Stored Procedure DWHPRO TITHONIZER

Table of Content

1. About this Document

This document describes the purpose, technical steps and chapters, and the concrete usage of the Teradata Stored Procedure DWHPRO TITHONIZER, as well as the assumptions and methodologies that are implicitly or explicitly incorporated in it.

2. The Purpose of DWHPRO TITHONIZER

The Stored Procedure DWHPRO TITHONIZER has been developed by Paul Timar.

It automatizes the process of finding and implementing values for compression on columns and values that are technically suitable for it, in the absence of any tool by Teradata or third parties that handles this task.

It was developed and tested for Teradata Versions 13 and 14.

2.1. Automatic Determination of the Best Possible Compression

DWHPRO TITHONIZER determines the set of compression candidate columns and the compression candidate values per column. It then selects the one combination of compressions over the entire column set that achieves the highest savings under the given technical constraints and with the current table demographics. For every candidate column, anything between not compressing and compressing at the maximum depth of 255 distinct values can be proposed by DWHPRO TITHONIZER.

2.2. Automatic Recompression on Demand

DWHPRO TITHONIZER can be restarted for the same table again, whether it is currently compressed to some extent or not. The result of a rerun for the same table can be an identical set of compression proposals or a different one, depending on the demographic changes the table underwent since it was compressed the last time. DWHPRO TITHONIZER is therefore also well suited for regular on on-demand revisions of compressions.

2.3. Storage of the Most Recent Compression as Loss Protection

The Stored Procedure does not apply a proposed compression directly. It creates the compression value sets per column and inserts them in a Staging table TITHONOS.

It is from there that any compressions can be retrieved and made ready for executions. In case of recreations of Tables from DDL statements that do not include given compressions, the most recent compression can easily be reapplied out of the Staging table without having to rerun the Stored Procedure.

Because of reservations against the compression of certain columns due to business reasons that lie outside the scope of what can be derived from the technical characteristics of columns, the execution of compression remains at the discretion of the user.

3. Assumptions

The Stored Procedure DWHPRO TITHONIZER is created under the following assumptions:

It is possible to find one optimal set of compressed values over all candidate columns that achieves the highest benefit.

The extent of compression is limited by the storage space of the table header, i.e. only so many values can be compressed over that no more than 8192 Bytes in total are consumed by storing the set of values in the table header.

More compression means more cost in terms of additional Presence Bytes, once the given Presence Bytes are used up.

Overly lengthy character entries shall not be subject to compression, as they most likely contain individual, non-standard or non-repeating entries that are costly to store relative to short and frequent standard values.

There is a minimum table size below which compression will not achieve a net space saving in principle.

Above the minimum size, suggested compressions shall achieve a net benefit. Cases of “overshooting” by suggesting too many values relative to the table size shall be made very unlikely.

Due to some degree of fragmentation always present on data blocks, the exact amount of Perm Space that a table used can never be determined with certainty, unless Teradata introduces means to investigate space and fragmentation on a block level.

4. Prerequisites for Deployment and Use

4.1. Technical Requirements for Deployment

The deployment user installing the Stored Procedure in an environment must have all of the following rights:

CREATE TABLE ('CT'), at least on the Stage database, and including the right to create volatile tables.

DROP TABLE ('DT'), at least on the Stage database, and including the right to create volatile tables.

CREATE PROCEDURE ('PC')

DROP PROCEDURE ('PD')

ALTER PROCEDURE ('AP')

EXECUTE PROCEDURE ('PE')

DELETE ('D')

INSERT ('I')

SELECT ('R')

UPDATE ('U')

DWHPRO TITHONIZER refers to a global temporary table G00_SP_LOG_TEMP for logging purposes. This table needs to be created either in the staging database directly or in any other database, together with a 1:1 view on it in the stage database. If G00_SP_LOG_TEMP is not present, DWHPRO TITHONIZER will fail.

Immediately before the deployment, i.e. the execution of a REPLACE PROCEDURE command, every Volatile Table that is created inside the Stored Procedure, must be created beforehand in the same session. These tables can be dropped immediately after a successful creation of DWHPRO TITHONIZER, or via a session logoff.

4.2. Technical Requirements for Usage

The user executing the stored procedure must have the aforementioned deployment rights plus select rights on the target database and table chosen.

The user implementing the compression proposal result of DWHPRO TITHONIZER to the target table must also be able to execute ALTER TABLE on the target table.

4.3. Input Parameter Setting

There are only two input parameters necessary:

ETLDB: The database that the target table resides in.

TABLENAME: The name of the target table to be analyzed.

Both must be provided as a string being equal to the full name as stored in dbc.

Any other information necessary for the completion of the analysis and compression proposal tasks is derived internally or queried from dbc views.

4.4. Logging

Every technical step executed will be logged temporarily in a global temporary table G00_SP_LOG_TEMP and can be queried from there as long as the session is active.

This table is composed as follows:

```
CREATE SET GLOBAL TEMPORARY TABLE DATABASE.G00_SP_LOG_TEMP,NO FALLBACK ,
   CHECKSUM = DEFAULT,
   DEFAULT MERGEBLOCKRATIO,
   LOG
   (
    TARGET_TABLE VARCHAR(32) CHARACTER SET LATIN NOT CASESPECIFIC NOT NULL,
    STEP_NUM SMALLINT NOT NULL,
    STEP_DESC VARCHAR(32) CHARACTER SET LATIN NOT CASESPECIFIC,
    TM_START TIMESTAMP(6) NOT NULL,
    STEP_START TIMESTAMP(6),
    SQL_REQUEST_TEXT VARCHAR(32000) CHARACTER SET LATIN NOT CASESPECIFIC,
    SQL_STATE CHAR(5) CHARACTER SET LATIN NOT CASESPECIFIC,
    SQL_CODE INTEGER,
    NM_ACTIVITY_COUNT INTEGER,
    STEP_END TIMESTAMP(6),
    LOAD_ID INTEGER)
UNIQUE PRIMARY INDEX ( TARGET_TABLE ,STEP_NUM ,TM_START )
ON COMMIT PRESERVE ROWS;
```

This structure corresponds to the expected one in the Stored Procedure.

Currently, there is no permanent logging established for the step level.

The entire Stored Procedure run is wrapped in a Query Band, so that its resource consumption can be detected and reported.

4.5. Call Statement Generation

One table requires one call of DWHPRO TITHONIZER.

For the generation of one or more call statements, use the following generation query:

```
-- CREATE THE CALL STATEMENTS FOR A DATABASE (AND TABLE GROUP)
SELECT
'CALL DWHPRO TITHONIZER('''||trim(databasename)||''','''||trim(tablename)||''');'
FROM DBC.TABLES
WHERE DATABASENAME='TARGET_DATABASE'
AND SUBSTR(TRIM(TABLENAME),1,1)='T'
ORDER BY databasename, tablename;
```

You can alter the tablename or databasename selection for your own purposes.

Pick the resulting call statements and execute them.

4.6. Manual Inspection of the Resulting Compression Proposal

Not every column that should be exempt from compression due to its content or due to other issues can be traced automatically, i.e. by looking at its technical characteristics alone. Therefore, the alter table statement created out of the proposed compression must be inspected before execution. Any undesired compression can then be removed or cut back to the set of desired level. The result will then be a less-than-optimal, but feasible compression.

If it turns out that individual columns shall be exempt from compression on a table-to-table or database-to-database basis, then this information must be stored somewhere so that the Stored Procedure can retrieve it and make it part of the compression candidate column set determination. The current version does not foresee for such a source in the form of a table or name pattern.

5. Technical Chapters and steps

DWHPRO TITHONIZER is divided into chapters and steps. Each chapter contains a series of technical preparatory or fulfillment steps that contribute to the achievement of one aspect of the total functionality of the Stored Procedure.

5.1. Chapter 0: Initialization and Prerequisite Check

All the variables necessary for future steps are initialized of filled with specific constants.

A Query Band is defined that covers the entire workload of DWHPRO TITHONIZER for the given input.

Next, the input parameters are validated. If either the database or the table input string does not equal an existing database or table, respectively, then the Stored Procedure is aborted in an ordinary way, i.e. any settings that occurred up to this point are removed again.

A set of volatile tables that will be used under all circumstances is created.

The number of rows in the table is counted and the figure is stored for further use.

A compression threshold check is executed next. Only Tables with a current number of rows of more than 5 times the Number of AMPs on the system are allowed to be subject to compression. With such a check, even an undifferentiated run of DWHPRO TITHONIZER over all tables of a database is possible. Users do not need to be informed over which tables are currently suitable for compression in size terms. A run over a table that is too small in that sense is only a matter of seconds and it will cause the Stored Procedure to terminate in an orderly manner, i.e. without errors or leftover intermediate objects and with a closed Query Band.

5.2. Chapter 1: Compressible Column Candidate Analysis

A volatile Table VT_COMP_CAND keeps the most frequent 255 values (or less, depending on the column content) of every compression candidate column. In order to populate this table, the necessary insert statements are generated first and stored in an intermediate Volatile Table INSERTITUDE.

A compression candidate column is one that fulfills all of the following requirements:

1. Is has a data type that can be subject to compression under the given Teradata System version. *(mandatory setting).*
2. It is not part of the Primary Key of the table *(mandatory setting).*
3. In is not used for Partitioning the table *(mandatory setting).*
4. It is not used for Secondary Indexing *(discretionary setting).*
5. It is not named exactly like a Teradata reserved word under the given Teradata System version *(discretionary setting).*
6. The column length is not more than 1000 *(discretionary setting).*

For some of the above information and some Teradata versions, a detour in the form of extra preparations instead of a direct dbc query is necessary.

Apart from requirements for the column as a whole, there are also restrictions on the value level, namely

1. When a character value in UNICODE, it must contain only characters that can be translated to LATIN
2. When a character value, it must not contain any of the following characters:

   (,), +, ?, ‘, ´, `.

These restrictions where introduced based on the experiences of polyglot environments with dominant codes and texts in English and German, where the occurrence of these tokens was the result of failed character translation or typing errors. In environments that depart from the typical character set expectations for English and German, the filter has to be adapted accordingly.

VT_COMP_CAND is filled with the most frequent 255 values, along with the following information:

1. the column data type
2. the column nullability
3. The calculatory value.

   For fixed-length columns, this is the number of Bytes consumed by the definition of the data type itself. For variable length columns, the Byte consumption is the character length of the value, multiplied by the CharType entry in dbc.columns, in order to account for the possible extra Byte consumption incurred by UNICODE character sets.
4. The frequency of occurrence of the value
5. The current column value rank. The most frequent value has rank 1, the second most frequent rank 2, and so on, down to rank 255 at most.
6. A flag indicating whether this value is currently compressed over in the table. A separate chapter of analysis of current compression will retrieve and update back this information at a later point.

Given a filled volatile table, a “Gravel Filter” is applied next. Gravel refers to tiny stones or stone splinters. In this context, gravel means single outliers of low frequency or various values with low frequency each. The filter is designed to prevent a net negative effect of compression for small tables by falsely concentrating on a large number of infrequent values over many columns, when not compressing over such a demographic landscape would in fact be the better choice.

The gravel filter condition states that in order for a value to be allowed for compression, the following condition must hold:

AMPNR * LN(rowcount) > col_calc_Value * Col_frequency.

AMPNR: The number of AMPs the system the SP works in has.

LN(rowcount): The natural logarithm of the number of rows in the table.

col_calc_Value: The Number of Bytes a value consumes

Col_frequency: The Number of times the value occurs in the table.

The AMPNR is a system constant. Cet. par., the more AMPs a system has, the higher the "gravel" threshold, This reflects the fact that it costs one time the table header space for compressed values for every AMP. More AMPs means more of these costs.

LN(rowcount) sets the threshold relatively high for very small tables that barely made it over the first hard limit of 5*AMPNR, while it becomes a very mild condition for moderately large tables and turns close to being insignificant for very large tables. It ensures that only a handful of very frequent values will be allowed for compression for small tables, whereas the threshold is not raised linearly as the table grows. Cet. par., the more rows, the higher the threshold, but at a rapidly decreasing rate relative to the row increase.

Cet.Par., the higher the calculatory value, the more likely the value will make it over the threshold. Bulky values that consume a lot of Bytes make it over the hump more easily than slim ones, other things being equal.

Cet. Par., the more frequent a value, the more likely the value will make it over the threshold.

Values below the "gravel filter" are exempt from further processing, i.e. they will not be compressed over.

5.3. Chapter 2: Current Compression Analysis

This chapter determines whether the target table has at least one column compressed and what the current compression means in terms of Presence Bytes used and Permanent Space consumed.

The current compression will, of course, only be analyzed if it exists. If not, this chapter is skipped. A volatile table VT_COMP_VALUES_IS is filled in a similar way as the VT_COMP_CAND table, only for the values that are currently subject to compression.

Any value that is found to be compressed over currently and that is also a candidate for the compression as of now, is marked back into VT_COMP_CAND.

5.4. Chapter 3: Space Consumption Calculation

First, the number of nullable columns and the total bits and full Presence Bytes consumed by nullability is calculated. Also, any current extra Presence Byte consumption by existing compression is determined. Estimates of the Perm Space consumption if uncompressed and, if it applies, under given compression are calculated and stored temporarily.

5.5. Chapter 4: Optimal Compression Determination

A new Volatile table, VT_COMP_LEVEL_SVG, summarizes the total of compression costs and benefits per column and level. A level is a number of values that can additionally be fit into compression by the consumption of one more Presence Bit. Ordered by rank, value 1, values 2,3, then values 4 to 7, then values 8 to 15, and so on, are summarized to levels.

Then, the dominant path for compression is determined.

The dominant path is the sequence of compression levels over columns, ordered by the SVG_PER_HDR_COST_RATIO, compression level and column. The SVG_PER_HDR_COST_RATIO is defined as the compression savings divided by the cost of storing the set of values that are part of the level in the table header, all expressed in Bytes.

First in line is one level1 compression of a column that has the highest Byte savings per table header byte cost incurred. Second in line is either level 2 of the same column or level 1 of another one, whichever has the higher savings per table header cost ratio and not necessarily the one with the absolutely highest byte savings.

The dominant path definition is table-header cost orientated in the sense that the compressed values that fit in to the table header limit shall be maximized, with preference to those levels of columns that cover larger parts of the column value distribution.

When the dominant path is determined, the entire set of columns and levels it put in a ranking order. Now, it might be necessary to cut the optimal path back to the last level that saves more bytes than it costs in the header over all AMPs along the path. No compression beyond this break-even point will be proposed. It might be that the entire path or only a small fraction passes this break-even test. This depends on the table demography, data types, character sets, nullability, and technical constraints to compression.

The point where the (remaining) dominant path ends could still be less than optimal from a Presence bit point of view. Therefore, it is tested if cutting back the dominant path so many steps back until the last opened Presence Bit is used up is a net benefit or loss, i.e. if saving one entire Presence Byte is a greater space saving than the forgone few more compressions would have saved. If so, the dominant path is cut back further to this local Presence Byte break-even-point.

With the final version of the dominant path, all the compression that shall be implemented has been found. It remains now to provide this set of values in a way that an ALTER TABLE statement can be created easily.

5.6. Chapter 5: The Optimal Compression Result

A permanent table TITHONOS is either created or the contents of the last run for that target table removed, if TITHONOS is already in place.

TITHONOS has the following structure:

```
CREATE SET TABLE '||STGDB||'.TITHONOS,NO FALLBACK ,
                    NO BEFORE JOURNAL,
                    NO AFTER JOURNAL,
                    CHECKSUM = DEFAULT
                    (
                     DatabaseName VARCHAR(30) CHARACTER SET LATIN NOT CASESPECIFIC NOT NULL,
                     TableName VARCHAR(30) CHARACTER SET LATIN NOT CASESPECIFIC NOT NULL,
                     ColumnName VARCHAR(30) CHARACTER SET LATIN NOT CASESPECIFIC NOT NULL,
                     COMPRESSIONLIST VARCHAR(30000) CHARACTER SET UNICODE NOT
CASESPECIFIC )
                   PRIMARY INDEX ( DatabaseName, TableName, ColumnName );
```

It is to be created at the Staging database of the respective environment. If the database environment knows no such thing as a Staging database, choose one where all the non-permanent, transient objects are typically stored.

Via a loop, the entire list of individual values to be compressed over for every column is built up along the dominant path, comma-separated and capsuled in single quotes or other accompanying formatting information if the data type of the column requires it.

Now that the optimal compression is determined, a space estimation for the Perm Space in Bytes under this compression can be calculated.

5.7. Chapter 6: Cleanup and Logging

The core tasks of the Stored Procedure are done by now. It remains to close and remove everything created for the purpose of intermediate storage and to close the Query Band in the end. A summarizing log entry gathers the key space estimates. If wished, they can be queried out of the logging table. Note that these values are only estimations that can and will depart from the empirical results of applying a given compression.

6. The DWHPRO TITHONIZER Result: Applying the Compression Proposal to the Table

The Stored Procedure returns a list of compressed values ready for application in an ALTER TABLE statement, stored in the staging table TITHONOS.

In order to apply the compression proposal to a table, use the following query to generate one(!) alter table statement that contains all changes to all columns of the table that are compression candidates in principle, either as a compression or as a decompression, if the optimal solution according to DWHPRO TITHONIZER does not foresee compression for the column. Both initial as well as recompressions can be executed with one and the same statement. The statement is created such that it will return the command lines in the correct order for one or several tables at once.

```
-- CREATE THE ALTER TABLE STATEMENT(s)
SELECT
databasename, tablename,ROWRANK  ,THESTATEMENT
FROM
(         SELECT
          DISTINCT databasename, tablename,
          CAST( 0 AS INTEGER) AS ROWRANK,
          CAST( 'ALTER TABLE '||trim(databasename)||'.'||trim(tablename) AS VARCHAR(9000)) AS
THESTATEMENT
          FROM
          XXX_STAGE.TITHONOS
          WHERE DATABASENAME='TARGET_DATABASE'
          AND  TABLENAME = 'TARGET_TABLE'
) A
UNION ALL
SELECT
 COALESCE(CP.databasename, CO.databasename) as databasename,
 COALESCE(CP.tablename,CO.tablename) as tablename ,
 COALESCE( CP.ROWRANK,
CAST(  ( RANK() OVER (PARTITION BY CO.DATABASENAME, CO.TABLENAME ORDER BY
CO.COLUMNNAME) ) AS INTEGER) *10000
 ) AS ROWRANK,
COALESCE(CP.THESTATEMENT,'ADD '||trim(CO.ColumnName)||' NO COMPRESS,'  ) AS THESTATEMENT
FROM
(         select * from DBC.COLUMNS
          WHERE DATABASENAME='TARGET_DATABASE'
          AND TABLENAME = 'TARGET_TABLE'
          and columnname NOT IN (TECHNICAL_COLUMN_LIST)
          and ColumnType NOT IN ('BO','CO','PD','PM','PS','PT','PZ','UT') -- TD 13 also: 'SZ','TS'
          AND ColumnLength <=1000
          AND (databasename, TABLENAME, columnname) NOT IN
	(SELECT DatabaseName, TableName, ColumnName FROM DBC.Indices )
) CO
LEFT OUTER JOIN
(
	SELECT databasename, tablename, COLUMNNAME,
	CAST( (CASE WHEN TRIM(THESTATEMENT) LIKE '%;' THEN ROWRANK*10000000 ELSE
ROWRANK END ) AS INTEGER) AS ROWRANK, THESTATEMENT
	FROM
	(
		SELECT databasename, tablename,COLUMNNAME, ROWRANK,
		CASE WHEN ROWRANK<ROWCNT THEN THESTATEMENT||',' ELSE THESTATEMENT||';'
END AS THESTATEMENT
		FROM
		(
			SELECT databasename, tablename,COLUMNNAME,
			RANK() OVER (PARTITION BY DATABASENAME, TABLENAME ORDER BY
COLUMNNAME) AS ROWRANK,
			COUNT(*) OVER (PARTITION BY DATABASENAME, TABLENAME) AS ROWCNT,
			'ADD '||trim(ColumnName)||
			CASE WHEN COMPRESSIONLIST ='*§N§U§L§L§*' THEN ' COMPRESS' ELSE  '
COMPRESS(' END
			||CAST(TRIM( CASE
						WHEN COMPRESSIONLIST = '*§N§U§L§L§*'
THEN ''
						WHEN COMPRESSIONLIST LIKE
'*§N§U§L§L§*%' THEN SUBSTR(TRIM( COMPRESSIONLIST),13,9999)
						ELSE COMPRESSIONLIST
						END ) AS VARCHAR(9000))||
			CASE WHEN COMPRESSIONLIST ='*§N§U§L§L§*' THEN '' ELSE ') ' END AS
			THESTATEMENT
			FROM
			XXX_STAGE.TITHONOS
			WHERE DATABASENAME='TARGET_DATABASE'
			AND  TABLENAME = 'TARGET_TABLE'
		) A
	) B
) CP
ON	CO.DATABASENAME = CP.DATABASENAME
AND CO.tablename = CP.tablename
AND CO.COLUMNNAME = CP.COLUMNNAME
ORDER BY 1,2,3;
```