QUERY OPTIMIZATION/TUNING

March 2015

AGENDA

- Identifying sub-optimal queries
- What tools to use
- Info we need
- Common problems and possible remedies:
  1. Unmatched Data Types
  2. Product Joins
  3. Resource Intensive Aggregations
  4. Long Running Merge Steps
  5. Large redistributions
  6. Skewed redistributions
  7. NOT IN conditions on NULL'able columns

BEFORE FIXING, WE NEED TO SEE WHETHER IT IS BROKEN…

Identifying sub-optimal queries

· High resource usage – High CPU, High IO consumers, High Spool Usage

· Skewed operations – Skewed operations take the massively parallel architecture of Teradata into a single or few parallel unit operation. Remediating these types of operations can significantly impact runtimes and congestion on the system.

· Long running - service time (Amp time) run long in a job stream or repeated process. This is the typical issue when the user complains about things running too long. Do we have long parsing time, by the way?

· Frequent execution - The frequency of execution should be given consideration. A query executed thousands of times with 1,000 CPU seconds is a better tuning candidate than a 100,000 CPU second query which is only executed once. **Identifying multiple occurrences of the same query can be a challenge.** Properly implemented, **query banding can prove invaluable in identifying multiple occurrences of specific queries**. In the absence of query banding a crude technique is to substring on the first 40 to 100 or so characters of the query text using this sub-stringed result to group on. This certainly is not a perfect method but has been found effective in a number of cases.

BEFORE FIXING, WE NEED TO SEE WHETHER IT IS BROKEN…

TERADATA. LABS

Queries candidates to be optimized/tuned fall mostly in this categories

- Duplicating a very large table on all amps
- Failed with an out of spool error
- Huge redistribution, (not skewed | skewed)
- Long running (merge | hash | product) join
- Long running query, started at 10:15 now 12:35
- Product Join, missing join conditions likely
- Product Join, obsolete statistics likely
- Product Join, OR-ed join condition
- Runaway Product Join, 163,000 CPU seconds and climbing
- Skewed on the NULL Amp; skewed on amp 964; skewed mostly on three amps
- Stuck in merge step, bad primary index likely
- User is asking for 4,895,931,800 rows
- Candidate for a pre-aggregation
- DISTINCT might be better than GROUP BY
- Blocking other users

:

WHAT TOOLS TO USE TO IDENTIFY QUERIES “GILTY AS CHARGED”

·**Real-time monitoring:** Viewpoint (in the past Teradata Manager and /or PMon) is the most often tool used to spot sub-optimal operations while they are in progress and hopefully before they have done too much damage. Real time monitoring is important on systems where there a body of ad hoc users submitting unproven SQL. Viewpoint will allow you to drill into the running query and the currently executing plan down to the currently active step. This tool cannot be effective when trying to capturing information on high frequency, short running operations.

· **DBQL:** Queries which have high CPU usage or those which are highly skewed make the best candidates for tuning/optimization activities. **There are many different approaches for coming up with a prioritized list, the most important measures for prioritization are: CPU consumption, service time and skew.**

WHAT DO WE NEED BEFORE TRYING TO OPTIMIZE/TUNE A QUERY

1. TSET
2. DBQL FOR QUERY
3. DBQL STEP FOR QUERY
4. EXPLAIN (IF POSSIBLE TRY TO GET THE FOLLOWING INFO FROM CUSTOMER OR FROM YOUR PERFECT MATCHED PLAN):

    DIAGNOSTIC VERBOSEEXPLAIN ON FOR SESSION;
    DIAGNOSTIC HELPSTATS ON FOR SESSION;

VERBOSEEXPLAIN adds information on the hash fields use to build intermediate spool files. This can be of significant value when diagnosing skewed redistributions.

HELPSTATS add a list of RECOMMENDED STATISTICS to the end of the explain plan. A word of caution: do not blindly implement this list of recommendations! This list is normally excessive and includes recommendations which may not help and will likely prove costly to capture and maintain.

COMMON PROBLEMS AND POSSIBLE REMEDIES

Unmatched Data Types

A TRANSLATE Function in the Explain plan positively identifies un-matched data types. In cases where the translation is character to numeric, the TRANSLATE key word is seen in the Explain as shown in the text below.

> 4) We do an all-AMPs JOIN step from Spool 4 (Last Use) by way of a RowHash match scan, which is joined to Spool 5 (Last Use) by way of a RowHash match scan. Spool 4 and Spool 5 are left outer joined using a merge join, with a join condition of ("(TRANSFER_DATE <= ENTRD_DT) AND ((**TRANSLATE**((ORIG_ACCT_NUM)USING LATIN_TO_UNICODE)(FLOAT, FORMAT '-9.99999999999999E-999'))…

The concerns with data translations in join conditions are:

· The Translate function, by itself, can be quite expensive when large volumes of data are involved.
· Collected statistics cannot be compared when the data types are not matched: the optimizer may not produce good estimates and usually will not recognize skewed data; costly product joins and skewed processing are the likely outcomes.
· Indexes will be ignored when data types do not match.

The basic remedy for an unmatched data type on a join condition is to materialize correctly matching data types before they participate in a join condition.

COMMON PROBLEMS AND POSSIBLE REMEDIES

Product Joins

- Product joins compare all rows on both sides of the join operation and will, especially when there are a lot of rows on both sides of the join; consume copious quantities of CPU resources. Not all product joins are bad, in fact product joins are a very effective join method when at least one side of the join has a low row count (under 10 is ideal – but up to 100 will typically be acceptable).

- The bad resource intensive product joins are many times caused by the optimizer underestimating the number of rows which will participate. Verify the row counts and compare the estimates with the actual row counts. Primary root causes for resource intensive product joins include:

  · Missing join conditions
  · OR-ed join conditions
  · Unequal join conditions
  · Missing, obsolete, or inappropriate statistics (obsolete statistics are most frequent cause)
  · A large number of predicates which may cause the optimizer to produce very low estimates
  · Unmatched data types will typically result in low estimates and costly join logic
  · Join conditions with complex case logic
  · Use of functions in join conditions: TRIM(), UPPER(), SUBSTRING(), etc

COMMON PROBLEMS AND POSSIBLE REMEDIES

Product Joins (Remedies)

- Common remedies for a bad product join depends on the root cause, in some cases such as unequal join conditions which are legally required, there may be no recourse, other than to limit the scale of the operation and schedule the operation to run during off hours.

- The typical solution for an OR-ed join condition is to split the conditions on two or more SELECT statements and combine the results with a UNION operator.

- It may be necessary to refresh collected statistics on specific columns after scheduled ETL
- operations for date columns where there is a predicate such as BUS_DT>=CURRENT_DATE-2.

- Several predicates which result in a low estimate can many times be addresses by using multicolumn statistics. Use the NOT CASESPECIFIC (NOT CS) attribute as a replacement for UPPER and LOWER functions as shown in these two examples which reduced the runtimes from well over an hour to less than three minutes:

  Original: ON UPPER(a.adj_num) = UPPER(d.operator_id)
  Change to: ON a.adj_num (NOT CS) = d.operator_id (NOT CS)

- Join conditions which include unmatched data types, functions or case logic are typically a sign of a poor physical data modeling or incomplete ETL practices. Workaround is to force matching data types on the join itself

COMMON PROBLEMS AND POSSIBLE REMEDIES

Resource Intensive Aggregations

- Large aggregations especially those with high cardinality (a large number of distinct values), and aggregations containing complex case logic in the GROUP BY clause are typically very resource intensive operations with high CPU and spool file usage.

- Common remedies for resource intensive aggregations depend on the specific operation. In Teradata 13, the optimizer has been enhanced to rewrite the query so this should become less of an issue going forward.

- Materializing the results of the case logic on raw data will sometimes proved more efficient especially if there is high cardinality in the aggregated result set. For these large aggregations it is sometimes beneficial to first create a temporary table of the raw data using the columns specified in the GROUP BY clause as the primary index of the temporary table. Then in a second step perform the aggregation on the temporary table.

COMMON PROBLEMS AND POSSIBLE REMEDIES

Long Running Merge Steps

- Long running merge steps in CREATE TABLE AS, INSERT, and UPDATE operations are generally the result of a bad primary index, secondary indexes on the target table, or update operations on index columns. Look for operations that are hung in a merge step as shown below:

  16) We do an all-AMPs MERGE into DBA_TEMP_TB.u61273_apr_may_reseg from Spool 13 (Last Use).

- Check to see if an abort is in progress on INSERT or UPDATE operations – if being aborted the merge will be stuck in a rollback operation.

- Primary index problems can result in some of the most severe performance problems. Common remedies include:

  1. Change the primary index to be unique or nearly unique will resolve primary index problems. Recommend that the maximum number of rows per distinct index value not exceed 1000 and that the average not exceed 50.
  2. Finding a unique or nearly unique index candidate is sometimes a bit of a trick. Build a surrogate index using IDENTITY columns employing ROW_NUMBER functions
  3. It is sometimes desirable to drop secondary indexes before performing large INSERT or UPDATE operations.

COMMON PROBLEMS AND POSSIBLE REMEDIES

Large redistributions

There are a number of abnormal conditions which can result in large redistributions and row duplications; these will prove costly in both CPU consumption and spool usage. A secondary symptom is that many of these queries will fail with out of spool errors.
The optimizer will duplicate the smaller table of a join operation which is perceived as having a skewed distribution on the join conditions, a costly example is shown here:

> **12) We do an all-AMPs RETRIEVE step from Spool 1 (Last Use) by way of an all-rows scan into Spool 29 (all_amps), which is duplicated on all AMPs. Then we do a SORT to order Spool 29 by row hash. The result spool file will not be cached in memory. The size of Spool 29 is estimated with high confidence to be 718,095,120 rows. …**
>
> **13) We do an all-AMPs JOIN step from Spool 28 (Last Use) by way of a RowHash match scan, which is joined to Spool 29 (Last Use) by way of a RowHash match scan. Spool 28 and Spool 29 are left outer joined using a merge join, with a join condition of ("(SYSTEM_ITEMS_KEY)= INVENTORY_ITEM_NUMBER"). …**

Look for unmatched data types and obsolete or missing statistics. These may cause the optimizer to pick the wrong table to duplicate.

COMMON PROBLEMS AND POSSIBLE REMEDIES

Large redistributions (Remedies)

- Correcting unmatched data types and statistics problems

- Separate the skewed portion of the join from the non-skewed portion in separate select statements using a UNION operator to reassemble the consolidated result set

- A Join Index can be employed to pre-join the costly operation – the costly join will still have to be performed when the join index is built but it is only when the index is built – not for every query

COMMON PROBLEMS AND POSSIBLE REMEDIES

Skewed redistributions

- Join steps which are heavily skewed will usually be skewed on a hash redistribution and will at times fail with an out of spool conditions. A skewed distribution will likely be a result of the optimizer having insufficient information on which to recognize the skewed condition.
- Look downstream to determine what join conditions are in play for the data currently being redistributed. Pay particular attention to statistics on those column(s); are statistics collected and current? If the downstream join operation involves multiple columns, is there a corresponding Multi-column statistics set?

Two diagnostics may prove useful in the diagnostic efforts:

· **VERBOSEEXPLAIN** adds information on the hash fields use to build intermediate spool files. This can be of significant value in figuring out which columns are being used as the primary key for the hash distribution.

· **HELPSTATS** add a list of RECOMMENDED STATISTICS to the end of the explain plan. A word of caution on DIAGNOSTIC HELPSTATS displays: Do not blindly implement this list of recommendations! This list is normally excessive and usually includes recommendations which will not help and prove costly to capture and maintain.

COMMON PROBLEMS AND POSSIBLE REMEDIES

NOT IN conditions on NULL'able columns

- NULL value checking can prove very costly with NOT IN conditions if the NULL attribute exists on columns employed in the NOT IN condition. Look for the words "Skip this join step if null exists" in the explain step:

  We do an all-AMPs JOIN step from Spool 9 (Last Use) by way of an all-rows scan, which is joined to ud151.IL_REWRITE_XREF by way of an all-rows scan with no residual conditions. Spool 9 and ud151.IL_REWRITE_XREF are joined using an exclusion merge join, with a join condition of ("ACCOUNT_NUMBER = ud151.IL_REWRITE_XREF.orig_acct"), and null value information in Spool 6. **Skip this join step if null exists.** The result goes into Spool 13 (all_amps), which is duplicated on all AMPs. Then we do a SORT to order Spool 13 by row hash. The result spool file will not be cached in memory…

- Common remedies include:

  1. Use NOT EXISTS as an alternative to NOT IN.
  2. If there is in fact no NULL data, the source tables should be changed to use NOT NULL attributes.
  3. Adding IS NOT NULL predicates to filter rows with NULL values may solve some of the logical issues with this condition – but usually does not solve the associated performance problems.