# The difference between TRUNCATE , DELETE and DROP is one of the most common interview question.

## TRUNCATE

- TRUNCATE is a DDL command
- TRUNCATE is executed using a table lock and whole table is locked for remove all records.
- We cannot use Where clause with TRUNCATE.
- TRUNCATE removes all rows from a table.
- Minimal logging in transaction log, so it is performance wise faster.
- TRUNCATE TABLE removes the data by deallocating the data pages used to store the table data and records only the page deallocations in the transaction log.
- Identify column is reset to its seed value if table contains any identity column.
- To use Truncate on a table you need at least ALTER permission on the table.
- Truncate uses the less transaction space than Delete statement.
- Truncate cannot be used with indexed views.

## DELETE

- DELETE is a DML command.
- DELETE is executed using a row lock, each row in the table is locked for deletion.
- We can use where clause with DELETE to filter & delete specific records.
- The DELETE command is used to remove rows from a table based on WHERE condition.
- It maintain the log, so it slower than TRUNCATE.
- The DELETE statement removes rows one at a time and records an entry in the transaction log for each deleted row.
- Identity of column keep DELETE retain the identity.
- To use Delete you need DELETE permission on the table.
- Delete uses the more transaction space than Truncate statement.
- Delete can be used with indexed views.

## DROP

- The DROP command removes a table from the database.
- All the tables' rows, indexes and privileges will also be removed.
- No DML triggers will be fired.
- The operation cannot be rolled back.
- DROP and TRUNCATE are DDL commands, whereas DELETE is a DML command.
- DELETE operations can be rolled back (undone), while DROP and TRUNCATE operations cannot be rolled back.

# Difference between Primary Key and Foreign Key

In SQL Server, there are two keys - primary key and foreign key which seems identical, but actually both are different in features and behaviours. In this article, I would like to share the key differences between primary key and foreign key.

For more information about the keys, please refer to the article Different Types of SQL Keys.

## Difference between Primary Key and Foreign Key

Primary Key

Foreign Key

Primary key uniquely identify a record in the table.

Foreign key is a field in the table that is primary key in another table.

Primary Key can't accept null values.

Foreign key can accept multiple null value.

By default, Primary key is clustered index and data in the database table is physically organized in the sequence of clustered index.

Foreign key do not automatically create an index, clustered or non-clustered. You can manually create an index on foreign key.

We can have only one Primary key in a table.

We can have more than one foreign key in a table.

## Defining Primary key and Foreign key

```
1.  --Create Parent Table
2. CREATE TABLE Department
3.  (
4.  DeptID int PRIMARY KEY, --define primary key
5.  Name varchar (50) NOT NULL,
6.  Address varchar(100) NULL
7. )
8.  GO
9.  --Create Child Table
10. CREATE TABLE Employee
11.  (
12.  EmpID int PRIMARY KEY, --define primary key
13.  Name varchar (50) NOT NULL,
14.  Salary int NULL,
```

```
15.   --define foreign key
16.   DeptID int FOREIGN KEY REFERENCES Department(DeptID)
17.   )
```

| **Primary Key** | **Foreign Key** |
|---|---|
| Primary key uniquely identify a record primary key in the table. | Foreign key is a field in the table that is in another table. |
| Primary Key can't accept null values. | Foreign key can accept multiple null value. |
| By default, Primary key is clustered index and data in the database table is physically organized in the sequence of clustered index. | Foreign key do not automatically create an index, clustered or non- clustered. You can manually clustered. You can manually create an index on foreign key |
| We can have only one Primary key in a table. | We can have more than one foreign key in a table |
| | Foreign key is a field in the table that is primary key in another table. |

# Oracle / PLSQL: Synonyms

This Oracle tutorial explains how to **create and drop synonyms** in Oracle with syntax and examples.

## Description

A **synonym** is an alternative name for objects such as tables, views, sequences, stored procedures, and other database objects.

You generally use synonyms when you are granting access to an object from another schema and you don't want the users to have to worry about knowing which schema owns the object.

## Create Synonym (or Replace)

You may wish to create a synonym so that users do not have to prefix the table name with the schema name when using the table in a query.

### Syntax

The syntax to create a synonym in Oracle is:

```
CREATE [OR REPLACE] [PUBLIC] SYNONYM [schema .] synonym_name
  FOR [schema .] object_name [@ dblink];
```

OR REPLACE

Allows you to recreate the synonym (if it already exists) without having to issue a DROP synonym command.

PUBLIC

It means that the synonym is a public synonym and is accessible to all users. Remember though that the user must first have the appropriate privileges to the object to use the synonym.

schema

The appropriate schema. If this phrase is omitted, Oracle assumes that you are referring to your own schema.

object_name

The name of the object for which you are creating the synonym. It can be one of the following:

- table
- view
- sequence
- stored procedure
- function
- package
- materialized view
- java class schema object
- user-defined object
- synonym

## Example

Let's look at an example of how to create a synonym in Oracle.

For example:

```
CREATE PUBLIC SYNONYM suppliers
FOR app.suppliers;
```

This first CREATE SYNONYM example demonstrates how to create a synonym called *suppliers*. Now, users of other schemas can reference the table called *suppliers* without having to prefix the table name with the schema named *app*. For example:

```
SELECT *
FROM suppliers;
```

If this synonym already existed and you wanted to redefine it, you could always use the *OR REPLACE* phrase as follows:

```
CREATE OR REPLACE PUBLIC SYNONYM suppliers
FOR app.suppliers;
```

# Drop synonym

Once a synonym has been created in Oracle, you might at some point need to drop the synonym.

### Syntax

The syntax to drop a synonym in Oracle is:

```
DROP [PUBLIC] SYNONYM [schema .] synonym_name [force];
```

PUBLIC

> Allows you to drop a public synonym. If you have specified *PUBLIC*, then you don't specify a *schema*.

force

> It will force Oracle to drop the synonym even if it has dependencies. It is probably not a good idea to use *force* as it can cause invalidation of Oracle objects.

### Example

Let's look at an example of how to drop a synonym in Oracle.

For example:

```
DROP PUBLIC SYNONYM suppliers;
```

This DROP statement would drop the synonym called *suppliers* that we defined earlier.

# What is subquery in SQL?

A subquery is a SQL query nested inside a larger query.

- A subquery may occur in :
  - - A SELECT clause
  - - A FROM clause
  - - A WHERE clause

- The subquery can be nested inside a SELECT, INSERT, UPDATE, or DELETE statement or inside another subquery.
- A subquery is usually added within the WHERE Clause of another SQL SELECT statement.
- You can use the comparison operators, such as >, <, or =. The comparison operator can also be a multiple-row operator, such as IN, ANY, or ALL.
- A subquery is also called an inner query or inner select, while the statement containing a subquery is also called an outer query or outer select.
- The inner query executes first before its parent query so that the results of an inner query can be passed to the outer query.

You can use a subquery in a SELECT, INSERT, DELETE, or UPDATE statement to perform the following tasks:

- Compare an expression to the result of the query.
- Determine if an expression is included in the results of the query.
- Check whether the query selects any rows.

**Syntax :**

```
SELECT    select_list
FROM      table
WHERE     expr operator
                (SELECT    select_list
                FROM       table);
```

- The subquery (inner query) executes once before the main query (outer query) executes.
- The main query (outer query) use the subquery result.

# Introduction to Transactions

A transaction is a logical unit of work that contains one or more SQL statements. A transaction is an atomic unit. The effects of all the SQL statements in a transaction can be either all committed (applied to the database) or all rolled back (undone from the database).

A transaction begins with the first executable SQL statement. A transaction ends when it is committed or rolled back, either explicitly with a `COMMIT` or `ROLLBACK` statement or implicitly when a DDL statement is issued.

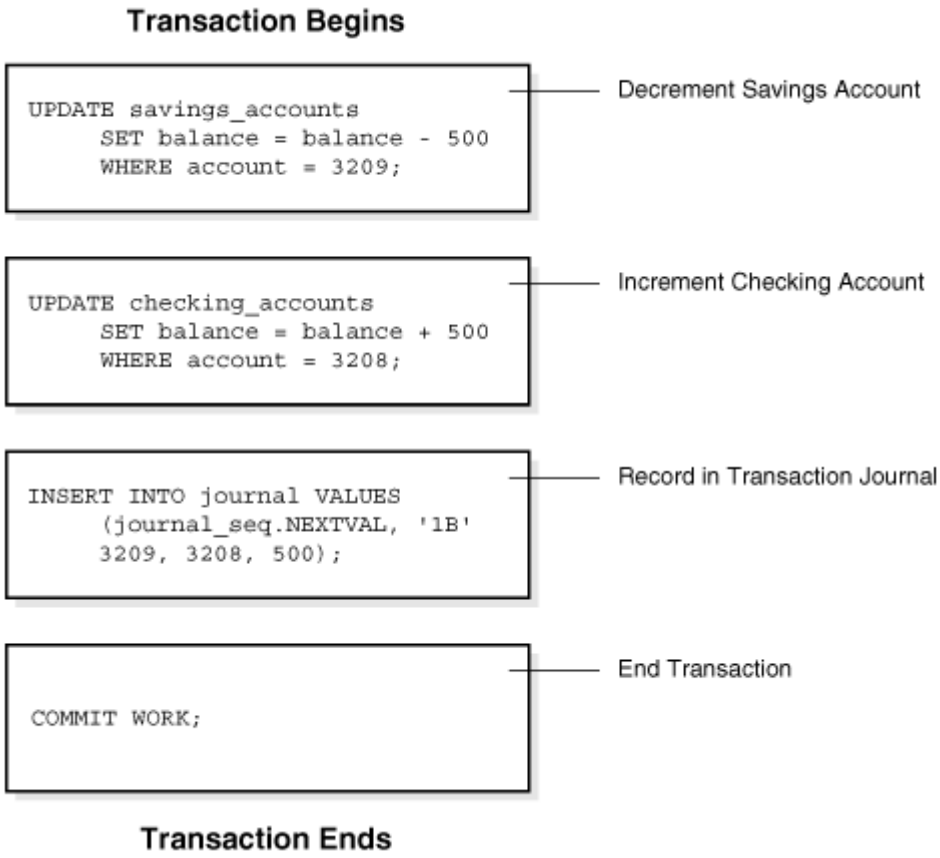
To illustrate the concept of a transaction, consider a banking database. When a bank customer transfers money from a savings account to a checking account, the transaction can consist of three separate operations:

- Decrement the savings account
- Increment the checking account
- Record the transaction in the transaction journal

Oracle must allow for two situations. If all three SQL statements can be performed to maintain the accounts in proper balance, the effects of the transaction can be applied to the database. However, if a problem such as insufficient funds, invalid account number, or a hardware failure prevents one or two of the statements in the transaction from completing, the entire transaction must be rolled back so that the balance of all accounts is correct.

Figure 4-1illustrates the banking transaction example.

Figure 4-1 A Banking 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 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;
```

# SQL GROUP Functions

Group functions are built-in SQL functions that operate on groups of rows and return one value for the entire group. These functions are: **COUNT, MAX, MIN, AVG, SUM, DISTINCT**

**SQL COUNT ():** This function returns the number of rows in the table that satisfies the condition specified in the WHERE condition. If the WHERE condition is not specified, then the query returns the total number of rows in the table.

**For Example:** If you want the number of employees in a particular department, the query would be:

```
SELECT COUNT (*) FROM employee
WHERE dept = 'Electronics';
```

The output would be '2' rows.

If you want the total number of employees in all the department, the query would take the form:

```
SELECT COUNT (*) FROM employee;
```

The output would be '5' rows.

**SQL DISTINCT():** This function is used to select the distinct rows.

**For Example:** If you want to select all distinct department names from employee table, the query would be:

```
SELECT DISTINCT dept FROM employee;
```

To get the count of employees with unique name, the query would be:

```
SELECT COUNT (DISTINCT name) FROM employee;
```

**SQL MAX():** This function is used to get the maximum value from a column.

To get the maximum salary drawn by an employee, the query would be:

```
SELECT MAX (salary) FROM employee;
```

**SQL MIN():** This function is used to get the minimum value from a column.

To get the minimum salary drawn by an employee, he query would be:

```
SELECT MIN (salary) FROM employee;
```

**SQL AVG():** This function is used to get the average value of a numeric column.

To get the average salary, the query would be

```
SELECT AVG (salary) FROM employee;
```

**SQL SUM():** This function is used to get the sum of a numeric column

To get the total salary given out to the employees,

```
SELECT SUM (salary) FROM employee;
```

# What is view ?

A database view is a virtual table or logical table which is defined as a SQL SELECT query with joins.

View is the simply subset of table which are stored logically in a database  means a view is a virtual table in the database whose contents are defined by a query.

To the database user, the view appears just like a real table, with a set of named columns and rows of data. SQL creates the illusion of the view by giving the view a name like a table name and storing the definition of the view in the database.

Views are used for security purpose in databases,views  restricts the user from viewing certain column and rows means by using view we can apply the restriction on accessing the particular rows and columns for specific user. Views display only those data which are mentioned in the query, so it shows only data which is returned by the query that is defined at the time of creation of the View.

## Advantages of views

**Security**

Each user can be given permission to access the database only through a small set of views that contain the specific data the user is authorized to see, thus restricting the user's access to stored data

**Query Simplicity**

A view can draw data from several different tables and present it as a single table, turning multi-table queries into single-table queries against the view.

**Structural simplicity**

Views can give a user a "personalized" view of the database structure, presenting the database as a set of virtual tables that make sense for that user.

**Consistency**

A view can present a consistent, unchanged image of the structure of the database, even if the underlying source tables are split, restructured, or renamed.

**Data Integrity**

If data is accessed and entered through a view, the DBMS can automatically check the data to ensure that it meets the specified integrity constraints.

**Logical data independence.**

View can make the application and database tables to a certain extent independent. If there is no view, the application must be based on a table. With the view, the program can be established in view of above, to view the program with a database table to be separated.

# Disadvantages of views

**Performance**

Views create the appearance of a table, but the DBMS must still translate queries against the view into queries against the underlying source tables. If the view is defined by a complex, multi-table query then simple queries on the views may take considerable time.

**Update restrictions**

When a user tries to update rows of a view, the DBMS must translate the request into an update on rows of the underlying source tables. This is possible for simple views, but more complex views are often restricted to read-only.

The syntax for creating a View is given below:

Hide Copy Code

```
Create View Viewname As
 Select Column1, Column2  From Tablename
 Where (Condition) Group by (Grouping Condition) having (having Condition)
```

For example:

Hide Copy Code

```
Create View View_Employeeinfo As s
       Select EmpId, EmpName, employmentdate  From EmployeeInfo
```

# What Is a Schema?

A *schema* is a collection of database objects (as far as this hour is concerned—tables) associated with one particular database username. This username is called the *schema owner*, or the owner of the related group of objects. You may have one or multiple schemas in a database. The user is only associated with the schema of the same name and often the terms will be used interchangeably. Basically, any user who creates an object has just created it in her own schema unless she specifically instructs it to be created in another one. So, based on a user's privileges within the database, the user has control over objects that are created, manipulated, and deleted. A schema can consist of a single table and has no limits to the number of objects that it may contain, unless restricted by a specific database implementation.

Say you have been issued a database username and password by the database administrator. Your username is `USER1`. Suppose you log on to the database and then create a table called `EMPLOYEE_TBL`. According to the database, your table's actual name is `USER1.EMPLOYEE_TBL`. The schema name for that table is `USER1`, which is also the owner of that table. You have just created the first table of a schema.

The good thing about schemas is that when you access a table that you own (in your own schema), you do not have to refer to the schema name. For instance, you could refer to your table as either one of the following:

```
EMPLOYEE_TBL
USER1.EMPLOYEE_TBL
```

The first option is preferred because it requires fewer keystrokes. If another user were to query one of your tables, the user would have to specify the schema, as follows:

```
USER1.EMPLOYEE_TBL
```

## A database schema can be divided broadly into two categories −

- **Physical Database Schema** − This schema pertains to the actual storage of data and its form of storage like files, indices, etc. It defines how the data will be stored in a secondary storage.
- **Logical Database Schema** − This schema defines all the logical constraints that need to be applied on the data stored. It defines tables, views, and integrity constraints.
-

Database Schema

A schema is a logical database object holder. A database schema of a database system is its structure described in a formal language supported by the database management system. The formal definition of a database schema is a set of formulas (sentences) called integrity constraints imposed on a database. These integrity constraints ensure compatibility between parts of the

schema. All constraints are expressible in the same language.
Creating schemas can be useful when objects have circular references, that is when we need to create two tables each with a foreign key referencing the other table. Different implementations treat schemas in slightly different ways.

**Syntax:**

```
CREATE SCHEMA [schema_name] [AUTHORIZATION owner_name]
[DEFAULT CHARACTER SET char_set_name]
[PATH schema_name[, ...]]
[ ANSI CREATE statements [...] ]
[ ANSI GRANT statements [...] ];
```

**Parameter:**

| Name | Description |
|---|---|
| schema_name | The name of a schema to be created. If this is omitted, the user_name is used as the schema name. |
| AUTHORIZATION owner_name | Identifies the user who is the owner of the schema. If not mentioned the current user is set as the owner. |
| DEFAULT CHARACTER SET char_set_name | Specify the default character set, used for all objects created in the schema. |
| PATH schema_name[, ...] | An optional file path and file name. |
| ANSI CREATE statements [...] | Contains one or more CREATE statements. |
| ANSI GRANT statements [...] | Contains one or more GRANT statements. |

**Examples :**

*Example-1*: As a user with authority, create a schema called STUDENT with the user STUDENT as the owner.

**SQL Code:**

```
CREATE SCHEMA STUDENT AUTHORIZATION STUDENT
```

Copy

*Example-2*: Create a schema that has an student details table. Give authority on the table to user DAVID.

**SQL Code:**

```
CREATE SCHEMA INVENTRY

     CREATE TABLE PART (IDNO  SMALLINT NOT NULL,
                        SNAME    VARCHAR(40),
                        CLASS INTEGER)

     GRANT ALL ON PART TO DAVID
```

# What is a role in a database?

A database role is a collection of any number of permissions/privileges that can be assigned to one or more users. A database role also is also given a name for that collection of privileges.

The majority of today's RDBMS's come with predefined roles that can be assigned to any user. But, a database user can also create his/her own role if he or she has the CREATE ROLE privilege.

## Advantages of Database Roles

Why are database roles needed? Well, let's go over some of the advantages of using database roles and why they would be necessary:

## Roles continue to live in database even after users are deleted/dropped

Many times a DBA (Database Administrator) has to drop user accounts for various reasons – say, for example, an employee quits the company so his/her user account is removed from the system. Now suppose that those same user accounts need to be recreated later on – just assume that same employee re-joins the company later on and needs his same account. That employee's user account probably had a lot of specific permissions assigned to it. So, when his/her account was deleted then all of those permissions were deleted as well, which creates a hassle for the DBA who has to reassign all of those permissions one by one. But, if a ***role*** was being used then all of those permissions could have just been bundled into one role – and then the process of re-instating that employee into the system would mean that the DBA simply reassigns the role to the employee. And, of course that role could also be used for other users as well. So, this is a big advantage of using a database role.

# Roles save DBA's time

Another advantage is the fact that a DBA can grant a lot of privileges with one simple command by assigning a user to a role.

# Database roles are present before users accounts are created

And finally, an advantage of database roles is that they can be used to assign a group of permissions that can be re-used for new users who belong to a specific group of people who need those permissions. For example, you may want to have a group of permissions in a role reserved just for some advanced users who know what they are doing and assign that role to a user only when a new advanced user needs that role. Or, you can have a group of privileges for users who are all working on the same project and need the same type of access.

# Disadvantages of Database Roles

The main disadvantage of using a database role is that a role may be granted to user, but that role may have more privileges than that user may actually need. This could cause a potential security issue if that user abuses his extra privileges and potentially ruins some part of the database.

An example of this is that in older versions of Oracle (before release 10.2), there is a role called CONNECT, which included privileges like CREATE TABLE, CREATE VIEW, CREATE SESSIONS, ALTER SESSION, and several other privileges. But, having all of these privileges is probably too much for a normal business user. That is probably why in newer versions of Oracle (since version 10.2), the CONNECT role has been changed so that it only has the CREATE SESSION privilege.

# How to create a database role

Most RDBMS's use the CREATE ROLE syntax to define a role. And then, the GRANT statement is used to give permissions to that database role. But, the exact details vary from one RDBMS to another so it's best to consult the documentation.

# Example of a database role

Here is an example of what creating a database role could look like:

```
CREATE ROLE advancedUsers;

GRANT UPDATE ON SOMETABLE
    TO advancedUsers;
```

# What happen before and after user Commit Fire in Oracle

The COMMIT statement ends a transaction successfully. All changes made by all SQL statements since
the transaction began are recorded permanently in the database. Before the COMMIT statement is
issued, the changes may not be visible to other transactions.
You can commit a transaction by using either of the following statements, which make the
changes permanent:
SQL> COMMIT;

## 3 things that happen Before Commit:

1. Oracle generates undo records in the undo segment buffers in the SGA. As you know, the
undo records contain the old values of the updated and deleted table rows.
2. Oracle generates redo log entries in the redo log buffers in the SGA.
3. Oracle modifies the database buffers in the SGA.

## 3 things that happen AFTER Commit:

1. The transaction tables in the redo records are tagged with the unique system change number
(SCN) of the committed transaction.
2. The log writer writes the redo log information for the transaction from the redo log buffer to
the redo log files on disk, along with the transaction's SCN. This is the point at which a commit
is considered complete in Oracle.
3. Any locks that Oracle holds are released, and Oracle marks the transaction as complete.