Assessment Report   
<Prepared For> / <Date>

**Mobilize.Net SnowConvert for Teradata**

***App Version Not Provided / Conversion Core 3.2.27***

The purpose of this document is to summarize the technical considerations and code analysis in migrating SQL to Snowflake from Teradata that either have an impact on the automated code conversion or cannot be handled by automated code conversion, as well as providing a high-level inventory and automation capability of the code that will need to be addressed.

**-----------------------------------------------------------------------------------------------------------------------------**

**OVERALL CONVERSION SUMMARY**

Total Files: 9

- SQL Files: 9

- Script Files: 0

Total Files Not Generated: 0

Conversion Speed: 1831 lines/sec

Conversion Time: 00:00:33

Total Conversion Errors: 3

Total Parsing Errors: 3

Total Warnings: 2204

Total Lines of Code (LOC): 62063

**SQL LINE CONVERSION SUMMARY**

Lines of Code: 62063

LOC Conversion Percentage: 99%

Unrecognized Lines of Code: 5

**SCRIPTS LINE CONVERSION SUMMARY**

Lines of Code: 0

LOC Conversion Percentage: -

Unrecognized Lines of Code: 0

**OBJECT CONVERSION SUMMARY**

Identified Objects: 2057

Identified Object Conversion Rate: 100 %

Unrecognized Objects: 3

**-------------------------------------------------------------------------------------------------------------------------------**

**Table of Contents**

[File and Object Level Breakdown 2](#_Toc69722205)

[Environment Settings 4](#_Toc69722206)

[Databases and Schema 4](#_Toc69722207)

[Tables 5](#_Toc69722208)

[Views, Join Indexes, Macros, and Procedures 8](#_Toc69722209)

[Considerations 9](#_Toc69722210)

# FILE AND OBJECT LEVEL BREAKDOWN

**SQL/DML/DDL – Files**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **File** | **Conversion Rate** | | **Total File Quantity** | **Total LOC** | **Parsing Errors** | **Conversion Errors** |
|  | **Files Generated** | **LOC** |
| SQL | 100% | 99% | 9 | 62063 | 3 | 3 |

**SQL/DML/DDL – Identified Objects**

The breakdown of all the database objects created or modified in all SQL files.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Object** | **Conversion Rate** | | **Total Object Quantity** | **Total LOC** | **Parsing Errors** | **Conversion Errors** |
| **Object** | **LOC** |
| Tables | 100% | 100% | 2047 | 61938 | 0 | 0 |
| Views | 100% | 100% | 3 | 28 | 0 | 0 |
| Join Indexes | 100% | 100% | 1 | 3 | 0 | 0 |
| Macros | 100% | 100% | 1 | 13 | 0 | 0 |
| Procedures | 100% | 100% | 3 | 18 | 0 | 0 |
| Functions | 100% | 100% | 2 | 24 | 0 | 0 |

**SCRIPTS – Files**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **File** | **Conversion Rate** | | **Total File Quantity** | **Total LOC** | **Parsing Errors** | **Conversion Errors** |
|  | **Files Generated** | **LOC** |  |
| BTEQ | 0% | - | 0 | 0 | 0 | 0 |
| FastLoad | 0% | - | 0 | 0 | 0 | 0 |
| MultiLoad | 0% | - | 0 | 0 | 0 | 0 |
| TPT | 0% | - | 0 | 0 | 0 | 0 |
| TPump | 0% | - | 0 | 0 | 0 | 0 |

**SCRIPTS – Identified Objects**:The breakdown of all the database objects created or modified in all script files (BTEQ, BTQ, FL, ML, TPUMP, TPT).

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Object** | **Conversion Rate** | | **Total Object Quantity** | **Total LOC** | **Parsing Errors** | **Conversion Errors** |
|  | **Object** | **LOC** |
| Tables | 0% | 0% | 0 | 0 | 0 | 0 |
| Views | 0% | 0% | 0 | 0 | 0 | 0 |
| Join Indexes | 0% | 0% | 0 | 0 | 0 | 0 |
| Macros | 0% | 0% | 0 | 0 | 0 | 0 |
| Procedures | 0% | 0% | 0 | 0 | 0 | 0 |
| Functions | 0% | 0% | 0 | 0 | 0 | 0 |

**ISSUES BREAKDOWN**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Low** | **Medium** | **High** | **Critical** |
| # of issues | 2210 | 1 | 0 | 3 |
| # of unique issues | 11 | 1 | 0 | 1 |

|  |  |  |  |
| --- | --- | --- | --- |
| **Error Code** | **Description** | **Instances** | **Level** |
| [MSCEWI1001](https://docs.mobilize.net/snowconvert/general/issues/mscewi1001) | Error parsing the source code | 3 | Critical |
| [MSCEWI1031](https://docs.mobilize.net/snowconvert/general/issues/mscewi1031) | Function not supported | 1 | Medium |
| [MSCEWI2040](https://docs.mobilize.net/snowconvert/for-teradata/issues/mscewi2040) | Format in a table not supported | 1850 | Low |
| [MSCEWI2015](https://docs.mobilize.net/snowconvert/for-teradata/issues/mscewi2015) | SET TABLE not supported | 306 | Low |
| [MSCEWI2008](https://docs.mobilize.net/snowconvert/for-teradata/issues/mscewi2008) | CLUSTER BY performance review | 36 | Low |
| [MSCEWI1039](https://docs.mobilize.net/snowconvert/general/issues/mscewi1039) | Semantic information could not be loaded | 7 | Low |
| [MSCEWI2001](https://docs.mobilize.net/snowconvert/for-teradata/issues/mscewi2001) | RECURSIVE FORWARD ALIAS error | 3 | Low |
| [MSCEWI2004](https://docs.mobilize.net/snowconvert/for-teradata/issues/mscewi2004) | Invalid default value | 3 | Low |
| [MSCEWI2016](https://docs.mobilize.net/snowconvert/for-teradata/issues/mscewi2016) | GLOBAL TEMPORARY TABLE not supported | 1 | Low |
| [MSCEWI1022](https://docs.mobilize.net/snowconvert/general/issues/mscewi1022) | Parameter missing dependency | 1 | Low |
| [MSCEWI1020](https://docs.mobilize.net/snowconvert/general/issues/mscewi1020) | Custom UDF inserted | 1 | Low |
| [MSCEWI1050](https://docs.mobilize.net/snowconvert/general/issues/mscewi1050) | Missing Dependencies Objects | 1 | Low |
| [MSCEWI2053](https://docs.mobilize.net/snowconvert/for-teradata/issues/mscewi2053) | Period types are handled as varchar | 1 | Low |

*If you are using the full version of SnowConvert, you can find out the exact file and location of each error in the Issues Report in the output reports folder created by SnowConvert.*

**Error Categorization**

|  |  |  |  |
| --- | --- | --- | --- |
| LOW | MEDIUM | HIGH | CRITICAL |
| Warnings for source code that there was no direct conversion to Snowflake. The code has been converted to a functionally equivalent output but should still be reviewed. | Errors that produce a functional or runtime difference. The user may have to invest a *low amount* of manual effort to complete the conversion. | Errors that produce a functional or runtime difference. The user may have to invest a *high amount* of manual effort to complete the conversion. | Parsing errors or errors that cause significant conversion exceptions in SnowConvert. If the source code is correct, a significant intervention will be required. |

*For more information on the error codes and their severity, please visit our documentation page on errors:*[*https://docs.mobilize.net/snowconvert/for-teradata/issues-and-troubleshooting*](https://docs.mobilize.net/snowconvert/for-teradata/issues-and-troubleshooting)

# ENVIRONMENT SETTINGS

These are key settings that need to be considered in the Teradata migration.

ANSI Mode versus Teradata Mode - Teradata mode which is typically used ignores only trailing spaces and ANSI mode ignores both leading and trailing spaces by default during comparison. This setting is important to understand when converting code.

<https://docs.teradata.com/reader/Daz9Bt8GiwSdtthYFn~vdw/6oXMWdRsjArKuCs36O7vmA>

*Parameters* -

QUOTED\_IDENTIFIERS\_IGNORE\_CASE (Snowflake) - Should be set to TRUE for the database for a Teradata migration. This is necessary as there are many identifiers that have to use quotes in Snowflake that are not necessary in Teradata (any name with a # symbol for example).

TIMESTAMP\_TYPE\_MAPPING (Snowflake) - Depending on customer requirements this alias mapping will need to be set to TIMESTAMP\_LTZ for the default behavior of the TIMESTAMP function to match the default behavior in Teradata.

ROUNDHALFWAYMAGUP (Teradata) - This Teradata parameter controls banker rounding and should be looked at to determine if banker rounding is being used. This parameter is currently in private preview for Snowflake.

# DATABASES & SCHEMAS

Teradata does not have the concept of schemas and a decision must be made when migrating Teradata environments and databases. The simplest approach is for a single Teradata environment to equate to a single Snowflake Database. The individual Teradata databases are then deployed as individual schemas within the Snowflake Database. However, there may be cases where workloads will need to split into separate Snowflake Databases for purposes of managing the resources (sizing) dedicated to specific database.

*DATABASES*

Number of Databases Containing Objects: 13

13 databases had objects defined (tables, views, macros, join indexes, procedures). The size of this list is important for conversion strategy if converting Teradata databases to Snowflake databases as opposed to Snowflake schemas within a single database.

No databases were found with the name PUBLIC or INFORMATION\_SCHEMA which are default Snowflake schemas.

TABLES**: 100% Identified Object Conversion Rate**

Conversion Rate by Lines of Code: 100%

Number of Tables: 2047

Lines of Code: 61938

Total Parsing Errors: 0

Conversion Errors: 0

Successfully converted objects have been converted to an equivalent table and data types capable of storing the same data values as Teradata but some functionality below needs to be further analyzed and considered based upon specific use cases.

*Constraints / Data Values* – Snowflake does not enforce or allow constraints. If your data input/ingestion process relies on these capabilities to validate data prior to being inserted into a table, these processes will need to be rearchitected to prevent duplicate or bad data.

|  |  |  |  |
| --- | --- | --- | --- |
| **Constraint / DataValue** | **Instances** | **Tables Impacted** | **Percent Impacted** |
| CHECK | 0 | 0/2047 | 0% |
| UNIQUE PRIMARY INDEX | 1212 | 1212/2047 | 59% |
| PRIMARY KEY | 59 | 59/2047 | 2% |
| FOREIGN KEY | 0 | 0/2047 | 0% |

Notes:

* CHECK – Not supported in Snowflake, checks the value of data being inserted to be a specific list of values or meet certain conditions.
* UNIQUE PRIMARY INDEX / PRIMARY KEY – Not enforced in Snowflake. Definition is allowed for documentation purposes only.
* FOREIGN KEY – Not enforced in Snowflake. Definition is allowed for documentation purposes only.

*Object Naming* – Object naming conventions that have significance in a migration.

|  |  |  |  |
| --- | --- | --- | --- |
| **Object Naming** | **Instances** | **Tables Impacted** | **Percent Impacted** |
| Invalid Column Names | 0 | 0/2047 | 0% |
| Identifiers Requiring Double Quotes | 34 | 14/2047 | 0% |

Notes:

* + Invalid column names – These column names are not supported in Snowflake and are converted to XXX\_<column name>\_XXX. This could require changes to external reporting/ETL systems. Examples include ‘current\_date’, ‘localtimestamp’.
  + Identifiers Requiring Double Quotes – These objects require double quotes for defining in Snowflake and as a result are case sensitive unless the parameter QUOTED\_IDENTIFIERS\_IGNORE\_CASE has been set to True.

*Table Types* – Table properties may not perfectly line up between Snowflake and your source code platform. Here are some key type differences between Snowflake and your source language.

|  |  |  |  |
| --- | --- | --- | --- |
| **Table Types** | **Instances** | **Tables Impacted** | **Percent Impacted** |
| SET TABLES | 306 | 306/2047 | 14% |
| TEMPORAL TABLES | 0 | 0/2047 | 0% |
| GLOBAL TEMPORARY TABLES | 1 | 1/2047 | 0% |
| QUEUE TABLES | 0 | 0/2047 | 0% |

Notes:

* *SET Tables* – Set tables in Teradata automatically prevent duplicate rows by checking for duplicates upon insert. If a primary key is defined it will base the duplicate check on the primary key, otherwise it compares the entire row - which is very performance intensive. Snowflake does not have this functionality and will require rearchitecting the data ingestion process to check for duplicates prior to insertion if the SET functionality of these tables is being leveraged for de-duplication.
* *Temporal Tables* – Temporal tables operate as pre-built slowly changing dimensions, capturing history as records are inserted/updated/deleted. Snowflake does not have an equivalent table type. Some of this functionality can be mimicked utilizing Streams, Tasks and Time Travel but use cases must be considered due to process timing and change capture requirements.
* *Global Temporary Tables* – Global temporary tables allow for a table to be permanently defined but the contents of the table are specific and temporary to users’ sessions. Snowflake does not currently have a comparable feature and use of these must be re-architected based upon the use case.
* *Queue Tables* – Persistent table type used to handle queue‑oriented data, such as event processing and asynchronous data loading applications. Snowflake does not have a like functionality and a process will need to be defined to emulate the functionality.

*Data Types* – Just as with table types, data properties may not perfectly line up between Snowflake and your source platform. Here are some key data type differences between Snowflake and your source language.

|  |  |  |  |
| --- | --- | --- | --- |
| **Data Types** | **Instances** | **Tables Impacted** | **Percent Impacted** |
| NUMBER without precision | 481 | 97/2047 | 4% |
| INTERVAL | 0 | 0/2047 | 0% |
| PERIOD | 2 | 1/2047 | 0% |
| BLOB | 0 | 0/2047 | 0% |
| CLOB | 0 | 0/2047 | 0% |
| GEOSPATIAL | 0 | 0/2047 | 0% |
| XML/JSON | 0 | 0/2047 | 0% |

*Notes:*

* This assessment can be used as a complement to a full data profile. A complete inventory for this assessment is coming soon.
* *NUMBER without precision* – Number when defined without a scale/precision in Teradata allows for a flexible scale value for any given record from 0 to 38 provided the total precision never exceeds 38. Snowflake does not allow this and always has a fixed scale/precision. Data Profiling results showed….
* *INTERVAL* – Interval data types are not supported in Snowflake and are converted to varchars and manipulated as such using custom functions. Heavy use of intervals can cause performance issues and may require some re-architecture of the DDL and SQL.
* *PERIOD* – Period data types are not supported in Snowflake and are converted to varchars. They are a pair of dates or timestamps. Heavy use of periods can cause performance issues and may require some re-architecture of the DDL and SQL.
* *BLOB* – BINARY can be used instead but has a maximum of 8,388,608 bytes. Data Profiling results showed….
* *CLOB* – VARCHAR can be used instead but has a maximum of 16,777,216 bytes (for single-byte). Data Profiling results showed….
* *GEOSPATIAL* – Geospatial types have recently had support released.  Some specific use cases may require evaluation for workarounds.
* *XML/JSON* – Special column types that are converted to VARIANT columns in Snowflake and may need additional attention in testing/data loading.

*Data Values* – Data values that have significance in a migration.

|  |  |  |  |
| --- | --- | --- | --- |
| **Data Types** | **Instances** | **Tables Impacted** | **Percent Impacted** |
| Binary Defaults | 0 | 0/2047 | 0% |
| TIME Defaults | 0 | 0/2047 | 0% |
| FLOAT Defaults | 0 | 0/2047 | 0% |
| IDENTITY Columns | 0 | 457/2047 | 22% |
| SESSION Defaults | 0 | 0/2047 | 0% |

*Notes:*

* *Binary Defaults* – Binary data types do not currently support a default value being defined.
* *TIME Defaults* – Time with TimeZone is not a supported type in Snowflake
* *FLOAT Default* – Timestamp can be defined as a default value for a FLOAT column in Teradata but is not supported in Snowflake. This is typically a column required for an auxiliary system such as SAS and many times will not be an issue once the system has been adjusted to work with Snowflake.
* *IDENTITY Columns* – These are converted to sequences and can represent an area that requires additional testing. 0 unique definitions were found. A large number of these can represent an area of increased testing, particularly if sequences are required to be kept in progression at cutover.
* *SESSION Defaults* – These columns are defined as integers in Teradata and must be varchars in Snowflake to hold the Snowflake current\_session value.

*Unique Conversion Elements* – Other elements that may not explicitly fit into one of the previous categories that have significance in a migration.

|  |  |  |  |
| --- | --- | --- | --- |
| **Data Types** | **Instances** | **Tables Impacted** | **Percent Impacted** |
| Formats | 1850 | 870/2047 | 42% |
| Partition By | 36 | 36/2047 | 1% |
| UPPERCASE | 0 | 0/2047 | 0% |

Notes:

* *Formats* – Default format definitions by column are not supported in Snowflake. This can have implications for reporting or data extracts that are created for consumption. 36 unique format definitions were found.
* *Partition By* – Partitions are not supported in Snowflake but have a similar functionality called CLUSTER BY. By default, the PARTITION BY statements are converted to CLUSTER BY but are commented out as CLUSTER BY does not support all the same cases that can be defined in Partitions, so some must be modified before enabling. Each CLUSTER BY statement will need to be individually evaluated based upon the necessity for performance.
* *UPPERCASE* – Not supported in Snowflake and must be analyzed to determine if case insensitive collation is sufficient.

**Identified Object Summary (Non Tables)**

VIEWS**: 100% Identified Object Conversion Rate**

Conversion Rate by Lines of Code: 100%

Number of Views: 3

Number of Views created with only SELECT \* FROM: 1

Number of Views with more than 10 nested Select Statements: 0

Lines of Code 28

Parsing Errors: 0

Conversion Errors: 0

**JOIN INDEXES: 100% Identified Object Conversion Rate**

*Join indexes can be converted to Materialized views if only one table is referenced in the join index. This can have cost impacts that need to be analyzed.  Join indexes referencing more than a single table would be converted to a traditional view.*

Conversion Rate by Lines of Code: 100%

Number of Join Indexes: 1

Lines of Code: 3

Parsing Errors: 0

Conversion Errors: 0

**MACROS: 100% Identified Object Conversion Rate**

Conversion Rate by Lines of Code: 100%

Number of Macros: 1

Lines of Code: 13

Parsing Errors: 0

Conversion Errors: 0

**PROCEDURES: 100% Identified Object Conversion Rate**

Conversion Rate by Lines of Code: 100%

Number of Procedures: 3

Number of Procedures with InOut or Out Parameters: 0

Lines of Code: 18

Parsing Errors: 0

Conversion Errors: 0

Dynamic Result Sets: 0

Cursors: 0

Dynamic SQL Statements: 0

ACTIVITY\_COUNT Occurrences: 0

**FUNCTIONS: 100% Identified Object Conversion Rate**

Conversion Rate by Lines of Code: 100 %

Number of Functions: 2

Lines of Code: 24

Parsing Errors: 0

Conversion Errors: 0

Functions with C Body: 0

Functions with C++ Body: 0

Functions with Java Body: 0

Functions with SQL Body: 2

# CONSIDERATIONS

**Numeric Data Operations**

*Calculation Precision* - Calculations in Teradata round after every step based upon the data types in the operation.  For decimal types, this will keep the larger precision used in the calculation.  NUMBER data types will keep full precision.  Since Snowflake stores everything as a number, it keeps full precision throughout the calculation which can result in different outcomes for any number type (decimal, integer, float...) other than NUMBER. This behavior is not modified through code conversion as it is typically not an intended desired result by programmer.

Teradata: SELECT (1.00/28) \* 15.00 = 0.60

Snowflake: SELECT (1.00/28) \* 15.00 = 0.535710 = 0.54

*Integer-Integer Division* – Teradata performs a truncation or floor when dividing two integer values in, whereas Snowflake will ultimately perform a round operation. This scenario is accounted for in the automated code conversion by adding a TRUNC statement wherever this happens.

Teradata: SELECT (5/3) = 1

Snowflake: SELECT (5/3) = 1.6666666 = 2

Converted Snowflake: SELECT TRUNC(5/3) = 1

*Banker Rounding* - Teradata has the option to use Banker rounding by setting the parameter ROUNDHALFWAYMAGUP while Snowflake only uses normal rounding.

|  |  |  |
| --- | --- | --- |
| SQL | Teradata | Snowflake |
| **CAST( 1.05 AS DECIMAL(9,1))** | 1.0 | 1.1 |
| **CAST( 1.15 AS DECIMAL(9,1))** | 1.2 | 1.2 |
| **CAST( 1.25 AS DECIMAL(9,1))** | 1.2 | 1.3 |
| **CAST( 1.35 AS DECIMAL(9,1))** | 1.4 | 1.4 |
| **CAST( 1.45 AS DECIMAL(9,1))** | 1.4 | 1.5 |
| **CAST( 1.55 AS DECIMAL(9,1))** | 1.6 | 1.6 |
| **CAST( 1.65 AS DECIMAL(9,1))** | 1.6 | 1.7 |
| **CAST( 1.75 AS DECIMAL(9,1))** | 1.8 | 1.8 |
| **CAST( 1.85 AS DECIMAL(9,1))** | 1.8 | 1.9 |
| **CAST( 1.95 AS DECIMAL(9,1))** | 2.0 | 2.0 |

*Decimal to Integer Conversion* - Teradata truncates the decimal values while Snowflake rounds to the nearest integer. This is accounted for in the conversion by inserting a TRUNC statement.

|  |  |  |
| --- | --- | --- |
| SQL | Teradata | Snowflake |
| **CAST( 1.0 AS INTEGER)** | 1 | 1 |
| **CAST( 1.1 AS INTEGER)** | 1 | 1 |
| **CAST( 1.2 AS INTEGER)** | 1 | 1 |
| **CAST( 1.3 AS INTEGER)** | 1 | 1 |
| **CAST( 1.4 AS INTEGER)** | 1 | 1 |
| **CAST( 1.5 AS INTEGER)** | 1 | 2 |
| **CAST( 1.6 AS INTEGER)** | 1 | 2 |
| **CAST( 1.7 AS INTEGER)** | 1 | 2 |
| **CAST( 1.8 AS INTEGER)** | 1 | 2 |
| **CAST( 1.9 AS INTEGER)** | 1 | 2 |

*Number without Precision/Scale* - Number when defined without a scale/precision in Teradata allows for a flexible scale value for any given record from 0 to 38 provided the total precision never exceeds 38.  Snowflake does not allow this and always has a fixed scale/precision.  An example of numbers defined in a table this way:

CREATE MULTISET TABLE DATABASEXYZ.TABLE\_NUMS

     (NUM\_COL1 NUMBER(\*),

      NUM\_COL2 NUMBER,

      NUM\_COL3 NUMBER(38,\*));

In this table, the example of 2 values below that would not fit into a single Snowflake column but could be found in any of the columns shown in the table above in Teradata:

Value 1:  123,345,678,901,234,567,891,012.0123456789

Value 2:  123.12345678901234567890

These two values would require a fixed precision/scale of NUMBER(42, 20) which exceeds the maximum available in Snowflake precision of 38.  Snowflake is currently developing the functionality for flexible precision/scale.

*Truncation on INSERT for SQL DML Statements* - Teradata will auto-truncate a string value upon insert if the string is too large to fit into the specified field.  SnowConvert will convert fields on a like to like basis (ex: VARCHAR(20) -> VARCHAR(20)).  If an ingestion process is relying on auto-truncation, that process will need to be adjusted manually to truncate the data using a LEFT() function.  SnowConvert does not automatically add this as there are multiple implications to doing this across an entire code base.

*Float Default Issue Example:*

/\* <sc-table> TABLE DUMMY.EXAMPLE </sc-table> \*/

/\*\*\*\* WARNING: SET TABLE FUNCTIONALITY NOT SUPPORTED \*\*\*\*/

CREATE TABLE DUMMY.PUBLIC.EXAMPLE (

LOGTYPE INTEGER,

OPERSEQ INTEGER DEFAULT 0,

RUNTIME FLOAT **/\*\*\*\* ERROR: DEFAULT CURRENT\_TIME NOT VALID FOR DATA TYPE \*\*\*\*/**

);

*Float Data Aggregation* - Float data types are by definition approximations and as such different databases may aggregate differently due to how these approximations are handled at different points in the internal calculations of the database.

**Other Considerations**

*Join Elimination* - Snowflake does not currently eliminate un-required joins in any SQL that is run. It executes the SQL assuming that all objects included will potentially affect the result sets. Teradata has built-in join elimination features by leveraging primary-foreign key relationships that are defined in the DDL. The primary reason for this is to help avoid improperly written queries and is rarely an issue where code has not been written to intentionally take advantage of this feature. If a design decision was taken in views or DML to purposefully exploit this feature, code conversion can not address this and re-architecting some of that solution design may be required.

**Using max() over (order by) and other non-rank-related window functions**

Teradata behavior and defaults

**Default**: In the presence of an ORDER BY clause and the absence of a ROWS or ROWS BETWEEN clause, Teradata SQL window aggregate functions use **ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING**.

Snowflake behavior and defaults:

**Default**: In the presence of an ORDER BY clause and the absence of a ROWS or ROWS BETWEEN clause, Snowflake window aggregate functions use **ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW**.  
  
**Example:**

Below is a TEST\_WIN table containing salary information for different employees in different departments.

|  |  |  |  |
| --- | --- | --- | --- |
| DEPT\_NM | DEPT\_NO | EMP\_NO | SALARY |
| SALES | 10 | 11 | 5000 |
| SALES | 10 | 12 | 6000 |
| HR | 20 | 21 | 1000 |
| HR | 20 | 22 | 2000 |
| PS | 30 | 31 | 7000 |
| PS | 30 | 32 | 9000 |

Executing the below code in Teradata returns the maximum salary of an employee across departments.

SELECT DEPT\_NM, SALARY ,DEPT\_NO,

MAX(SALARY) over ( ORDER BY DEPT\_NO ) As MAX\_DEPT\_SALARY

FROM TEST\_WIN;

|  |  |  |  |
| --- | --- | --- | --- |
| **DEPT\_NM** | **SALARY** | **DEPT\_NO** | **MAX\_DEPT\_SALARY** |
| SALES | 6000 | 10 | 9000 |
| SALES | 5000 | 10 | 9000 |
| HR | 2000 | 20 | 9000 |
| HR | 1000 | 20 | 9000 |
| PS | 7000 | 30 | 9000 |
| PS | 9000 | 30 | 9000 |

However, executing the same code (code converted by Mobilize-SnowConvert) produces different results (highlighted values), which are correct as per the defaults of Snowflake.

SELECT DEPT\_NM, SALARY ,DEPT\_NO,

MAX(SALARY) over ( ORDER BY DEPT\_NO ) As MAX\_DEPT\_SALARY

FROM TEST\_WIN;

|  |  |  |  |
| --- | --- | --- | --- |
| **DEPT\_NM** | **SALARY** | **DEPT\_NO** | **MAX\_DEPT\_SALARY** |
| SALES | 5000 | 10 | 6000 |
| SALES | 6000 | 10 | 6000 |
| HR | 1000 | 20 | 6000 |
| HR | 2000 | 20 | 6000 |
| PS | 7000 | 30 | 9000 |
| PS | 9000 | 30 | 9000 |

In order to produce the same results as in Teradata, it requires adding the ROWS/RANGE value as per the below code.

SELECT DEPT\_NM, SALARY ,DEPT\_NO,

MAX(SALARY) over ( ORDER BY DEPT\_NO RANGE BETWEEN UNBOUNDED PRECEDING and UNBOUNDED FOLLOWING) As MAX\_DEPT\_SALARY

FROM TEST WIN;

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| DEPT\_NM | SALARY |  | DEPT\_NO | MAX\_DEPT\_SALARY |
| SALES | 5000 |  | 10 | 9000 |
| SALES | 6000 |  | 10 | 9000 |
| HR | 1000 |  | 20 | 9000 |
| HR | 2000 |  | 20 | 9000 |
| PS | 7000 |  | 30 | 9000 |
| PS | 9000 |  | 30 | 9000 |

The above addition of the RANGE/ROWS clause is to explicitly specify how the ORDER BY clause is working, similar behavior can also be achieved by omitting the order by clause altogether.

**References:**   
Snowflake: <https://docs.snowflake.com/en/sql-reference/functions-analytic.html>  
Teradata: <https://docs.teradata.com/r/756LNiPSFdY~4JcCCcR5Cw/dIV_fAtkK3UeUIQ5_uucQw>