Chapter 14 – Things to Consider

“Never advise anyone to go to war or to marry.”

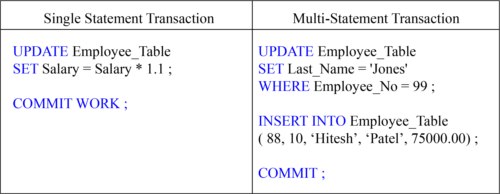
-Spanish Proverb

*Teradata Mode vs. ANSI Mode*

|  |  |
| --- | --- |
| Teradata Mode | ANSI Mode |
| Transactions are implied (implicit). Explicit Transactions must use the Begin Transaction (BT) and End Transaction (ET) commands. | All Transactions are explicit and require a COMMIT WORK or COMMIT statement to commit work. |
| All table creates default to SET Tables. | All table creates default to Multiset. |
| Data comparison is NOT case specific. | Data comparison is case specific. |
| Allows truncation of display data. | No truncation of display data allowed. |

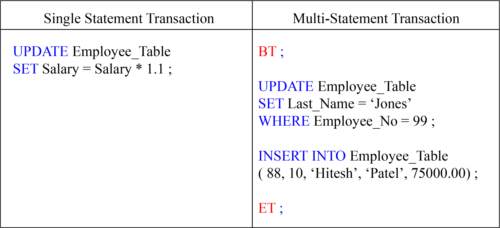
Teradata sessions can operate in one of two modes: ANSI mode and Teradata (BTET) mode. The BTET stands for Begin Transaction End Transaction. The choice of mode affects such things as case sensitivity defaults, collating sequences, data conversions and display functions. All SQL can be done in either mode. Above are the major differences.

*ANSI Mode Transactions*



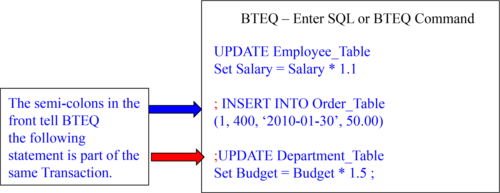
Work is not committed in ANSI mode without the COMMIT or COMMIT WORK statements. If a user logs off without entering the COMMIT statement the work is rolled back. ANSI transactions are always explicit.

*Teradata Mode Transactions*



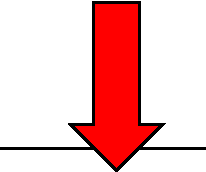
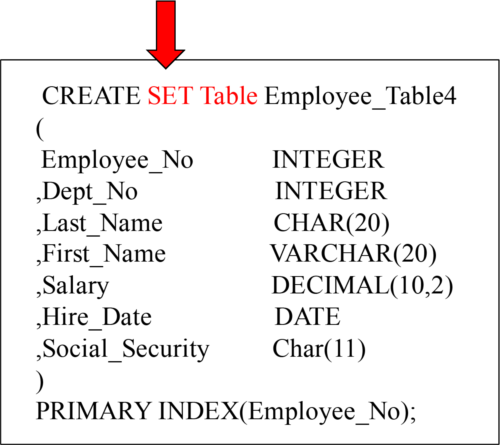
Work is implicitly committed in Teradata mode. If a user wants a Multi-Statement transaction then Begin Transaction (BT) and End Transaction (ET) are used.

*Multi- Statement Transaction in BTEQ*



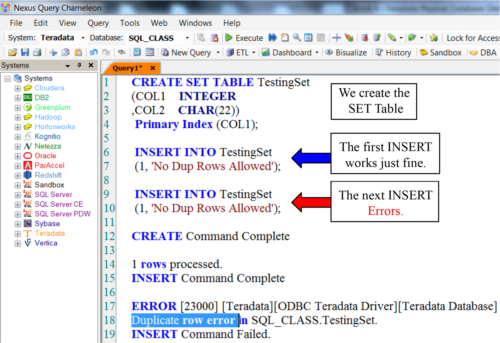
The 3 statements in the example above are considered one transaction in BTEQ because of the placements of the Semi-Colons.

*Teradata Mode Default is a Set Table*



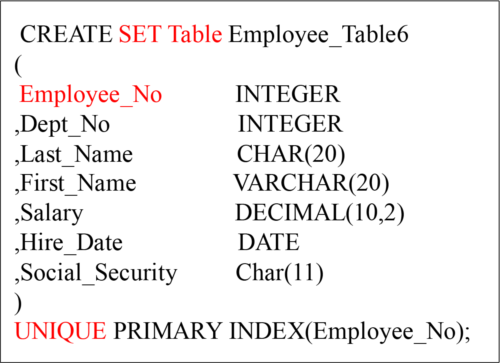
SET TABLE means that Duplicate ROWS are rejected. If your system is in Teradata mode, then SET tables will be the default. You can be in Teradata mode and explicitly define a Multiset table.

*An Example of a Duplicate Row Error*



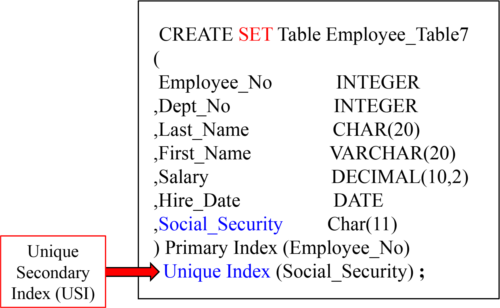
A Set Table will kick out duplicate rows. Every time there is an INSERT or an UPDATE they system has to find all rows with the same value in the Primary Index. Then, the system compares each of those rows, column by column, to see if there is a duplicate row. This can take a lot of time.

*Creating a Set Table with a Unique Primary Index*



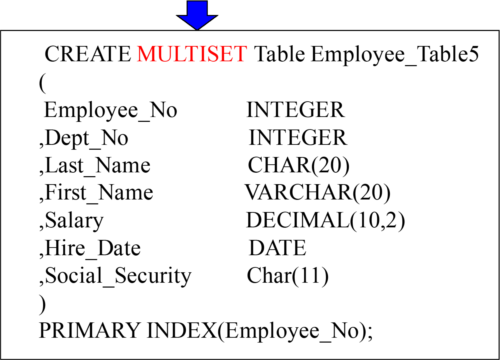
It is very important, when dealing with a SET table, to have either a Unique Primary Index or a Unique Secondary Index. They eliminate the Duplicate Row Check. Because SET tables won’t allow duplicate rows, a “Duplicate Row Check” is done on all new INSERTS or UPDATES. But, if any column has a UNIQUE constraint, then the system knows that no row can be a duplicate because the specific column is UNIQUE. This saves a lot of time. Do your best to stay away from the Duplicate Row Check.

*Creating a Set Table with a Unique Secondary Index*



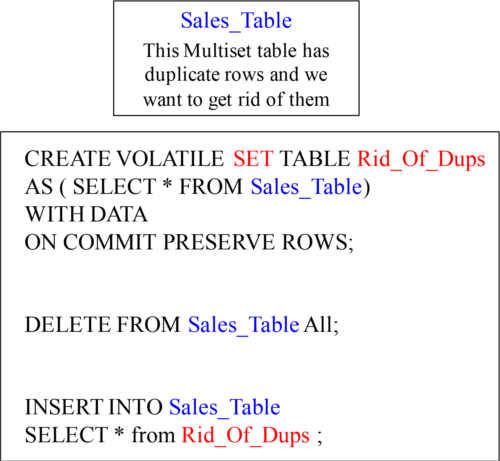
It is very important, when dealing with a SET table, to have either a Unique Primary Index or a Unique Secondary Index. They eliminate the Duplicate Row Check. Here, we have created a UNIQUE Secondary Index (USI), and this will ensure no duplicate Social\_Security values exist. So, the system won’t do the duplicate row check.

*ANSI mode defaults to a Multiset Table*



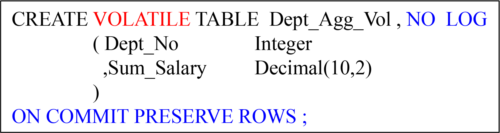
A MULTISET Table means the table will ALLOW duplicate rows. If your system is in ANSI mode, then MULTISET tables will be the default. In either Teradata mode or ANSI mode, you can specifically state (SET or MULTISET) for the table type desired. The problem with Multiset tables is if you have a Non-Unique Primary Index, and accidentally load the table twice, you have duplicate rows. The next page will show you how to correct that situation.

*Using a Volatile Table to Get Rid of Duplicate Rows*



If you have a Multiset table that accidentally gets unwanted duplicate rows, you can use the technique above to get rid of them. We first create a SET Volatile table, and when the data is copied, the duplicate rows are eliminated. Then, we can delete all the rows from the Multiset table and reinsert the rows from the Volatile, and all is good.

*Creating a Volatile Table*

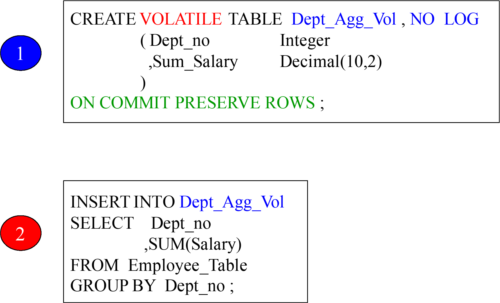


NO Log is the default and it means don’t use the Transient Journal, which gives you Rollback capabilities and better data integrity, but since this is a Volatile Table who cares. No Log is faster when doing Maintenance.

ON COMMIT PRESERVE ROWS **is** NOT the default . You must us these Keywords if you want your data to stay in the Volatile Table after you populate it, otherwise after the load transaction the data is deleted. That is referred to as ON COMMIT DELETE ROWS .

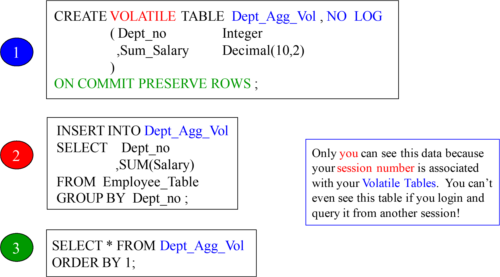
This statement creates a Volatile Table!

*You Populate a Volatile Table with an INSERT/SELECT*



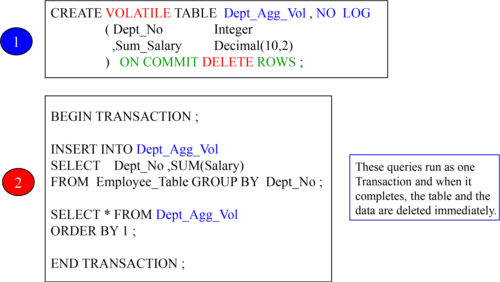
1) A USER Creates a Volatile Table and then 2) populates the Volatile Table with an INSERT/ SELECT Statement. The space to materialize this table comes from the User’s Spool space. Now you can query this table all session long. When the session is logged off, the table and the data are automatically deleted.

*The Three Steps to Use a Volatile Table*



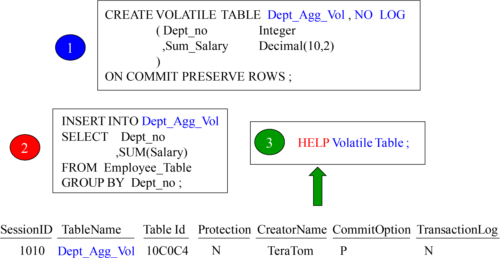
1) A USER Creates a Volatile Table and then 2) populates the Volatile Table with an INSERT/ SELECT Statement, and then 3) Queries it until he/ she logs off.

*Why Would You Use the ON COMMIT DELETE ROWS?*



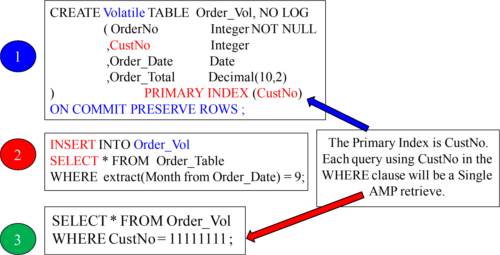
If you want to populate a Volatile Table, and then only run one query, then why not have it go away when you are done? That is what will happen in the above example.

*The HELP Volatile Table Command Shows your Volatiles*



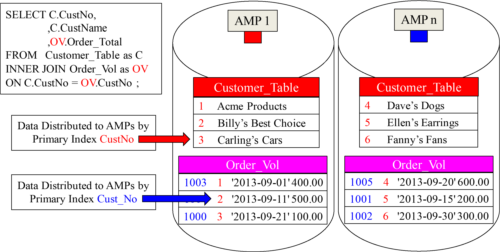
The HELP Volatile Table command above is exactly what you type in. This shows you all the Volatile tables you have materialized in your current session.

*A Volatile Table with a Primary Index*



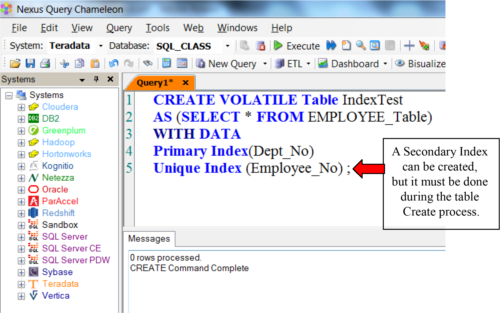
It is a great idea to give your Volatile Table a Primary Index so you can control how it is distributed and the best way you want to query it. In the above example, we knew we would be querying on the column CustNo so we made it the Primary Index. Check out the next page and see how clever our Primary Index of CustNo is for joining tables.

*The Joining of Two Tables Using a Volatile Table*



When Teradata does a join, the matching rows need to be on the same AMP. We gave our Volatile Table a great Primary Index, fully knowing we were going to populate it with September orders and then join it to the Customer\_Table on the join condition of CustNo. Now, no data movement is required. Brilliant!

*Creating a Secondary Index on a Volatile Table*



When initially creating a Volatile table, you can define secondary indexes as part of the CREATE VOLATILE TABLE statement. You cannot add secondary indexes (via CREATE INDEX) to a Volatile table after it has been created. Secondary indexes have to be specified with the initial CREATE VOLATILE TABLE statement. You cannot create a join index or a hash index on a VOLATILE table.

*You Can Collect Statistics on Volatile Tables*

You can Collect Statistics on Volatile Tables. This can help improve query performance. You should consider collecting on:

•All Non -Unique Primary Indexes (NUPI)

•The Unique Primary Index of small tables (less than 1,000 rows per AMP)

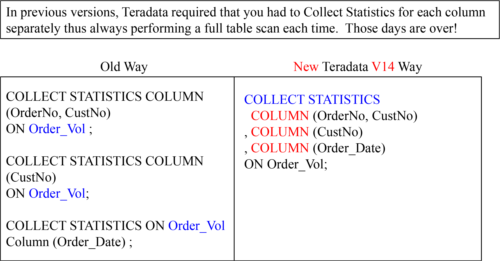
•Columns that frequently appear in WHERE search conditions

•Non-indexed columns used in joins

•Partitioning column of a PPI Table

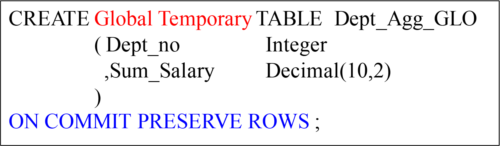
You don’t have to collect statistics on Volatile tables, but sometimes you will find if you are having performance problems that collecting statistics on a volatile table can greatly enhance performance. Above, are some great guidelines for collecting statistics on volatile tables.

*The New Teradata V14 Way to Collect Statistics*



The new way to collect statistics in Teradata V14 is to do it all at the same time. This is a much better strategy. Only a single table scan is required instead of 3 table scans using the old approach. This is an incredible improvement.

*CREATING A Global Temporary Table*

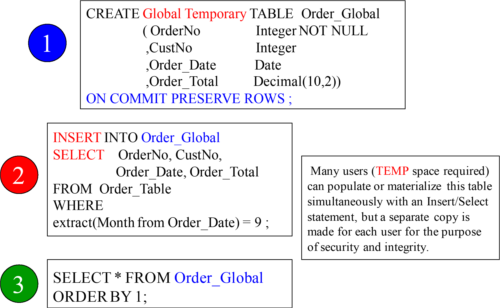


ON COMMIT PRESERVE ROWS is NOT the default . You must us these Keywords if you want your data to stay in the Volatile Table after you populate it, otherwise after the load transaction the data is deleted. That is referred to as ON COMMIT DELETE ROWS.

The Table Definition stays Permanently . When a user logs off the data Inside the Global Temporary Table is deleted , but the definition stays around ready to be populated again.

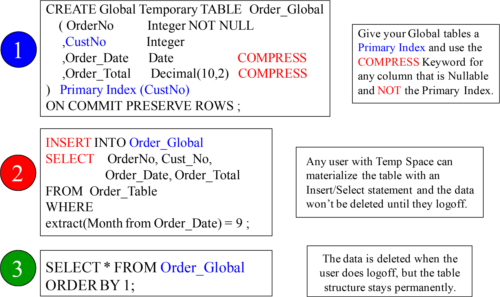
This syntax creates a Global Temporary Table, which is stored in the Data Dictionary of Teradata. A Global Temporary Table survives a Teradata Restart.

*Using a Simple Global Temporary Table*



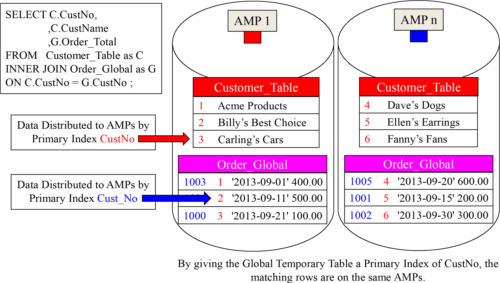
1) A USER Creates a Volatile Table and then 2) populates the Volatile Table with an INSERT/ SELECT Statement, and then 3) Queries it until he/she logs off. All data is deleted when a user logs off, but the table definition stays forever unless dropped.

*Two Brilliant Techniques for Global Temporary Tables*



Give your Global Temporary Tables a Primary Index, and also compress any Nullable column. If a null is present, then Teradata will compress it and save space.

*The Joining of Two Tables Using a Global Temporary Table*



We gave our Global Temporary Table a great Primary Index fully knowing we were going to populate it with September orders, and then join it to the Customer\_Table on the join condition of CustNo. Now, no data movement is required. Brilliant!