………………………………………………………………………………………………………………………………………………………….

**WHERE v/s HAVING Clause:**

1) Apart from SELECT queries, you can use WHERE clause with UPDATE and DELETE clause but HAVING clause can only be used with SELECT query. For example following query, which involve WHERE clause will work but other which uses HAVING clause will not work :  
  
2) WHERE clause is used for filtering rows and it applies on each and every row, while HAVING clause is used to filter groups in SQL.

3) One syntax level difference between WHERE and HAVING clause is that, former is used before GROUP BY clause, while later is used after GROUP BY clause.

4) When WHERE and HAVING clause are used together in a SELECT query with aggregate function

WHERE clause is applied first on individual rows and only rows which pass the condition is included for creating groups. Once group is created, HAVING clause is used to filter groups based upon condition specified.

That's all on difference between WHERE and HAVING clause in SQL. As I said this is very popular question and you can't **afford** not to prepare it. Always remember key difference between WHERE and HAVING clause in SQL, if WHERE and HAVING clause is used together, first WHERE clause is applied to filter rows and only after grouping HAVING clause is applied.

**Foreign Key:**

The advantage of Foreign Keys is Referential Integrity. This means that for every row in a Child table that has a foreign key, there will be a matching row in the Parent table.

The disadvantage of using Foreign Keys is the CPU cost incurred when Foreign Key lookups occur during INSERT operations. When a row is inserted into a table with a Foreign Key, the database engine has to look in the parent table to make sure there is a matching Primary Key value. This CPU cost adds latency to INSERTs.

An additional side-benefit is that it allows tools to automatically generating database diagrams with relationships inferred from the schema itself.

Think of them as a way to maintain data integrity, especially as a safeguard against orphaned records.

For example, if you had a database relating many PhoneNumber records to a Person, what happens to PhoneNumber records when the Person record is deleted for whatever reason? They will still exist in the database, but the ID of the Person they relate to will no longer exist in the relevant Person table and you have orphaned records.

Yes, you could write a trigger to delete the PhoneNumber whenever a Person gets removed, but this could get messy if you accidentally delete a Person and need to rollback.

Yes, you may remember to get rid of the PhoneNumber records manually, but what about other developers or methods you write 9 months down the line?

By creating a Foreign Key that ensures any PhoneNumber is related to an existing Person, you both insure against destroying this relationship and also add 'clues' as to the intended data structure.

## Update SQL VIEW

You can modify the definition of a SQL VIEW without dropping it by using the SQL CREATE OR REPLACE VIEW Statement.

Syntax

The syntax for the SQL CREATE OR REPLACE VIEW Statement is:

CREATE OR REPLACE VIEW view\_name AS

SELECT columns

FROM table

[WHERE conditions];

**Question:** Can you update the data in a SQL VIEW?

**Answer:** A VIEW in SQL is created by joining one or more tables. When you update record(s) in a view, it updates the records in the underlying tables that make up the SQL View.

So, yes, you can update the data in a SQL VIEW providing you have the proper privileges to the underlying SQL tables.

**Question:** Does the SQL View exist if the table is dropped from the database?

**Answer:** Yes, in Oracle, the SQL VIEW continues to exist even after one of the tables (that the SQL VIEW is based on) is dropped from the database. However, if you try to query the SQL VIEW after the [table has been dropped](https://www.techonthenet.com/sql/tables/drop_table.php), you will receive a message indicating that the SQL VIEW has errors.

If you [recreate the table](https://www.techonthenet.com/sql/tables/create_table.php) (the table that you had dropped), the SQL VIEW will again be fine

## Create views and stored procedures

A view is a stored SELECT statement, and a stored procedure is one or more Transact-SQL statements that execute as a batch.

Views are queried like tables and do not accept parameters. Stored procedures are more complex than views. Stored procedures can have both input and output parameters and can contain statements to control the flow of the code, such as IF and WHILE statements. It is good programming practice to use stored procedures for all repetitive actions in the database.

For this example, you will use CREATE VIEW to create a view that selects only two of the columns in the **Products** table. Then, you will use CREATE PROCEDURE to create a stored procedure that accepts a price parameter and returns only those products that cost less than the specified parameter value.

##### **So, in short, We Need Views in SQL Server**

1. To protect the data. If we have a table containing sensitive data in certain columns, we might wish to hide those columns from certain groups of users. For instance, customer names, addresses, and their social security numbers might all be stored in the same table; however, for lower level employees like shipping clerks, you can create a view that only displays customer name and address. You can grant permissions to a view without allowing users to query the original tables.
2. A view is a logical table but what it stores internally is a select statement that is used for creating the view. So that whenever a user performs an operation on the view like select, insert, update or delete internally the view performs those operations on a table.
3. Simply we can say that view will act as an interface between the data provider (Table) and the User.
4. The view is created based on a table any changes that are performed on the table reflects into the view any changes performed on the view reflect on the table also.

* Encrypt sensitive data.
* Access the database using an account with the least privileges necessary.
* Install the database using an account with the least privileges necessary.
* Ensure that data is valid.
* Do a code review to check for the possibility of second-order attacks.
* Use parameterised queries.
* Use stored procedures.
* Re-validate data in stored procedures.
* Ensure that error messages give nothing away about the internal architecture of the application or the database.

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.

============================

ALTER Procedure [dbo].[DeleteStudentTransaction]@Id

INT AS

BEGIN TRY

BEGIN TRANSACTION

DELETE FROM Student WHERE Id=@Id

-- some other codes

COMMIT

END TRY

BEGIN CATCH

ROLLBACK

END CATCH

### A. Using an explicit transaction

**APPLIES TO:** SQL Server (starting with 2008), Azure SQL Database, Azure SQL Data Warehouse, Parallel Data Warehouse

This example uses AdventureWorks.

BEGIN TRANSACTION;

DELETE FROM HumanResources.JobCandidate

WHERE JobCandidateID = 13;

COMMIT;

### B. Rolling back a transaction

**APPLIES TO:** SQL Server (starting with 2008), Azure SQL Database, Azure SQL Data Warehouse, Parallel Data Warehouse

The following example shows the effect of rolling back a transaction. In this example, the ROLLBACK statement will roll back the INSERT statement, but the created table will still exist.

CREATE TABLE ValueTable (id int);

BEGIN TRANSACTION;

INSERT INTO ValueTable VALUES(1);

INSERT INTO ValueTable VALUES(2);

ROLLBACK;

### C. Naming a transaction

**APPLIES TO:** SQL Server (starting with 2008), Azure SQL Database

The following example shows how to name a transaction.

DECLARE @TranName VARCHAR(20);

SELECT @TranName = 'MyTransaction';

BEGIN TRANSACTION @TranName;

USE AdventureWorks2012;

DELETE FROM AdventureWorks2012.HumanResources.JobCandidate

WHERE JobCandidateID = 13;

COMMIT TRANSACTION @TranName;

GO

### D. Marking a transaction

**APPLIES TO:** SQL Server (starting with 2008), Azure SQL Database

The following example shows how to mark a transaction. The transaction CandidateDelete is marked.

BEGIN TRANSACTION CandidateDelete

WITH MARK N'Deleting a Job Candidate';

GO

USE AdventureWorks2012;

GO

DELETE FROM AdventureWorks2012.HumanResources.JobCandidate

WHERE JobCandidateID = 13;

GO

COMMIT TRANSACTION CandidateDelete;

GO

## Nested Sub-queries versus Correlated Sub-queries:

With a normal nested sub-query, the inner SELECT query runs first and executes once, returning values to be used by the main query. A correlated sub-query, however, executes once for each candidate row considered by the outer query. In other words, the inner query is driven by the outer query.