Explore the possibilities

A Teradata Certified Master answers readers' technical questions.

Carrie Ballinger
Senior database analyst
Teradata Certified Master

My grandfather was an Arctic explorer, and as part of the U.S. Geological Survey, he mapped Alaska's Arctic coast in the early 1900s. Most land-based exploration was exhausted long ago. But exploring how computers and databases function, and how they can be used to solve unique problems, remains an uncharted opportunity today.

One reason I enjoy putting together this column of questions I have received and answers I have provided is that it supports the spirit of modern-day exploration.

Multi-column statistics on join constraints

Dear Carrie:

I'm joining two tables and I've got statistics collected on my join columns, which are made up of multiple columns. I understand that the order of multi-column statistics is determined by the column order in the base table. The problem is that the columns are in a different sequence on each table. Am I getting the best plan I can from the Optimizer? Should I redefine the column positions in one of the tables? Will the rows hash differently during a redistribution?

—*Orderly Conduct*

Dear Orderly:

Not a problem. You do not need to redefine the column positions in either table. The multiple columns that make up the join columns can have different column sequences within the multi-column statistics and not affect Optimizer decisions. With the different ordering of the columns, you'll get different values in your histogram's detailed intervals, starting with interval one. However, that information is used primarily for single-table selection estimates and will be applied to each table separately.

Join planning looks at data that is contained in interval zero of the histogram, such as the number of distinct values, the high-mode frequency or the number of NULLs. This data will be the same no matter how your multi-column statistics are sequenced.

In a special case, column ordering can influence the joins. If there are multi-column statistics covering the single table selection *and* join columns, it is recommended to keep the single table selection columns as the leading columns. If the query contains a value for the selection column expressed in an equality condition, the optimizer can then use the multi-column statistic histogram to determine the exact correlation between that particular selection value and the join columns. To achieve this, you may need to change the column ordering in the table.

The ordering of the columns within the statistic does not influence the hashing process at all. The hashing algorithm is order-independent: hashrow(10,20) = hashrow(20,10).

©2008 Teradata Corporation

Lowering the exception interval

Dear Carrie:

I'm currently implementing Teradata Active System Management at an insurance company in Europe, and their system has many single-AMP and all-AMP tactical queries. The all-AMP queries take a second or two to run, but we are planning on putting heavy SAS queries on the same platform, so we might occasionally see the queries run longer. How low can we put the exception interval if we want to manage tactical queries that run beyond a few seconds?

—*Playing it Safe*

Dear Safe:

The exception interval is the time between asynchronous checking for exceptions, if you have defined exceptions on your Workload Definitions. The shorter the interval, the sooner an exception can be detected.

I checked with Steve Woodrow from the Performance and Workload Management Center of Excellence in Teradata Professional Services to tap into his experiences implementing Teradata Active System Management at customer sites: "Most exception intervals I've seen are currently in the 30- to 60-second range, although I set up Workload Management at one production site where a 10-second exception interval is being used successfully," he says. "We've avoided setting the interval too low out of concern for the overhead that frequent exception checking could generate, especially on a large system."

Instead of relying on the exception interval to manage tactical queries that run too long, you could define a query milestone that will cause a change in priority based on CPU consumed per node. Teradata Active System Management allows a single query milestone demotion, but only for tactical queries.

Two Orange Books on Teradata Active System Management have an in-depth discussion concerning the trade-offs involved in using query milestones in situations such as yours. My advice is to study the topic before moving in that direction. The books are "Teradata Active System Management Usage Considerations and Best Practices" and "Tips and Techniques Using Workloads with Teradata Active System Management."

The space of NULL values

Dear Carrie:

I came across this explanation of how NULL works from a posting on the Teradata Forum:

It depends on the type of field and whether or not the column is compressed. If a column contains NULL and is compressed (remember that if you specify compression on any value in Teradata the NULLs are also compressed) then the only physical space required is for the presence bit.

If a column contains NULL and is not compressed, then the physical space required is the space that would normally be used by a non-NULL value. As an example, a DATE column will require 4 bytes, a CHAR(10) will require 10 bytes. If the column is a VARCHAR and contains NULL then it requires 0 bytes, but will use 2 bytes for the variable length indicator.

—Dave Wellman
Technical director
Ward Analytics Ltd.

Based on what Dave Wellman said, NULLs take up disk space if you don't have compression on the column. I always thought that a stored NULL only turned on the presence bit but did not take up space, regardless of the compression settings. Am I wrong?

—Second-Opinion Seeker

Dear Seeker:

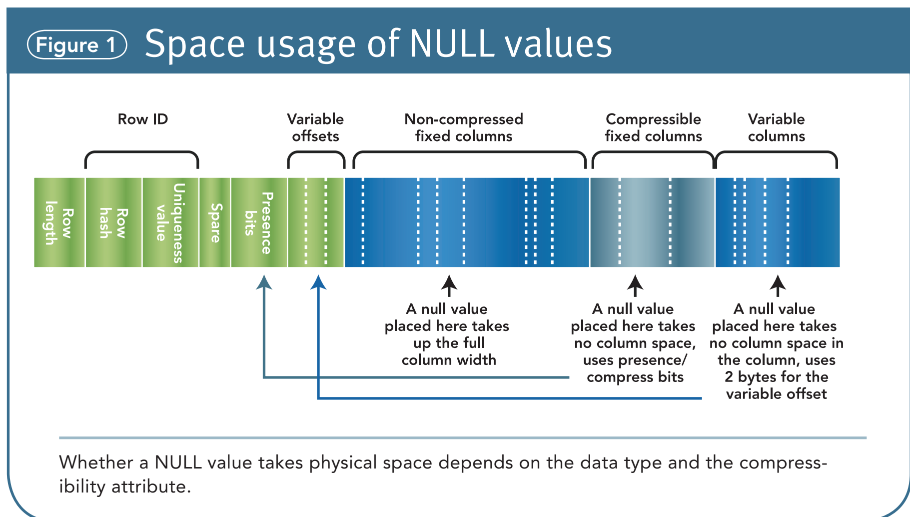
Dave is correct. Fixed-length non-compressible columns always use exactly the same amount of space in the row, regardless of their value. Variable-length columns that are NULL use no space other than the allocated presence bit and 2-byte offset word—plus a 2-byte-per-row overhead, if any variable-length columns are defined. (See table and figure 1.)

In addition, on 64-bit systems, row and field alignment rules also affect space usage. For example, if the row size before compression is 64 bytes, compressing a single integer column won't have any effect on space usage, because the row size will be extended out to a multiple of eight. Row alignment is required to be on an 8-byte boundary with 64-bit platforms (64 bits = 8 bytes).

PHYSICAL SPACE REQUIREMENT CHART

Column specification	NULL is inserted	Non-NULL is inserted
Fixed - NOT NULL	N/A	Size of column (depending on data type)
Fixed - NULLs allowed (default)	Size of column + presence bit	Size of column + presence bit
Fixed, COMPRESS NULLs	Presence bit	Size of column + presence bit
Varchar, NOT NULL	N/A	2 byte offset + value
Varchar, NULLs allowed	2 byte offset + presence bit	2 byte offset + presence bit + value

Figure 1 Space usage of NULL values



Whether a NULL value takes physical space depends on the data type and the compressibility attribute.

Hash index versus single-table join index

Dear Carrie:

Are there any advantages to using a hash index rather than a single-table join index (STJI)? They seem to provide the same functionality.

—Indecisive Indexer

Dear Indexer:

A hash index and a similarly defined STJI are functionally equivalent. However, if you look at the syntax, the hash index silently carries the primary index (PI) of the base table, while the STJI must explicitly define each column it includes, as well as a unique identifier (usually row ID) of the base table row. It is up to the users to decide which syntax they prefer.

I usually recommend using the STJI, because it offers more flexibility. For example, the hash index always gives the compressed format (fixed and repeating groups within each row), while it's an option in the STJI. The STJI explicitly shows all columns that are carried and whether the compressed format is being used, making it easier to understand the structure.

To delve into your question a bit, I ran an informal test to compare tactical query access times between a hash index and a similar STJI to see which offers a better response time.

With the STJI, an uncompressed format (no repeating groups) was used for this test, because the column being indexed was almost unique. In such cases, using the compressed format adds some additional row overhead to manage the compressed format. Consequently, fewer rows will fit into one data block. The compressed format becomes an advantage when there are many base table row IDs that match to the same index value. The following is the specific syntax I used to test the different response times in the hash index and STJI:

Join Index:

```
CREATE JOIN INDEX  PriceJI   AS
SELECT  o_price, o_orderkey , ordertbl.
    ROWID
FROM  ordertbl
PRIMARY INDEX ( o_price );
```

Hash Index:

```
CREATE HASH INDEX  HashPriceJI
(o_price )
ON  ordertbl
BY (o_price )
ORDER BY HASH (o_price );
```

Query 1:

```
USING key1 (char(20))
SELECT o_orderkey
FROM ordertbl
WHERE o_price = :key1;
```

In the test, I compared the total time to execute a string of 100,000 single-AMP queries back to back. In the first test these queries were completely satisfied by the index structure, and each query used a different value for the STJI and hash index PIs. In this case both the STJI and the hash index performed about the same.

I changed the second test so that the index structure no longer fully satisfied the query, and the base table had to be accessed. A similar comparison of total execution time showed the hash index taking about 15% longer than the STJI to perform the 100,000 selects:

Query 2:

```
USING key1 (char(20))
SELECT o_custkey, o_orderkey
FROM ordertbl
WHERE o_price = :key1;
```

A comparison of the Explain text shows that a merge join was used to access the base table rows via the hash index, while the more efficient row ID join was used when the STJI was defined. Both Explains are shown below:

Explain text with the hash index defined:

```
Explanation
   1) First, we do a single-AMP RETRIEVE step
from ADW.HASHPRICEJI by way of the primary
index "ADW.HASHPRICEJI.O_TOTALPRICE =
87925.67" with no residual conditions into
Spool 2 (group_amps), which is redistrib-
uted by hash code to all AMPs. Then we do a
SORT to order Spool 2 by row hash. The size
of Spool 2 is estimated with high confidence
to be 4 rows. The estimated time for this step
is 0.00 seconds.
   2) Next, we do a group-AMPs JOIN step
from ADW.ordertbl by way of a RowHash
match scan with no residual conditions,
which is joined to Spool 2 (Last Use) by way
of a RowHash match scan. ADW.ordertbl
and Spool 2 are joined using a merge join,
with a join condition of ("(Field_1027 =) AND
(Field_1026 = (SUBSTRING((ADW.ordertbl.
RowID) FROM (7) FOR (4 ))))"). The input table
ADW.ordertbl will not be cached in memory,
but it is eligible for synchronized scanning.
The result goes into Spool 1 (group_amps),
which is built locally on that AMP. The size of
Spool 1 is estimated with low confidence to
be 4 rows. The estimated time for this step is
0.05 seconds.
   3) Finally, we send out an END
TRANSACTION step to all AMPs involved in
processing the request.
```

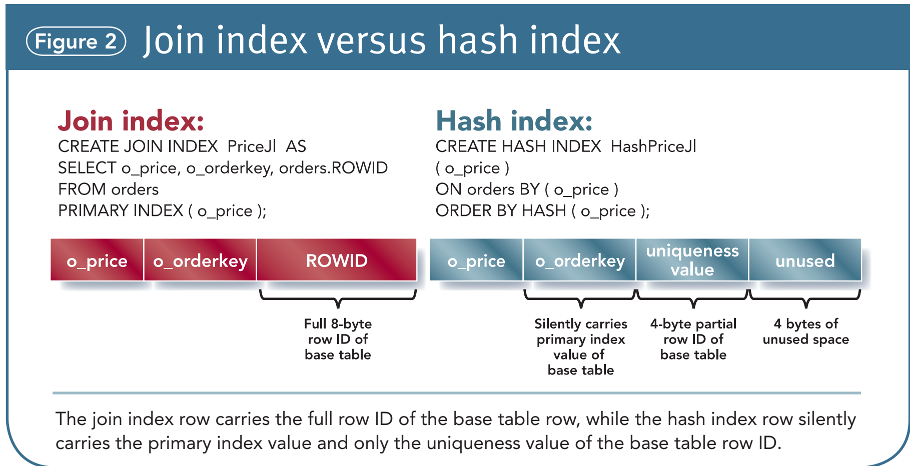
Explain text with the STJI defined:

```
Explanation
   1) First, we do a single-AMP RETRIEVE step
from ADW.PRICEJI by way of the primary index
"ADW.PRICEJI.O_TOTALPRICE = 87925.67" with
no residual conditions into Spool 2 (group_
amps), which is redistributed by hash code to
all AMPs. Then we do a SORT to order Spool 2
by the sort key in spool field1. The input table
will not be cached in memory, but it is eligible
for synchronized scanning. The size of Spool 2
is estimated with high confidence to be 4 rows.
The estimated time for this step is 0.00 seconds.
   2) Next, we do a group-AMPs JOIN step
from Spool 2 (Last Use) by way of an all-rows
```

scan, which is joined to ADW.ordertbl by way of an all-rows scan with no residual conditions. Spool 2 and ADW.ordertbl are joined using a row id join, with a join condition of ("Field_1 = ADW.ordertbl. RowID"). The input table ADW.ordertbl will not be cached in memory. The result goes into Spool 1 (group_amps), which is built locally on the AMPs. The size of Spool 1 is estimated with index join confidence to be 4 rows. The estimated time for this step is 0.05 seconds.

3) Finally, we send out an END TRANSACTION step to all AMPs involved in processing the request.

Figure 2 Join index versus hash index



The join index row carries the full row ID of the base table row, while the hash index row silently carries the primary index value and only the uniqueness value of the base table row ID.

The hash index may be somewhat slower when it is used as a bridge to the base table because, unlike the STJI, it does not carry the complete row ID. It silently carries the PI value along with the uniqueness value associated with the base table. (See figure 2.) So either the full-row ID must be calculated if access to the base table is required, or a slightly slower join approach will be used. To summarize, I like the STJI approach because it:

- Provides a more straightforward syntax
- Creates a small performance advantage if a large number of accesses are going to the base table
- Offers additional capabilities, such as sparseness and aggregation
- Accommodates multi-table coverage
- Makes the compressed format optional, not required