

# Go ahead and ask

A Teradata Certified Master answers readers' technical queries.



**Carrie Ballinger**  
Senior Technical Consultant  
Teradata Certified Master V2R5

I'm often asked technical questions defensively, almost apologetically. I've been on the asking end a lot myself, and I used to have that same tendency to couch my questions with self-deprecation. But lately I've been using my granddaughter as a role model. She's five years old, and while she asks a fair amount of what you and I would consider "naive" questions, I've never once heard her apologize for asking, or act like it wasn't her right to know.

It's a natural thing to want to understand, and problem solving often involves reaching out to others for help. This column is a collection of questions I've received and answers I've provided that support that healthy interaction. Going forward, let's all remember our childhood roots. And the next time you have a question, simply ask.

## Rewriting data blocks during updates

**Dear Carrie:**

As I remember it, a row/block is rewritten to a new location every time there is an update. Is the before image of the row that is being updated visible to a query using an access lock as long as the update has not been committed?

—Blocked in Brisbane

**Dear Blocked:**

First, updated rows don't always require a rewrite. Rows can grow and shrink inside a data block to some degree without necessitating a completely new block being written. If the degree of change is less than a sector (512 bytes), then the block is not rewritten to a different location. If the change makes the block grow more than a sector, the block is relocated.

Now let's consider what happens if you are reading a data block being updated and your query uses an access lock. Keep in mind that access locking is not the same as the repeatable read isolation level. If, at the time of the access locking query, the data block is about to change, but it has not yet been changed, then the access locking query will

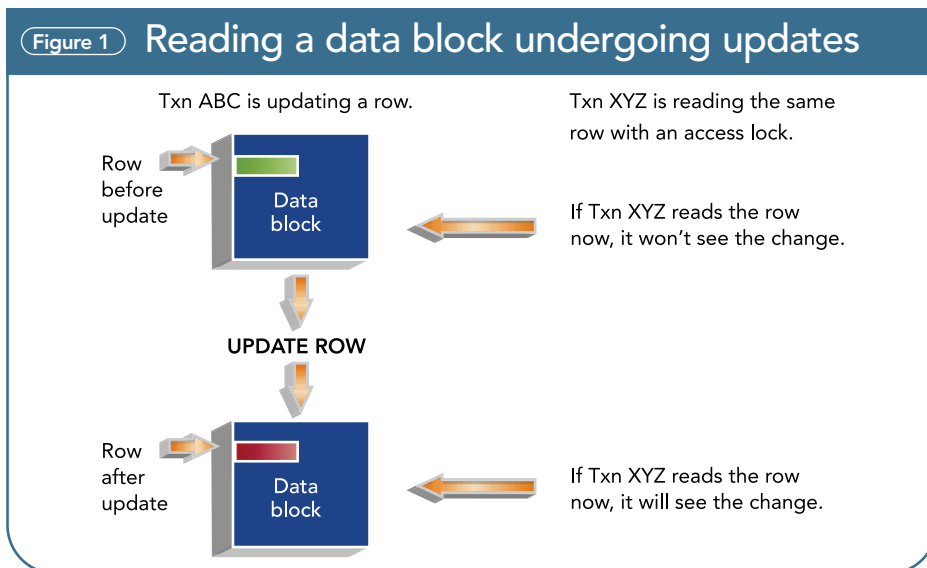
see the previous view of that data block. If the data block has just been changed, even if the transaction has not yet committed, the access locking query sees the change. (See figure 1, below.)

Access locking simply means you can read through the write lock. But you will see the block in whatever state it is currently in at the time of the access locking read. You might see the before image, you might see the after image. That's why it's called a dirty read.

## Holding AMP worker tasks while waiting for a lock

**Dear Carrie:**

Does a query hold an AMP worker task (AWT) while waiting for a table-level lock to be set? In other words, what happens if



A query reads a row undergoing a change with an access lock.

the query requests a lock but can't get it right away?

—*Rather Not Wait*

**Dear Rather Not:**

An enhancement was made in Teradata Database V2R5 so that an AWT will not be held by the query when table-level locks are being requested. All table-level locks that a query requires are set in one step (called a multi-lock, or MLK, step) near the top of the plan. Once this lock step is submitted on each AMP to the respective lock managers, the AWT that initiated the lock request is released—regardless of whether the locks are granted immediately or not.

Any incomplete lock request is queued and handed over to the Lock Manager along with whatever information is needed to reply. The last granted lock request triggers the Lock Manager to send a success response to the MLK step initiator.

Waiting for a row-hash level lock is a different story. This more granular type of lock is always set within the query step where the database processing takes place. An AWT will be held if a row-hash lock is blocked by a lock originating in another session.

**Tuning parameters for cylinder reads**

**Dear Carrie:**

Are there tuning parameters that would help to increase the chances of cylinder reads and help the performance of large scan operations?

—*Always Tuning*

**Dear Tuning:**

A cylinder read allows operations, such as table scans, to perform more efficiently by reading one cylinder of data blocks at a time. The AMP will decide at query execution time whether or not to issue a cylinder read, and there are several factors that can influence its use. (See figure 2, below.)

Because a cylslot (a location in memory reserved for a cylinder read) must be available at the moment your query needs it, the number of cylinder slots per AMP that are defined is one factor in determining cylinder read success. That number of cylslots is a tunable parameter and can be from two to 40 slots per AMP, with a current default of four.

But there are trade-offs. The space for the cylslots comes out of the FSG cache memory, which is shared by all AMPs on the node. The fewer AMPs/node you have defined, the

more cylslots you can support for the same amount of memory. If the percent of the FSG cache dedicated to cylslots is too high, it could limit the memory available to other active work, reducing throughput.

The amount of table data per AMP is another ingredient. If your table doesn't have at least one cylinder of data on the AMP, a cylinder read won't be attempted. And for a given cylinder to be read, the cylinder-resident data blocks must exceed a break-even point of six data blocks. Consequently, this feature helps most when reading tables that have a large number of blocks per AMP.

High concurrency can force queries processing a cylinder read to share CPU with a greater number of other processes. As a result, the query may take longer to process its cylinder of data and hold on to its cylslot longer. Object throttles, because they control the level of concurrency, can help such queries to use their cylslots more efficiently, freeing them up sooner for other queries to use. Just a reminder, because they transfer more data, cylinder reads take more time to perform, and can cause queries doing block reads that are caught behind them to wait longer to complete their I/O.

**Global temp tables and tactical queries**

**Dear Carrie:**

We've had to resort to global temporary tables (GTTs) with our tactical queries to get good performance on our fact table. Are there any known issues using global or volatile temp tables with tactical applications?

—*Searching in the South*

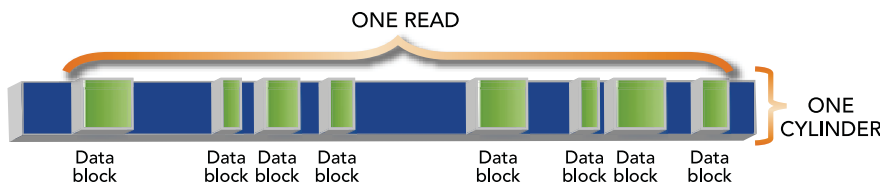
**Dear Searching:**

A GTT is a structure that is defined once in the data dictionary; any number of sessions can insert into and reference their own local copy of the table structure. Volatile temp tables (VTT), by way of contrast, are created and used only within the context of a single session.

**Figure 2** When a scanned table becomes a candidate for a cylinder read

A cylinder's worth of data blocks may be read as one physical read:

- > If there is an available cylslot on the AMP.
- > If the cylinder contains at least six data blocks for the table.



A cylinder read reduces I/Os by reading one cylinder of data blocks at a time.

Be careful of using GTTs with tactical queries because the plans that they produce will not be able to be re-used by queries from other sessions. This is true even when the SQL that references them is in a macro and you have parameterized the SQL correctly.

Each session will have a different table ID for the GTT, and that table ID is not known until the query is parsed. Without all table IDs in the query being consistent across all executions of the query, a generalized, re-usable plan cannot be cached.

In addition, there is slightly more overhead when GTTs execute than would be the case with VTTs. The DBC.Next table must be processed by the first query in a session that is using a GTT to get the table ID. Since DBC.Next has only one row, access from multiple sessions will be single-threaded. VTTs, however, have no need to go to DBC.Next table to get a table ID, as they use Spool ID instead, and Spool ID is available locally.

If you would like more information on how to code for cacheable plans, be sure to read my column in the next issue of *Teradata Magazine* Vol. 7 No. 2, as I'll answer a reader's question and provide coding examples on this topic.

## Local vs. Global Aggregation

Dear Carrie:

Is there any special benefit to aggregation when the primary index (PI) of a table is the same as the GROUP BY columns? And if so, does this extend to non-unique secondary indexes (NUSIs) and join indexes?

—Querying in Quebec

Dear Querying:

You will get a plan that uses "local" aggregation when the query's GROUP BY columns are the same as the table's PI. Local aggregation eliminates row redistribution and the need for additional memory for a global aggregation buffer, so it's always more efficient.

In figure 3, Query 1 only requires local aggregation because the PI of the Txn table (txn\_key) is the same as the GROUP BY column (txn\_key).

If the GROUP BY columns are *not* on PI columns of the base table or join columns from the previous step, then global aggregation has to be performed. This is illustrated by Query 2 in figure 1.

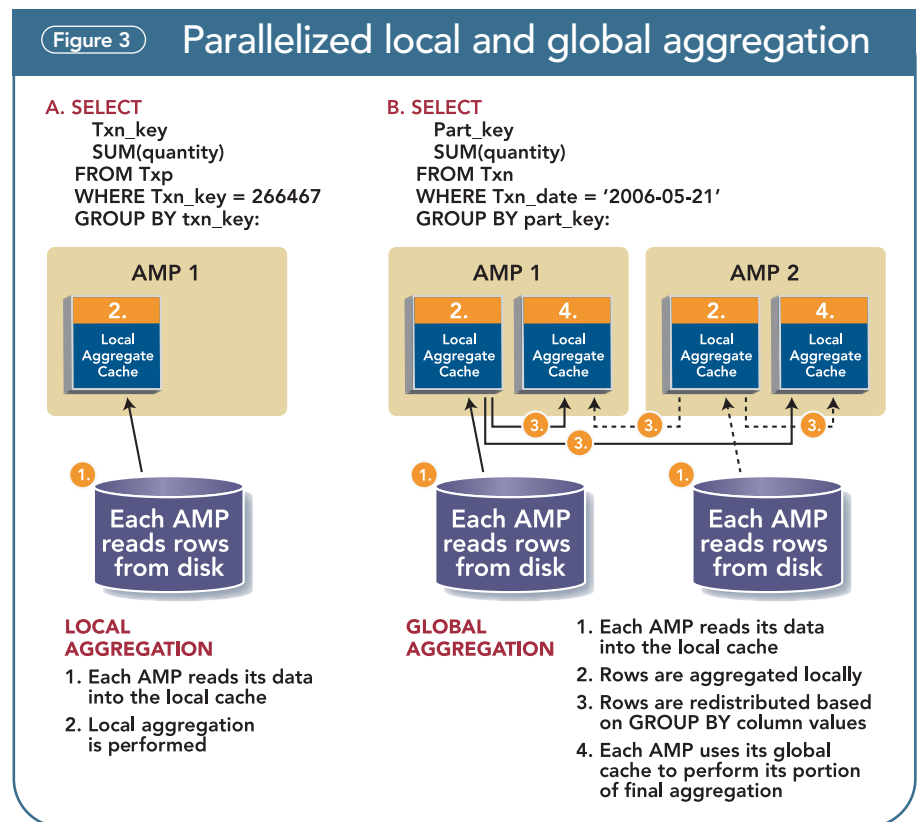
During global aggregation, each AMP will aggregate its rows into its local aggregate cache. When all the rows have been aggregated, the entries will be sorted and redistributed based on hashing their GROUP BY values. The AMP that receives the redistributed entries places them into its global aggregate cache.

While each AMP could have entries in its local cache for all possible GROUP BY values, it will only have to work on a subset of GROUP BY values in its global cache—the

ones that hash to that AMP. It is possible that the other AMPs will send many duplicate entries; these can be consolidated. Global aggregation was designed so that the preliminary subtotals can be done locally on each AMP, while the final grouping effort is shared in parallel across all AMPs in the system.

Global aggregation requires two AMP worker tasks (AWTs) on each AMP: one to send locally aggregated rows and one to receive such rows from other AMPs. In addition, 2MB of memory per AMP will be reserved at the start of the step: 1MB for the local aggregation buffer and 1MB for the global aggregation buffer. Local aggregation only requires a single AWT and only a single cache on each AMP.

Local aggregation would not apply to a NUSI because the base table rows are not physically located based on the hash of the



More efficient local aggregation will be performed if the GROUP BY columns match the primary index of the table being aggregated.

NUSI, but rather on the hash of the base table's PI. Because this is a NUSI (a local structure), some index values could appear on multiple AMPs, so there has to be redistribution and global aggregation when you GROUP BY a NUSI.

A join index will work the same as a base table. If the GROUP BY columns are the same as the join index NUPI, you would get local aggregation, assuming the join index is selected for use and that it fully covers the query's needs.

## BASE TABLE DEFINITION

```
CREATE TABLE ADW.LINEITEM
(
  L_ORDERKEY INTEGER NOT NULL,
  L_PARTKEY INTEGER NOT NULL,
  L_LINENUMBER INTEGER NOT NULL,
  L_QUANTITY DECIMAL(15,2) NOT NULL,
  L_EXTENDEDPRICE DECIMAL(15,2) NOT NULL,
  L_SHIPDATE DATE FORMAT 'yyyy-mm-dd' NOT NULL)
PRIMARY INDEX ( L_ORDERKEY );
```

## LOCAL AGGREGATION PLAN

```
explain
select
  l_orderkey
  , sum(l_quantity)
from lineitem
where l_shipdate = '1992-03-15'
group by l_orderkey;
```

### Explanation

- 1) First, we lock a distinct ADW."pseudo table" for read on a RowHash to prevent global deadlock for ADW.lineitem.
- 2) Next, we lock ADW.lineitem for read.
- 3) We do an all-AMPs SUM step to aggregate from ADW.lineitem by way of an all-rows scan with a condition of ("ADW.lineitem.L\_SHIPDATE = DATE '1992-03-15'"), and the grouping identifier in field 1025. **Aggregate Intermediate Results are computed locally**, then placed in Spool 3. The input table will not be cached in memory, but it is eligible for synchronized scanning. The size of Spool 3 is estimated with low confidence to be 9,914 rows. The estimated time for this step is 6 minutes and 53 seconds.
- 4) We do an all-AMPs RETRIEVE step from Spool 3 (Last Use) by way of an all-rows scan into Spool 1 (group\_amps), which is built locally on the AMPs. The size of Spool 1 is estimated with low confidence to be 9,914 rows. The estimated time for this step is

0.05 seconds.

- 5) 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 6 minutes and 54 seconds.

## GLOBAL AGGREGATION PLAN

```
explain
select
  l_partkey
  , sum(l_quantity)
from lineitem
where l_shipdate = '1992-03-15'
group by l_partkey
```

### Explanation

- 1) First, we lock a distinct ADW."pseudo table" for read on a RowHash to prevent global deadlock for ADW.lineitem.
- 2) Next, we lock ADW.lineitem for read.
- 3) We do an all-AMPs SUM step to aggregate from ADW.lineitem by way of an all-rows scan with a condition of ("ADW.lineitem.L\_SHIPDATE = DATE '1992-03-15'"), and the grouping identifier in field 1026. **Aggregate Intermediate Results are computed globally**, then placed in Spool 3. The input table will not be cached in memory, but it is eligible for synchronized scanning. The size of Spool 3 is estimated with low confidence to be 1,322 rows. The estimated time for this step is 6 minutes and 54 seconds.
- 4) We do an all-AMPs RETRIEVE step from Spool 3 (Last Use) by way of an all-rows scan into Spool 1 (group\_amps), which is built locally on the AMPs. The size of Spool 1 is estimated with low confidence to be 1,322 rows. The estimated time for this step is 0.05 seconds.
- 5) 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 6 minutes and 54 seconds.

If you would like more information on how to code for cacheable plans, be sure to read my column in the next issue of *Teradata Magazine*, Vol. 7 No. 2, as I'll answer a reader's question and provide coding examples on this topic. **T**