From Chaos to Clarity: A Journey Through Random SQL Code: Practical Examples for Developers

In SQL development, we often encounter recurring challenges that can be addressed with small, reusable pieces of code. Here, I've put together a few random yet handy SQL snippets that tackle common scenarios—from formatting output to managing permissions. Whether you're a beginner or seasoned database professional, these examples could help simplify your daily SQL tasks.

Generating Sequential Numbers Without a Table

This snippet creates a sequence of numbers without requiring a pre-existing table of numbers. It's helpful when you need to generate a set range quickly.

```
with numbers as
(
select 1 as number
  union all
select number + 1
from numbers
where number < 100
)
select number
from numbers
option (maxrecursion 100);
```

Pivot Data for Better Analysis

Pivoting is a great way to turn rows into columns, making certain analyses easier. An example of how you can use a PIVOT query in SQL to transform sales data by month into a pivot table. Basically, you want to create a pivot table showing total sales per month for each year.

Sample Input Data

SaleID	SaleAmount	SaleMonth	Year
1	1000	Jan	2024
2	1500	Feb	2024
3	2000	Mar	20

The Query

```
select [year],
isnull([jan], 0) as [jan],
isnull([feb], 0) as [feb],
isnull([mar], 0) as [mar],
isnull([apr], 0) as [apr],
isnull([may], 0) as [may],
isnull([jun], 0) as [jun],
isnull([jul], 0) as [jul],
isnull([aug], 0) as [aug],
isnull([sep], 0) as [sep],
isnull([oct], 0) as [oct],
isnull([nov], 0) as [nov],
isnull([dec], 0) as [dec]
from
(
  select
  [saleamount],
  [salemonth],
  [year]
  from [dbo].[sales]
) as [sourcetable]
PIVOT
```

```
        (
          sum(saleamount)
          for [salemonth] in
          (
            [jan],
            [feb],
            [mar],
            [apr],
            [may],
            [jun],
            [jul],
            [aug],
            [sep],
            [oct],
            [nov],
            [dec]
          )
        ) as [pivottable]
        order by [year]
```

Explanation

The PIVOT function sums the sales for each month. If there were no sales in a month, then that cell returns 0 (isnull).

SourceTable is a derived table containing SaleAmount, SaleMonth, and Year columns from Sales.

PIVOT is used to aggregate SaleAmount by SaleMonth.

SUM(SaleAmount) computes the total sales amount for each month.

FOR SaleMonth IN (...) specifies the months as columns in the pivot table.

ISNULL([Month], 0) ensures that months without sales data show up as 0 rather than NULL.

<u>Sample Output</u>

Year	**Jan**	**Feb**	**Mar**	**Apr**	**May**	**Jun**	**Jul**	**Aug**	**Sep**	**Oct**	**Nov**	**Dec**
2024	1000	1500	2000	0	0	0	0	0	0	0	0	0

Check for Table Existence Before Creating

```
if not exists (select * from information_schema.tables where table_name = 'newtable')
begin
  set ansi_nulls on
  set quoted_identifier on

  create table [dbo].[newtable]
  (
    [pkid] uniqueidentifier primary key, -- unique identifier which serves as primary key for table
    [newcolumn] nvarchar(50)
    )
end
```

Create Table with Unique Identifier, Primary Key, and Clustered Index

```
if not exists (select * from information_schema.tables where table_name = 'the_table_to_create')
begin
  set ansi_nulls on
  set quoted_identifier on

  create table [dbo].[the_table_to_create]
  (
    [pkid] [uniqueidentifier] not null,
    [int_field] [int] not null,
    [varachar_field] [varchar](20) not null,
    [bit_field] [bit] not null,
    constraint [pk_settings] primary key clustered
    (
      [pkid] asc
    ) with (pad_index = off, statistics_norecompute = off, ignore_dup_key = off, allow_row_locks = on,
allow_page_locks = on) on [primary]
  ) on [primary]

  -- insert records into the new table
  insert into [the_table_to_create] values (newid(), 123, 'ABC', 1)
  insert into [the_table_to_create] values (newid(), 456, 'DEF', 0)
end
```

Explanation

This block creates a new table called "the_table_to_create" with the following columns:

pkid: A unique identifier (GUID) as the primary key.

int_field: An integer field.

varachar_field: A variable-length character field of up to 20 characters.

bit_field: A bit field for storing boolean values (0 or 1).

It also defines a clustered primary key constraint on the "pkid" column, with specific index options (e.g., allowing row and page locks).

Queries Representing Different Time/Date

These types of date-calculation SQL queries are often essential in scenarios where specific time-based data segmentation is required. They are especially useful in financial, reporting, and data warehousing contexts.

```
-- date four days ago
select dateadd(day,-4,getdate())

-- yesterday
select (dateadd(dd, datediff(dd,0,getdate()), 0)-1)

-- first day of current month
select dateadd(mm, datediff(mm,0,getdate()), 0)

-- first day of the year
select dateadd(yy, datediff(yy,0,getdate()), 0)

-- monday of the current week
select dateadd(wk, datediff(wk,0,getdate()), 0)

-- first day of the quarter
select dateadd(qq, datediff(qq,0,getdate()), 0)
```

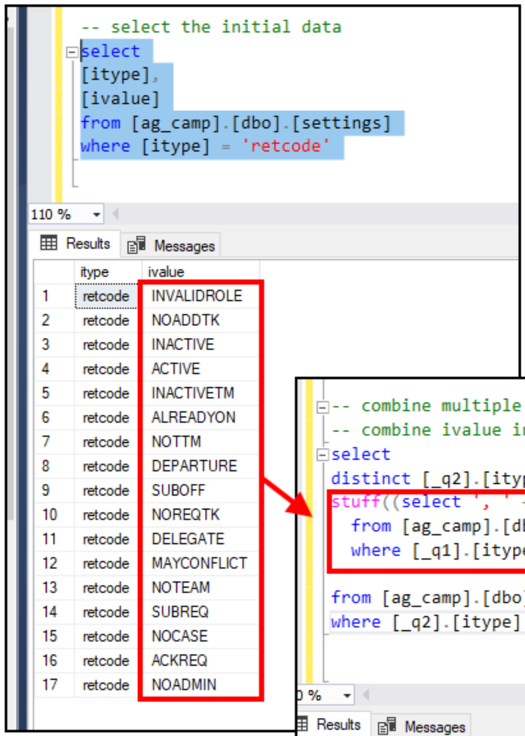
```
-- midnight for the current day
select dateadd(dd, datediff(dd,0,getdate()), 0)

-- last day of prior month
select dateadd(ms,-3,dateadd(mm, datediff(mm,0,getdate()  ), 0))

-- last day of prior year
select dateadd(ms,-3,dateadd(yy, datediff(yy,0,getdate()  ), 0))

-- last day of current month
select dateadd(ms,-3,dateadd(mm, datediff(m,0,getdate()  )+1, 0))

-- last day of current year
select dateadd(ms,-3,dateadd(yy, datediff(yy,0,getdate()  )+1, 0))

-- first monday of the month
select dateadd(wk, datediff(wk,0, dateadd(dd,6-datepart(day,getdate()),getdate())), 0)

-- for only the date (no time), stuff it all in a convert with a target type of date
select convert(date, (dateadd(dd, datediff(dd,0,getdate()), 0)+1)) as [tomorrow_date]

-- to show as month-day-year, stuff it all in a format with 'MM-dd-yyyy'
select format(convert(date, (dateadd(dd, datediff(dd, 0, getdate()), 0) +1)), 'MM-dd-yyyy') as [tomorrow_date]
```

Combine Multiple Rows into One Combined Result Set using FOR XML PATH

Combining multiple rows into one combined result set is useful in scenarios where you need to consolidate data into a single view for clarity, reporting, or further processing, such as aggregated reporting and summaries, displaying related data in a single row, reducing redundant rows in BI and data warehousing, and data transformation for exporting.

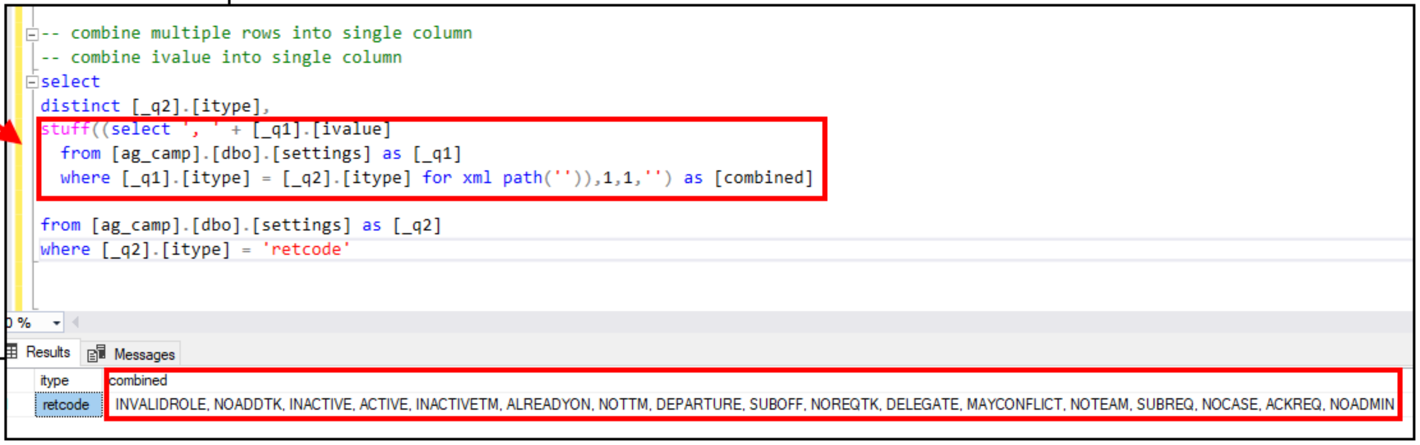
```
-- select the initial data
select [itype], [ivalue]
from [ag_camp].[dbo].[settings]
where [itype] = 'retcode'

-- combine multiple rows into single column
-- combine ivalue into single column
select
```

```
distinct [_q2].[itype], stuff((select ', ' + [_q1].[ivalue]
from [ag_camp].[dbo].[settings] as [_q1]
where [_q1].[itype] = [_q2].[itype] for xml path('')),1,1,'') as [combined]
from [ag_camp].[dbo].[settings] as [_q2]
where [_q2].[itype] = 'retcode'
```



```
-- select the initial data
select
[itype],
[ivalue]
from [ag_camp].[dbo].[settings]
where [itype] = 'retcode'
```

	itype	ivalue
1	retcode	INVALIDROLE
2	retcode	NOADDTK
3	retcode	INACTIVE
4	retcode	ACTIVE
5	retcode	INACTIVETM
6	retcode	ALREADYON
7	retcode	NOTTM
8	retcode	DEPARTURE
9	retcode	SUBOFF
10	retcode	NOREQTK
11	retcode	DELEGATE
12	retcode	MAYCONFLICT
13	retcode	NOTEAM
14	retcode	SUBREQ
15	retcode	NOCASE
16	retcode	ACKREQ
17	retcode	NOADMIN



```
-- combine multiple rows into single column
-- combine ivalue into single column
select
distinct [_q2].[itype],
stuff((select ', ' + [_q1].[ivalue]
  from [ag_camp].[dbo].[settings] as [_q1]
  where [_q1].[itype] = [_q2].[itype] for xml path('')),1,1,'') as [combined]

from [ag_camp].[dbo].[settings] as [_q2]
where [_q2].[itype] = 'retcode'
```

itype	combined
retcode	INVALIDROLE, NOADDTK, INACTIVE, ACTIVE, INACTIVETM, ALREADYON, NOTTM, DEPARTURE, SUBOFF, NOREQTK, DELEGATE, MAYCONFLICT, NOTEAM, SUBREQ, NOCASE, ACKREQ, NOADMIN

Common Table Expression

Common Table Expressions (CTEs) are versatile tools in SQL that can improve query organization, readability, and efficiency across a variety of scenarios such as, simplifying complex queries, improving readability, encapsulating reusable logic, handling aggregate functions, etc.

Create a CTE to calculate the average sales per product from a Sales table.

Sample Data

ProductID	SaleAmount	SaleDate
1	100	11/1/2024
1	150	11/2/2024
2	200	11/1/2024
2	300	11/2/2024
3	250	11/1/2024
3	300	11/2/2024

Query the Sales table using CTE

```
-- the CTE query
with [averagesales_cte] as
(
  select
  [productid],
  avg([saleamount]) as [avgsaleamount]
  from [dbo].[sales]
  group by [productid]
)

-- selecting from the CTE table
select
[productid],
[avgsaleamount]
from averagesales_cte
```

This simple CTE query computes the average sales for each product and presents the results in a clear manner. CTEs are particularly useful for organizing complex queries, making them easier to read and maintain.

Sample Output

ProductID	AvgSaleAmount
1	125
2	250
3	275

Explanation

CTE Definition: The WITH clause defines a CTE called AverageSales. Within the CTE, we select the ProductID and calculate the average sale amount for each product by using the AVG function. We group the results by ProductID to ensure we get the average for each individual product.

Main Query: After defining the CTE, we select the ProductID and AvgSaleAmount from the CTE in the main query.

SQL Table Variable

A SQL table variable is a type of variable used to store a set of rows and columns in memory. It functions like a temporary table but is scoped to the batch, stored procedure, or function in which it is declared. Table variables allow you to hold intermediate results for processing within your queries, making them useful for situations where you need to perform calculations or manipulations on small to moderate-sized datasets.

Calculate a bonus for employees based on their current salary and department from an Employees table.

Sample Employees Table

EmployeeID	EmployeeName	Salary	DepartmentID
1	John Doe	50000	1
2	Jane Smith	60000	2
3	Sam Johnson	70000	1
4	Lucy Brown	80000	3

The Query

Query the Employees table into a table variable, calculate the bonus, and update Employees table.

```
-- declare a table variable to hold the bonus calculations
declare @employeebonuses table
(
  employeeid int,
  bonusamount decimal(10, 2)
)


-- insert calculated bonuses into the table variable from the employees table
insert into @employeebonuses
select [employeeid], [salary * 0.10] as [bonusamount]
from [dbo].[employees]
where [departmentid] in (1, 2);  -- calculate bonuses only for departments 1 and 2

-- update the original employees table with the bonuses
update [emp]
set [emp].[salary] = [emp].[salary] + [bonusinfo].[bonusamount]
from [dbo].[employees] as [emp]
join @employeebonuses as [bonusinfo] on [emp].[employeeid] = [bonusinfo].[employeeid]

-- select to verify updated results
select * from [dbo].[employees]
```

Explanation

Declare Table Variable: The @EmployeeBonuses table variable is declared with two columns: EmployeeID and BonusAmount.

Inserting Data: The INSERT INTO statement populates the table variable with the EmployeeID and calculated BonusAmount (10% of the current salary) for employees in departments 1 and 2.

Updating Original Table: The UPDATE statement updates the Salary column of the Employees table by adding the calculated bonus from the @EmployeeBonuses table variable. This is done using a JOIN to match the correct employees.

Verifying Results: The final SELECT statement retrieves the updated rows from the Employees table to verify that the salaries have been adjusted accordingly.

When to Use a Table Variable

Small to Moderate Data Sets: When you need to store intermediate results for a small number of records. Table variables are best for datasets with fewer than 1,000 rows, as they can perform better than temporary tables in such cases.

Limited Scope: When you need a temporary storage solution that only exists within the scope of the batch, stored procedure, or function. There is no need to explicitly drop the variable as it is automatically cleaned up at the end of its scope.

Benefits of Using Table Variables

Performance: For smaller datasets, table variables can offer better performance because they incur less overhead than temporary tables. They are stored in memory and don’t require disk space unless they exceed memory limits.

Ease of Use: Table variables are simple to declare and use, which can lead to clearer, more maintainable code. You don't have to manage their lifecycle explicitly - no need to drop them. Once the batch or procedure finishes execution, the table variable is automatically dropped, reducing the risk of lingering temporary data in your database.

Less Blocking: Since they are scoped to the batch, table variables can result in less locking/blocking in highly concurrent environments.

Conclusion

Using a table variable in SQL is good for temporary data storage of small(ish) datasets. They provide a straightforward way to perform intermediate calculations or manipulations without the overhead and complexity of managing temporary tables.

Insert a Record into an Existing Table From a Different Table With Identity Column Defined

[source_table] contains the data you want to insert.

[target_table] has an identity column (pkid) and other columns to receive data from [source_table].

Note: While it is generally advisable to let SQL Server automatically manage identity values, this script is specifically designed for situations where I need to restore a single record from the source table to the target table while preserving the [pkid] identity value. It is important to explicitly list the columns of both the target and source tables when performing an insert with an identity column.

```
-- allow explicit values to be inserted into the identity column
set identity_insert [target_table] on;

-- insert data into [dbo].[target_table]
-- from [dbo].[source_table]
insert into [dbo].[target_table] ([pkid], [name], [age], [status])
  select
  [source].[pkid],
  [source].[name],
  [source].[age],
  [source].[status]
  from [dbo].[source_table] as [source]

-- turn off explicit value insert
set identity_insert [target_table] off;
```

The Case Expression

A CASE expression is a conditional statement in SQL that enables you to perform CRUD operations based on specific conditions, making it valuable for creating dynamic, conditional logic within a single query instead of multiple separate queries.

Selecting data using the select case

Query the Employees table. If [performancerating] is 'Excellent' then show 'Well Done' else show 'Pretty Good'

```
select
[emp].[fname],
[emp].[lname],
[emp].[salary],
case when [emp].[performancerating] = 'Excellent'
  then 'Well Done'
  else 'Pretty Good'
end as [rating]
from [dbo].[employees] as [emp]
where [departmentid] in (1, 2);  -- only for departments 1 and 2
```

Updating data using the select case

Update the [salary] column value based on [performancerating]. If 'Excellent' then update the [salary] value with the existing salary value plus 1000, else update the [salary] value with the existing salary value plus 500

```
update [dbo].[employees]
set [salary] = [salary] +
case
  when [performancerating] = 'Excellent' then 1000
  when [performancerating] = 'Good' then 500
  else 100
end
```

Use a JOIN To Update a Column in One Table with a Value From a Different Table

Update the Employees.DepartmentName column value based on the DepartmentName value in Departments.DepartmentName table based on specific criteria such as EmployeeId.

Sample Code

```
update [emp]
set [emp].[departmentname] = [dept].[departmentname]
from [dbo].[employees] as [emp]
join [departments] as [dept] on [emp].[departmentid] = [dept].[departmentid]
where [emp].[employeeid] = 1234 -- example condition
```

This query updates DepartmentName in the Employees table with the value from the Departments table where there is a matching DepartmentID.

Top-N Per Group

Find the top 3 orders per customer. This query retrieves the top 3 orders per customer based on the order amount.

Sample Code

```
select *
from
(
  select
  [customer_id],
  [order_id],
  [amount],
  row_number() over (partition by customer_id order by amount desc) as [rank]
  from [dbo].[orders]
) as [ranked_orders]
where [rank] <= 3;
```

Explanation

Inner Query (Ranked Orders):

The inner query selects columns customer_id, order_id, and amount from the orders table.

The ROW_NUMBER() window function assigns a rank to each order within each customer_id group.

The PARTITION BY customer_id clause restarts the row numbering for each customer.

ORDER BY amount DESC ensures that the highest amounts receive the lowest rank values (i.e., rank 1 is the highest order amount within each customer group).

Outer Query (Top 3 Filtering):

The outer query wraps the ranked orders, filtering to retain only those rows where rank <= 3. This limits the results to the top 3 orders by amount for each customer.

Result

The final output includes the top 3 orders (by amount) for each customer, with each row containing customer_id, order_id, amount, and rank (indicating the position within the top 3 for each customer).This approach is commonly used to retrieve "Top-N" items within each category or group in SQL.

OVER PARTITION BY: The OVER PARTITION BY clause in SQL is used with window functions to define how rows are divided, grouped, and ordered for calculations across subsets of data within a result set. It is particularly helpful for applying functions like ROW_NUMBER(), RANK(), SUM(), and AVG() to each group separately without collapsing the result into one row per group.

Breakdown of OVER PARTITION BY

OVER: Introduces a window function and specifies the "window" or subset of rows on which the function will operate.

PARTITION BY: Defines how the data is divided into partitions or groups. Each unique value (or combination of values) in the PARTITION BY clause creates a separate partition.

ORDER BY (Optional): Specifies the order of rows within each partition, which is necessary for functions like ROW_NUMBER(), RANK(), and NTILE().

SQL window functions: Functions that perform calculations across a set of table rows related to the current row. Unlike aggregate functions, which summarize data by collapsing rows, window functions operate on a "window" of rows defined by the OVER clause, allowing row-by-row operations while still retaining each row in the output.

Key Characteristics of Window Functions

Operate Over Windows: Window functions use the OVER clause to specify the "window" of rows they should consider for each calculation. This window can be defined by PARTITION BY and/or ORDER BY clauses.

Do Not Collapse Rows: Unlike aggregate functions (like SUM or COUNT) that reduce groups of rows into single rows, window functions maintain each row in the result set, enabling row-by-row analysis within groups.

Can Use PARTITION BY and ORDER BY: Window functions can group (partition) rows with PARTITION BY and define the order of rows within each partition with ORDER BY.

Types of SQL Window Functions

Here are some commonly used SQL window functions:

1. Ranking Functions:

ROW_NUMBER(): Assigns a unique, sequential number to each row within a partition, based on a specified order.

RANK(): Assigns a rank to each row within a partition. Rows with the same value in the order column receive the same rank, leaving gaps for duplicate ranks.

DENSE_RANK(): Similar to RANK(), but without gaps. Rows with the same rank have consecutive ranking numbers.

NTILE(n): Divides rows into n roughly equal-sized groups, assigning a group number (1 to n) to each row.

2. Aggregate Functions:

SUM(), AVG(), MIN(), MAX(), COUNT(): These can be used as window functions, calculating running totals, averages, minimums, maximums, or counts within each partition.

Example: SUM(amount) OVER (PARTITION BY customer_id ORDER BY order_date)

3. Value Functions:

LAG(): Returns the value from a previous row within the window. Useful for comparing the current row to a prior row.

LEAD(): Returns the value from a subsequent row within the window. Useful for looking ahead in the dataset.

FIRST_VALUE() and LAST_VALUE(): Return the first or last value in the window, respectively.

4. Cumulative and Moving Aggregates:

Window functions can compute cumulative totals or moving averages by defining the "frame" of rows relative to the current row.

Benefits of Window Functions

Efficient Analysis: Great for analytics without needing subqueries or joins.

Enhanced Flexibility: You can calculate running totals, ranks, and differences without collapsing rows.

Simpler Syntax: Reduces complex SQL code for common analytical tasks.

Window functions are a powerful feature in SQL, offering capabilities for row-by-row data analysis, comparisons, and trend calculations within partitions. They are especially useful in reporting, financial analysis, and time-series data analysis.

Conditional Update with a JOIN

Update a table conditionally based on another table. This query updates the status of employees in the [employees] table to 'Active' for those who have been rehired, as recorded in the [employee_changes] table.

```
update [emp]
set [emp].[status] = 'Active'
from [dbo].[employees] as [emp]
join [employee_changes] as [empchange] on [emp].[id] = [empchange].[employee_id]
where [empchange].[change_type] = 'Rehired';
```

<u>Breakdown of the Query</u>

1 Target Table:

update [emp] specifies that the target for this update is the [employees] table, aliased as [emp].

2 Join Condition:

join [employee_changes] as [empchange] on [emp].[id] = [empchange].[employee_id] joins [employees] with [employee_changes] using the id field from [employees] and employee_id from [employee_changes].

This join ensures that only employees with corresponding records in [employee_changes] are considered for the update.

3 Update Condition:

where [empchange].[change_type] = 'Rehired' filters the joined results to only those rows where the change_type in [employee_changes] is 'Rehired'.

4 Set Clause:

set [emp].[status] = 'Active' sets the status field in [employees] to 'Active' for all employees who meet the where criteria.

<u>Purpose and Result</u>

The query updates the status of employees who have a record of being rehired in the [employee_changes] table, marking them as 'Active' in the [employees] table. This is useful for maintaining an up-to-date employee status based on changes recorded in a separate table.

Bulk Insert CSV file to Table

This query performs a bulk insert operation to load data from a CSV file into a SQL Server table named [target_table].

```
bulk insert [target_table]
from 'C:\Temp\data_to_insert.csv'
with
(
  format = 'CSV',
  firstrow = 2
)
```

Breakdown of the Query

bulk insert [target_table]: this command specifies the operation to bulk load data into the table called [target_table].

from 'C:\Temp\data_to_insert.csv': This part of the query indicates the source file from which the data will be read. In this case, it specifies a CSV file located at C:\Temp\data_to_insert.csv.

with (...): This section specifies additional options for the bulk insert operation.

format = 'CSV': This option specifies that the data in the source file is formatted as CSV (Comma-Separated Values). It informs SQL Server how to interpret the incoming data.

firstrow = 2: This option indicates that the bulk insert operation should start reading data from the second row of the CSV file. This is commonly used to skip a header row (which typically contains column names) that is not needed for the insert operation.

Purpose and Result

The purpose of this query is to efficiently load a large volume of data from the specified CSV file into the [target_table] in the database. By using the bulk insert method, the operation is optimized for performance, making it suitable for importing large datasets quickly.

Delete Duplicate Rows (two methods)

Using Having Count

Method One: Delete duplicate records from the [Employee] table based on [emp_id] where there is more than one record in the table for [emp_id] while retaining one instance of each [emp_id] (the not in portion).

```
delete from [Employee]
where [emp_id] in
(
  select
  [emp_id]
  from [Employee]
  group by [emp_id]
  having count(*) > 1
)
and [emp_id] not in
(
  select min([emp_id])  -- or other unique identifier to keep
  from [Employee]
  group by [emp_id]
  having count(*) > 1
)
```

Using a CTE, ROW_NUMBER(), and PARTITION BY

Method Two: Delete duplicate records from the [Employee] table based on [emp_id] using a Common Table Expression (CTE) to delete the duplicate records while retaining one instance of each [emp_id].

```
with cte as
(
  select
  [emp_id],
  row_number() over (partition by [emp_id]) as [rownum]
  from [Employee]
)
delete from cte where [rownum] > 1
```

Retrieve Vital Configuration and Resource Details About a SQL Server Instance

The purpose of this query is to gather and display essential information about the current SQL Server instance and its operating environment. This can be useful for system diagnostics, performance monitoring, or general server administration. "xp_msver" is a built-in stored procedure in SQL Server that provides various system-related information.

The Query

```
exec xp_msver 'ProductName', 'ProductVersion', 'Language', 'Platform', 'WindowsVersion',
'PhysicalMemory', 'ProcessorCount'
```

Conclusion

These SQL snippets cover a variety of needs, from simple data management to more complex transformations.

I will be adding new content to this document as time allows.