SQL

1. [Query to delete duplicate records](#_Query_to_delete)
2. [Magic Tables](#_Magic_Tables)
3. Isolation Levels
4. RANK and DENSE\_RANK

# Query to delete duplicate records

DELETE FROM [SampleDB].[dbo].[Employee]

    WHERE ID NOT IN

    (

        SELECT MAX(ID) AS MaxRecordID

        FROM [SampleDB].[dbo].[Employee]

        GROUP BY [FirstName],

                 [LastName],

                 [Country]

    );

# Magic Tables

There are Magic Tables (virtual tables) in SQL Server that hold the temporal information of recently inserted and recently deleted data in the virtual table. INSERTED and DELETED are two types of magic tables in SQL Server. An INSERTED magic table is populated with INSERT and UPDATE operations and DELETED magic table is populated with UPDATE and DELETE operations.

The INSERTED magic table stores the before version of the row, and the DELETED table stores the after version of the row for any INSERT, UPDATE, or DELETE operations.

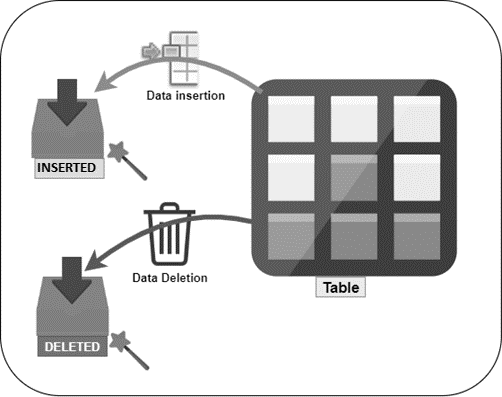
A magic table can be utilized in INSERT, UPDATE, and DELETE activity with the table in a trigger, which is the common understanding of people. SQL Server uses magic tables outside the TRIGGER also for a lot of other purposes too. Use of Magic tables in SQL Server with the usual update statement lessens the information dependency and makes your information consistent with your transaction.

A magic table is stored in the temp DB.

**INSERT**: The INSERTED magic table will have the newly inserted rows on the top in the table with an insert operation. It can be used to manage an audit of the table to another history table.

**DELETE**: The DELETED magic table will have the recently deleted rows on the top in the table with a delete operation. It can be used to manage a previous version of the row for the audit purpose in the history table.

**UPDATE**: Both INSERTED and DELETED virtual tables will be part of an update statement. Update statement returns deleted magic table with the previous version of the row and the inserted magic table with the new version of a row, which is going to be replaced or updated with the earlier values in the table. The important thing is that whenever users perform the update statement inside the trigger or outside the trigger, INSERTED and DELETED magic tables are being used.



A common use of Magic tables in SQL Server is the DML (Data Manipulation Language) trigger. SQL Server DML trigger allows using these two virtual tables INSERTED and DELETED. The ideal use of the trigger is auditing and managing a before and after version of the table row on INSERT, UPDATE, or DELETE operation within the transaction statement. Even users can write data manipulation logic as well with these magic tables inside the trigger.

# SQL ISOLATIONS

**References:**<https://gavindraper.com/2012/02/18/sql-server-isolation-levels-by-example/>

1)      **Read Uncommitted:** One transaction can read the uncommitted data (dirty read) while other transaction still updating the data.

Query1

BEGIN TRAN

UPDATE IsolationTests SET Col1 = 2

--Simulate having some intensive processing here with a wait

WAITFOR DELAY '00:00:10'

ROLLBACK

Query2

SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED

SELECT \* FROM IsolationTests

OR

2)      **Read Committed (Default):**This is the default isolation level and means selects will only return committed data. Select statements will issue shared lock requests against data you’re querying this causes you to wait if another transaction already has an exclusive lock on that data. Once you have your shared lock any other transactions trying to modify that data will request an exclusive lock and be made to wait until your Read Committed transaction finishes.

Query1

BEGIN TRAN

UPDATE IsolationTests SET Col1 = 2

--Simulate having some intensive processing here with a wait

WAITFOR DELAY '00:00:10'

ROLLBACK

Query2

SELECT \* FROM IsolationTests

**3)      Repeatable Read:**

This is similar to Read Committed but with the additional guarantee that if you issue the same select twice in a transaction you will get the same results both times. It does this by holding on to the shared locks it obtains on the records it reads until the end of the transaction, this means any transactions that try to modify these records are forced to wait for the read transaction to complete.

As before run Query1 then while its running run Query2.

Query1

SET TRANSACTION ISOLATION LEVEL REPEATABLE READ

BEGIN TRAN

SELECT \* FROM IsolationTests

WAITFOR DELAY '00:00:10'

SELECT \* FROM IsolationTests

ROLLBACK

Query2

UPDATE IsolationTests SET Col1 = -1

**4)      Serializable:**

This isolation level takes Repeatable Read and adds the guarantee that no new data will be added eradicating the chance of getting Phantom Reads. It does this by placing range locks on the queried data. This causes any other transactions trying to modify or insert data touched on by this transaction to wait until it has finished.

You know the drill by now run these queries side by side…

Query1

5)

SET TRANSACTION ISOLATION LEVEL SERIALIZABLE

BEGIN TRAN

SELECT \* FROM IsolationTests

WAITFOR DELAY '00:00:10'

SELECT \* FROM IsolationTests

ROLLBACK

Query2

6)

INSERT INTO IsolationTests(Col1,Col2,Col3)

VALUES (100,100,100)

You’ll see that the insert in Query2 waits for Query1 to complete before it runs eradicating the chance of a phantom read. If you change the isolation level in Query1 to repeatable read, you’ll see the insert no longer gets blocked and the two select statements in Query1 return a different amount of rows.

**5) Snapshot:**

This provides the same guarantees as serializable. So what's the difference? Well it’s more in the way it works, using snapshot doesn't block other queries from inserting or updating the data touched by the snapshot transaction. Instead row versioning is used so when data is changed the old version is kept in tempdb so existing transactions will see the version without the change. When all transactions that started before the changes are complete the previous row version is removed from tempdb. This means that even if another transaction has made changes you will always get the same results as you did the first time in that transaction.

So on the plus side you’re not blocking anyone else from modifying the data whilst you run your transaction but…. You’re using extra resources on the SQL Server to hold multiple versions of your changes.

To use the snapshot isolation level, you need to enable it on the database by running the following command

ALTER DATABASE IsolationTests

SET ALLOW\_SNAPSHOT\_ISOLATION ON

If you rerun the examples from serializable but change the isolation level to snapshot you will notice that you still get the same data returned but Query2 no longer waits for Query1 to complete.

# RANK and DENSE\_RANK

**RANK()** will assign the same number for the row which contains the same value and skips the next number.

**DENSE\_RANK ()** will assign the same number for the row which contains the same value without skipping the next number.

#### Example:

Our database has a table named **sales\_assistant** with data in the following columns: id (primary key), first\_name, last\_name, month, and sold products.

| **id** | **first\_name** | **last\_name** | **month** | **sold products** |
| --- | --- | --- | --- | --- |
| 1 | Lisa | Black | 5 | 2300 |
| 2 | Mary | Jacobs | 5 | 2400 |
| 3 | Lisa | Black | 6 | 2700 |
| 4 | Mary | Jacobs | 6 | 2700 |
| 5 | Alex | Smith | 6 | 2900 |
| 6 | Mary | Jacobs | 7 | 1200 |
| 7 | Lisa | Black | 7 | 1200 |
| 8 | Alex | Smith | 7 | 1000 |

Let’s display each sales assistant’s first and last name and number of sold products. We also want to rank them in terms of the number of products sold in descending order.

#### Solution 1:

|  |
| --- |
| SELECT RANK() OVER(ORDER BY sold products DESC) AS r,    DENSE\_RANK() OVER(ORDER BY sold products DESC) AS dr,    first\_name,    last\_name,    month,    sold products  FROM sales\_assistant; |

This query returns two rankings: one produced by RANK and another by DENSE\_RANK. What’s the difference?

Simply put, RANK skips the number of positions after records with the same rank number. The ranking RANK\_DENSE returns position numbers from 1 to 6 because it doesn’t skip records with the same rank number:

| **r** | **dr** | **first\_name** | **last\_name** | **month** | **sold products** |
| --- | --- | --- | --- | --- | --- |
| 1 | 1 | Alex | Smith | 6 | 2900 |
| 2 | 2 | Lisa | Black | 6 | 2700 |
| 2 | 2 | Mary | Jacobs | 6 | 2700 |
| 4 | 3 | Mary | Jacobs | 5 | 2400 |
| 5 | 4 | Lisa | Black | 5 | 2300 |
| 6 | 5 | Mary | Jacobs | 7 | 1200 |
| 6 | 5 | Lisa | Black | 7 | 1200 |
| 8 | 6 | Alex | Smith | 7 | 1000 |

#### Discussion:

If you’d like to rank rows in the result set, SQL offers the RANK() and DENSE\_RANK functions. These functions are used in SELECT with others columns. After RANK or DENSE\_RANK, we call the OVER() function, which takes an ORDER BY clause with the name of the column to sort before assigning a ranking.

Unlike DENSE\_RANK, RANK skips positions after equal rankings. The number of positions skipped depends on how many rows had an identical ranking. For example, Mary and Lisa sold the same number of products and are both ranked as #2. With RANK, the next position is #4; with DENSE\_RANK, the next position is #3.

Both RANK and RANK\_DENSE work on partitions of data:

#### Solution 1:

|  |
| --- |
| SELECT RANK() OVER(PARTITION BY month ORDER BY sold products DESC) AS r,    DENSE\_RANK() OVER(PARTITION BY month ORDER BY sold products DESC) AS dr,    first\_name,    last\_name,    month,    sold products  FROM sales\_assistant; |

| **r** | **dr** | **first\_name** | **last\_name** | **month** | **sold products** |
| --- | --- | --- | --- | --- | --- |
| 1 | 1 | Mary | Jacobs | 5 | 2400 |
| 2 | 2 | Lisa | Black | 5 | 2300 |
| 1 | 1 | Alex | Smith | 6 | 2900 |
| 2 | 2 | Lisa | Black | 6 | 2700 |
| 2 | 2 | Mary | Jacobs | 6 | 2700 |
| 1 | 1 | Mary | Jacobs | 7 | 1200 |
| 1 | 1 | Lisa | Black | 7 | 1200 |
| 3 | 2 | Alex | Smith | 7 | 1000 |

You can split records into groups according to a given column (in our example, month). In this situation, records are ranked as part of a partition: