Compare SQL Server Temp Tables to Table Variables, Subqueries, Derived Tables, CTEs and Physical Tables

The temporary data stores tips included: temp tables, table variables, uncorrelated subqueries, correlated subqueries, derived tables, Common Table Expressions (CTEs) and staging tables implemented with permanent tables. By a temporary data store, this tip means one that is not a permanent part of a relational database or a data warehouse.

Each section of this tip drills down on a temporary data store in two ways. First, this tip describes the data store in a way that distinguishes it from other temporary data stores. Second, this tip covers some use cases that are typical ones for that temporary data store type. The tip closes with a summary section offering general guidelines about which database tasks can benefit the most from the different types of temporary data stores.

Memory-optimized versions of temporary data stores are outside the scope focus of this tip. Their access varies by different SQL Server editions for SQL Server versions 2014 and later. As of the preparation date for this tip, you can get recent information on memory-optimized data storage from Microsoft at these two resources (here and here).

SQL Server Temp Tables

Temp tables are materialized versions of temporary data stores. They are also appropriate for use in many different types of use cases. This tip uses the term materialized version to refer to a copy of a row set as of a point in time. If the original row set is modified after the copy is made, then the materialized version and the row set are no longer synchronized with each other.

Defining features of SQL Server Temp Tables

A temp table is a SQL Server object. Therefore, you can create a temp table with a create table statement. Also, you can drop a temp table object with a drop table statement. SQL Server knows that the table is a temporary table because its name starts with a hash sign. You can populate temp tables similarly to the way you populate permanent tables. For example, with insert statements and select…into statements. Finally, you can populate a temp table with a bulk insert statement; this is another way in which a temp table is like a permanent table.

Temp tables reside in the system tempdb database, which can serve multiple users and applications on a SQL Server instance. This feature distinguishes temp tables from permanent tables, which are normally created within a user-defined database. Both temp tables and permanent tables can have primary keys as well as clustered and non-clustered indexes, but temp tables cannot have foreign keys.

Compare SQL Server Temp Tables to Table Variables, Subqueries, Derived Tables, CTEs and Physical Tables

There are two types of temp tables: local and global. These two types have different naming conventions and scopes. A local temp table name must begin with a single hash sign, such as #my_first_local_temp_table. The scope for a local temp table is the connection that SQL Server uses to create the temp table. When the connection creating a local temp table closes, the local temp table goes out of scope. Another way to have a local temp table go out of scope is to remove the local temp table from the tempdb database with a drop table statement. Local temp tables created within a stored procedure can be referenced by child stored procedures, but you cannot pass a local temp table out of a stored procedure to a parent stored procedure or a script.

A global temp table name must begin with at least two hash signs, such as ##_my_first_global_temp_table. A global temp table has a scope that enables it to be referenced by any active connection – not just the one in which it was created. In this way, a global temp table is more like a permanent table than a local temp table. Also, a global temp table stays in scope until the connection for creating the global temp table closes and no other connection has an active reference to the table. This feature distinguishes a global temp table from a local temp table whose scope closes as soon as the connection creating the local temp table closes.

In case you are not familiar with the term scope within SQL Server, you can think of it as a module. Examples of modules include the connection for a tab in SSMS, a batch of T-SQL commands within a connection, a trigger, a stored procedure, or a function. An object is in scope within a module after its creation or definition by some other means (such as a declaration statement for a table variable). An object can go out of scope when a drop statement is issued, when a connection is closed, or when control passes to a statement outside a batch in an open connection. In the context of this tip, when a temporary data store is out of scope, you can no longer reference it.

Use cases for SQL Server temp tables

Temp tables are different than permanent tables in that they do not require a drop table statement for the table to go out of scope. Local temp tables can go out scope when the originating connection closes, and global temp tables can go out of scope when the originating session closes, and no other connection is referencing the table. Therefore, temp tables require less active management than permanent tables, which require a drop table statement to go out of scope.

You can import data to a temp table from either an external file or another database source, such as a filtered subset of rows from a permanent SQL Server table. Because a temp table can accept data from a file, it is a potential resource for staging data for insertion into a permanent table from an external data source. Some other temp data

Compare SQL Server Temp Tables to Table Variables, Subqueries, Derived Tables, CTEs and Physical Tables

stores, such as CTEs, require a relational data source instead of an external data source.

Temp tables are well able to handle some large data sources because you can assign indexes to temp tables. These indexes can speed processing of temp tables beyond what is possible with other temporary data stores that cannot be indexed, such as derived tables or Common Table Expressions.

SQL Server Table Variables

Table variables are somewhat like temp tables, but table variables are not SQL Server objects like temp tables. That is, you do not create a table variable with a create statement. Table variables are best suited for working with small row sets.

Defining features of SQL Server Table Variables

Table variables are SQL Server local variables, but they also have some properties that resemble temp tables. Because a table variable is a type of local variable, T-SQL scripts do not create table variables with a create table statement. Instead, use a declaration statement to make available a fresh table variable. The local variable type is table; table is the type of variable. A table variable name must begin with an @ sign, such as @_my_first_table_variable.

You can create a temp table with either a create table statement or the into clause in a select statement. Neither of these approaches work for table variables. After declaring a table variable, you can populate a table variable for individual rows with an insert statement having one or more value clauses as well as with an insert statement followed by a select statement returning one or more rows.

SQL Server does not maintain row statistics for table variables, but SQL Server does maintain them for temp tables. Therefore, select statements referencing table variables versus temp tables can run slower when the data source has a larger number of rows. A prior MSSQLTips.com tip confirmed this outcome using the SQL Server Profiler and data sources of 2000 and 1000000 rows. Because table variables do not support statistics, at least a couple of SQL Server professionals prefer restricting their use to very small row sets, such as a few up to less than 100 or at the very most 1000 (here and here).

In case you are not familiar with statistics in SQL Server, you can think of them as lightweight objects maintained by SQL Server to help optimize query designs. Statistics contain information about the distribution of values in one or more columns of a table or

Compare SQL Server Temp Tables to Table Variables, Subqueries, Derived Tables, CTEs and Physical Tables

indexed view. Because table variables do not support statistics, they cannot be optimized like temp tables which do support statistics.

Table variables, like other types of local variables, have a scope restricted to the batch or stored procedure in which they are created. Local temp tables have a scope limited to their connection, such as for a connection or a stored procedure. Since one connection can have multiple batches, local temp tables can have a wider scope that table variables. Global temp tables can be accessed from other connections besides the one in which SQL Server creates the global temp table. Therefore, the scope of a global temp table is wider than either table variables or local temp tables.

Temp tables and table variables act differently in transaction sets. Update, insert, delete and merge transactions can be rolled back for temp tables in a transaction's scope. Table variables within the scope of a transaction cannot have modifications rolled back.

The process for returning temporary data stores is different depending on the type of temporary data store and the T-SQL script container.

- You can use local temp tables, global temp tables, or table variables inside a stored procedure. However, only a global temp table can receive data created inside a stored procedure for use outside the stored procedure. You can assign a row set to a global temp table in a stored procedure. Then, you can reference the global temp table in a from clause outside the stored procedure.
- User-defined functions do not enable the creation of temp tables or references to previously created temp tables. However, the inline table-valued type of user-defined function can return a filtered subset of rows from a permanent table (see a code sample here). Therefore, you can create an inline table-valued function that references a permanent table populated with the contents of a temp table (a select…into statement is an easy way to accomplish this). After populating the permanent table and creating the function, a select statement can reference the function in its from clause with parameter value(s) that the function uses to derive a returned subset of rows from the permanent table.
- You can also declare, populate, and return a table variable from within user-defined function. Invoking the user-defined function in the from clause of a select statement returns the table variable. A code sample from this source demonstrates how to accomplish this.

Local temp tables go out of scope when the connection creating the local temp table is closed or when you drop the local temp table. Global temp tables go out of scope when the connection creating the global temp table is closed, and no other connection is

Compare SQL Server Temp Tables to Table Variables, Subqueries, Derived Tables, CTEs and Physical Tables

referencing the global temp table. You can also drop a global temp table. Table variables go out of scope when control flows away from the batch or other containing module for a table variable. There is no drop statement for a table variable.

Use cases for SQL Server Table Variables

Table variables are best reserved for use cases involving a relatively small number of rows. Additionally, table variables should not be used when the same row set needs to occur across two or more different batches within the same connection or across multiple connections. This is partly because it is known that table variables do not support fast access to data sources with a large number of rows. Additionally, it is because table variables have a more limited scope relative to temp tables -- especially global temp tables that can be referenced from more than one connection.

When your code can benefit from generating and returning a row set created via a user-defined function, then a table variable represents a better temporary data store than either a local or a global temp table. This is because a table variable is the only one of the three capable of being returned from a user-defined function.

If you have multiple transactions in the same transaction scope and some transactions may need conditional roll backs while other transactions should be executed unconditionally, then you can store those row sets for which roll backs are not allowed in table variables. Recall that table variables do not participate in transaction roll backs.

SQL Server Uncorrelated Subqueries

An uncorrelated subquery is a select statement that does not depend on the current row of an outer query when it runs. An uncorrelated subquery can be nested within a select, insert, update, or delete statement.

Defining features for SQL Server Uncorrelated Subqueries

An uncorrelated subquery is a select statement nested inside of another select statement or other T-SQL statement, such as a select, insert, update, or delete statement. The nested select statement is called uncorrelated when its outcome does not depend on the current row of the result set from the outer T-SQL statement.

The most common place for an uncorrelated subquery to appear is in the where clause of an outer query. When used within a where clause, an uncorrelated subquery can return one or more values. Comparisons between the column values of an outer query

Compare SQL Server Temp Tables to Table Variables, Subqueries, Derived Tables, CTEs and Physical Tables

with an uncorrelated subquery result set can be used to constrain rows returned from an outer query. Uncorrelated subqueries behave in having clauses for grouped data in a similar way to where clauses for ungrouped data.

When an uncorrelated subquery appears in the where clause of an outer query, then the subquery can be compared by any of a set of operators to column values from the outer query. Selected comparison operators include : =, !=, >, >=, <, <= , in, not in, exists, not exists, any, all. Rows with column values returning a value that satisfies the comparison operator are retained in the result set for the outer query.

Another place in which to use an uncorrelated subquery is among the list items for an outer select statement. When a subquery is a list item in an outer query, then it can return just one value. In this role, an uncorrelated subquery can place the same value on each row in the result set of an outer query. A subquery as a list item can benefit from an alias to assign a name to the returned value in a result set.

An uncorrelated subquery is different than a temp table in that it is a nested inside of another outer T-SQL statement. Because an uncorrelated subquery is not an object or variable, you cannot reference it outside of the outer T-SQL statement in which it resides. You can, of course, copy the same subquery to two or more different outer queries. However, SQL Server does not have built-in features for managing the consistency of copied subqueries used in different outer queries.

Use cases for SQL Server Uncorrelated Subqueries

The primary use case for an uncorrelated subquery is to draw a subset of rows from an outer query. While the returned value from an uncorrelated subquery does not depend on row values in the outer query, the selected subset rows do depend on row values in the outer query and how they compare to the values in the uncorrelated subquery. This capability to draw a subset applies to grouped rows from the having clause of an outer query as well as ungrouped rows from the where clause of an outer query.

Another valuable use of uncorrelated subqueries can be for specifying a subset of rows from a full set of rows that you want to update or delete from a data source. Use the subquery in the where clause within the from clause for a delete or update query statement. Similarly, if you are populating a new source or adding rows to an existing data source based on a subset of rows in another data source, an uncorrelated query can prove useful for specifying the subset of rows that you want to insert into a new or existing data source.

Compare SQL Server Temp Tables to Table Variables, Subqueries, Derived Tables, CTEs and Physical Tables

When an uncorrelated subquery is an item in outer select statement, then it can add a constant to each row of the outer query. The feature is particularly convenient for adding an aggregate value, such as the sum across all sales order amounts to a row set of individual sales orders.

SQL Server Correlated Subqueries

A correlated subquery is a select statement that depends on the current row of an outer query when the subquery runs. A correlated subquery can be nested within a select, insert, update, or delete statement.

Defining features for SQL Server Correlated Subqueries

Correlated subqueries are very similar to uncorrelated subqueries with one important difference. Correlated subqueries change their result set based on the current row of their outer query. The capability of a correlated subquery to adapt its output based on the outer query's current row makes the operation of a correlated subquery dynamic relative to the static output from an uncorrelated subquery.

The dynamic operation of a correlated subquery relative to an uncorrelated subquery is particularly apparent when adding aggregated column values as select list items to an outer query. With an uncorrelated subquery, each list item based on the subquery adds the same value to a row in the outer query. This is fine so long as you want to add just a single value, such as the sum of sales for all sales orders. However, what if you need to add a column with total sales by subcategory to a row set with one row per product subcategory? In this case, the added aggregate column value needs to vary from row to row depending on the subcategory column value for a row. Correlated subqueries facilitate this goal because they can change their output based on one or more current row values in the outer query, such as product subcategory value. The temporary data stores tutorial (url to be provided by editor at publication) includes an example query that demonstrates both uncorrelated and correlated subqueries in select list items for an outer query.

Use cases for SQL Server Correlated Subqueries

In SQL development projects, it is often desirable to add a new column to the outer query based on a subquery.

- If the rules for assigning a row value in the outer query is the same for all rows in the outer query, then you should use an uncorrelated subquery.

Compare SQL Server Temp Tables to Table Variables, Subqueries, Derived Tables, CTEs and Physical Tables

- On the other hand, if the rules for assigning a row value varies depending on outer query row value(s), then a correlated subquery is the right kind of subquery to use.

Instead of assigning multiple values to a single column, you can add multiple columns – one to receive values based on each subset of rows in an outer query. When each added column is for a separate subset of rows in the outer query, then you can use multiple uncorrelated subqueries. Each distinct uncorrelated subquery can define values for a different column in the outer query.

When performing updates, deletes and inserts via an outer query with a subquery, then you can allow for variations in the data manipulation operation based on the subquery. For example, if the same update is made to all outer query rows, then an uncorrelated subquery makes a good data source for the update statement. In contrast, if different assignments are made to column values for different sets of rows in the outer query, then a correlated subquery makes a good choice for the update statement. The correlated subquery can uniquely identify each subset of rows and provide an appropriate assignment value for each subset of rows.

SQL Server Derived Tables

A derived table is a subquery that behaves like a table.

Defining features for SQL Server Derived Tables

A derived table is a type of subquery that is in a parenthesis, assigned a name, and in the from clause of an outer select statement. The subquery returns a result set from a select statement. When the subquery is used to define a derived table, it can return more than one column for multiple rows. Common functions handled by derived tables include aggregating column values, performing complex calculations, specifying windows functions, filtering rows for the subquery's data source, or even just taking a subset of the columns from a data source.

Assigning names to the columns of a derived table allows the outer query for a derived table to process expressions by name instead of expression formula. For example, if a column named sales_after_taxes in a derived table is computed as the product of quantity_ordered_by_line_item multiplied by unit_price_by_line_item plus sales_tax_amount, you can reference the name sales_after_taxes instead of its computing expression in the outer query for the derived table. Assigning names to columns inside a derived table can make queries relying on derived tables more readable and self-documenting.

Compare SQL Server Temp Tables to Table Variables, Subqueries, Derived Tables, CTEs and Physical Tables

A derived table is sometimes called a nested query because it is embedded within an outer query. You can designate multiple levels of nesting so that the outer query for one derived table is itself nested inside another outer query and so forth. When you are computing complex expressions with multiple steps that must be executed in a fixed order, you may find it convenient to develop deep levels of nested derived tables. The column name for inner nested expressions, can be referenced by its name in the its outer query. In this way, successive layers of nested queries can inherit values by name from inner nested derived tables that make the code easier to understand.

A very simple example of successive nesting is as follows.

- In the inner-most derived table compute sales_before_taxes as quantity_ordered_by_line_item multiplied by price_by_line_item.
- In the outer derived table to inner-most derived table, compute sales_after_taxes as sales_before_taxes plus sales_tax_amount.
- In the outer-most query, compute sales_after_taxes summed and grouped by some other column, such as company_name, customer_name, state, or any other categorizing variable of interest.

Use cases for SQL Server Derived Tables

When developing a solution with agile technologies or there is a need to compute a quantity based on parts, it is common to compute the parts and then combine the parts in a final solution step. You can calculate and save the values for parts with any of several temporary data stores, such as derived tables, CTEs, or temp tables. An interesting question is: how to assess which temporary data store has the best performance in a particular situation? As with many database development solution questions, a good way to answer it is to compare the performance from all good candidate solution frameworks for a representative type of data to which the solution will be applied.

You can often express solutions based on nested derived tables or with multiple CTEs. When using CTEs, you need to return two or more CTEs from a single with keyword to accomplish this outcome. Each CTE can return results for a key part to the overall solution. The returning of two or more named CTEs from a with keyword is demonstrated in the MSSQLTips.com temporary data stores tutorial (url to be provided by editor at publication) as well as in a blog by Kathi Kellenberger.

Kathi's blog confirms with a pair of examples that a CTE can sometimes deliver faster execution than a derived table, and other times CTEs do not deliver superior performance. She used elapsed run times as well as percent of run time for

implementations based on CTEs versus derived tables. By testing both formulations, she was able to verify that the best approach depends on the details of the problem requiring a solution.

Kathi didn't test temporary tables as an alternative solution strategy, but temporary tables offer the potential to deliver superior value in some contexts as well. For example, it is known that performance for derived tables over multiple nesting levels or for joins between derived tables deteriorates as the size of a data source increase.

- One potential workaround to the degraded performance is to save the parts as values in temp tables.
- Follow the initial step by combining parts as values from temp tables as opposed to code for parts from either derived tables or CTEs.
- In my prior experience as an ETL developer, I have found this approach to be of value multiple times.

Finally, you can think of derived tables as a way of enabling view-like capabilities. Developers implementing view-like capabilities via derived tables do not require create permission for SQL Server objects. By taking advantage of T-SQL code, developers can take a subset of columns and/or rows from a single data source, combine multiple distinct data sources into one temporary data store, and add computed fields based on one data source or multiple data sources. Changes to the original data sources for solutions based on derived tables will be reflected whenever the derived-table solution is re-run.

If you need to run your solution from more than one point within a single solution or you want to run the solution without exposing the underlying code, then consider translating a derived-table or a CTE solution offering view-like capabilities to a view. This will enable developers to use the solution merely by referencing the view. Also, developers will be empowered to build new solutions that easily combine the result sets from views with other additional data sources.

SQL Server CTEs

CTEs, which stands for Common Table Expressions, are temporary data stores that are highly regarded for their "readability". Also, a special type of CTE known as a recursive CTE has built-in features for returning a result set that shows recursive relationships, such as in an organization chart or a bill of materials. This section does not include coverage of CTE applications to SQL Data Warehouse and Parallel Data Warehouse; see this link for the special CTE rules that apply to these Microsoft database products.

Compare SQL Server Temp Tables to Table Variables, Subqueries, Derived Tables, CTEs and Physical Tables

Defining features for SQL Server CTEs

A CTE is a query whose result set can be referenced in select, insert, delete, update, or merge statements. CTEs gained popularity as a temporary data store because of a couple of main reasons. First, they can encapsulate complex queries that are relatively easy to read – especially in comparison to derived tables. Second, some CTEs can reference themselves to generate a recursive result set in which some rows of output serve as parents to other rows of output. CTEs that reference themselves are called recursive CTEs, and those that do not reference themselves are called non-recursive CTEs (or sometimes just CTEs).

A CTE has a scope that extends to the line of code following it. This is part of what makes CTEs so easy to use. You specify the code for the CTE, and then you immediately follow the CTE's definition with a reference to it. If you want to re-use a CTE in multiple places within a script, then you need to copy it to a new location for each re-use.

The with keyword is a signal to the T-SQL compiler that one or more CTEs are to follow. The keyword is followed by a name for the CTE and optionally a list of fields returned from the CTE in parentheses. Next, a trailing as keyword separates the CTE name and field name from a second set of parentheses. The code inside the second set of parentheses must as a minimum define values for the returned fields from the CTE. T-SQL code inside the parentheses with the defining query for a non-recursive CTE can include such T-SQL keywords as join, where, group by, and union. If you have a need to make data sets from several different data sources trailing the initial with keyword, you can separate multiple CTEs from one another by commas. After defining enough CTEs for your requirement, insert on the next line of code a T-SQL query statement for using the one or more CTEs specified following the with keyword.

The structure of the code inside a recursive CTE must follow precise rules. Both recursive and non-recursive CTEs begin after the with keyword, a CTE name, and an optional list of field names. However, the code inside the second set of parentheses after the as keyword for a recursive CTE must follow these special rules.

- First, you must designate a select statement for the anchor row in the parent-child result set.
- Second, the anchor row select statement must be followed by a union all operator.
- Third, a select statement designates the child rows for the anchor row. The from clause in the select statement for child rows must include a reference back to the

Compare SQL Server Temp Tables to Table Variables, Subqueries, Derived Tables, CTEs and Physical Tables

CTE name. This reference back to the CTE is part of the reason the CTE is called a recursive CTE.

A CTE is sometimes compared to a view. The fields returned by a CTE can come from different data sources that are joined, concatenated, filtered, and/or grouped; the result set from these different kinds of T-SQL statements presents a different view of the original data sources. A CTE developer does not require create permission. This means that a less-privileged developer can achieve some of the benefits of a view in a CTE. In order to create a view, developers require a higher level of permissions than for creating a CTE. Of course, a CTE does not provide an object, such as a view, that can be re-used with a simple reference to an object name. However, you can copy the code for a CTE to multiple points in the same script or different scripts to re-use a CTE.

Use cases for SQL Server CTEs

There are three primary use cases for CTEs. These primary CTE capabilities are demonstrated in the MSSQLTips.com temporary data stores tutorial (url to be provided by editor at publication).

- First, you can encapsulate some code inside a CTE, and then reference the result set defined by the CTE name and its return fields in a select, insert, update, delete, or merge statement immediately following the CTE. This is how to use a single non-recursive CTE.
- Second, you can define more than one CTE following a with keyword. This allows you to combine the results from multiple CTEs in the T-SQL query statement immediately following the with keyword and its CTE definitions.
- Third, you can build a result set for a section of an organization chart in a recursive CTE. The result set can depict a managing employee and those reporting directly or indirectly to the managing employee. When combined with the hierarchyid data type, this result set can show both parent-child and sibling relationships.

It may also be worth denoting some cases in which a CTE is not especially well suited for use along with some potential remedial actions

- When a temporary data store needs to be re-used at multiple points in a connection or in different stored procedure or in a user-defined function.
    - CTEs cannot export their row set directly from a stored procedure; however, you can insert a row set from a CTE into a global temp table inside a stored procedure and return the values from the global temp table with the CTE result set outside the scope of the stored procedure

Compare SQL Server Temp Tables to Table Variables, Subqueries, Derived Tables, CTEs and Physical Tables

    - CTEs cannot export their row set directly from a user-defined function, but there are at least two ways to make the result set from a CTE outside a T-SQL user-defined function; see this link for code samples illustrating each approach
        - you can insert a row set from the CTE into a table variable and return the table variable from a user-defined table-valued function
        - for performance reasons, you may care to convert the CTE to a select statement that returns a result set and that returns its value through an inline, table-valued function
    - CTEs need to be copied to multiple locations in the script within a connection if the CTE must be re-used at multiple points within a script; there is no built-in capability for maintaining the consistency of the copied CTEs in different locations
- When you need a materialized version of the CTE query for easy re-use without the CTE statement in different locations
    - Consider copying the CTE result set to a table variable if the result set is small and will be re-used in the same batch
    - Consider copying the CTE result set to a local temp table if the result set is not small and will be re-used in the same connection
    - Consider copying the CTE result set to a global temp table if the result set is not small and will be re-used in one or more other connections besides the one used to create the global temp table
- When you need a permanent single object that combines multiple data sources and can be re-used in different connections, consider using a view instead of a CTE

Staging tables implemented with Permanent Tables in SQL Server

A permanent table as its name implies is not really a temporary data store, but a permanent table can be used to great advantage to hold data that is needed on a temporary basis. When using a permanent table, you will often derive your best value from it by creating a user-defined database for the table.

Defining features for staging tables implemented with permanent tables in SQL Server

Staging tables are a landing zone between external data sources, such as input file(s) for an ETL operation, and destination tables in a SQL Server database. It is important to understand that data in a staging table is always temporary whether you are using permanent tables in a staging database or temp tables in the tempdb. As the amount of data grows and/or the scope of ETL processing expands, the value of a permanent

Compare SQL Server Temp Tables to Table Variables, Subqueries, Derived Tables, CTEs and Physical Tables

staging table rises. Permanent tables in a user-defined database offer a dedicated database for your staging requirements, and if your requirements change, then you can change the database to meet your needs. Temp tables are quick and easy to setup and manage relative to permanent tables, but they cannot be dedicated to staging requirements in the same way as permanent tables within a custom user-defined database.

Here are a couple of guidelines for custom user-defined databases holding permanent staging tables.

- Consider creating a database that does not automatically grow. This can improve the performance of your staging operations by eliminating delays associated with automatically growing a database. You can learn more about managing auto database growth functionality from this source.
- You can also allocate space for your permanent staging tables on resources that are very fast, such as a solid-state storage device, or on local drives that are not used when your staging applications run. The purpose of these kinds of actions is to make available fast, dedicated storage for your staging requirements.

If your staging operations make use of lookup tables or user-defined functions that rarely change, consider storing these objects in the staging database for your permanent tables to managing staging operations.

- These actions can save you from having to populate lookup table(s) every time you stage an external data source for a destination table in a data mart or a relational database.
- By creating user-defined functions in a staging database you can expedite processing of staging data by simplifying access to code for routinely run code and eliminating the need for ETL developers to become familiar with advanced processing techniques that are special for a staging operation.

There are a variety of data management practices that you may want to consider in your staging operations.

- If you are receiving external data sources that frequently have errors, then you can consider a two-step loading practice.
    - You can initially load all data with varchar or nvarchar data types having wide maximum field lengths in one staging table
    - Then, you can parse the varchar and/or nvarchar data into more precisely typed data fields that match your target tables in a data mart or relational database

Compare SQL Server Temp Tables to Table Variables, Subqueries, Derived Tables, CTEs and Physical Tables

table. During the processing of data from the initial load to the staging table for a destination table you can flag and/or assess remedies for bad data

- Another potentially useful processing step for data in a staging database can be to re-shape the data for destination tables.
    - You may care to split data from one flat file into several different staging tables that correspond to different target tables in a relational database
    - If your destination table is not normalized, but your source data is normalized, then you can de-normalize data transferred to the staging database to match the layout of your destination table
- If you have many computed fields or complex calculations for some of your computed fields, consider adding columns for computed fields to a staging table along with previously cleansed data. With complex computed fields that draw on staging data and other computed fields,
    - you can initially compute the first round of basic computed fields from the staging data
    - in a second round of processing, you can calculate more complex fields that may draw on a combination of cleansed data and previously computed fields

Use cases for staging tables implemented with permanent tables in SQL Server

Permanent tables that are part of a dedicated database can be relatively expensive to setup and manage, but the rewards may be worth it when you have recurring staging requirements for high-volume, fast-turnaround ETL projects. These kind of ETL projects can be associated with the successive loading of data sets for a series of different clients from a legacy application. The data can be migrated in parts until all the data for all the clients are transferred from the old system to the new one. Between loads for successive clients or types of clients, you can re-customize your staging database in ways that best serve the needs of the ETL for the next client or batch of clients.

Permanent staging tables in a custom database are also suitable for multi-step ETL operations. This is because you may need a non-transient storage medium that can hold data over several days or weeks during the time that the different steps are executed in sequence. This kind of ETL implementation is particularly appropriate for those that may require some fine-tuning or custom adjustments between the steps in a multi-step ETL project.

Permanent staging tables are not suitable for low-volume ETL requirements with clean data arriving from a small set of clients. Another example of an ETL project type for which permanent tables may not be suitable is when there is no tight timeframe for the transfer of data from an external source to one or more destination tables. For this kind

Compare SQL Server Temp Tables to Table Variables, Subqueries, Derived Tables, CTEs and Physical Tables

of ETL requirements, you can use temp tables that reside in the system tempdb database instead of a dedicated custom staging database.

Comparative use case summary across temporary data types in SQL Server

As you can see from the preceding sections, there are many similarities and differences between temporary data stores. Also, different use cases can favor implementation via different temporary data stores. For example, a table variable is a good choice for returning a few rows from a user-defined function, but a table variable would be a very poor choice for an ETL application that needed to import tens of millions of rows (permanent tables in a staging database would be a far better choice for this scenario). Temp tables are more suitable for use cases that fall in between a few and tens of millions of rows.

Another major way of categorizing temporary data stores is via their scope. Temp tables, table variables and permanent tables support different scopes.

- Table variables are declared for use in a batch, and they go out of scope so they cannot be referenced when control flows out of the batch in which they are declared.
- Local temp tables are for use in the connection in which they are created; you cannot reference a local temp table created in one connection from another connection.
- On the other hand, global temp tables can be referenced from any connection.
- There is no drop statement for a table variable; table variables go out of scope when control flows out of the batch in which they are declared. Local and global temp tables can go out of scope with a drop table statement.
- Local temp tables can also go out of scope when you close the connection in which a local table is created.
- Global temp tables go out of scope when the connection in which they are created closes and no other connection is actively referencing the table.

Permanent tables are, by definition, not necessarily dedicated to temporary data, but they can hold data temporarily. Permanent tables do not go out of scope when an application using them closes. Even when the application closes and there are no active connections to a permanent data source, permanent tables still retain data. You can drop a permanent table or remove values from a permanent table via delete or truncate statements when temporary data are no longer needed.

Compare SQL Server Temp Tables to Table Variables, Subqueries, Derived Tables, CTEs and Physical Tables

In addition, to materialized temporary data stores, there are other temporary data sources that are defined exclusively by T-SQL code. Within the context of this tip, these data sources include uncorrelated subqueries, correlated subqueries, derived tables, and CTEs. These code-based temporary data stores never exist as an object or variable. In essence, they are all select statements that are referenced by another T-SQL statement. These code-based temporary data stores define a temporary data as a result set from a select statement – or even a pair of select statements in the case of a recursive CTE.

This tip section closes with the following summary table matching each of the seven temporary data stores to a selection of eleven use cases. Notice that most use cases can be served by more than just one temporary data store type, but there are a couple of exceptions.

- Recursive CTEs are especially customized for building recursive data sources.
- When importing tens or millions of records on a tight schedule, you will frequently benefit from the use of permanent tables in a customized staging database.
- Even while you could use a permanent table in a staging table for ad hoc applications involving several thousand rows, using this approach for medium-sized and small-sized ETL projects is not a best practice. Consider temp tables for smaller ETL operations without a tight schedule.

The summary table below also reveals one use case that is supported by all seven temporary data store types. The use case has the name: Can write to a global temp table from within a stored procedure with select…into. It is especially useful to be able to populate a global temp table because a global temp table can be accessed from any connection. This use case depends on the into clause for a select statement inside a stored procedure. The into clause copies values from a select statement's result set into a table that is dynamically created and populated by the select statement. The into clause operates the same way inside and out of a stored procedure.

- Uncorrelated subquery, correlated subquery, derived tables, and CTEs all can operate from a select statement. Therefore, the result set values from any of these temporary data store types can be transferred to a global temp table for easy access outside of a stored procedure.
- The result set values for a local temp table inside a stored procedure can be copied to a global temp table so the temp table result set values are accessible outside the stored procedure.
- You can also copy global temp table result set values to another global temp table if you ever need two copies of the same result with different names inside versus outside of a stored procedure.

Compare SQL Server Temp Tables to Table Variables, Subqueries, Derived Tables, CTEs and Physical Tables

- If your requirements dictate a need, you can even copy the result set from a permanent table, with a few edge case exceptions, to another permanent table with a different name.

Here are some guidelines for how to derive value from the following table of use cases and temporary data stores.

- If you have a use case for which you are trying to implement a solution, scan the Use Case column to find a use case that most closely matches your requirement. Then, develop trial solutions using the temporary data stores identified as appropriate for the use case. If you need help programming the solution, review our temporary data stores tutorial (url to be provided by editor at publication) and linked resources in this tip for learning more about programmatically managing a temporary data store. Finally, pick the best approach for your organization based on performance and maintenance considerations.
- If you are trying to learn more about the ways that a temporary data store can be used, scan all the use cases that match a temporary data store.

Select Use Cases for Temporary Data Sources in SQL Server

User Case	Temp table	Table Variable	Uncorrelated subquery	Correlated subquery	Derived table	CTE	Permanent table
Temp data must be available throughout a connection	Yes for local and global temp tables	No unless the connection has only one batch	No	No	No	No	Yes
Temp data must be available through any connection	Yes for global temp table, but not for local temp table	No	No	No	No	No	Yes
Temp data must be available only through a	No	Yes; portion is a batch within a connection	Yes; scope is limited to its outer query	Yes; scope is limited to its outer query	Yes; scope is limited to its outer query	Yes; scope is limited to first statemen	No

Compare SQL Server Temp Tables to Table Variables, Subqueries, Derived Tables, CTEs and Physical Tables

portion of a connection						t after the CTE	
You can insert data from an external data source into it	Yes	Yes	No	No	No	No	Yes
You need to assign indexes to the temporary data store	Limited relative to permanent tables	Limited relative to temp tables	No	No	No	No	Yes
Supports statistics	Yes	No	No	No	No	No	Yes
Can write to a global temp table from within a stored procedure with select…into	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Is customized for building a recursive result set such as an organization chart or a bill of materials	No	No	No	No	No	Yes; recursive CTEs are ideal for this use case	No
Can return a row set from a user-defined function	No, but you can insert temp table into a permanent table and return a filtered result set	Yes	No	No	No	No, but you can insert CTE into table variable and return the table variable	Yes, where table name is in the from clause of a select statement within an inline table-valued function

Compare SQL Server Temp Tables to Table Variables, Subqueries, Derived Tables, CTEs and Physical Tables

Want to use nested select statements as a source for result set from an outer-most query	No	No	No	No	Yes	No	No
Want to use custom staging database for ETL	No	No	No	No	No	No	Yes

Next Steps

- Start by picking the combination of temporary data store and use case from the "Select Use Cases for Temporary Data Source" table that interests you most.
- Review the summary information in the table.
- Examine the extended comments about features and use cases in this tip for whatever temporary data source you are trying to learn about.
- Examine the code samples in the temporary data stores tutorial for the temporary data store you are seeking to use.
- Pick any of the remaining combinations of temporary data store and use case that interest you. Perform the preceding three steps for the combination.

Source

https://www.mssqltips.com/sqlservertip/6021/compare-sql-server-temp-tables-to-table-variables-subqueries-derived-tables-ctes-and-physical-tables/?utm_source=dailynewsletter&utm_medium=email&utm_content=headline&utm_campaign=20190612