BOSTON
TRAINING ACADEMY

SQL – STRUCTURE QUERY LANGUAGE

	7	Muffy

30. SQL – Indexes

Indexes are **special lookup tables** that the database search engine can use to speed up data retrieval. Simply put, an index is a pointer to data in a table. An index in a database is very similar to an index in the back of a book.

For example, if you want to reference all pages in a book that discusses a certain topic, you first refer to the index, which lists all the topics alphabetically and are then referred to one or more specific page numbers.

An index helps to speed up **SELECT** queries and **WHERE** clauses, but it slows down data input, with the **UPDATE** and the **INSERT** statements. Indexes can be created or dropped with no effect on the data.

Creating an index involves the **CREATE INDEX** statement, which allows you to name the index, to specify the table and which column or columns to index, and to indicate whether the index is in an ascending or descending order.

Indexes can also be unique, like the **UNIQUE** constraint, in that the index prevents duplicate entries in the column or combination of columns on which there is an index.

The CREATE INDEX Command

The basic syntax of a **CREATE INDEX** is as follows.

```
CREATE INDEX index_name ON table_name;
```

Single-Column Indexes

A single-column index is created based on only one table column. The basic syntax is as follows.

```
CREATE INDEX index_name
ON table_name (column_name);
```

Unique Indexes

Unique indexes are used not only for performance, but also for data integrity. A unique index does not allow any duplicate values to be inserted into the table. The basic syntax is as follows.

```
CREATE UNIQUE INDEX index_name
on table_name (column_name);
```

Composite Indexes

A composite index is an index on two or more columns of a table. Its basic syntax is as follows.

```
CREATE INDEX index_name
on table_name (column1, column2);
```

Whether to create a single-column index or a composite index, take into consideration the column(s) that you may use very frequently in a query's WHERE clause as filter conditions.

Should there be only one column used, a single-column index should be the choice. Should there be two or more columns that are frequently used in the WHERE clause as filters, the composite index would be the best choice.

Implicit Indexes

Implicit indexes are indexes that are automatically created by the database server when an object is created. Indexes are automatically created for primary key constraints and unique constraints.

The DROP INDEX Command

An index can be dropped using SQL **DROP** command. Care should be taken when dropping an index because the performance may either slow down or improve.

The basic syntax is as follows:

```
DROP INDEX index_name;
```

You can check the INDEX Constraint chapter to see some actual examples on Indexes.

When should indexes be avoided?

Although indexes are intended to enhance a database's performance, there are times when they should be avoided.

The following guidelines indicate when the use of an index should be reconsidered.

- Indexes should not be used on small tables.
- Tables that have frequent, large batch updates or insert operations.
- Indexes should not be used on columns that contain a high number of NULL values.
- Columns that are frequently manipulated should not be indexed.

SQL - INDEX Constraint

The INDEX is used to create and retrieve data from the database very quickly. Index can be created by using a single or group of columns in a table. When the index is created, it is assigned a ROWID for each row before it sorts out the data.

Proper indexes are good for performance in large databases, but you need to be careful while creating an index. Selection of fields depends on what you are using in your SQL queries.

Example

For example, the following SQL creates a new table called CUSTOMERS and adds five columns in it.

```
CREATE TABLE CUSTOMERS(
        ID      INT                 NOT NULL,
        NAME VARCHAR (20)           NOT NULL,
        AGE   INT                   NOT NULL,
        ADDRESS CHAR (25),
        SALARY    DECIMAL (18, 2),
        PRIMARY KEY (ID)
);
```

Now, you can create an index on a single or multiple columns using the syntax given below.

```
CREATE INDEX index_name
    ON table_name ( column1, column2 .....);
```

To create an INDEX on the AGE column, to optimize the search on customers for a specific age, you can use the following SQL syntax:

```
CREATE INDEX idx_age
    ON CUSTOMERS ( AGE );
```

DROP an INDEX Constraint

To drop an INDEX constraint, use the following SQL syntax.

```
ALTER TABLE CUSTOMERS
    DROP INDEX idx_age;
```

31. SQL – ALTER TABLE Command

The SQL **ALTER TABLE** command is used to add, delete or modify columns in an existing table. You should also use the ALTER TABLE command to add and drop various constraints on an existing table.

Syntax

The basic syntax of an ALTER TABLE command to add a **New Column** in an existing table is as follows.

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

The basic syntax of an ALTER TABLE command to **DROP COLUMN** in an existing table is as follows.

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

The basic syntax of an ALTER TABLE command to change the **DATA TYPE** of a column in a table is as follows.

```
ALTER TABLE table_name MODIFY COLUMN column_name datatype;
```

The basic syntax of an ALTER TABLE command to add a **NOT NULL** constraint to a column in a table is as follows.

```
ALTER TABLE table_name MODIFY column_name datatype NOT NULL;
```

The basic syntax of an ALTER TABLE command to **ADD UNIQUE CONSTRAINT** to a table is as follows.

```
ALTER TABLE table_name
ADD CONSTRAINT MyUniqueConstraint UNIQUE(column1, column2...);
```

The basic syntax of an ALTER TABLE command to **ADD CHECK CONSTRAINT** to a table is as follows.

```
ALTER TABLE table_name
ADD CONSTRAINT MyUniqueConstraint CHECK (CONDITION);
```

The basic syntax of an ALTER TABLE command to **ADD PRIMARY KEY** constraint to a table is as follows.

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

The basic syntax of an ALTER TABLE command to **DROP CONSTRAINT** from a table is as follows.

```
ALTER TABLE table_name
DROP CONSTRAINT MyUniqueConstraint;
```

If you're using MySQL, the code is as follows:

```
ALTER TABLE table_name
DROP INDEX MyUniqueConstraint;
```

The basic syntax of an ALTER TABLE command to **DROP PRIMARY KEY** constraint from a table is as follows.

```
ALTER TABLE table_name
DROP CONSTRAINT MyPrimaryKey;
```

If you're using MySQL, the code is as follows:

```
ALTER TABLE table_name
DROP PRIMARY KEY;
```

Example

Consider the CUSTOMERS table having the following records:

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

```
+----+----------+---------+----------+----------+
```

Following is the example to ADD a **New Column** to an existing table:

```
ALTER TABLE CUSTOMERS ADD SEX char(1);
```

Now, the CUSTOMERS table is changed and following would be output from the SELECT statement.

```
+----+----------+-----+-----------+----------+------+
| ID | NAME     | AGE | ADDRESS   | SALARY   | SEX  |
+----+----------+-----+-----------+----------+------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 | NULL |
|  2 | Ramesh   |  25 | Delhi     |  1500.00 | NULL |
|  3 | kaushik  |  23 | Kota      |  2000.00 | NULL |
|  4 | kaushik  |  25 | Mumbai    |  6500.00 | NULL |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 | NULL |
|  6 | Komal    |  22 | MP        |  4500.00 | NULL |
|  7 | Muffy    |  24 | Indore    | 10000.00 | NULL |
+----+----------+-----+-----------+----------+------+
```

Following is the example to DROP sex column from the existing table.

```
ALTER TABLE CUSTOMERS DROP SEX;
```

Now, the CUSTOMERS table is changed and following would be the output from the SELECT statement.

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Ramesh   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | kaushik  |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

32. SQL - TRUNCATE TABLE Command SQL

The SQL **TRUNCATE TABLE** command is used to delete complete data from an existing table.

You can also use DROP TABLE command to delete complete table but it would remove complete table structure form the database and you would need to re-create this table once again if you wish you store some data.

Syntax

The basic syntax of a **TRUNCATE TABLE** command is as follows.

```
TRUNCATE TABLE table_name;
```

Example

Consider a CUSTOMERS table having the following records:

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

Following is the example of a Truncate command.

```
SQL > TRUNCATE TABLE CUSTOMERS;
```

Now, the CUSTOMERS table is truncated and the output from SELECT statement will be as shown in the code block below:

```
SQL> SELECT * FROM CUSTOMERS;
Empty set (0.00 sec)
```

33. SQL – Using Views

A view is nothing more than a SQL statement that is stored in the database with an associated name. A view is actually a composition of a table in the form of a predefined SQL query.

A view can contain all rows of a table or select rows from a table. A view can be created from one or many tables which depends on the written SQL query to create a view.

Views, which are a type of virtual tables allow users to do the following:

- Structure data in a way that users or classes of users find natural or intuitive.
- Restrict access to the data in such a way that a user can see and (sometimes) modify exactly what they need and no more.
- Summarize data from various tables which can be used to generate reports.

Creating Views

Database views are created using the **CREATE VIEW** statement. Views can be created from a single table, multiple tables or another view.

To create a view, a user must have the appropriate system privilege according to the specific implementation.

The basic **CREATE VIEW** syntax is as follows:

```
CREATE VIEW view_name AS
SELECT column1, column2.....
FROM table_name
WHERE [condition];
```

You can include multiple tables in your SELECT statement in a similar way as you use them in a normal SQL SELECT query.

Example

Consider the CUSTOMERS table having the following records:

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad | 2000.00  |
|  2 | Khilan   |  25 | Delhi     | 1500.00  |
|  3 | kaushik  |  23 | Kota      | 2000.00  |
|  4 | Chaitali |  25 | Mumbai    | 6500.00  |
```

```
|  5 | Hardik    |  27| Bhopal    |  8500.00 |
|  6 | Komal     |  22| MP        |  4500.00 |
|  7 | Muffy     |  24| Indore    | 10000.00 |
+----+-----------+----+-----------+----------+
```

Following is an example to create a view from the CUSTOMERS table. This view would be used to have customer name and age from the CUSTOMERS table.

```
SQL > CREATE VIEW CUSTOMERS_VIEW AS
SELECT name, age
FROM CUSTOMERS;
```

Now, you can query CUSTOMERS_VIEW in a similar way as you query an actual table. Following is an example for the same.

```
SQL > SELECT * FROM CUSTOMERS_VIEW;
```

This would produce the following result.

```
+----------+-----+
| name     | age |
+----------+-----+
| Ramesh   |  32 |
| Khilan   |  25 |
| kaushik  |  23 |
| Chaitali |  25 |
| Hardik   |  27 |
| Komal    |  22 |
| Muffy    |  24 |
+----------+-----+
```

The WITH CHECK OPTION

The WITH CHECK OPTION is a CREATE VIEW statement option. The purpose of the WITH CHECK OPTION is to ensure that all UPDATE and INSERTs satisfy the condition(s) in the view definition.

If they do not satisfy the condition(s), the UPDATE or INSERT returns an error.

The following code block has an example of creating same view CUSTOMERS_VIEW with the WITH CHECK OPTION.

```
CREATE VIEW CUSTOMERS_VIEW AS
```

```
SELECT name, age
FROM CUSTOMERS
WHERE age IS NOT NULL
WITH CHECK OPTION;
```

The WITH CHECK OPTION in this case should deny the entry of any NULL values in the view's AGE column, because the view is defined by data that does not have a NULL value in the AGE column.

Updating a View

A view can be updated under certain conditions which are given below –

- The SELECT clause may not contain the keyword DISTINCT.
- The SELECT clause may not contain summary functions.
- The SELECT clause may not contain set functions.
- The SELECT clause may not contain set operators.
- The SELECT clause may not contain an ORDER BY clause.
- The FROM clause may not contain multiple tables.
- The WHERE clause may not contain subqueries.
- The query may not contain GROUP BY or HAVING.
- Calculated columns may not be updated.
- All NOT NULL columns from the base table must be included in the view in order for the INSERT query to function.

So, if a view satisfies all the above-mentioned rules then you can update that view. The following code block has an example to update the age of Ramesh.

```
SQL > UPDATE CUSTOMERS_VIEW
        SET AGE = 35
        WHERE  name='Ramesh';
```

This would ultimately update the base table CUSTOMERS and the same would reflect in the view itself. Now, try to query the base table and the SELECT statement would produce the following result.

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  35 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

Inserting Rows into a View

Rows of data can be inserted into a view. The same rules that apply to the UPDATE command also apply to the INSERT command.

Here, we cannot insert rows in the CUSTOMERS_VIEW because we have not included all the NOT NULL columns in this view, otherwise you can insert rows in a view in a similar way as you insert them in a table.

Deleting Rows into a View

Rows of data can be deleted from a view. The same rules that apply to the UPDATE and INSERT commands apply to the DELETE command.

Following is an example to delete a record having AGE = 22.

```
SQL > DELETE FROM CUSTOMERS_VIEW
      WHERE age = 22;
```

This would ultimately delete a row from the base table CUSTOMERS and the same would reflect in the view itself. Now, try to query the base table and the SELECT statement would produce the following result.

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  35 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
+----+----------+-----+-----------+----------+
```

```
|  7 |  Muffy   |  24 | Indore   |  10000.00 |
+----+----------+-----+----------+-           +
```

Dropping Views

Obviously, where you have a view, you need a way to drop the view if it is no longer needed. The syntax is very simple and is given below:

```
DROP VIEW view_name;
```

Following is an example to drop the CUSTOMERS_VIEW from the CUSTOMERS table.

```
DROP VIEW CUSTOMERS_VIEW;
```

34. SQL – Having Clause

The **HAVING Clause** enables you to specify conditions that filter which group results appear in the results.

The WHERE clause places conditions on the selected columns, whereas the HAVING clause places conditions on groups created by the GROUP BY clause.

Syntax

The following code block shows the position of the HAVING Clause in a query.

```
SELECT
FROM
WHERE
GROUP BY
HAVING
ORDER BY
```

The HAVING clause must follow the GROUP BY clause in a query and must also precede the ORDER BY clause if used. The following code block has the syntax of the SELECT statement including the HAVING clause:

```
SELECT column1, column2
FROM table1, table2
WHERE [ conditions ]
GROUP BY column1, column2
HAVING [ conditions ]
ORDER BY column1, column2
```

Example

Consider the CUSTOMERS table having the following records.

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad | 2000.00  |
|  2 | Khilan   |  25 | Delhi     | 1500.00  |
|  3 | kaushik  |  23 | Kota      | 2000.00  |
```

```
|  4 | Chaitali |  25| Mumbai    |  6500.00 |
|  5 | Hardik   |  27| Bhopal    |  8500.00 |
|  6 | Komal    |  22| MP        |  4500.00 |
|  7 | Muffy    |  24| Indore    | 10000.00 |
+----+----------+----+-----------+----------+
```

Following is an example, which would display a record for a similar age count that would be more than or equal to 2.

```
SQL > SELECT ID, NAME, AGE, ADDRESS, SALARY
FROM CUSTOMERS
GROUP BY age
HAVING COUNT(age) >= 2;
```

This would produce the following result:

```
+----+--------+-----+-----------+---------+
| ID | NAME   | AGE | ADDRESS   | SALARY  |
+----+--------+-----+-----------+---------+
|  2 | Khilan |  25 | Delhi     | 1500.00 |
+----+--------+-----+-----------+---------+
```

35. SQL – Transactions

A transaction is a unit of work that is performed against a database. Transactions are units or sequences of work accomplished in a logical order, whether in a manual fashion by a user or automatically by some sort of a database program.

A transaction is the propagation of one or more changes to the database. For example, if you are creating a record or updating a record or deleting a record from the table, then you are performing a transaction on that table. It is important to control these transactions to ensure the data integrity and to handle database errors.

Practically, you will club many SQL queries into a group and you will execute all of them together as a part of a transaction.

Properties of Transactions

Transactions have the following four standard properties, usually referred to by the acronym **ACID**.

- **Atomicity:** ensures that all operations within the work unit are completed successfully. Otherwise, the transaction is aborted at the point of failure and all the previous operations are rolled back to their former state.
- **Consistency:** ensures that the database properly changes states upon a successfully committed transaction.
- **Isolation:** enables transactions to operate independently of and transparent to each other.
- **Durability:** ensures that the result or effect of a committed transaction persists in case of a system failure.

Transaction Control

The following commands are used to control transactions.

- **COMMIT:** to save the changes.
- **ROLLBACK:** to roll back the changes.
- **SAVEPOINT:** creates points within the groups of transactions in which to ROLLBACK.
- **SET TRANSACTION:** Places a name on a transaction.

Transactional Control Commands

Transactional control commands are only used with the **DML Commands** such as – INSERT, UPDATE and DELETE only. They cannot be used while creating tables or dropping them because these operations are automatically committed in the database.

The COMMIT Command

The COMMIT command is the transactional command used to save changes invoked by a transaction to the database. The COMMIT command saves all the transactions to the database since the last COMMIT or ROLLBACK command.

The syntax for the COMMIT command is as follows.

```
COMMIT;
```

Example

Consider the CUSTOMERS table having the following records:

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

Following is an example which would delete those records from the table which have age = 25 and then COMMIT the changes in the database.

```
SQL> DELETE FROM CUSTOMERS
     WHERE AGE = 25;
SQL> COMMIT;
```

Thus, two rows from the table would be deleted and the SELECT statement would produce the following result.

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
+----+----------+-----+-----------+----------+
```

```
|    6| Komal     |    22| MP          |    4500.00 |
|    7|  Muffy    |    24| Indore      | 10000.00  |
+-------+----------+-----+-----------+----------  +
```

The ROLLBACK Command

The ROLLBACK command is the transactional command used to undo transactions that have not already been saved to the database. This command can only be used to undo transactions since the last COMMIT or ROLLBACK command was issued.

The syntax for a ROLLBACK command is as follows:

```
ROLLBACK;
```

Example

Consider the CUSTOMERS table having the following records:

```
+-------+----------+-----+-----------+---------- +
| ID | NAME       |AGE|    ADDRESS   | SALARY    |
+-------+----------+-----+-----------+---------- +
|    1 | Ramesh   |    32| Ahmedabad |    2000.00 |
|    2 |  Khilan  |    25 |  Delhi    |    1500.00 |
|    3| kaushik   |    23 |  Kota     |    2000.00 |
|    4|  Chaitali |    25 | Mumbai    |    6500.00 |
|    5| Hardik    |    27 |  Bhopal   |    8500.00 |
|    6 |  Komal   |    22 | MP        |    4500.00 |
|    7 |  Muffy   |    24| Indore     | 10000.00  |
+-------+----------+-----+-----------+---------- +
```

Following is an example, which would delete those records from the table which have the age = 25 and then ROLLBACK the changes in the database.

```
SQL> DELETE FROM CUSTOMERS
       WHERE AGE = 25;
SQL>  ROLLBACK;
```

Thus, the delete operation would not impact the table and the SELECT statement would produce the following result.

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

The SAVEPOINT Command

A SAVEPOINT is a point in a transaction when you can roll the transaction back to a certain point without rolling back the entire transaction.

The syntax for a SAVEPOINT command is as shown below.

```
SAVEPOINT SAVEPOINT_NAME;
```

This command serves only in the creation of a SAVEPOINT among all the transactional statements. The ROLLBACK command is used to undo a group of transactions.

The syntax for rolling back to a SAVEPOINT is as shown below.

```
ROLLBACK TO SAVEPOINT_NAME;
```

Following is an example where you plan to delete the three different records from the CUSTOMERS table. You want to create a SAVEPOINT before each delete, so that you can ROLLBACK to any SAVEPOINT at any time to return the appropriate data to its original state.

Example

Consider the CUSTOMERS table having the following records.

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
```

```
|  4 | Chaitali |  25| Mumbai   |  6500.00 |
|  5 | Hardik   |  27| Bhopal   |  8500.00 |
|  6 | Komal    |  22| MP       |  4500.00 |
|  7 | Muffy    |  24| Indore   | 10000.00 |
+----+----------+----+----------+----------+
```

The following code block contains the series of operations.

```
SQL> SAVEPOINT SP1;
Savepoint created.
SQL> DELETE FROM CUSTOMERS WHERE ID=1;
1 row deleted.
SQL> SAVEPOINT SP2;
Savepoint created.
SQL> DELETE FROM CUSTOMERS WHERE ID=2;
1 row deleted.
SQL> SAVEPOINT SP3;
Savepoint created.
SQL> DELETE FROM CUSTOMERS WHERE ID=3;
1 row deleted.
```

Now that the three deletions have taken place, let us assume that you have changed your mind and decided to ROLLBACK to the SAVEPOINT that you identified as SP2. Because SP2 was created after the first deletion, the last two deletions are undone:

```
SQL> ROLLBACK TO SP2;
Rollback complete.
```

Notice that only the first deletion took place since you rolled back to SP2.

```
SQL> SELECT * FROM CUSTOMERS;
+----+----------+-----+----------+----------+
| ID | NAME     |AGE| ADDRESS  | SALARY   |
+----+----------+-----+----------+----------+
|  2 | Khilan   |  25 | Delhi    |  1500.00 |
|  3 | kaushik  |  23 | Kota     |  2000.00 |
|  4 | Chaitali |  25 | Mumbai   |  6500.00 |
|  5 | Hardik   |  27 | Bhopal   |  8500.00 |
```

```
|    6 |Komal     |    22| MP          |    4500.00 |
|    7| Muffy     |    24| Indore      | 10000.00   |
+--------+----------+-----+-------------------+----------  +
6 rows selected.
```

The RELEASE SAVEPOINT Command

The RELEASE SAVEPOINT command is used to remove a SAVEPOINT that you have created.

The syntax for a RELEASE SAVEPOINT command is as follows.

```
RELEASE SAVEPOINT SAVEPOINT_NAME;
```

Once a SAVEPOINT has been released, you can no longer use the ROLLBACK command to undo transactions performed since the last SAVEPOINT.

The SET TRANSACTION Command

The SET TRANSACTION command can be used to initiate a database transaction. This command is used to specify characteristics for the transaction that follows. For example, you can specify a transaction to be read only or read write.

The syntax for a SET TRANSACTION command is as follows.

```
SET TRANSACTION [ READ WRITE | READ ONLY ];
```

36. SQL – Wildcard Operators

We have already discussed about the SQL **LIKE** operator, which is used to compare a value to similar values using the wildcard operators.

SQL supports two wildcard operators in conjunction with the LIKE operator which are explained in detail in the following table .

Wildcard Operators	Description
The percent sign (%)	Matches one or more characters. **Note:** MS Access uses the asterisk (*) wildcard character instead of the percent sign (%) wildcard character.
The underscore (_)	Matches one character. **Note:** MS Access uses a question mark (?) instead of the underscore (_) to match any one character.

The percent sign represents zero, one or multiple characters. The underscore represents a single number or a character. These symbols can be used in combinations.

Syntax

The basic syntax of a **'%'** and a **'_'** operator is as follows.

```
SELECT FROM table_name
WHERE column LIKE '%XXXX%'

or

SELECT FROM table_name
WHERE column LIKE '%XXXX%'

or

SELECT FROM table_name
WHERE column LIKE 'XXXX_'

or
```

```
SELECT FROM table_name
WHERE column LIKE '_XXXX'

or

SELECT FROM table_name
WHERE column LIKE '_XXXX_'
```

You can combine N number of conditions using the AND or the OR operators. Here, XXXX could be any numeric or string value.

Example

The following table has a number of examples showing the WHERE part having different LIKE clauses with '%' and '_' operators.

Statement	Description
WHERE SALARY LIKE '200%'	Finds any values that start with 200.
WHERE SALARY LIKE '%200%'	Finds any values that have 200 in any position.
WHERE SALARY LIKE '_00%'	Finds any values that have 00 in the second and third positions.
WHERE SALARY LIKE '2_%_%'	Finds any values that start with 2 and are at least 3 characters in length.
WHERE SALARY LIKE '%2'	Finds any values that end with 2.
WHERE SALARY LIKE '_2%3'	Finds any values that have a 2 in the second position and end with a 3.
WHERE SALARY LIKE '2___3'	Finds any values in a five-digit number that start with 2 and end with 3.

Let us take a real example, consider the CUSTOMERS table having the following records.

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

The following code block is an example, which would display all the records from the CUSTOMERS table where the SALARY starts with 200.

```
SQL> SELECT * FROM CUSTOMERS
WHERE SALARY LIKE '200%';
```

This would produce the following result.

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
+----+----------+-----+-----------+----------+
```