Tuning Hash Operators in ASE 15.0.3 for Optimal Performance

Hashing Out Maximum Query Performance

ASE 15 has various query processing algorithms that use hashing techniques to process particular parts of a query. In this article, we'll look at various ways of tuning the associated hash operators. Additional aspects of the 'plancost' output will also be examined to assist us with our tuning.

By Eric Anderson and Rob Verschoor

ASE 15 introduced various new query processing algorithms, including the so-called hash operators. This article looks at ways of tuning these hash operators.

In a previous article (*ASE 15 Query Tuning with 'plancost'*, published in the March 2012 edition of the *ISUG Technical Journal*), it was described how to use the output of **set statistics plancost**, specifically the estimated number of rows (**er:**) and the actual rows (**r:**), for query tuning purposes. In this article, we will use other aspects of the 'plancost' output for tuning hash operators in ASE 15.

Hashing Operators

ASE 15 has various query processing algorithms that use hashing techniques to process particular parts of a query. For example, you may see a **HashVectAgg** operator in a query plan for a "group by", or a **HashDistinct** for a "select distinct". The performance of hashing operators is usually superior to the classic non-hashing algorithms that were the sole option in ASE 12.x (those are still supported in ASE 15, but the ASE 15 query optimizer will often choose the hash-based operators).

"The performance of hashing operators is usually superior to the classic non-hashing algorithms that were the sole option in ASE 12.x (those are still supported in ASE 15, but the ASE 15 query optimizer will often choose the hash-based operators)."

Spilling to Disk

When hashing operators process data, they create a hash table in memory. We will skip the technical details, but when processing large amounts of data, the hash table used by a hash operator may become so large that some parts of it will temporarily be moved to a worktable to avoid grabbing too much memory. This can be the start of a sequence of events whereby parts of the hash table are continuously shifted around between the hash table and the worktable; this can potentially have a big negative impact on the performance of these parts of the query. This process is also known as "spilling to disk". In the plancost output, spilling

Eric Anderson is a Senior Staff Software Engineer in the ASE query optimizer team.

Rob Verschoor is a Technical Director in the data management evangelism group. He can be reached at robv@sybase.com

is visible as a non-zero number reported for the **p:** property (meaning: actual physical I/O). Here is an example of a hash operator where massive spilling (98115 physical I/Os) had a big impact on performance:

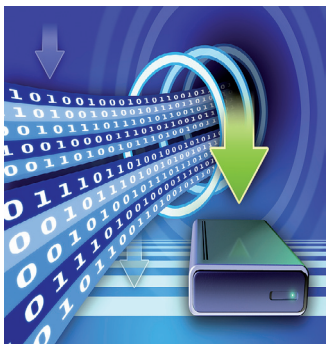
```
HashVectAgg
Sum
(VA = 10)
r:149433 er:122966
l:71142 el:44287
p:98115 ep:44283
```

The 'max buffers per lava operator' Parameter

To achieve optimal query performance, hash operator spilling should be avoided. Unfortunately, in early ASE 15 versions there was little you could do about this problem if it occurred. This has changed in 15.0.3: the configuration parameter **'max buffers per lava operator'** can be used to increase the maximum allowed size of a hash table (with **sp_configure**). The default setting is 2048 (more about the unit below). By increasing this parameter (maximum is 65535), the hash table is allowed to dynamically grow larger for **HashVectAggr** and **HashDistinct** operators, thus hopefully avoiding spilling. For **Sort** and **HashJoin** operators, a larger setting for **'max buffers per lava operator'** allows a larger initial size of the hash table (dynamic growth is currently not supported).

In the context of **'max buffers per lava operator'**, a 'buffer' is actually a data page-sized buffer from the data cache that is used by **tempdb**. Note that this parameter has a server-wide scope, so its setting applies to all queries by all users.

When looking at query plan performance, you should always check whether hash operators perform any physical I/O during execution. If physical I/O is happening (meaning that spilling occurs), then try increasing the setting for **'max buffers per lava operator'** until the physical I/O usage by the hash operator becomes zero. In practice, you may well find that just a small increase of **'max buffers per lava operator'** is already enough to avoid spilling and reduce physical I/O to zero. For the example above, the 98115 physical I/Os disappeared when we increased **'max buffers per lava operator'** to only 2500:



```
HashVectAgg
Sum
(VA = 10)
r:149433 er:122966
l:5 el:44287
p:0 ep:44283
```

Note: the estimated physical I/O as reported by ***ep:*** *is not relevant for this discussion.*

Even when no spilling occurs, the plancost output may still provide clues for improving the performance of hash operators. Consider the following example, which was taken from a problematic query that ran too slow:

```
HashVectAgg
Sum
(VA = 6)
r:8492 er:3
l:7 el:2
p:0 ep:0
```

In this case, the issue is clearly not spilling since **p:** is zero. The problem here is that the optimizer has significantly under-estimated the number of rows that will stream through the **HashVectAgg** operator: it estimated 3 rows, but in reality, there were 8492 – that is about 2800 times more than expected by the optimizer. Such a significant underestimation can have a negative impact on the performance of the hash operator, but in a different way than described above. This has to do with the 'hash buckets'.

Hash Tables

First, some background: A hash table maps a large number of values onto a smaller number of hash buckets: data values are put into the hash table at the bucket location defined by the hash function. The modulo function (or a variant of it) is often used for this. A quick example: in a hash table with

"A hash table maps a large number of values onto a smaller number of hash buckets: data values are put into the hash table at the bucket location defined by the hash function. The modulo function (or a variant of it) is often used for this."

64 hash buckets, the value 87654321 would be placed into the hash table at hash bucket 49 (since 87654321 modulo 64=49). Multiple values that hash to the same bucket are placed in a chain originating at that bucket. Efficient hashing achieves a relatively balanced distribution of the values across all hash buckets, and avoids long hash chains.

Now, back to the ASE query plan: when an ASE hash operator initializes the hash table, it picks the number of 'hash buckets' based on the optimizer estimates. This means that if few rows are expected to stream through the hash operator, as in the example above, correspondingly few hash buckets will be used. When the number of rows during actual execution appears to be much larger than the estimate, all those values will need to be stored in the small number of initially created hash buckets. Unlike the number of buffers used by a hash table (as described above), the number of buckets is fixed.

As a result, in the example above, where the optimizer has dramatically under-estimated the number of rows (3 vs. 8492), very long chains of values need to be created for each hash bucket during query execution. Such long 'hash chains' can negatively impact overall performance since maintaining and accessing such long hash chains can cost a disproportionate amount of CPU time. Indeed, this was the cause of the disappointing performance of the query that contained the hash operator above.

Therefore, if you find a hash operator with such a significantly under-estimated rowcount, it might be a good idea to try updating statistics to see if it causes the estimate to become more realistic, and the query to run faster. You may well find that the estimated-vs-actual rowcount discrepancy in a hash operator originates from an **IndexScan** or **TableScan** operator at a lower level in the query plan. As discussed in the earlier article, such discrepancies can give you clues for which tables you may need to update the statistics.

Recommendations

Here is an ordered list of recommendations for troubleshooting query performance issues.

1. When suspecting query plan problems related to index choices and/or join order, look in the plancost output for **TableScan** and **IndexScan** operators showing big discrepancies between **er:** and **r:**; try updating the statistics to get the estimated rowcount more realistic.

2. When seeing hash operators, especially **HashVectAggr** and **HashDistinct**, report significant physical I/O (indicated by **p:** in the plancost output), try increasing the configuration parameter **'max buffers per lava operator'** (in ASE 15.0.3 or later) until the physical I/O disappears.

3. When experiencing poor performance for queries involving hash operators, look in the plancost output for big discrepancies between **er:** and **r:**. If the estimated number of rows (**er:**) is significantly lower than the actual number of rows (**r:**), this can be the reason for suboptimal performance. Determine which **TableScan** and **IndexScan** operator seems to be generating this estimate, and try to get the estimate more realistic by updating statistics or changing the index.

"As mentioned many times before: having good statistics matters. Since ASE 15 uses statistics in many more ways and places than ASE 12.5 did, it is more important than ever before to pay close attention to the quality of your statistics in ASE 15."

As mentioned many times before: having good statistics matters. Since ASE 15 uses statistics in many more ways and places than ASE 12.5 did, it is more important than ever before to pay close attention to the quality of your statistics in ASE 15.

Conclusion

Knowing how to tune hash operators via accurately configuring the **'max buffers per lava operator'** parameter and by understanding other aspects of the 'plancost' output such as the estimated number of rows that will stream through the hash operators will allow you to be more effective in fine-tuning ASE 15 queries. ■