Questions lead to knowledge
A Teradata Certified Master answers readers’ technical queries.

When MultiLoad is in the acquisition phase, the following actions occur:

1. All sender AWTs are initially set to inactive status.
2. The client sends multiple rows in a message to a sender AWT.
3. The sender AWT that receives rows from the client performs the following steps:
   > Sets its state to active
   > Unpacks the rows in the message and converts them to internal format
   > Sends converted rows to the correct AMP’s receiver AWT, based on hash codes of the rows
   > Sets its state to inactive
   > Sends a response to the client and waits for the next message
4. Meanwhile, receiver AWTs process received rows independently.

A sking questions as a method of self-education has a long history. The first well-known thinker who systematically introduced questioning into Western civilization was the Greek philosopher Socrates. Although Socrates started out poor and illiterate, he established himself by traversing his native Athens, asking questions and absorbing knowledge. Part of Socrates’ contribution was that he recognized the importance of constantly questioning and evaluating ourselves and the world around us.

This column is a compilation of questions I have received and the answers I have provided. And while these are small technical questions, as opposed to the grander questions Socrates may have asked, they carry on the age-old tradition of self-education through asking.

Inactive sessions with MultiLoad jobs
Dear Carrie:
We use the default number of sessions (one per AMP) for MultiLoad jobs, which for us is more than 70 sessions per load. However, many of these sessions show up as inactive through Query Session at any point in time. Are we holding AMP worker tasks (AWTs) for all of these inactive sessions?

—Confused in Auckland

Dear Confused:
Those inactive MultiLoad sessions are indeed holding AWTs. All AWTs required in the acquisition phase will be held until the end of the data transfer from client to database. “Inactive” means the session/AMP is waiting for data from the client at the moment; “active” means the session/AMP is actively processing data from the client.

A particular session may be changing between active and inactive many times during a load job. A client machine (and/or network) may not be powerful enough to keep all sessions/AMPs busy at the same time. When that happens, only a few sessions will be active simultaneously and most will be inactive. Because the client sends data round-robin to all sessions, a different set of sessions/AMPs is likely to be active at a given point.

On each AMP, one sender AWT and one receiver AWT are always allocated and held until the end of the data transfer, regardless of the number of sessions specified. If the number of sessions is high, a subset of sender tasks is likely to be inactive at any point in time. Yet all sender tasks are involved in processing—just not all at the same time.

On the other hand, if you specified max sessions = 5, only five sessions could be used for data transfer, and it is likely they would show as “active” all of the time. But even with a lower number of sessions specified, the sender AWTs of the other 65 sessions (assuming 70 AMPs) will still be allocated and held until the end of the data transfer. They would be used only during initialization and for spawning receiver tasks, but not during data transfer.
Dear Carrie:
What is the Estimated Proc Time in the Database Query Log (DBQL) step table? Is it clock seconds or CPU seconds? And is it the same time as the estimated processing time that appears in steps in the Explain text?

—Classifying in California

Dear Classifying:
Estimated Proc Time comes from the same source whether it’s within DBQL or the Explain text. It represents the Optimizer’s cost estimate for a given query step, or for the entire query. Estimated processing time takes into account CPU, I/O and network (BYNET) estimated costs combined, and self-adjusts for higher-powered hardware. So, technically, it is neither CPU nor clock seconds.

The original purpose of estimated processing time was to allow the Optimizer to compare different costs for different database operations when building a plan. Even today, it really has no absolute meaning except within the Optimizer’s algorithms. However, if good statistics have been collected, estimated processing time can give you a reasonable sense of whether the query is likely to be very short, very long or somewhere in between. Because it usually correlates with the resulting resource requirements of the request, estimated processing time is a popular secondary classification criterion when using Teradata Active System Management.

Review the DBQL detail to determine where to set the estimated processing time so it can be used as a classification criterion for a Workload Definition (WD). What detail you will need to look at will depend on which software release is installed on your platform.

The purpose of analyzing the estimated processing time of previously run queries is to correlate the resource-usage levels of these queries with the estimates they produced. This correlation can increase confidence that when an estimated processing time value is specified for classification purposes, queries with a predictable level of resource-usage demand will classify to that WD. Analyzing DBQL data can help to take the guesswork out of the classification process.

Teradata Database V2R6.x
The following is an example of where to look for the estimated processing times of previously executed queries when you are using Teradata Database V2R6.1 or V2R6.2. The assumption is that Teradata Active System Management has not yet been enabled.

First, consider the actual Explain text shown below, then compare the estimates it produces with the DBQLStepTbl estimates that are illustrated later.

```
EXPLAIN
SELECT *
FROM customer, ordertbl
WHERE c_custkey = o_custkey;
```

Explanation
1) First, we lock a distinct TPCD."pseudo table" for read on a RowHash to prevent global deadlock for TPCD.ordertbl.
2) Next, we lock a distinct TPCD."pseudo table" for read on a RowHash to prevent global deadlock for TPCD.customer.
3) We lock TPCD.ordertbl for read, and we lock TPCD.customer for read.
4) We do an all-AMPS RETRIEVE step from TPCD.ordertbl by way of an all-rows scan with no residual conditions into Spool 2 (all_amps), which is redistributed by hash code to all AMPs. Then we do a SORT to order Spool 2 by row hash. The result spool file will not be cached in memory. The size of Spool 2 is estimated with high confidence to be 5,974,999 rows. The estimated time for this step is 40.29 seconds.
5) We do an all-AMPS JOIN step from TPCD.customer by way of a RowHash match scan with no residual conditions, which is joined to Spool 2 (LastUse by way of a RowHash match scan. TPCD.customer and Spool 2 are joined using a merge join, with a join condition of ((TPCD.customer.C_CUSTKEY = O_CUSTKEY)). The result goes into Spool 1 (group_amps), which is built locally on the AMPs. The result spool file will not be cached in memory. The size of Spool 1 is estimated with low confidence to be 5,974,999 rows. The estimated time for this step is 18.77 seconds.
6) Finally, we send out an END TRANSACTION step to all AMPs involved in processing the request.

Prior to Teradata Database 12.0, data from

**Estimated processing time takes into account CPU, I/O and network (BYNET) estimated costs combined, and self-adjusts for higher-powered hardware. So, technically, estimated processing time is neither CPU nor clock seconds.**
the DBQL step table (DBC.DBQLStepTbl) will be required to understand the estimated processing time for a query that has already run. The "EstProcTime" column values for each step will need to be summed to get the total estimated time. In this example, the sum of EstProcTime values for the retrieve and the join step is the same value as in the final step of the Explain above: 59.06 seconds. (See table 1.)

Teradata Database 12.0
The following illustrates a Teradata Database 12.0 example. This test was run on a similar but somewhat different hardware platform from the one illustrated in table 1. The plan and estimates slightly differ.

EXPLAIN
SELECT *
FROM customer, ordertbl
WHERE c_custkey = o_custkey;

Explanation
1) First, we lock a distinct TPCD."pseudo table" for read on a RowHash to prevent global deadlock for TPCD.ordertbl.
2) Next, we lock a distinct TPCD."pseudo table" for read on a RowHash to prevent global deadlock for TPCD.customer.
3) We lock TPCD.ordertbl for read, and we lock TPCD.customer for read.
4) We do an all-AMPS RETRIEVE step from TPCD.customer by way of an all-rows scan with no residual conditions into Spool 2 (all_amps) fanned out into 18 hash join partitions, which is built locally on the AMPs. The size of Spool 2 is estimated with high confidence to be 600,000 rows (72,000,000 bytes). The estimated time for this step is 0.99 seconds.
5) We do an all-AMPS RETRIEVE step from TPCD.ordertbl by way of an all-rows scan with no residual conditions into Spool 3 (all_amps) fanned out into 18 hash join partitions, which is redistributed by the hash code of (TPCD.ordertbl.O_CUSTKEY) to all AMPs. The input table will not be cached in memory, but it is eligible for synchronized scanning. The result spool file will not be cached in memory. The size of Spool 3 is estimated with low confidence to be 5,971,680 rows (573,281,280 bytes). The estimated time for this step is 17.91 seconds.
6) We do an all-AMPS JOIN step from Spool 2 (Last Use) by way of an all-rows scan, which is joined to Spool 3 (Last Use) by way of an all-rows scan. Spool 2 and Spool 3 are joined using a hash join of 18 partitions, with a join condition of ("C.CUSTKEY = O.CUSTKEY"). The result goes into Spool 1 (group_amps), which is built locally on the AMPs. The result spool file will not be cached in memory. The size of Spool 1 is estimated with low confidence to be 5,971,680 rows (1,260,024,480 bytes). The estimated time for this step is 14.79 seconds.
7) Finally, we send out an END TRANSACTION step to all AMPs involved in processing the request.

The contents of Spool 1 are sent back to the user as the result of statement 1. The total estimated time is 33.69 seconds.

With this newer release, the DBQLogTbl has a new column named “EstProcTime” that provides the same estimated processing time that can be seen in the bottom of the Explain. (See table 2).

While there is no longer any need to access the DBQL step table to get the total estimated processing time, the contents of DBQLStepTbl are shown in table 3 for validation purposes. The values in the "EstProcTime" column match to steps in the Explain shown above.
Redistribution buffers

Dear Carrie:

We have two nodes and increased from 10 AMPs per node to 14 AMPs per node on our MP-RAS platform. We set the RedistBufSize setting in the DBS Control record at the maximum of 63, but I am wondering if we shouldn’t be at a smaller buffer size to get better query performance. The default value is 4KB. I’ve read that if you have many AMPs per node, a small redistribution buffer size is generally better.

—Super-sizing in Cincinnati

Dear Super-sizing:

To avoid the overhead of sending lots of little messages across the BYNET, buffers are used to batch up individual rows during the row redistribution process. Both load utilities and queries involve such redistributions, but their approach to outbound buffering is different. (See figure 1.)

In the past, the RedistBufSize setting was used by all types of redistributions, but today it only influences the size of row redistribution buffers used by the load utilities. This is a setting in the DBS Control record that determines the number of kilobytes for the units that make up those buffers.

Some recommendations come with this setting, because if this setting is set high and a large number of AMPs are in the system, it is possible that memory demand could be excessive. When MultiLoad or FastLoad are running, each AMP must set up a separate buffer in memory for every AMP in the configuration, and it must allocate space equal to the RedistBufSize number of kilobytes for each of those AMPs.

RedistBufSize works the same on MP-RAS, Linux and Windows platforms, and the default value, 4KB, is the same across all platforms.

On your platform each AMP would require 4KB * 28 AMPs = 112KB just for its redistribution buffer when the default value of 4KB is being used. When you change this setting to 63, each AMP will require 63KB per AMP, totaling 1,764KB system-wide. That’s approaching 2MB of memory per AMP to support a single load utility’s redistribution buffering needs. On small or moderate configurations, this may not put pressure on memory resource. But if you were to double the number of AMPs, for example, the buffer size per AMP will double as well.

Node-level buffering is used for query row redistributions. Node buffering on MP-RAS requires that each AMP set up a 32KB buffer for every node in the system when a query is doing row redistribution. On your platform, each AMP would have a buffer with its size calculated as 32KB * 2 nodes = 64KB. As a side comment, this 32KB-per-node buffer doubles to 64KB on 64-bit platforms, to take advantage of the larger memory available on those platforms.

Bottom line: While you may be getting some improved throughput on your loads, increasing the default RedistBufSize to the maximum is not helping to deliver better query performance.

![Figure 1: AMP-level redistribution for load utilities versus node-level redistributions for queries](image)
**Priority prior to classification**

**Dear Carrie:**
I am a new user of Teradata Active System Management. I know that query priority is determined by the Workload Definition (WD) to which the query classifies, but can you explain to me what happens before the query gets classified to a WD? Specifically, since query classification to a WD doesn’t happen until the final module in the parsing engine (PE) (namely, the dispatcher), and since classification determines priority, what priority controls query optimization and other processing that happen upstream of classification?

—*Active in Atlanta*

**Dear Active:**
Typically, a query is part of an ongoing session that has completed one or more queries at an earlier time. The WD used by the most recent query in a session is remembered. That WD will control the next query in the session until such time as the new query has been successfully classified. (See figure 2.) All PE processing that is required before query classification takes place will be under the control of the session’s previous WD.

If the previous query was demoted to another WD before it completed, the original classification WD will be used for the next-up query’s PE work, as a demotion is considered an exception. Whatever allocation group (AG) the previous query’s WD maps to will be responsible for all processing prior to AMP execution. Under normal circumstances this should make very little difference, since PE CPU is generally a very small percentage of total query CPU time.

If the session has just logged on and this is the first query, the PE processing will be under the control of an AG already set up in Teradata Active System Management named “M.” If a tactical application is composed of queries that are intended to have a very high priority, the initial query in the session will be running at a lower priority during its parsing and optimizing phases. However, once the initial query within that session is completed, both the PE and AMP processing will be controlled by the same high priority—assuming the session’s queries consistently classify to the same, high-priority WD and the session remains logged on.

**Figure 2** Which Workload Definition controls pre-AMP processing?

The part of the query processing that takes place in the parsing engine (PE) before query classification occurs will be under the control of the Workload Definition (WD) into which the previous query in that session was classified.