
REFERENCES Persons(P_Id)
)
```

- http://www.w3schools.com/sql/sql_foreignkey.asp

Creating FOREIGN KEY Constraints

You can create a FOREIGN KEY constraint by using one of the following methods:

- Creating the constraint when the table is created (as part of the table definition)
- Adding the constraint to an existing table, provided that the FOREIGN KEY constraint is linked to an existing PRIMARY KEY constraint or a UNIQUE constraint in another (or the same) table

Foreign Key Constraints

```
CREATE TABLE Table1
 (
  Col1 INT PRIMARY KEY,
   Col2 INT  REFERENCES Employees(EmployeeID)

 )
```

table-level FOREIGN KEY constraint

```
CREATE TABLE Table1
(
 Col1 INT PRIMARY KEY,
 Col2 INT,
 CONSTRAINT col2_fk FOREIGN KEY (Col2)
 REFERENCES Employees (EmployeeID)
)
```

ALTER TABLE statement to add a FOREIGN KEY

```
ALTER TABLE Table1
ADD CONSTRAINT col2_fk FOREIGN KEY
(Col2) REFERENCES Employees (EmployeeID)
```

DROP TABLE

- Indexes, tables, and databases can easily be deleted/removed with the DROP statement
- The DROP TABLE statement removes a table definition and all data, indexes, triggers, constraints, and permission specifications for that table.
- Any view or stored procedure that references the dropped table must be explicitly dropped by using the DROP VIEW or DROP PROCEDURE statement.

DROP TABLE

- The following statement drops the Importers table from the Northwind database.

```
USE Northwind
DROP TABLE Importers
```

Data Control Language

- Data control language is used to control permission on database objects. Permissions are controlled by using the GRANT and REVOKE statements and the SQL DENY statement.

GRANT

- The GRANT statement creates an entry in the security system that enables a user in the current database to work with data in that database or to execute specific Transact-SQL statements.
- The following statement grants the Public role SELECT permission on the Customers table in the Northwind database:

GRANT

```
USE Northwind
GRANT SELECT
ON Customers
TO PUBLIC
```

REVOKE

- The REVOKE statement removes a previously granted or denied permission from a user in the current database.
- The following statement revokes the SELECT permission from the Public role for the Customers table in the Northwind database:

```
USE Northwind
REVOKE SELECT
ON Customers
TO PUBLIC
```

DENY

- The DENY statement creates an entry in the security system that denies a permission from a security account in the current database and prevents the security account from inheriting the permission through its group or role memberships.

  ```
  USE Northwind
  DENY SELECT
  ON Customers
  TO PUBLIC
  ```

Data Manipulation Language

Data manipulation language is used to select, insert, update, and delete in the objects defined with DDL.

SELECT

The SELECT statement retrieves rows from the database and enables the selection of one or many rows or columns from one or many tables. SELECT statement, sometimes called a **mapping** or a **select-from-where block**, is formed of the three clauses SELECT, FROM, and WHERE and has the following form

```
SELECT <attribute list>
FROM   <table list>
WHERE  <condition>
```

SELECT

Where

<attribute list> is a list of attribute names whose values are to be retrieved by the query.

<table list> is a list of the relation names required to process the query.

<condition> is a conditional (Boolean) expression that identifies the tuples to be retrieved by the query.

QUERY 1: Retrieve the birth date and address of the employee(s) whose name is 'John B. Smith'.

```
SELECT      Bdate, Address
FROM   EMPLOYEE
WHERE Fname = 'John' AND Minit = 'B'
AND Lname = 'Smith';
```

QUERY 2: Retrieve the name and address of all employees who work for the 'Research' department.

-
  ```
  SELECT   FNAME, LNAME,
  ADDRESS
  FROM     EMPLOYEE,
  DEPARTMENT
  WHERE    DNAME='Research'
  AND DNUMBER=DNO
  ```

QUERY 3: For every project located in 'Stafford', list the project number, the controlling department number, and the department manager's last name, address, and birthdate.

```
SELECT          PNUMBER, PNAME, DNUM,
DNAME, FNAME, MINIT, LNAME,
                BDATE, ADDRESS
FROM   PROJECT, DEPARTMENT,
EMPLOYEE
WHERE PLOCATION='Stafford' AND
DNUM=DNUMBER AND MGRSSN=SSN
```

QUERY 4: The following statement retrieves the CustomerID, CompanyName, and ContactName data for companies who have a CustomerID value equal to alfki or anatr. The result is ordered according to the ContactName value:

```
USE Northwind
SELECT CustomerID, CompanyName,
   ContactName
            FROM Customers
WHERE (CustomerID = ‘alfki’ OR CustomerID
   =‘anatr’)
ORDER BY ContactName
```

Ambiguous Attribute Names, Aliasing, and Tuple Variables

- In SQL the same name can be used for two (or more) attributes as long as the attributes are in different relations.
- If this is the case, and a query refers to two or more attributes with the same name, we must qualify the attribute name with the relation name to prevent ambiguity.
- This is done by prefixing the relation name to the attribute name and separating the two by a period.

Example: Ambiguous Attribute Names

```
SELECT Fname, EMPLOYEE.Name, Address
FROM   EMPLOYEE, DEPARTMENT
WHERE DEPARTMENT.NAME='Research'
AND DEPARTMENT.Dnumber= EMPLOYEE.
.Dnumber ;
```

Ambiguous Attribute Names, Aliasing, and Tuple Variables

- Ambiguity also arises in the case of queries that refer to the same relation twice, as in the following example.

- QUERY 5: For each employee, retrieve the employee's name, and the name of his or her immediate supervisor.

```
Q5:SELECT    E.FNAME, E.LNAME, S.FNAME,
             S.LNAME
   FROM      EMPLOYEE AS E, EMPLOYEE AS S
   WHERE     E.SUPERSSN=S.SSN
```

Ambiguous Attribute Names, Aliasing, and Tuple Variables

- In this case, we are allowed to declare alternative relation names E and S, called **aliases** or **tuple variables,** for the EMPLOYEE relation. An alias can follow the keyword **AS,** as shown in Q5,

Unspecified WHERE Clause and Use of the Asterisk

- A missing WHERE clause indicates no condition on tuple selection; hence all tuples of the relation specified in the FROM clause qualify and are selected for the query result.

- If more than one relation is specified in the FROM clause and there is no WHERE clause, then CROSS PRODUCT – all possible tuple combinations – of these relations is selected.

- For example, Query 6 selects all EMPLOYEE Ssns and Query 7 selects all combinations of an EMPLOYEE Ssn and a DEPARTMENT Dname.

- Q6: **SELECT** Ssn
  FROM EMPLOYEE

- Q7: **SELECT** Ssn, Dname
  FROM EMPLOYEE, DEPARTMENT

Use of **Asterisk**

- To retrieve all the attribute values of the selected tuples, we do not have to list the attribute names explicitly in SQL; we just specify an asterisk (*), which stands for all the attributes.

```
SELECT   *
FROM     EMPLOYEE
WHERE    Dno = 5
```

Use of Asterisk

```
SELECT   *
FROM     EMPLOYEE, DEPARTMENT
WHERE    Dname = 'Research' AND Dno=
         Dnumber
      ORDER BY  fname
SELECT   *
FROM     EMPLOYEE, DEPARTMENT
```

Aggregate Functions in SQL

- Grouping and aggregation are required in many database applications, SQL has features that incorporate these concepts.
- A number of built-in functions exist: **COUNT, SUM, MAX, MIN,** and **AVG.**
- The COUNT function returns the number of tuples or values as specified in a query.

Aggregate Functions in SQL

- The functions SUM, MAX, MIN, and AVG are applied to a set of multiset of numeric values and return, respectively, the sum, maximum value, minimum value, and average (mean) of those values.

- These functions can be used in the SELECT clause or in a HAVING clause (which we introduce later).

Aggregate Functions in SQL

- The functions MAX and MIN can also be used with attributes that have nonnumeric domains if the domain values have a total ordering among one another.
- We illustrate the use of these functions with example queries.

Find the sum of the salaries of all employees, the maximum salary, the minimum salary, and the average salary.

- **SELECT** **SUM**(Salary), **MAX**(Salary), **MIN**(Salary),**AVG**(Salary)

  FROM EMPLOYEE

Query 9: Find the sum of the salaries of all employees of the 'Research' department, as well as the maximum salary, the minimum salary, and the average salary in this department.

```
Q9:   SELECT  SUM(Salary), MAX(Salary),
        MIN(Salary), AVG(Salary)
        FROM EMPLOYEE, DEPARTMENT
          WHERE Dname = 'Research' AND
                Dno = Dnumber
```

Queries 10 and 11. Retrieve the total number of employees in the company (Q10) and the number of employees in the 'Research' department (Q11).

- Q10:
```
SELECT   COUNT(*)
FROM     EMPLOYEE
```

- Q11:
```
SELECT   COUNT(*)
FROM     EMPLOYEE, DEPARTMENT
WHERE    Dno=Dnumber AND
         Dname='Research'
```

Grouping: The GROUP BY and HAVING Clauses

- In many cases we want to apply the aggregate functions to subgroups of tuples in a relation, where the subgroups are based on some attribute values.

- For example, we may want to find the average salary of employees in each department or the number of employees who work on each project.

Grouping: The GROUP BY and HAVING Clauses

- In these cases we need to **partition** the relation into nonoverlapping subsets (or groups) of tuples.

- Each group (partition) will consist of the tuples that have the same value of some attribute(s), called the **grouping attribute(s).**

Grouping: The GROUP BY and HAVING Clauses

- We can then apply the function to each such group independently. SQL has a **GROUP BY** clause for this purpose.

- The GROUP BY clause specifies the grouping attributes, which should also appear in the SELECT clause, so that the value resulting from applying each aggregate function to a group of tuples appears along with the value of the grouping attribute(s).

For each department, retrieve the department number, the number of employees in the department, and their average salary.

```
SELECT   Dno, COUNT(*), AVG(Salary)
FROM     EMPLOYEE
GROUP BY Dno
```

GROUP BY EXAMPLE

```
SELECT productid, count(*), sum(quantity)
FROM [order details]
GROUP BY productid
```

The above will group by productid, count the orders per product and return the total quantity of each product that has been ordered from the Order Details table

GROUP BY EXAMPLE

For each project, retrieve the project number, the project name, and the number of employees who work on that project.

```
SELECT      Pnumber, Pname, COUNT(*)
FROM        PROJECT, WORKS_ON
WHERE       Pnumber =Pno
GROUP BY    Pnumber, Pname;
```

HAVING Clauses

- When we want to retrieve the values of functions for only groups that satisfy certain conditions, then we use the HAVING Clause.
- For example, suppose that we want to modify previous so that only projects with more than two employees appear in the result.
- SQL provides a **HAVING** clause, which can appear in conjunction with a GROUP BY clause, for this purpose.

HAVING Clause

- HAVING provides a condition on the group of tuples associated with each value of the grouping attributes. Only that satisfy the condition are retrieved in the result of the query.

HAVING Clause Example

Query: For each project on which more than two employees work, retrieve the project number, the project name, and the number of employees who work on the project.

```
SELECT      Pnumber, Pname, COUNT(*)
FROM    PROJECT, WORKS_ON
WHERE   Pnumber =Pno
GROUP BY    Pnumber, Pname
HAVING COUNT(*) > 2
```

The ORDER BY Clause

- The ORDER BY clause sorts a query result by one or more columns (up to 8060 bytes).
- A sort can be ascending (ASC) or descending (DESC). If neither is specified, ASC is assumed. If more than one column is named in the ORDER BY clause, sorts are nested.

The ORDER BY Clause - Example

```
USE Pubs
SELECT Pub_id, Type, Title_id, Price
FROM Titles
ORDER BY Pub_id DESC, Type,
Price
```

INSERT, DELETE and UPDATE Statements in SQL

- In its simplest form, INSERT is used to add a single tuple to a relation.
- We must specify the relation name and a list of values for the tuple.
- The values should be listed in the same order in which the corresponding attributes were specified in the CREATE TABLE command.

INSERT -Example

- For example, to add a new tuple to the EMPLOYEE relation ; we can use

```
INSERT INTO          EMPLOYEE
    VALUES ('Richard','K','Marini', '653298653',
    '30-DEC-52','98 Oak Forest,Katy,TX', 'M',
   37000,'987654321', 4 )
```

INSERT – Example 2

```
USE Northwind
INSERT INTO Territories
VALUES (98101, ‘Seatle’, 2)
```

INSERT –Type 2

- A second form of INSERT statement allows the user to specify explicit attribute names that correspond to the values provided in the INSERT command.
- This is useful if a relation has many attributes but only a few of those attributes are assigned values in the new tuple.

INSERT

- The values must include all attributes with NOT NULL specification and no default value.

- Attributes with NULL allowed or DEFAULT values are the ones that can be left out.

INSERT –Type 2 Example

- For example, to enter a tuple for a new EMPLOYEE for whom we know only the Fname, Lname, Dno, and Ssn attributes, we can use the statement below:

```
INSERT INTO EMPLOYEE (FNAME,
LNAME, Dno, SSN)
VALUES ('Richard', 'Marini',4, '653298653')
```

UPDATE

- The UPDATE command is used to modify attribute values of one or more selected tuples.
- As in the DELETE command, a WHERE clause in the UPDATE command selects the tuples to be modified from a single relation.

UPDATE

- However, updating a primary key value may propagate to the foreign key values of tuples in other relations if such a referential triggered action is specified in the referential integrity constraints of the DDL.
- An additional **SET** clause in the UPDATE command specifies the attributes to be modified and their new values.

UPDATE - Example

- For example, to change the location and controlling department number of project number 10 to ‘Bellaire’ and 5 respectively, we use the statement below:

```
UPDATE      PROJECT
SET         PLOCATION = 'Bellaire', DNUM = 5
WHERE       PNUMBER=10
```

DELETE STATEMENT

The DELETE statement removes rows from a table

```
DELETE
FROM [table]
WHERE {condition}
```

DELETE STATEMENT - Example

This example deletes all rows from the **authors** table.

```
USE pubs
DELETE authors
```

DELETE STATEMENT - Example

- This example deletes all rows in which **au_lname** is McBadden.

  ```
  USE pubs
  DELETE FROM authors
  WHERE au_lname = 'McBadden'
  ```

VIEWS

- A **view** in SQL/MySQL terminology is a single table that is derived from other tables.
- These other tables can be base tables or previously defined views.
- A view does not necessarily exist in physical form; it is considered a **virtual table,** in contrast to base tables, whose tuples are actually stored in the database.

VIEWS

- In SQL, a view is a virtual table based on the result-set of an SQL statement.
- A view contains rows and columns, just like a real table.
- The fields in a view are fields from one or more real tables in the database.
- View doesn't actually store information itself, but just pulls it out of one or more existing tables.
- Although impermanent, a view may be accessed repeatedly by storing its criteria in a query.

VIEWS

- We may frequently issue queries that retrieve the employee name and the project names that the employee works on.

- Rather than having to specify the join of the EMPLOYEE, WORKS_ON, and PROJECT tables every time we issue that query, we can define a view that is a result of these joins.

VIEWS

- Then we can issue queries on the view, which are specified as single table retrievals rather than as retrievals involving two joins on three tables.

- We call the EMPLOYEE, WORKS_ON, and PROJECT tables the **defining tables** of the view.

SYNTAX

```
CREATE
    [OR REPLACE]
    [ALGORITHM = {UNDEFINED | MERGE | TEMPTABLE}]
    [DEFINER = { user | CURRENT_USER }]
    [SQL SECURITY { DEFINER | INVOKER }]
    VIEW view_name [(column_list)]
    AS select_statement
    [WITH [CASCADED | LOCAL] CHECK OPTION]
```

- *If the view does not exist, CREATE OR REPLACE VIEW is the same asCREATE VIEW*
- *If the view does exist, CREATE OR REPLACE VIEW is the same as ALTER VIEW.*

- The view definition is “frozen” at creation time, so changes to the underlying tables afterward do not affect the view definition.
- For example,

  if a view is defined as SELECT * on a table, new columns added to the table later do not become part of the view

VIEW – Example

```
CREATE VIEW  WORKS_ON1
 AS SELECT      Fname, Lname, Pname, Hours
 FROM           EMPLOYEE, PROJECT, WORKS_ON
   WHERE        Ssn=Essn AND Pno =Pnumber

CREATE VIEW   DEPT_INFO(Dept_name, No_of_emps, Total_sal)
 AS SELECT      Dname, COUNT (*), SUM(Salary)
       FROM     DEPARTMENT,EMPLOYEE
       WHERE    Dnumber=Dno
    GROUP BY    Dnumber
```

VIEW – Example

```
mysql> CREATE TABLE t (qty INT, price INT);
mysql> INSERT INTO t VALUES(3, 50);
mysql> CREATE VIEW v AS SELECT qty, price, qty*price AS value FROM t;
mysql> SELECT * FROM v;
+------+-------+-------+
| qty  | price | value |
+------+-------+-------+
|    3 |    50 |   150 |
+------+-------+-------+
```

Querying A View

- we can utilise the WORKS_ON1 view and specify the query as below:

```
SELECT   Fname, Lname
FROM     WORKS_ON1
WHERE    Pname = ‘ProjectX’
```

Using Advanced Query Techniques To Access Data

- One of these techniques is to combine the contents of two or more tables to produce a result set that incorporates rows and columns from each table.
- Another technique is to use subqueries, which are SELECT statements nested inside other SELECT, INSERT, UPDATE, or DELETE statements.
- Subqueries can also be nested in other subqueries.

Using Joins To Retrieve Data

- By using joins, you can retrieve data from two or more tables based on logical relationships between the tables. Joins can be specified in either the FROM or WHERE clauses.

- The join conditions combine with the WHERE and HAVING search conditions to control the rows that are selected from the base tables referenced in the FROM clause.

- Specifying the join conditions in the FROM clause, however, helps separate them from any other search conditions that might be specified in a WHERE clause and is the recommended method for specifying joins.

JOINS

- When multiple tables are referenced in a single query, all column references must be unambiguous@.

- The table name must be used to qualify any column name that is duplicated in two or more tables referenced in a single query.

JOINS

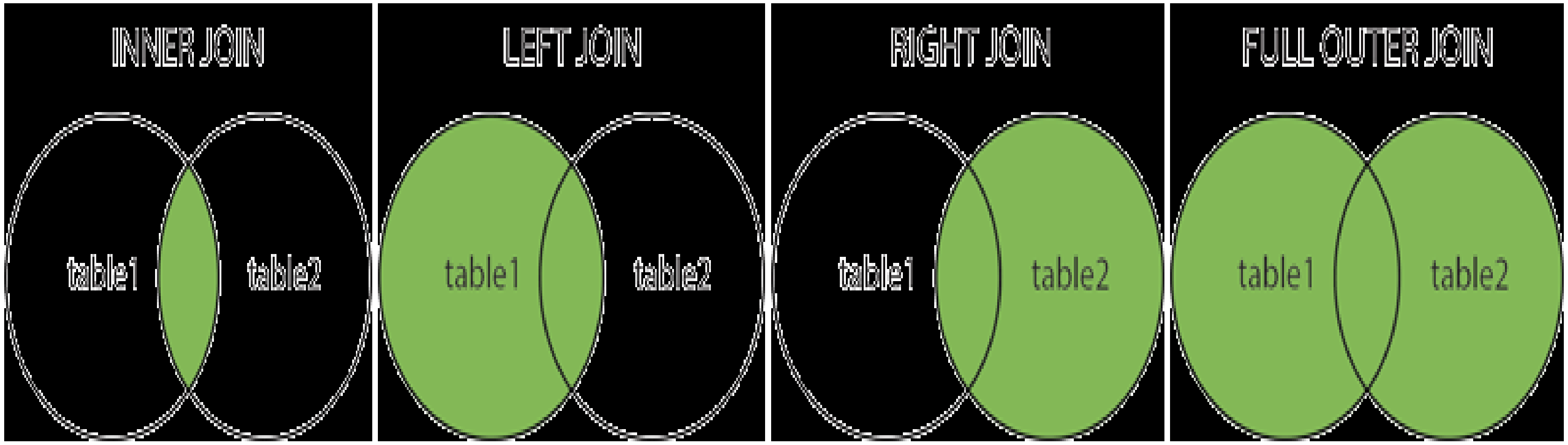
- The select list for a join can reference all of the columns in the joined tables or any subset of the columns.

- The select list is not required to contain columns from every table in the join.

JOINS

- Although join conditions usually use the equals sign (=) comparison operator, other comparison or relational operators can be specified (as can other predicates).
- Most joins can be categorized as inner joins or other joins.

JOINS


INNER JOIN
table1
table2
LEFT JOIN
table1
table2
RIGHT JOIN
table1
table2
FULL OUTER JOIN
table1
table2

INNER JOIN

- Inner joins return rows only when there is at least one row from both tables that matches the join condition, eliminating the rows that do not match with a row from the other table.
- An inner join is a join in which the values in the columns being joined are compared through the use of a comparison operator.

INNER JOIN

- The following SELECT statement uses an inner join to retrieve data from the Publishers table and the Titles table in the Pubs database:

```
SELECT t.Title, p.Pub_name
FROM Publishers AS p INNER JOIN
Titles AS t
ON p.Pub_id = t.Pub_id
ORDER BY Title ASC
```

Below is a selection from the "Customers" table:

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
1	Alfreds Futterkiste	Maria Anders	Obere Str. 57	Berlin	12209	Germany
2	Ana Trujillo Emparedados y helados	Ana Trujillo	Avda. de la Constitución 2222	México D.F.	05021	Mexico
3	Antonio Moreno Taquería	Antonio Moreno	Mataderos 2312	México D.F.	05023	

And a selection from the "Orders" table:

OrderID	CustomerID	EmployeeID	OrderDate	ShipperID
10308	2	7	1996-09-18	3
10309	37	3	1996-09-19	1
10310	77	8	1996-09-20	2

```
SELECT Customers.CustomerName, Orders.OrderID
FROM Customers
INNER JOIN Orders
ON Customers.CustomerID=Orders.CustomerID
ORDER BY Customers.CustomerName;
```

Number of Records: 196

CustomerName	OrderID
Ana Trujillo Emparedados y helados	10308
Antonio Moreno Taquería	10365
Around the Horn	10355
Around the Horn	10383
B's Beverages	10289
Berglunds snabbköp	10278
Berglunds snabbköp	10280

Outer Joins

- Outer joins, however, return all rows from at least one of the tables or views mentioned in the FROM clause as long as these rows meet any WHERE or HAVING search conditions.
- There are three types of outer joins: left, right, and full.
- All rows retrieved from the left table are referenced with a left outer join, and all rows from the right table are referenced in a right outer join.
- All rows from both tables are returned in a full outer join.

Using Left Outer Joins

- A result set generated by a SELECT statement that includes a left outer join includes all rows from the table referenced to the left of LEFT OUTER JOIN.

- The only rows that are retrieved from the table to the right are those that meet join condition.

Using Left Outer Joins

- In the following SELECT statement, a left outer join is used to retrieve the authors' first names, and (when applicable) the names of any publishers that are located in the same cities as the authors:

Using Left Outer Joins

```
USE Pubs
SELECT a.Au_fname, a.Au_lname, p.Pub_name
FROM    Authors a LEFT OUTER JOINPublishers p
ON a.City = p.City
ORDER BY p.Pub_name ASC, a.Au_lname ASC,
a.Au_fname ASC
```

Using Left Outer Joins

```
SELECT Customers.CustomerName, Orders.OrderID
FROM Customers  LEFT JOIN Orders
ON Customers.CustomerID=Orders.CustomerID
ORDER BY Customers.CustomerName;
```

Number of Records: 213

CustomerName	OrderID
Alfreds Futterkiste	null
Ana Trujillo Emparedados y helados	10308
Antonio Moreno Taquería	10365
Around the Horn	10355
Around the Horn	10383
B's Beverages	10289
Berglunds snabbköp	10278
Berglunds snabbköp	10280
Berglunds snabbköp	10384
Blauer See Delikatessen	null

Using Right Outer Joins

- A result set generated by a SELECT statement that includes a right outer join includes all rows from the table referenced to the right of RIGHT OUTER JOIN.
- The only rows that are retrieved from the table to the left are those that meet the join condition.
- In the following SELECT statement, a right outer join is used to retrieve the list of publishers and authors’ first names, and last names, if those authors are located in the same cities as the publishers:

Using Right Outer Joins

```
USE Pubs
SELECT a.Au_fname, a.Au_lname,
p.Pub_name
FROM    Authors a RIGHT OUTER JOIN
Publishers p
ON a.City = p.City
ORDER BY p.Pub_name ASC, a.Au_lname
ASC, a.Au_fname ASC
```

Using Full Outer Joins

- A result set generated by a SELECT statement that includes a full outer join includes all rows from both tables, regardless of whether the tables have a matching value (as defined in the join condition).

- In the following SELECT statement, a full outer join is used to retrieve the list of publishers and authors’ first and last names:

Using Full Outer Joins

```
USE Pubs
SELECT a.Au_fname, a.Au_lname, p.Pub_name
FROM    Authors a FULL OUTER JOIN Publishers p
ON a.City = p.City
ORDER BY p.Pub_name ASC, a.Au_lname ASC,
                a.Au_fname ASC
```

Using Full Outer Joins

```
SELECT Customers.CustomerName, Orders.OrderID
FROM Customers FULL OUTER JOIN Orders
ON Customers.CustomerID=Orders.CustomerID
ORDER BY Customers.CustomerName;
```

A selection from the result set may look like this:

CustomerName	OrderID
Alfreds Futterkiste	
Ana Trujillo Emparedados y helados	10308
Antonio Moreno Taquería	10365
	10382
	10351

Defining Subqueries inside SELECT Statement

- A subquery is a SELECT statement that returns a single value and is nested inside a SELECT, INSERT, UPDATE or DELETE statement or inside another subquery.

- A subquery can be used anywhere an expression is allowed. A subquery is also called an inner query or inner select, while the statement containing a subquery is called an outer select.

Subqueries

- Subqueries are an alternate way of returning data from multiple tables.
- Subqueries can be used with the following SQL statements along with the comparision operators like =, <, >, >=, <= etc.

- In the following example, a subquery is nested in the

  WHERE clause of the outer SELECT statement:

Guidelines for Using Subqueries

- Enclose subqueries in parentheses.
- Place subqueries on the right side of the comparison operator.
- Do not add an ORDER BY clause to a subquery.
- Use single-row operators with single row subqueries.
- Use multiple-row operators with multiple-row subqueries.

Subquery - Example

```
USE Northwind
SELECT ProductName
FROM Products
WHERE UnitPrice =
      {
       SELECT UnitPrice
       FROM Products
       WHERE ProductName = ‘Bel~Aqua’

       }
```

Subqueries

- If a table only appears in a subquery and not in the outer query, then columns from that table cannot be included in the output (the select list of the outer query).

Types of Subqueries

Subqueries can be specified in many places within a SELECT statement. Statements that include a subquery usually take one of the following formats:

WHERE <expression> [NOT] IN (<subquery>)

WHERE <expression> <comparison_operator> [ANY | ALL] (<subquery>)

WHERE [NOT] EXISTS (<subquery>)

Comparision Operators	Description
LIKE	column value is similar to specified character(s).
IN	column value is equal to any one of a specified set of values.
BETWEEN...AND	column value is between two values, including the end values specified in the range.
IS NULL	column value does not exist.

Subqueries that Are used with IN or NOT IN

- The result of a subquery introduced with IN (or with NOT IN) is a list of zero or more values.

  ie. Usually, a subquery should return only one record, but sometimes it can also return multiple records when used with operators LIKE, IN, NOT IN in the where clause.

- After the subquery returns the result, the outer query makes use of it.

Subquery - Example

```
SELECT name
FROM ingredients
WHERE vendorid =
(SELECT vendorid
FROM vendors
WHERE companyname = ‘OP & D');
```

Note: *SQL marks the boundaries of a subquery with parentheses.*

In the above query, the = operator makes the connection between the queries.

Because = expects a single value, the inner query may only return a result with a

single attribute and row.

If subquery returns multiple rows, we must use a different operator.

We use the IN operator, which expects a subquery result with zero or

more rows.

Find the name of all ingredients supplied by OP & D or Spring Water Supply

```
SELECT name
FROM ingredients
WHERE vendorid IN
(SELECT vendorid FROM vendors WHERE
companyname = ‘OP & D' OR companyname =
'Spring Water Supply');
```

Find the average unit price for all items provided by OP & D

```
SELECT AVG(unitprice) AS avgprice
FROM ingredients
WHERE vendorid IN
(SELECT vendorid
FROM vendors
WHERE companyname = 'OP & D');
```

SQL LIKE Operator

- The LIKE operator is used to list all rows in a table whose column values match a specified pattern.
- It is useful when you want to search rows to match a specific pattern, or when you do not know the entire value.
- For this purpose we use a wildcard character '%'.

For example:

- To select all the students whose name begins with 'S‘

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

The output would be similar to:

```
first_name        last_name
------------- -------------
Stephen   Wadoodo
Shekar    Powerful
```

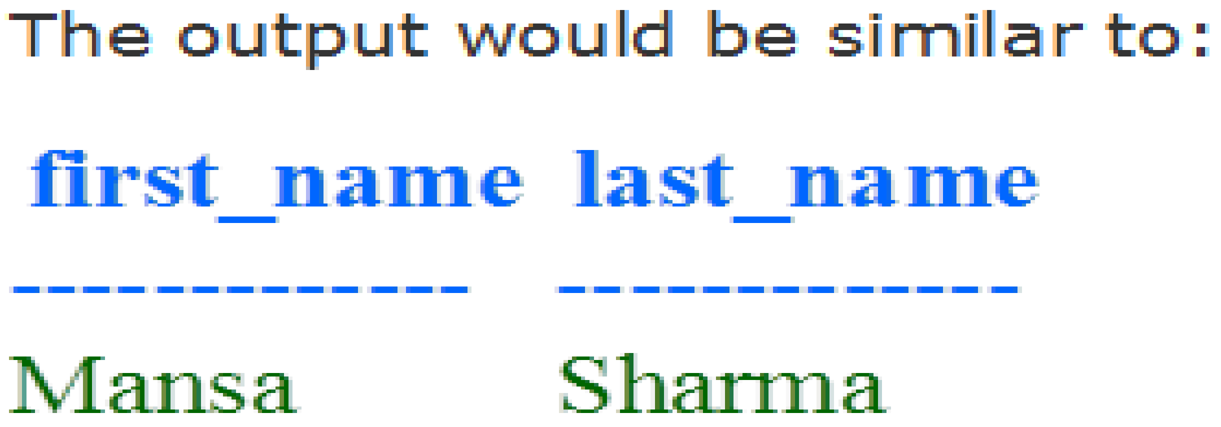
SQL LIKE Operator

- The above select statement searches for all the rows where the first letter of the column first_name is 'S' and rest of the letters in the name can be any character.
- There is another wildcard character you can use with LIKE operator. It is the underscore character, ' _ ' . In a search string, the underscore signifies a single character.

Example

- To display all the names with 'a' second character

SELECT first_name, last_name
FROM student_details
WHERE first_name LIKE '_a%';



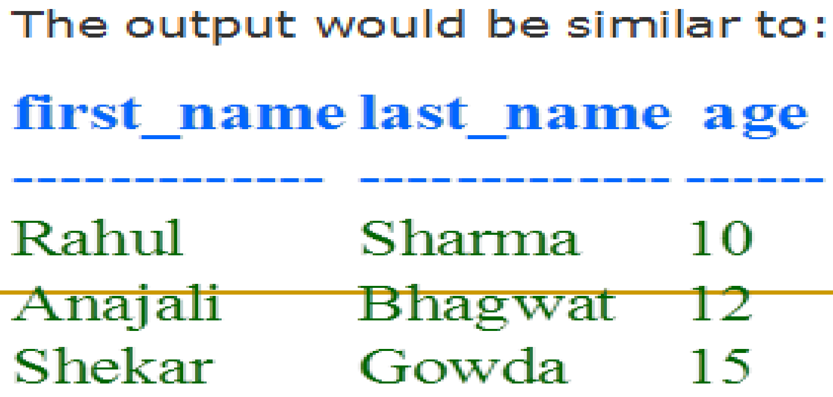
The output would be similar to:

```
first_name  last_name
----------  ---------
Mansa       Sharma
```

SQL BETWEEN ... AND Operator

- The operator BETWEEN and AND, are used to compare data for a range of values.
- For Example: to find the names of the students between age 10 to 15 years, the query would be like,

```
SELECT first_name, last_name, age
FROM student_details
WHERE age BETWEEN 10 AND 15;
```



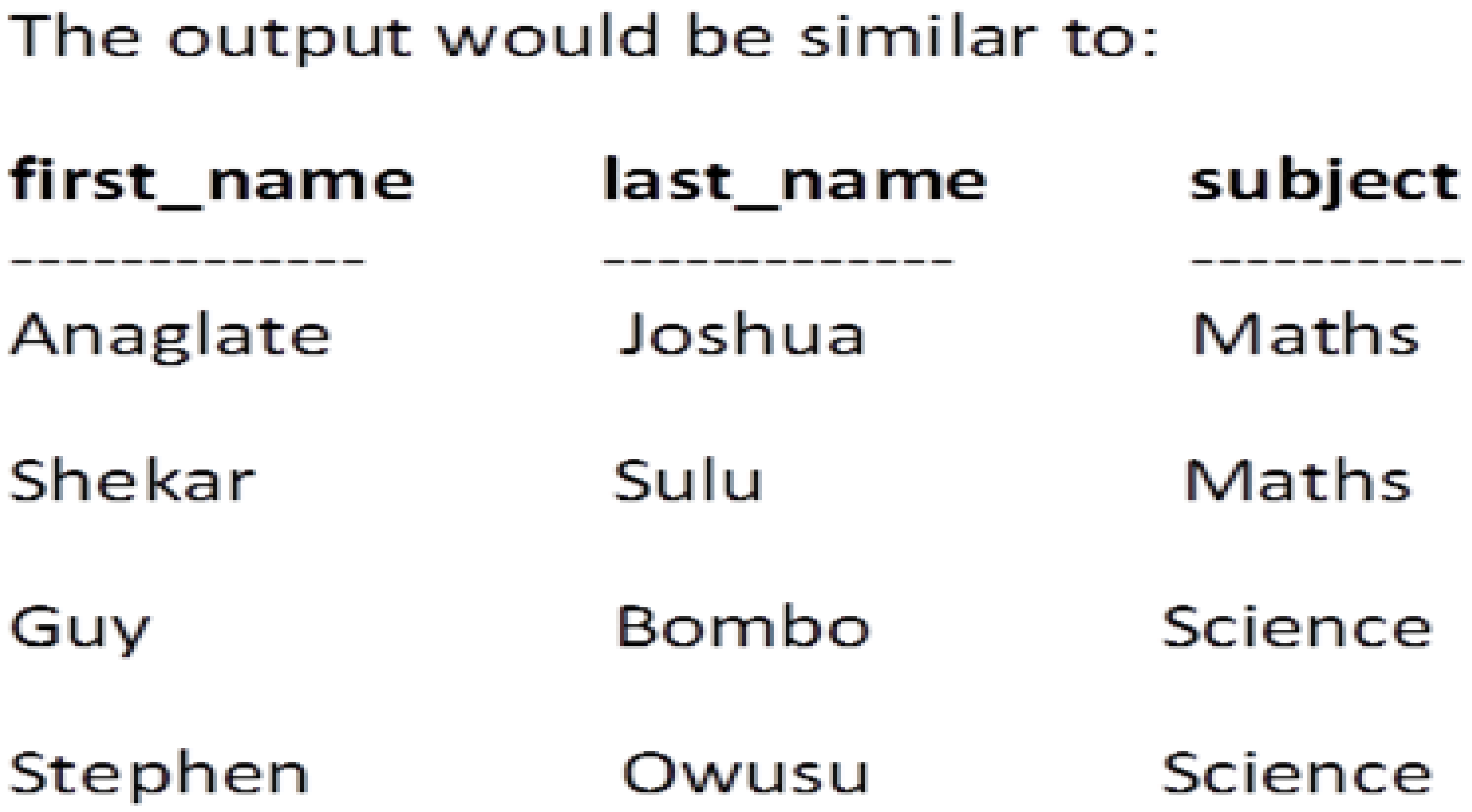
The output would be similar to:

```
first_name last_name  age
---------- ---------- ------
Rahul      Sharma     10
Anajali    Bhagwat    12
Shekar     Gowda      15
```

SQL IN Operator:

- The IN operator is used when you want to compare a column with more than one value. It is similar to an OR condition.
- **For example:** If you want to find the names of students who are studying either Maths or Science, the query would be like,

```
SELECT first_name, last_name, subject
FROM student_details
WHERE subject IN ('Maths', 'Science');
```



The output would be similar to:

```
first_name      last_name       subject
------------    ------------    ----------
Anaglate        Joshua          Maths
Shekar          Sulu            Maths
Guy             Bombo           Science
Stephen         Owusu           Science
```

- You can include more subjects in the list like ('maths','science','history')
- **NOTE:**The data used to compare is case sensitive.

Subqueries Using NOT IN:

- Find all of the ingredients supplied by someone other than OP & D

```
SELECT name
FROM ingredients
WHERE vendorid NOT IN
(SELECT vendorid FROM vendors WHERE
companyname = ‘OP & D');
```

Find the company name of the small vendors who don’t provide any ingredients with large (>100) inventories

```
SELECT companyname
FROM vendors
WHERE vendorid NOT IN
(SELECT vendorid FROM ingredients WHERE inventory > 100);
```

Subqueries with INSERT statement

- INSERT statement can be used with subqueries. Here are the syntax and an example of subqueries using INSERT statement.

```
INSERT INTO table_name [ (column1 [, column2 ]) ]
SELECT [ *|column1 [, column2 ]
FROM table1 [, table2 ]
[ WHERE VALUE OPERATOR ];
```

Example

- If we want to insert those orders from 'orders' table which have the advance_amount 2000 or 5000 into 'neworder' table the following SQL can be used:
- Sample table : orders

```
   ORD_NUM ORD_AMOUNT ADVANCE_AMOUNT ORD_DATE  CUST_CODE       AGENT_CODE ORD_DESC
---------- ---------- -------------- --------- --------------- -------------
    200114       3500           2000 15-AUG-08 C00002          A008
    200122       2500            400 16-SEP-08 C00003          A004
    200118        500            100 20-JUL-08 C00023          A006
    200119       4000            700 16-SEP-08 C00007          A010
    200121       1500            600 23-SEP-08 C00008          A004
    200130       2500            400 30-JUL-08 C00025          A011
    200134       4200           1800 25-SEP-08 C00004          A005
    200108       4000            600 15-FEB-08 C00008          A004
    200103       1500            700 15-MAY-08 C00021          A005
    200105       2500            500 18-JUL-08 C00025          A011
    200109       3500            800 30-JUL-08 C00011          A010
    200101       3000           1000 15-JUL-08 C00001          A008
    200111       1000            300 10-JUL-08 C00020          A008
    200104       1500            500 13-MAR-08 C00006          A004
    200106       2500            700 20-APR-08 C00005          A002
    200125       2000            600 10-OCT-08 C00018          A005
    200117        800            200 20-OCT-08 C00014          A001
    200123        500            100 16-SEP-08 C00022          A002
    200120        500            100 20-JUL-08 C00009          A002
```

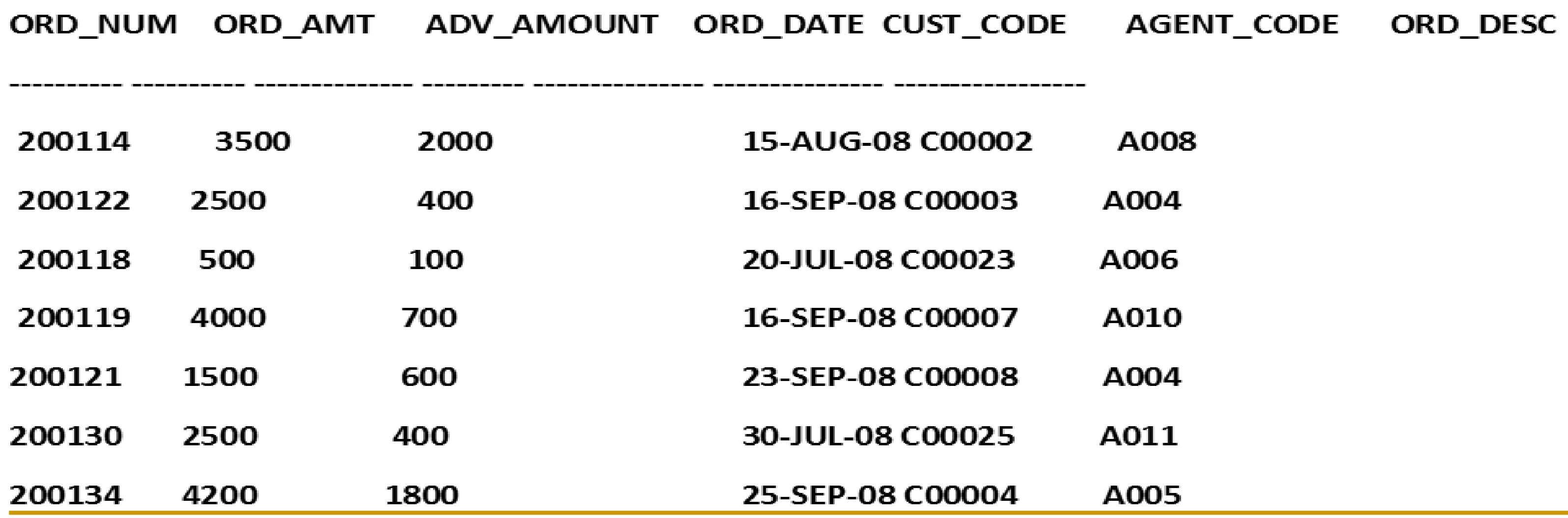
```
INSERT INTO neworder
SELECT * FROM  orders
WHERE advance_amount in(2000,5000);
```

Subqueries with UPDATE statement

```
UPDATE table  SET column_name = new_value
[ WHERE OPERATOR [ VALUE ]
(SELECT COLUMN_NAME
FROM TABLE_NAME)
[ WHERE) ]
```

Example

- If we want to update that ord_date in 'neworder' table with '15-JAN-10' which have the difference of ord_amount and advance_amount is less than the minimum ord_amount of 'orders' table the following SQL can be used:
- Sample table : neworder



```
ORD_NUM    ORD_AMT    ADV_AMOUNT   ORD_DATE  CUST_CODE    AGENT_CODE    ORD_DESC
---------- ---------- -------------- --------- -------------- -------------- -----------------
 200114      3500          2000           15-AUG-08 C00002      A008
 200122     2500           400            16-SEP-08 C00003     A004
 200118      500           100            20-JUL-08 C00023     A006
 200119     4000          700             16-SEP-08 C00007     A010
200121     1500           600             23-SEP-08 C00008     A004
200130     2500          400              30-JUL-08 C00025     A011
200134     4200          1800             25-SEP-08 C00004     A005
```

```
UPDATE neworder
SET ord_date='15-JAN-10'
WHERE ord_amount-advance_amount<
(SELECT MIN(ord_amount) FROM orders);
```

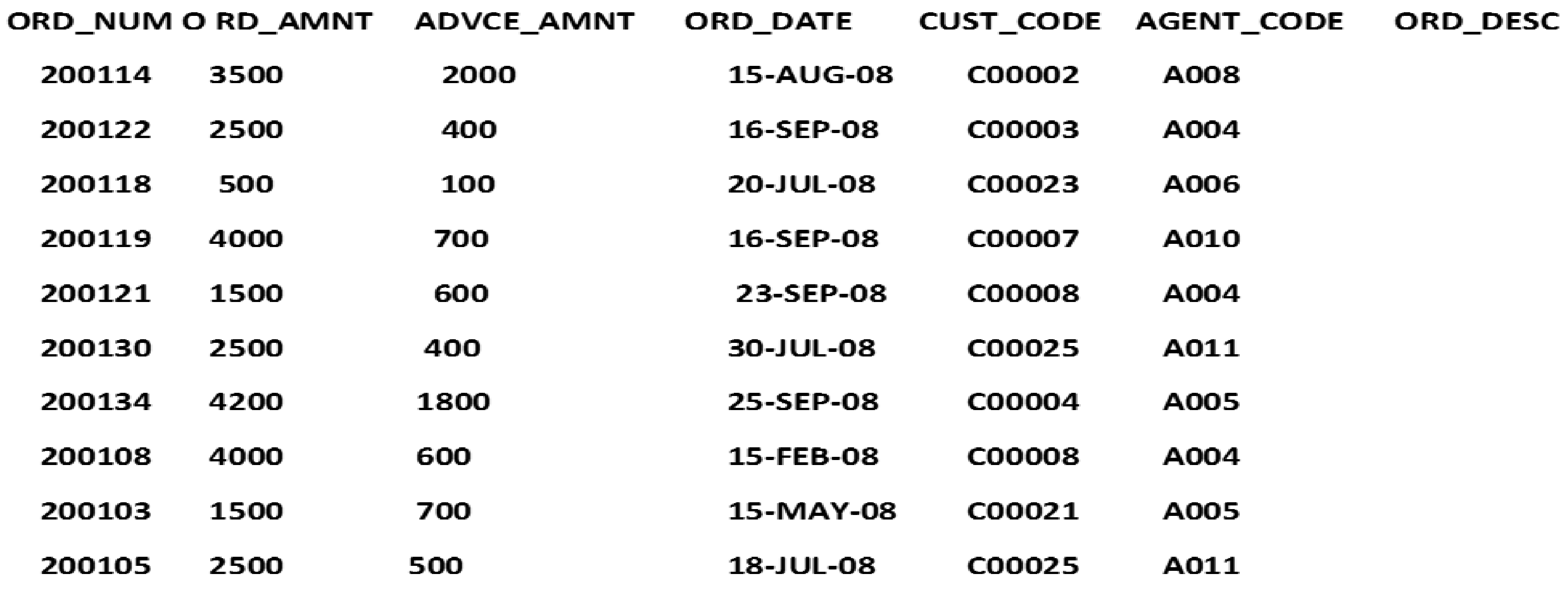
Subqueries with DELETE statement

- DELETE statement can be used with subqueries. Here are the syntax and an example of subqueries using DELETE statement.

```
DELETE FROM TABLE_NAME
[ WHERE OPERATOR [ VALUE ]
(SELECT COLUMN_NAME
FROM TABLE_NAME)
[ WHERE) ]
```

- If we want to delete those orders from 'neworder' table which advance_amount are less than the maximum advance_amount of 'orders' table, the following SQL can be used:

- Sample table : neworder



ORD_NUM	O RD_AMNT	ADVCE_AMNT	ORD_DATE	CUST_CODE	AGENT_CODE	ORD_DESC
200114	3500	2000	15-AUG-08	C00002	A008	
200122	2500	400	16-SEP-08	C00003	A004	
200118	500	100	20-JUL-08	C00023	A006	
200119	4000	700	16-SEP-08	C00007	A010	
200121	1500	600	23-SEP-08	C00008	A004	
200130	2500	400	30-JUL-08	C00025	A011	
200134	4200	1800	25-SEP-08	C00004	A005	
200108	4000	600	15-FEB-08	C00008	A004	
200103	1500	700	15-MAY-08	C00021	A005	
200105	2500	500	18-JUL-08	C00025	A011	

```
DELETE FROM neworder
WHERE advance_amount<
(SELECT MAX(advance_amount) FROM orders);
```

SQL IS NULL Operator

- A column value is NULL if it does not exist. The IS NULL operator is used to display all the rows for columns that do not have a value.
- For Example: If you want to find the names of students who do not participate in any games, the query would be as given below

```
SELECT first_name, last_name
FROM student_details
WHERE games IS NULL
```

Subquery - Example

```
USE Pubs
SELECT Pub_name
FROM Publishers
WHERE Pub_id IN
    (
    SELECT Pub_id
    FROM Titles
    WHERE Type = ‘business’
    )
```

Subqueries that Are Used with Comparison Operators

- Comparison operators that introduce a subquery can be modified with the keyword ALL or ANY.
- Subqueries introduced with a modified comparison operator return a list of zero or more values and can include a GROUP BY or HAVING clause.
- These subqueries can be restated with EXISTS.

Subqueries that Are Used with Comparison Operators

- The ALL and ANY keywords each compare a scalar value with a single-column set of values.
- The ALL keyword applies to every value, and the ANY keyword applies to at least one value.
- In the following example, the greater than (>) comparison operator is used with the ANY keyword:

Subquery - Example

```
USE Pubs
SELECT Title
FROM Titles
WHERE Advance > ANY
      (
         SELECT Advance
          FROM Publishers INNER JOIN Titles
          ON Titles.Pub_id = Publishers.Pub_id
          AND Pub_name = ‘Algodata Infosystems’
      )
```

Subqueries that Are Used with EXISTS and NOT EXISTS

- When a subquery is introduced with the keyword EXISTS, it functions as an existence test.
- The WHERE clause of the outer query tests for the existence of rows returned by the subquery.
- The subquery does not actually produce any data; instead, it returns a value of TRUE or FALSE.
- In the following example, the WHERE clause in the outer SELECT statement contains the subquery and uses the EXISTS keyword.

Subquery - Example

```
USE Pubs
SELECT Pub_name
FROM Publishers
WHERE  EXISTS
        (
        SELECT * FROM Titles
        WHERE Titles.Pub_id = Publishers.Pub_id
                AND Type = ‘Business’
         )
```

Subqueries: Guidelines

1. A subquery must be enclosed in parentheses.
2. A subquery must be placed on the right side of the comparison operator.
3. Subqueries cannot manipulate their results internally, therefore ORDER BY clause cannot be added into a subquery. You can use an ORDER BY clause in the main SELECT statement (outer query) which will be the last clause.
4. Use single-row operators with single-row subqueries.
5. If a subquery (inner query) returns a null value to the outer query, the outer query will not return any rows when using certain comparison operators in a WHERE clause.

End