TECHNICAL BLOG

Performance Troubleshooting Tips for Actian VectorH

This post lists the common areas that should be investigated when trying to improve query performance. Most of the items apply equally well to Actian Vector (single node) as to Actian VectorH (a multi-node cluster on Hadoop) but the focus is on the more complex environment.

Unlike SQL query engines on Hadoop (Hive, Impala, Spark SQL, etc.) VectorH is a true columnar, MPP, RDBMS with full capabilities of SQL, ACID transactions (i.e. support for updates and deletes in place), built-in robust security options, etc. This flexibility allows VectorH to be optimized for complex workloads and environments.

Note that Vector and VectorH are very capable of running queries efficiently without using any of the following techniques, but these techniques will come in handy for demanding workloads and busy Hadoop environments and will allow you to get the best results from your platform.

In order of priority, the main areas that should be investigated are listed below.

Partition your tables

One very important consideration in schema design for any Massively Parallel Processing (MPP) system like VectorH is how to spread data around a cluster so as to balance query execution evenly across all of the available resources. If you do not explicitly partition your tables when they are created, VectorH will by default create non-partitioned tables – but for best performance, you should always partition the largest tables in your database.

In VectorH, tables are either partitioned or not. The following sections describe each type, and where you would use them.

Non-partitioned tables

Non-partitioned tables are not associated with any specific node for data locality. Often, they are stored on the master node and the table data is replicated randomly over two other VectorH nodes by HDFS, but this is not guaranteed and therefore we assume there is no locality anywhere when dealing with non-partitioned tables.

When querying non-partitioned tables, the parallelization step in the engine decides the number of nodes and threads that will participate in the query, and each node will read the (remote) data it needs. When joining a non-partitioned table with a partitioned table, each node will read the complete non-partitioned table to perform the join - non-partitioned tables are sometimes referred to as “replicated tables” for this reason.

If you frequently access non-partitioned tables, they will mostly be resident in the memory buffer cache so the remote data will not influence performance in that case. Non-partitioned tables should generally be the smaller tables by number of rows in the schema (for example, smaller dimension tables).

The advantage of this approach is that you are not limited to only using the specific join key shared by both tables since you will have local access to all data in the non-partitioned table so there is no reshuffling of data around the cluster when you join a dimension table with a fact table with a different key. There is also no overhead from partitioning to deal with, which can be significant when dealing with small amounts of data.

Partitioned tables

The biggest tables in your VectorH cluster should always be partitioned – see the next section for details of how to choose the number of partitions to use.

Partitioned tables are split into multiple “chunks” where a given chunk (partition) has only guaranteed locality on one machine in the cluster. The table data is replicated to two other machines by HDFS as usual, but as the data is randomly spread out, there is no other machine that has the full table stored locally. Medium to large tables (for example, large dimension and fact tables) are more efficiently processed as partitioned tables because each node need only deal with the partitions of the table that are stored locally to itself. The VectorH processing engine is able to efficiently allocate work to the nodes which have the data, so as to minimize the amount of data transfer that takes place during execution of a query.

VectorH is designed to operate efficiently, making best use of cluster resources, when at least one table in any joining query is partitioned. Lack of appropriate partitioning can result in poorer performance; partitioning, therefore, must be considered and implemented at the beginning of the physical database design process. Ingesting data is also much more efficient for partitioned tables; non-partitioned tables are loaded in a single thread, while for partitioned tables we can utilize a thread per partition.

The "location" of a partition refers to the node on which the primary copy of the blocks is stored. HDFS will create block replicas and place them elsewhere in the cluster for resiliency, but VectorH keeps track of the nodes storing the primary blocks for a partition and directs query execution to those nodes first.

VectorH uses a hash based partitioning scheme. The partition key is hashed to determine which partition a row belongs to. Partitions are mapped to physical nodes at startup time to determine the node that manages a given partition.

Partition Keys and Co-Located Joins

Where possible, tables that are joined together should have the same partition key, especially if both of them are partitioned. Doing so will mean the rows of the tables that are to be joined together will reside on the same physical nodes (since they will share the hash value, used by the partition algorithm) and allow the join to be performed on the node itself without requiring the data to be moved around the network. This is referred to as a co-located join, and is much more efficient than a join which requires data to be shuffled around the cluster.

Too Many/Too Few Table Partitions

If a table is large enough to need to be spread around the cluster (i.e. it has more than a few million rows per node, say), then choosing a ‘good’ number of partitions to split a table into is an important consideration for performance.

Partitioned tables should have a partition count that is an integer multiple of the number of nodes so there is the same number of partitions on each data node in the cluster and at least one partition per node.

A small number of partitions per node (e.g. one) means that the table data is contained in a smaller number of files on disk, meaning that file handling workload is reduced. This can be an important consideration when VectorH is one component in a very busy Hadoop system.

VectorH is able to split partition processing across threads (i.e. having multiple threads processing the same partition) so erring on the smaller side of the number of partitions is relatively safe.

However, having more than one partition on a node will increase the parallelism with which data can be loaded or inserted (which includes both CREATE TABLE AS SELECT and INSERT … SELECT operations). While during query execution that does not involve inserts VectorH can split partitions per node on-the-fly to increase parallelism, this increases the workload, so for maximum query performance, multiple partitions per node are beneficial. The system can also decide to merge partition on-the-fly if it thinks it is beneficial to operate with fewer threads than partitions, e.g. under heavy concurrent query load. The number of partitions does not have a one-to-one relationship with the number of execution threads for a query, but a large deviation will result in slower performance. For write queries, the relationship between partitions and parallelism is much stronger.

There are two practical upper limits for the number of partitions per node. The first is that the maximum number of threads used in this way for any given query is governed by the 'max parallelism_level' setting in vectorwise.conf for this database, so there is no point in having more partitions than this value.

Secondly, the recommended upper limit for partitions per node is the number of cores on that node, so never allocate more than one partition per core in the system.

In general, a good default is to use a number of partitions for the largest tables that is equal to a quarter of the total number of cores in the system. This is to allow for multiple queries to execute at the same time without the need for merging partitions. Highly concurrent workloads might warrant queries using fewer threads, which can be achieved either by changing the level of parallelism for that query, or changing the system-wide default level of parallelism for all queries.

As a side note it is worth mentioning that high levels of parallelism for complex queries can result in an overhead in the planning phase of query processing and can overload the network, as sometimes there is an all-thread-to-all-thread communication step. Complex queries (potentially but not necessarily combined with high concurrency) might warrant experimentation with lower levels of parallelism (either per query or system-wide) to combat this.

Another observation made while testing highly intensive workloads is to always ensure some CPU cores have capacity to handle work required by the operating system. If you over-stress a cluster with purely VectorH activities and the operating system is starved of resources to complete tasks like file-system management, then that can have a measurable impact on performance, since VectorH then becomes dependent on a task that never gets sufficient CPU time.

A tool to help calculate this default number of partitions for any given cluster can be found within the Vector Tools repository of Actian's Github account, here: https://github.com/ActianCorp/VectorTools. For a VectorH system, this tool simply does the following calculation:

```
NODES=`cat $II_SYSTEM/ingres/files/hdfs/slaves|wc -l`
CORES=`cat /proc/cpuinfo|grep 'cpu cores'|sort|uniq|cut -d: -f 2`
CPUS=`cat /proc/cpuinfo|grep 'physical id'|sort|uniq|wc -l`
PARTITIONS=`expr $CORES "*" $CPUS "*" $NODES "/" 4`
```

This could be built into a table creation script as follows:

```
export PARTITIONS=`sh partitions.sh`
sql <database> <<EOF
```

```
create table lineitem (
 l_orderkey             INTEGER         not null
,l_partkey              INTEGER         not null
,l_suppkey              INTEGER         not null
,l_linenumber           INTEGER         not null
,l_quantity             DECIMAL(18,2)   not null
,l_extendedprice        DECIMAL(18,2)   not null
,l_discount             DECIMAL(18,2)   not null
,l_tax                  DECIMAL(18,2)   not null
,l_returnflag           CHAR(1)         not null
,l_linestatus           CHAR(1)         not null
,l_shipDATE             DATE            not null
,l_commitDATE           DATE            not null
,l_receiptDATE          DATE            not null
,l_shipinstruct         CHAR(25)        not null
,l_shipmode             CHAR(10)        not null
,l_comment              VARCHAR(44)     not null
)
with partition = (hash on l_orderkey $PARTITIONS partitions);
commit;\p\g
EOF
```

If your table creation schema is in a separate SQL file but you still want to substitute the PARTITIONS value as in the above example, the same thing can be achieved by using the Linux envsubst utility like this:

```
cat create_data_schema.sql | envsubst | sql <database name>
```

In this way, the table creation schema is made dynamic according to the size of the cluster it is being executed on, hence making it flexible across development, test and production clusters which might be of different sizes.

Avoid Data skew

Unbalanced data, where a small number of machines have much more data than most of the others, is known as data skew, and can cause severe performance problems for queries, since the machine with the disproportionate amount of data governs the overall query speed and can become a bottleneck.

Choice of partition key should be such that both data skew (the unevenness of data volume across partitions) and computation skew (the unevenness of processing across partitions) are avoided; typically, data skew causes processing skew, however processing skew can be evident even when there is no data skew.

Highly unique columns are a good choice of partition key; but of no single column is highly unique, then partition keys can even be made up from multiple columns if required.

Beware of using just date columns as partition keys since although these are often good with respect to data skew, queries that specify a given date in the search criteria will likely cause computation skew since all activity for the chosen date will take place on the same machine, rather than be spread around the cluster.

As the key must be hashed to determine the partition for the row, avoid selecting very wide columns as partition keys as these will impact performance. Integer partition keys will provide the best performance. Character string partition keys require more processing and introduce more overhead than numeric partition keys, so avoid these if possible.

You can check for data skew either within SQL, or else from the command line, as follows.

Detecting data skew from within SQL

In the example below, the line item table has around 600m rows. These are spread evenly over 12 partitions.

```
SELECT COUNT(*) FROM lineitem;\g

+----------------------+
|col1                  |
+----------------------+
|             600037902|
+----------------------+


SELECT tid/10000000000000000 AS partition_id,
       COUNT(*) AS num_rows
FROM lineitem
GROUP BY 1
ORDER BY 1;\g

+--------------+---------------+
|partition_id  |num_rows       |
+--------------+---------------+
|             0|       50007922|
|             1|       50012456|
|             2|       49995738|
|             3|       50002838|
|             4|       49998552|
|             5|       49998553|
|             6|       49995271|
|             7|       49993994|
|             8|       50020583|
|             9|       50006224|
|            10|       49998459|
|            11|       50007312|
+--------------+---------------+
```

The row numbers all being roughly the same show that the data is evenly distributed among the 12 partitions. A more sophisticated version of the same check could also be done, that shows the ratio of minimum partition size to the maximum partition size for each partition:

```
SELECT partition_id, num_rows, ((max_num_rows - num_rows)/max_num_rows) * 100.0 as variation
FROM  (
        SELECT partition_id, num_rows, cast(max(num_rows) over () as float8) as max_num_rows
        FROM (
                SELECT   tid/10000000000000000 AS partition_id,
                           count(*) as num_rows
                 FROM partitioned_fact_table
                 GROUP by 1
                ) X
        ) Y
ORDER BY variation DESC
```

Or to boil the whole table down to a single number, this query below gives the worst-case ratio between the smallest partition and the largest one. If the number is less than 2, then data skew in this table is probably ok. If it's much more than this, then data skew is likely to lead to inefficient query processing and an alternative partition key should be sought.

```
SELECT MAX(variation_ratio)
FROM (
    SELECT partition_id, num_rows, (max_num_rows/num_rows) as variation_ratio
    FROM  (
        SELECT partition_id, num_rows, CAST(MAX(num_rows) over () as FLOAT8) as max_num_rows
        FROM (SELECT   tid/10000000000000000 AS partition_id,
                       COUNT(*) as num_rows
              FROM partitioned_fact_table
              GROUP by 1
              ) X
        ) Y
    ) Z
```

Detecting data skew from the Command Line

The -T option of the vw info tool shows details of each physical partition within a table. In the example below, the table is named line item and the partition number is given after the '@' symbol in the table_name column:

```
$ vwinfo -T <database> -t lineitem

+---------------+----------------+----------------+
|schema_name    |table_name      |block_count     |
+---------------+----------------+----------------+
|actian         |lineitem@0      |            4873|
|actian         |lineitem@1      |            4873|
|actian         |lineitem@10     |            4873|
|actian         |lineitem@11     |            4872|
|actian         |lineitem@2      |            4873|
|actian         |lineitem@3      |            4872|
|actian         |lineitem@4      |            4873|
|actian         |lineitem@5      |            4872|
|actian         |lineitem@6      |            4873|
|actian         |lineitem@7      |            4872|
|actian         |lineitem@8      |            4874|
|actian         |lineitem@9      |            4872|
+---------------+----------------+----------------+
```

The details above show that the number of blocks per partition are evenly distributed, so the partition key in this case is a good one.

Missing statistics

Data distribution statistics are essential to the proper creation of a good query plan. In the absence of statistics, the VectorH query optimizer makes certain default assumptions about things like how many rows will match when two tables are joined together. When dealing with larger data sets, it is much better to have real data about the actual distribution of data, rather than to rely on these estimates.

Statistics don't have to be updated every day; instead, they should be updated whenever the distribution of data in a table materially changes. So for example a large data load that appends a lot of new data to the end of what was previously known in the table would qualify as a material change, since the statistics for data distribution in that table would not take into account this new data. This could mean that statistics need to be updated during the day, while users are still using the system.

Statistics can be generated in one of three ways:

- The `optimizedb` command
- The `‘-- stats’` option on the vwload utility, which creates statistics as data is loaded
- The SQL statement ‘`CREATE STATISTICS`’

Statistics can be generated on a table-by-table (or even individual column) basis. In addition, the CREATE STATISTICS command can create column correlation statistics using the WITH COLUMN GROUP clause.

The simplest approach is to generate statistics for all columns and all tables that the current user owns, which can be done as follows:

```
optimizedb <database name>
```

This will take some time for a large schema, so you can apply fine tuning. To speed up the operation and reduce the amount of statistics data generated, generate statistics only for columns involved in joins, restrictions, or grouping, using the `optimizedb □r <table name> -a <column name>` flags, or table/column listing with CREATE STATISTICS.

Statistics should be generated after every significant change to the distribution of data in a database, including on first loading the database, or on subsequent data loads that change the distribution of key values. Out of date statistics is a common reason for queries not performing as expected, or for worsening performance as data is loaded over time.

To find out if your table has statistics or not, simply execute ‘help table <table name>’ in a SQL terminal window. For example:

```
* help table webservice_data \g
Executing . . .

Name:                 webservice_data
Owner:                actian
Created:              23-mar-2016 04:06:54
Location:             ii_database
Type:                 user table
Cache priority:       0
Alter table version:  0
Alter table totwidth: 35
Row width:            35
Number of rows:       3805725
Storage structure:    vectorwise
Duplicate Rows:       allowed
Journaling:           enabled after the next checkpoint
Base table for view:  no
Optimizer statistics: none

Column Information:

Column Name              Type             Length Nulls Defaults Seq Key
date_timestamp           ansidate                  no   value
servicename              varchar              17   no     yes
response_code            integer               4   no     yes
response_time            integer               8   no     yes

Secondary indexes:    none
```

In addition, the query profile tool will show both the estimated and actual number of rows processed in a given step. Large differences (an order of magnitude) between these figures would suggest that statistics are either missing or out of date.

Finally, recent VectorH releases have included an optional configuration setting to improve to statistics heuristics for tables that are appended to on a regular basis, such as time series data. This is done through the use of a configuration setting in `config.dat`, which is enabled through the `iisetres` command as follows:

```
iisetres ii.<hostname>.dbms.*.opf_histhack ON
```

When set ON like this, if the optimizer sees a predicate of the form ‘column = value’ and ‘value’ is larger than the maximum value currently stored in the statistics for that table and column, then the optimizer will use the average repeat factor for the existing values of the column instead of assuming a single row would be returned for this new restriction (which is the normal default assumption).

The intent is to deal with the "missing stats" problem where new data is added to a table but has not yet had statistics gathered for it before a query is run which selects the new data. For example, you updated statistics last night, but today you loaded a new day’s worth of data, and immediately after loading you want to SELECT from that new data. There are no statistics for this new data, so the optimizer will make very different row estimates compared with SELECTing data from yesterday (which does have statistics).

The downside is that for queries where the restriction really does select few or no rows, the optimizer will guess wrong in the ‘too many rows’ direction, so for this reason the setting is currently disabled by default, pending feedback from customer testing against real-world scenarios. This may change in future, but if you find a problem with row estimates of queries due to missing stats for recently loaded data in the meantime, please try this setting and report results to Actian.

Sorting Data

The relational model of processing does not require that data is sorted on disk – instead, an ORDER BY clause is used on a query that needs the data back in a particular sequence.

However, by using what is known as MinMax indexes (maintained automatically within a table’s structure without user intervention), VectorH is able to use ordered data to more efficiently eliminate unnecessary blocks of data from processing and hence speed up query execution, when queries have a WHERE clause or join restriction on a column that the table is sorted on. A separate blog entry deals with this subject in some detail, but in general, if data is loaded in the ‘natural’ order of a column, then through MinMax indexes VectorH will be able to detect this and use it to optimize query execution. This works very well with time series data for instance, or other sources where the new data is naturally arriving in the same order that is often used when querying the data (e.g. searching for specific dates in a time series).

If a different sort order is required, then the DBA can enforce a sort order by using the `create index` command. The index can created then be immediately dropped, and the data thus sorted will be retained, but the sort order will not be actively maintained for data inserted after that point. By immediately dropping the index, the overhead of maintaining the sort order with freshly inserted data is avoided – though of course the sort order will degrade over time for new data, and so a periodic index ‘creation and drop’ cycle may be needed to obtain the best of both worlds.

Even though it is a columnar database, a table can only be sorted on one column at a time though, so choose the sort column carefully after you have examined the system workload.

Optimizing for time series data in the schema design

For time-series use cases where data is continuously loaded, one very common usage pattern is to create a table for a subset of the data over a specific time period (e.g. a day or a month) and have a view that is a UNION ALL over all tables.

The tables that are not actively being loaded into can be kept sorted where the data is sorted when we switch to a new table for loading - this way we only have to ever sort a small subset of the data. This can also be very beneficial for easily purging old data (since it is then easy to simply drop a month's worth of data).

If there are queries with a selection predicate that filters out most of those tables (e.g. a query that only queries data from the last month instead of for the last two years), performance will also benefit from explicitly adding in WHERE clauses to the master View definition that defines the minimum and maximum data value of each of the tables. This way, the SQL optimizer can efficiently filter out those tables very early on during query processing. There is an optional vectorwise.conf configuration setting that can also help in this case: `[engine] zero_tuples_branch_pruning=true`, which is false by default.

Using the Most Appropriate Data Types

Like any database, choosing the right data type for your schema and queries can make a big difference to VectorH's performance, so don't make it a practice to use the maximum column size for convenience. Instead, consider the largest values you are likely to store in a VARCHAR column, for example, and size your columns accordingly.

Because VectorH compresses column data very effectively, creating columns much larger than necessary has minimal impact on the size of data tables. As VARCHAR columns are internally stored as null-terminated strings, the size of the VARCHAR actually has no effect on query processing times. However, it does influence the frontend communication times, as the data is stored as the maximum defined length after it leaves the engine. Note however that storing data that is inherently numeric (IDs, timestamps, etc) as VARCHAR data is very detrimental to the system, as VectorH can process numeric data much more efficiently than character data.

During processing for complex queries, memory is needed to compare the values from tables being joined. If the data types being compared are larger than needed, more memory is needed than is essential, meaning that joins will run out of memory and resort to disk spilling earlier than necessary. Effectively, VectorH is not able to operate on as large data sets as it would be if types were better-matched to the data. Again, using VARCHAR types for inherently numeric data is the biggest culprit here. This will also increase disk storage space, as VectorH can compress numeric data much more efficiently. Hashing algorithms and communication also works much more efficiently on numeric data.

The issue exists not just for VARCHAR columns that should be numeric, but other data types should be sized appropriately too. In general the guidelines are:

- Use 'not null' if the data has no Null values. Allowing for the possibility of Null values takes up more space in each column and slows down processing.

- Where possible use Integer and Decimal data types over floating point or character types.
- Use the smallest integer type possible- don't use 128-bit integers everywhere.
- Use the smallest character string size that is large enough for your data.
- Try to avoid character types as join columns where possible.

Increasing efficiency of "CREATE TABLE AS SELECT"

When you issue a 'Create Table as Select' statement on a large table in VectorH, you need to remember to use the 'with partition' clause to spread that large table around the cluster, for the reasons set out above.

If you don't remember to use the partitions clause, then the table is created with no partitions, and it will be treated as a replicated table and distributed in full to every node that needs it. This usually results in significantly worse performance than if it had been properly partitioned.

To use the partition clause, use a statement of this form:

```
CREATE TABLE customer_sorted AS
SELECT * FROM customer
ORDER BY custid
WITH PARTITION = (HASH ON custid 8 PARTITIONS);
```

Memory Management for Small Changes

VectorH has a patent-pending mechanism for efficiently dealing with many small changes to data, called Positional Delta Trees (PDTs). These also allow the use of update and delete statements on data stored in an append-only file system like HDFS.

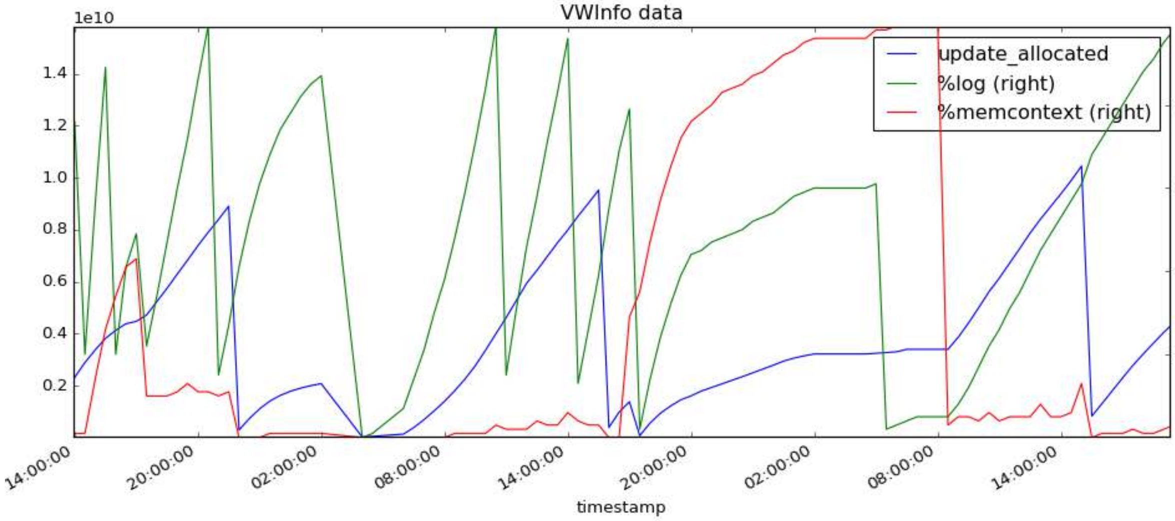
However, if a lot of update, insert or delete statements are being executed, memory usage for the PDT structures can grow quickly. If large amounts of memory are used, the system can get slower to process future changes, and eventually memory will be exhausted. Management of this memory is handled automatically, however the user can also directly issue a 'combine' statement, which will merge the changes from the PDT back into the main table in a process called update propagation. There are a number of triggers that make the system perform this maintenance automatically in the background (such as thresholds for the total memory used for PDTs, or the percentage of rows updated), so this is usually transparent for the user. However, update propagation can consume significant resources (mostly disk I/O), so for optimal performance a DBA might want to explicitly delay automatic activation during the day in favour of manually executing this maintenance during quiet hours, e.g., overnight.

To work out whether this is needed or not, the vwinfo statement will report on memory usage information. Overall organisation of memory used by Vector and VectorH is outlined in this Knowledge Base document (visible to customers with a Support contract): Actian-Vector-Memory-Internals. One of the three key memory areas that is critical to achieving performance is a pool called 'MemContext', and it is primarily this that is reported by vwinfo:

```
$ vwinfo sample
+------------------------------------+--------------------------------------------+
|stat                                |value                                       |
+------------------------------------+--------------------------------------------+
|memory.memcontext_allocated         |104857600                                   |
|memory.memcontext_maximum           |4294967296                                  |
|memory.memcontext_peak              |157286400                                   |
|memory.memcontext_virtual_allocated |104857600                                   |
|memory.memcontext_virtual_maximum   |4398046511104                               |
|memory.memcontext_virtual_peak      |4455363578                                  |
|memory.index_allocated              |6812207                                     |
|memory.update_allocated             |4860                                        |
|memory.update_maximum               |1073741824                                  |
|memory.committed_transactions       |0                                           |
|memory.bufferpool_maximum           |2147483648                                  |
|bm.block_size                       |524288                                      |
|bm.group_size                       |8                                           |
|bm.columnspace_used_blocks          |93                                          |
|bm.bufferpool_total_blocks          |4096                                        |
|bm.bufferpool_free_blocks           |4096                                        |
|bm.bufferpool_used_blocks           |0                                           |
|bm.bufferpool_cached_blocks         |72                                          |
|system.active_sessions              |1                                           |
|system.log_file_size                |825194                                      |
|system.threshold_log_condense       |33554432                                    |
|server.port_number                  |62032                                       |
|server.pid                          |6652                                        |
+------------------------------------+--------------------------------------------+
```

The amount of memory used for PDTs is displayed as memory.update_allocated. This memory is part of the total query execution memory in use (memory.memcontext_allocated). If the latter gets too close to the maximum available memory (memory.memcontext_maximum), the system performance and functionality can degrade. We can see how close the system gets to this point with memory.memcontext_peak: this is the most memory the system has had in use at a single point in time. If memory.update_allocated gets to about 10% of memory.memcontext_maximum, it might be advised to manually tune the update propagation triggers or perform regular maintenance. If memory.update_allocated reaches memory.update_maximum, the system will perform automatic update propagation.

This behavior is illustrated in the graph below from a busy system:



The blue line shows the absolute total PDT memory in use growing and then shrinking as the system automatically triggers update propagation. These peaks and drops were correlated with decreasing system performance that suddenly improved again when PDT memory was freed up.

At one point, an open write transaction had the effect of preventing total query execution memory from being freed, causing memcontext_allocated to reach 100%, as illustrated by the red line (against the right-hand Y axis). At this point, the system ground to a halt for a while until this transaction was closed and normal operations could continue again.

The other thing illustrated here in the green line is the percentage of the Vector LOG file that was used over time, on a system that was continually ingesting large volumes of data. LOG file usage grew until a LOG condense operation was triggered to recover space used by committed transactions. Again, this happens automatically or can be manually triggered on demand. In this case, a LOG condense operation was happening too frequently and was impacting performance, so the LOG file size was increased to reduce this frequency.

The above data was gathered periodically over time by a simple vwinfo script (available on Actian's Github repository here) and then analysed by another tool in the same collection for easier graphing, in this case using Python's matplotlib library via a Jupyter Notebook like this:

```
from matplotlib import pyplot as plt
import pandas as pd
%matplotlib notebook

vw = pd.read_csv('vwinfo.csv', parse_dates=True, index_col='timestamp', \
                 usecols=['timestamp','update_allocated','%log','%memcontext'])
vw.plot(title="VWInfo data", secondary_y=['%log', '%memcontext'])
```

Memory used in transaction management

Due to the snapshot isolation transaction model used by VectorH, each transaction only has the view of the world that existed as at transaction start (at which time we make a snapshot). If a transaction commits changes, we need to figure out if the world changed during the lifetime of the transaction and if those changes conflict with other changes.

To do so, any transaction that changes the database and that started after another running transaction will keep a trace of its changes for this conflict resolution. In practice, this means a long running transaction that runs concurrently with other transactions that change the database will result in a large amount of such traces, just in case this long running transaction needs to know how the world changed while it was running.

The memory occupied by this is recorded as memory.committed_transactions and counts towards the total execution memory as well. If this gets high, e.g., more than 10% of the maximum execution memory, we can run into performance degradation issues. To avoid this, make sure old, unused transactions and connections are committed and closed. It also helps a lot if long running transactions are explicitly marked as read only when we know that they will not change the database, as this indicates to VectorH that it does not need a trace of all changes that occurred while this transaction is running. This can be done through using the JDBC driver (which is read-only by default), or from other client drivers (e.g. the SQL command line) by issuing a ‘`set session isolation readonly`’ statement.

Another important part of the memory overview is memory.index_allocated. In most cases, this is memory consumed by MinMax indexes. If this gets too high (again, >10% of maximum query memory is a good rule of thumb), consult the separate blog article on MinMax indexes for minimising and reducing this.

Another useful Actian Support Knowledge Base article that discusses memory management and minimising memory usage is this one: Why-does-Committed-Transaction-Memory-Keep-Increasing.

Which tables are using PDT memory ?

We can see which tables are using the PDT memory by looking at the output from vwinfo –M:

```
+----------------+--------------------+------------------+------------------+------------------+------------------+
|schema_name     |table_name          |pdt_mem           |pdt_inserts_count |pdt_deletes_count |pdt_modifies_count|
+----------------+--------------------+------------------+------------------+------------------+------------------+
|actian          |breakdown           |                 0|                 0|                 0|                 0|
|actian          |channel_code        |                 0|                 0|                 0|                 0|
|actian          |bravo               |           8450401|             26756|                 0|                 0|
|actian          |insight             |                 0|                 0|                 0|                 0|
|actian          |quotes              |          11861453|             26760|                 0|                 0|
|actian          |rated_areas         |                 0|                 0|                 0|                 0|
|actian          |raw_quote_store     |        2186032162|            147984|                 0|                 0|
|actian          |replication_control |                 0|                 0|                 0|                 0|
+----------------+--------------------+------------------+------------------+------------------+------------------+
```

In this example, there are three tables which are consuming PDT memory, caused by INSERT statement operations. In order to release that memory, either wait for the system to automatically do so, or else simply ‘MODIFY <table> TO COMBINE’ in a process which is known as forcing update propagation.

This Support Knowledge Base article describes update propagation management in much more detail: Update-Propagation-COMBINE-and-Conflict.

Optimizing for Concurrency

VectorH is designed to allow a single query to run using as many parallel execution threads as possible to attain maximum performance. However, perhaps atypically for an MPP system, it also designed to allow for high concurrency with democratic allocation of resources when there is a high number of queries present to the system. VectorH will handle both these situation with "out-of-the-box" settings, but can be tuned to suit the needs of the application (for example if wanting to cater for a higher throughput of heavy duty queries by curtailing the maximum resources any one query can acquire).

The number of concurrent connections (64 by default) that a given VectorH instance will accept is governed by the connect_limit parameter, stored in config.dat and managed through the CBF utility. But there are usually more connections than executing queries, so how are resources allocated among concurrent queries?

By default, VectorH tries to balance single-query and multi-query workloads. The key parameters in balancing this are:

- The number of CPU cores in the VectorH cluster
- The number of threads that a query is able to use
- The number of threads that a query is granted by the system
- The number of queries currently executing in the system

The first query to run in an installation is allowed to use as many threads as it is able to, up to the system-wide limit defined by max_parallelism_level, which defaults to all of the cores in the cluster. In this way, an "expensive" query will run on as many threads as can be executed on all of the CPU cores in the cluster, if the system is otherwise idle.

Starting from the second concurrent query, VectorH tries to evenly share the resources in a cluster among new queries, so that, for example, if there are five queries running in a 100-core system, then each query will be allocated up to 20 threads (if it can use them). This CPU allocation strategy continues (for example, the 20th query will be allocated 5 threads) until each new query receives only one thread per node. So for a cluster with N cores, query number (N/2) + 1 will execute on only one thread per node; for example with 100 cores, the 51st concurrent query will execute with only one thread per node.

However, the query is not physically tied to execution on a specific machine or a number of cores, once the execution plan has been generated. The operating system controls how each query thread is given execution time on the CPUs in the machine, and query affinity is not used.

In fact, by default the number of cores in a system is over-allocated by 25% so the above calculation would actually operate on a maximum of 125 cores, rather than 100.

Although for the first few queries this thread allocation method leads to oversubscription of the number of threads versus available cores, if the query load is fairly constant this method results in a stable situation with properly divided resources as early, resource-heavy queries finish and later queries using fewer threads start. The operating system handles the initial oversubscription, but for concurrency tests the first and last N queries (where N is the level of concurrency) should be discarded from the test results to allow for test startup and shutdown lead and lag time.

If you know from the outset that you want to optimize for concurrent query execution, then you can constrain the early queries to use no more than N threads, even where they would otherwise have used more, in order to more evenly spread the CPU usage out across the queries (since default strategy gives greater weight to the "early" queries within a set). To do this, you can add a clause to the end of SQL statement like this:

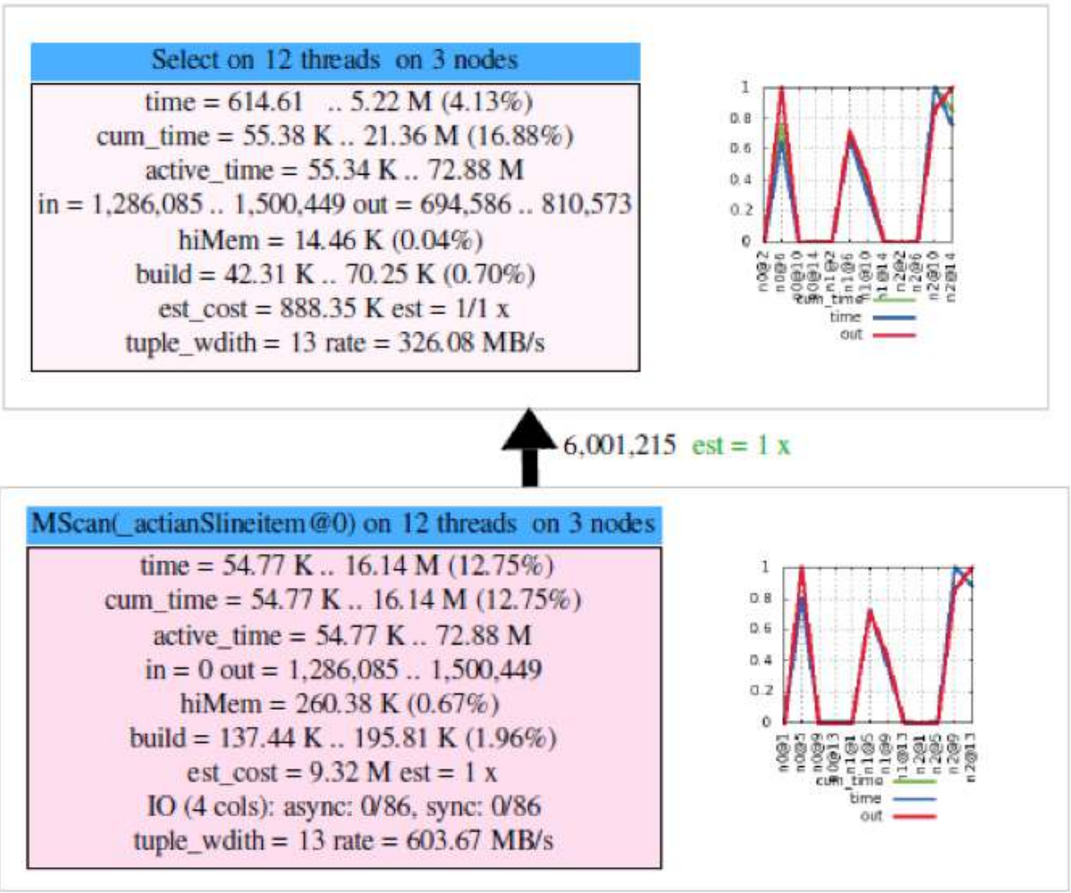
```
SELECT *
FROM lineitem
WITH MAX_PARALLEL 12/g
```

Alternatively you can change the installation default setting to affect all users and all queries, by changing the `max_parallelism_level` value in the vectorwise.conf file and restarting VectorH.

Finally, you can also change the setting for all queries in a session by executing this command at the start of the session:

```
CALL VECTORWISE (SETCONF 'engine, max_parallelism_level, ''12''')
```

For any given query, you can see how many threads it has been allocated by looking at the query profile, a fragment of which (created by the x100profgraph tool) is illustrated here. This query is using twelve threads in total over three nodes, and the spiky shapes of the graphs indicate that some processing skew is being seen:



```
Select on 12 threads on 3 nodes
time = 614.61 .. 5.22 M (4.13%)
cum_time = 55.38 K .. 21.36 M (16.88%)
active_time = 55.34 K .. 72.88 M
in = 1,286,085 .. 1,500,449 out = 694,586 .. 810,573
hiMem = 14.46 K (0.04%)
build = 42.31 K .. 70.25 K (0.70%)
est_cost = 888.35 K est = 1/1 x
tuple_wdith = 13 rate = 326.08 MB/s

6,001,215 est = 1 x

MScan(_actianSlineitem@0) on 12 threads on 3 nodes
time = 54.77 K .. 16.14 M (12.75%)
cum_time = 54.77 K .. 16.14 M (12.75%)
active_time = 54.77 K .. 72.88 M
in = 0 out = 1,286,085 .. 1,500,449
hiMem = 260.38 K (0.67%)
build = 137.44 K .. 195.81 K (1.96%)
est_cost = 9.32 M est = 1 x
IO (4 cols): async: 0/86, sync: 0/86
tuple_wdith = 13 rate = 603.67 MB/s
```

Summary

Vector and VectorH are very capable of running queries efficiently without using any of the techniques and tips above, but the more demanding your workload, in terms of either data volumes, query complexity or user concurrency, the more that applying some of the above tips will allow you to get the best results from your platform.

About Actian

Actian transforms big data into business value for organizations of all sizes, no matter where they are on their analytics journey. We help companies win by empowering them to connect to data of any type, size or location; analyze it quickly wherever it resides; and take immediate action on accurate insights gained to delight their customers, gain competitive advantage, manage risk and find new sources of income. The 21st century software architecture of the Actian Analytics Platform delivers extreme performance on off-the-shelf hardware, overcoming key technical and economic barriers to broad adoption of Big Data. Actian also makes Hadoop enterprise-grade by providing high-performance data enrichment, visual design and SQL analytics in Hadoop without the need for MapReduce skills. Among tens of thousands of organizations using Actian are innovators in the financial services, healthcare, telecommunications, digital media and retail industries. The company is headquartered in Silicon Valley and has offices worldwide. Stay connected with Actian Corporation at www.actian.com or on Facebook, Twitter and LinkedIn.

# # #

Actian, Actian Analytics Platform, Accelerating Big Data 2.0, and Big Data for the Rest of Us are trademarks of Actian Corporation and its subsidiaries. All other trademarks, trade names, service marks, and logos referenced herein belong to their respective companies.

www.actian.com

2300 Geng Rd., Ste. 150, Palo Alto, CA 94303
+1.888.446.4737 [Toll Free] | +1.650.587.5500 [Tel]