

**Teradata Architecture**

An **amp** is instance (virtual processor) of database management data (tables, rows, indices) with their associated data manipulation processes and their data context (Transaction in Progress table, lock information, disk access information).

A **clique** is a set of Teradata nodes that share a common set of disk arrays. Cabling a subset of nodes to the same disk arrays creates a clique.

**The BYNET is responsible for**

Point-to-point communications between nodes and virtual processors

Merging answer sets back to the PE

Making Teradata parallelism possible

**The Parsing Engine is responsible for**

Managing Individual sessions (up to 120)

Parsing and optimizing SQL requests

Dispatching the optimized plan to the AMPs

Sending the answer set response back to the requesting client

**The AMPs is responsible for**

Storing and retrieving rows to and from the disks

Lock Management

Sorting rows and aggregating columns

Join Processing

Output conversions and formatting

Creating answer sets for clients

Disk space management and accounting

**Tables:**

1. SET table - Strictly NO duplicate values [By-Default]

2. MULTISET table - Allows Duplicate Values.

Temporary Tables:

1. Global Temporary Table - supports [Compression in table], when session logged out, Table persists and its record gets Vanished.

2. Volatile Table - When session logged out, TABLE and its records are not available further.

# Teradata Space Management

Teradata is designed in such a fashion, to reduce the DBA’s administrative functions when it comes to space management. Space is configured in the following ways in Teradata system –

**1)      PERMANENT SPACE**

**2)      SPOOL SPACE**

**3)      TEMPORARY SPACE**

**1) PERMANENT SPACE** – Permanent space is where the objects (i.e. – databases, users, tables) are created and stored. PERM space is distributed evenly across all the AMPs. Equal distribution is necessary, because then there is a high percentage that the objects will be shared across all the AMPs, and at the time of data retrieval all AMPs will work parallel to fetch the data.

Unlike other relational databases the Teradata database does not physically defined the PERM space at the time of object creation, instead of that it defines the upper limit for the PERM space and then PERM space is used dynamically by the objects.

**2) SPOOL SPACE** – Spool space is the amount of space on the system that has not been allocated. The primary reason for the SPOOL space is to store intermediate results or queries that are being processed in Teradata. For example, when executing conditional query all the qualifying rows which satisfies the given condition will be store in the SPOOL space for further processing by the query. Any PERM space currently unassigned is available as a SPOOL space.

Defining a SPOOL space limit is not required when Users and Databases are created. But it is highly recommended to define the upper limit of SPOOL space for any object (i.e. users, database, and tables) which you create. Because in case there is no upper limit define for SPOOL space for the object then the processing query for that object might consume all the space in the system and cause “**runaway transaction**”.

3) **TEMP SPACE** – The amount of space used for **Global Temporary Tables** is known as TEMP space. These results remain available to the user until the session is terminated. Tables created in TEMP space will survive a restart. Permanent space not being used for tables is available for TEMP space.

**STATS:** Collect stats is an important concept in Teradata, collect stats gives PE to come up with a plan with least cost for a requested query. Collect stats defines the confidence level of PE in estimating "how many rows it is going to access? How many unique values does a table have, null values etc. and all this info is stored in data dictionary. Once you submit a query in TD the parsing engine checks if the stats are available for the requested table, if it has collected stats earlier PE generates a plan with "high confidence". In absence of collect stats plan will be with "low confidence".

There are 4 kind of locks in Teradata:   
  
**Exclusive -** Exclusive locks are placed when a DDL is fired on the database or table, meaning that the Database object is undergoing structural changes. Concurrent accesses will be blocked.  
  
Compatibility: NONE  
  
**Write -** A Write lock is placed during a DML operation. INSERT, UPDATE and DELETE will trigger a write lock. It may allow users to fire SELECT queries. But, data consistency will not be ensured.  
  
Compatibility: Access Lock - Users not concerned with data consistency. The underlying data may change and the user may get a "Dirty Read"  
  
  
**Read -** This lock happens due to a SELECT access. A Read lock is not compatible with Exclusive or Write locks.  
  
Compatibility: Supports other Read locks and Access Locks  
  
  
**Access -** When a user uses the LOCKING FOR ACCESS phrase. An Access lock allows users to read a database object that is already under write-lock or read-lock. An access lock does not support Exclusive locks. An access lock does not ensure Data Integrity and may lead to "Stale Read"  
  
  
Categorized on Levels, we can have locks at Database, Table or Row-level.  
  
  
**Row Hash Lock:** A Row Hash lock is a 1-AMP operation where the Primary Index is utilized in the WHERE clause of the query.   
  
How it helps: Rather than locking the entire table, Teradata locks only those rows that have the same Row Hash value as generated in the WHERE clause.  
  
Syntax: Locking Row for Access

#### **Practical Example:**

This is the situation faced today:  
  
SQL1: Insert into FACT\_CUST (col1,col2) Select(col1,col2) from WRK\_CUST   
  
SQL2: Select \* from FACT\_CUST  
  
Since, SQL1 was submitted first, there is a Write lock on FACT\_CUST. SQL2 needs a Read Lock on FACT\_CUST. So, it will wait until SQL1 is complete.  
  
  
When inserting or updating rows from a query (Insert/ Select or Update where the primary index is not specified), a Write lock is placed on the table.  
  
If you Insert Values or Update where PIVal = 12345 (via SQL or Load Utility) then a Write lock is placed at the row level.  
  
If you do a select without specifying the PI values, a Read lock is placed at table level. A Read Lock will not read through an existing write lock (and vice versa) so the one who gets in second will be delayed or bounced if NOWAIT is specified.  
  
  
If you put a LOCKING ROW (or Tablename) FOR ACCESS on the Select query, the Select will read through any Write lock at row or table level. (So-called "Dirty Read".) This only applies to select - a Write lock cannot be downgraded to an Access Lock.  
  
To overcome "Stale Read", we can allow read access through Views - put the LOCKING FOR ACCESS clause in the views.  
  
Let me explain the strategy I use:   
  
I have a view that is refreshed every day.  
View definition:   
  
replace view test\_v  
as  
locking row for access   
(  
select \* from test1  
);  
  
Now I load the delta rows into test2. Then once the processing completes and test2 table is ready, refresh the view as:  
  
replace view test\_v  
as  
locking row for access  
(  
select \* from test2  
);  
  
Next step will be to move the contents of test2 into test1:  
  
delete from test1 all;  
insert into test1 select \* from test2;  
  
This will always give consistent data and very little downtime (required only during view refresh)

# Comparison of the Teradata loading utilities

The article contains comparison and main features of the data loading tools provided by Teradata. The tutorial illustrates main features of Teradata Multiload, FastLoad and TPump (Parallel Data Pump) and provides sample real-life uses of those tools.   
  
Scroll down for the sample scripts which illustrate different ways to load a sample fixed-length extract into a Teradata database using FastLoad, MultiLoad and Parallel Data Pump (TPump).

### **Teradata Fast Load**

Main use: to load empty tables at high speed.

The target tables must be empty in order to use FastLoad

Supports inserts only - it is not possible to perform updates or deletes in FastLoad

Although Fastload uses multiple sessions to load the data, only one target table can be processed at a time

Teradata Fastload does not support join indexes, foreign key references in target tables and tables with secondary index defined. It is necessary to drop any of the constraints listed before loading and recreate them afterwards.

The maximum number of concurrent Teradata Fastload tasks can be adjusted by a system administrator.

Fastload runs in two operating modes: Interactive and Batch

Duplicate rows will not be loaded

### **Teradata Multi Load**

Main use: Load, update and delete large tables in Teradata in a bulk mode

Efficient in loading very large tables

Multiple tables can be loaded at a time.

Updates data in a database in a block mode (one physical write can update multiple rows)

Uses table-level locks

Resource consumption: loading at the highest possible throughput

Duplicate rows allowed

## **TERADATA PARALLEL DATA PUMP (TPUMP)**

Main use: to load or update a small amount of target table rows

Sends data to a database as a statement which is much slower than using bulk mode

TPump uses row-level hash locks

Resource consumption: loading speed can be adjusted using a built-in resource consumption management utility. The throughput can be turned down in peak periods.

TPump does not support MULTI-SET tables.

# Teradata FastLoad example

The following script attached below will load a sample fixed-length columns extract into a Teradata database using FastLoad.   
  
Use the following command to run load the ggclients.fastload file using Teradata FastLoad script:

fastload < ggclients.fastload

Contents of a ggclients.fastload script: 

SESSIONS 4;

ERRLIMIT 25;

logon tdpid/username,password;

create table gg\_cli (

wh\_cust\_no integer not null,

cust\_name varchar(200),

bal\_amt decimal(15,3) format ‘ZZZ,ZZ9.999’

)

unique primary index( wh\_cust\_no ) ;

SET RECORD UNFORMATTED;

define

wh\_cust\_no(char(10)), delim1(char(1)),

cust\_name(char(200)), delim2(char(1)),

bal\_amt(char(18)), delim3(char(1))

newlinechar(char(1))

file=insert.input;

SHOW;

BEGIN LOADING gg\_cli errorfiles error\_1, error\_2;

insert into gg\_cli (

:wh\_cust\_no,

:cust\_name,

:bal\_amt

);

END LOADING;

logoff;

# Teradata MultiLoad example

The following script attached below will load a sample fixed-length columns extract into a Teradata database using MultiLoad.   
  
Use the following command to run load the ggclients.mload file using Teradata FastLoad script:

mload < ggclients.mload

Contents of a ggclients.mload mload script: 

.logtable inslogtable;

.logon tdpid/username,password;

create table gg\_cli (

wh\_cust\_no integer not null,

cust\_name varchar(200),

bal\_amt decimal(15,3) format ‘ZZZ,ZZ9.999’

)

unique primary index( wh\_cust\_no ) ;

.BEGIN IMPORT MLOAD tables gg\_cli;

.layout ggclilayout;

.field wh\_cust\_no 1 char(10);

.field cust\_name 12 char(200);

.field bal\_amt 213 char(18);

.dml label insertclidml;

insert into gg\_cli.\*;

.import infile insert.input

format text

layout ggclilayout

apply insertclidml;

.END MLOAD;

.logoff;

# Teradata TPump example

The sample script attached below loads a sample fixed-length columns extract into a Teradata database using Parallel Data Pump - Teradata TPump.   
  
Contents of a ggclients.tpump script:

.logtable tpumplogtable;

.logon tdpid/username,password;

.BEGIN LOAD SESSION 4;

.layout ggclilayout;

.field wh\_cust\_no 1 char(10);

.field cust\_name 12 char(200);

.field bal\_amt 213 char(18);

.dml label insertclidml;

insert into gg\_cli.\*;

.IMPORT INFILE insert.input

layout ggclilayout

apply insertclidml;

.END LOAD;

.logoff;