VoltDB and Memory Usage

Copyright © 2010 VoltDB, Inc.

Introduction

VoltDB is an in-memory database. Storing data in memory has the advantage of eliminating the performance penalty of disk accesses (among other things). However, with the complex interaction of VoltDB memory usage and how operating systems allocate and deallocate memory, it can be tricky understanding exactly how much memory is being used at any given time. For example, deleting rows of data can result in a temporary increase in memory usage, which seems counterintuitive at first.

The purpose of this whitepaper is to explain how VoltDB used memory in the past, recent and upcoming changes in memory management, the impact of system memory allocation and deallocation functions on your database's memory utilization, and variables available to you to help control memory usage.

How VoltDB Uses Memory

VoltDB's use of memory can be grouped, loosely, into three buckets:

- Persistent
- Semi-persistent
- Temporary

Persistent memory is, as you might expect, the memory used for storing actual database records, including tables, indexes, and views. The larger the volume of data in the database, the more memory required to store it. String columns longer than 63 bytes are not stored in line. Instead they are stored as pointers to the content in a separate string storage area, which is also part of persistent memory.

Semi-persistent memory is used for temporary storage while processing SQL statements and certain system procedures. In particular, semi-persistent memory includes temporary tables and the undo buffer.

- Temporary tables are where data is processed as part of an SQL statement. For example, if you execute an SQL statement like `SELECT * FROM flight WHERE DESTINATION="LAX"`, all of the tuples meeting the selection criteria are copied into temporary tables before being returned to the initiator. If the stored procedure is multi-partitioned, each partition creates a copy of its tuples and the initiator merges the multiple copies.
- The undo buffer is also associated with the execution of SQL statements. Any tuples that are modified or deleted as part of an SQL statement are recorded in the undo buffer until the transaction is committed or rolled back.

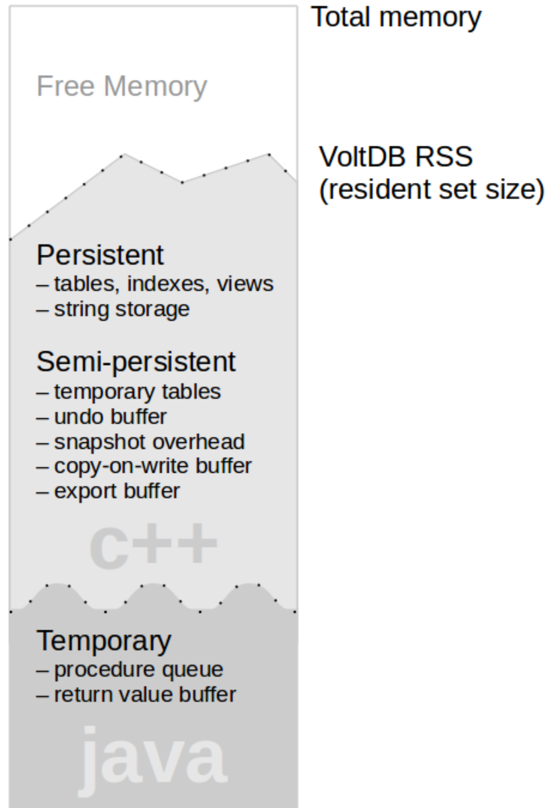
Semi-persistent memory is also used for buffers related to system activities such as snapshots and export. While a snapshot is occurring, a certain amount of memory is required for overhead, as well as copy-on-write buffers. Normally, snapshots are written directly from the tables in memory, thus not requiring additional overhead. However, if snapshots are non-blocking (performed asynchronously while other transactions are executing), any tuples that need to be modified before they are written to the snapshot get copied into semi-persistent memory. This technique is known as "copy-on-write". The consequence is that mixing asynchronous snapshots with frequent deletes and updates will increase the memory usage.

Similarly, when export is enabled, any modifications to exported tables are written to an export buffer in semi-persistent memory until an export receiver retrieves them.

Temporary memory is used by VoltDB to manage the queueing and distribution of procedures to the individual partitions. Temporary memory includes the queue of pending procedure invocations as well as buffers for the return values for the completed procedures (until the client application retrieves them).

Figure 1, "The Three Types of Memory in VoltDB" illustrates how the three types of memory are allocated in VoltDB.

Figure 1. The Three Types of Memory in VoltDB



The sum of the persistent, semi-persistent, and temporary memory is what makes up the total memory (what is referred to as resident set size, or RSS) used by VoltDB on the server.

Actions that Impact Memory Usage

There are a number of actions that impact the amount of memory VoltDB uses during operation. Obviously, the more data that is stored within the partition (including all tables, indexes, and views), the more memory is required for persistent storage. Similarly for snapshotting and export, when these functions are enabled, they require some amount of semi-persistent storage. However, under normal conditions, the memory requirements for snapshotting and export should be relatively consistent over time.

Temporary storage, on the other hand, fluctuates depending on the workload and type of transactions being executed. If the client applications are "firehosing" (sending stored procedure requests faster than the servers can process them), the temporary storage required for pending procedure invocations will grow. Similarly, if the parameters being submitted to the procedures or the data being returned is large in size (up to 2 megabytes per procedure), the buffer for return values can grow significantly.

The nature of the workload also has an impact on the amount of semi-persistent storage. Read-only queries do not require space in the undo buffer. However, complex queries and queries that return large data sets require space

for temporary tables. On the other hand, update and delete queries can take up significant space in the undo buffer, especially when a single transaction (or stored procedure) performs multiple queries, each requiring undo support.

The use of the temporary and semi-persistent storage explains fluctuations that can be seen in overall memory utilization of servers running VoltDB. Although delete operations do eventually release memory used by the persistent storage, they initially require more memory in the undo buffer and for any temporary table operations. Once the entire transaction is complete and committed, the space in persistent storage and undo buffer is freed up. Note, however, that the unused space may not immediately be visible in the system RSS reports. The amount of memory *in use* and the amount of memory *allocated* can vary as a result of the interaction of several different memory management schemes that all come into play.

When VoltDB frees up space in persistent storage, it does not immediately return that memory to the operating system. Instead, it keeps track of unused space, which is then reused the next time a tuple is stored. Two events occurred in the past that would impact the use of memory within VoltDB:

- First, in previous versions (prior to 1.2.1) VoltDB managed memory in blocks. When a delete operation removed a tuple, it may only free up part of a block. The rest of the block may still be in use. The next time a query adds a tuple, VoltDB searches for free space in existing blocks and reuses it, rather than allocate a new block.
- Only when entire blocks of memory were freed did VoltDB actually deallocate the space and return it for use by other processes.

Figure 2. Details of Memory Usage During and After an SQL Statement

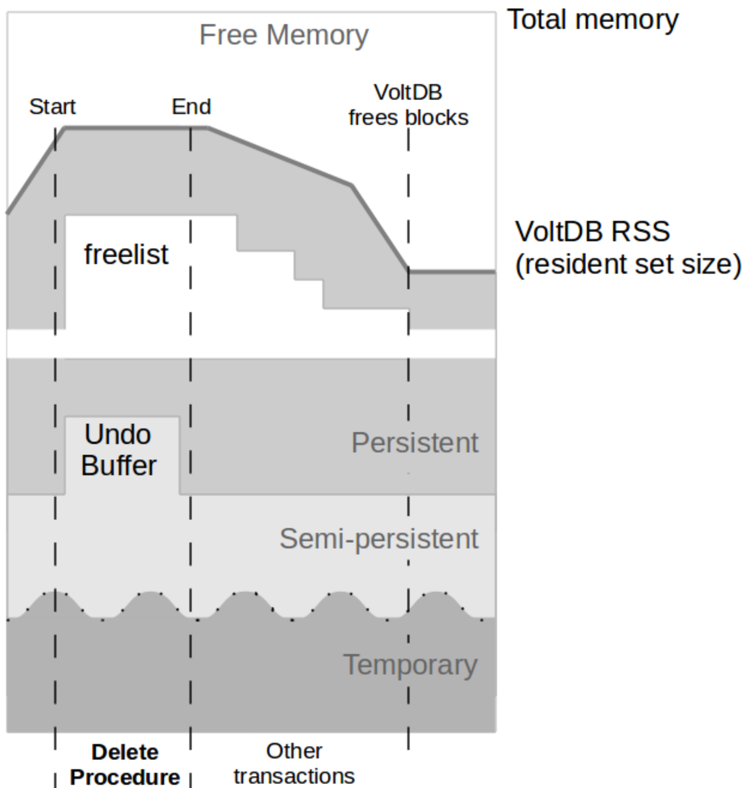


Figure 2, “Details of Memory Usage During and After an SQL Statement” illustrates how a delete operation could have a delayed effect on overall memory allocation.

1. At the beginning of the transaction, the deleted tuples are recorded in the semi-persistent undo buffer, increasing memory usage. Any freed persistent storage is returned to the VoltDB list of free space.

2. At the end of the transaction, the undo buffer is freed. However, the storage for the deleted tuples in persistent storage is managed and may not be released immediately.

3. Over time, free memory is used for new tuples, until...

4. At some point, VoltDB releases unused blocks to the system.

Finally, there are some combinations of factors that can aggravate the fluctuations in memory usage. The memory required for snapshotting is usually not significant. However, if non-blocking snapshots are intermixed with update-heavy transactions, the snapshot copy-on-write buffer can grow rapidly.

Perhaps the most dangerous memory condition is if export is enabled but no client receiver is running to clear the export buffer. Over time, the export buffer will grow without constraint and will, eventually, use up all available memory. This is why it is essential that you run a client receiver attached to all nodes of the cluster when export is enabled.

Changes to How VoltDB Manages Memory

Although the previous memory management algorithms were effective, they had two drawbacks:

- Over time, free space becomes fragmented and can take up more space than necessary.

- Due to internal memory management and delays in deallocating freed storage, the RSS of the VoltDB process is not a valid indicator of how much memory is actually being used.

Starting with V1.2.1, VoltDB is changing how it manages the memory it uses for persistent and semi-persistent storage to resolve these issues. Note, RSS will never be an exact measurement of actual memory usage. However, in the future it will more accurately reflect how much memory is in use. In addition, VoltDB now offers statistics providing a detailed breakdown of how it is using the memory that it has currently allocated. These statistics can be used to provide a more meaningful representation of VoltDB's memory usage than the lump sum allocation reported by the operating system RSS.

The basic plan is that VoltDB will manage memory for persistent and semi-persistent storage more aggressively to ensure unused space is compacted and released when available. In some cases, memory will be returned to the operating system, making the RSS more responsive to changes in the database contents. In other cases, where memory is managed as a pool of resources, VoltDB will provide detailed statistics on what memory is allocated and what is actually in use.

The work to complete this effort will span two separate releases. For V1.2.1, two major changes are being implemented. Persistent storage for database tables (tuples) and tree indexes is being compacted to eliminate fragmentation and allow more memory to be returned to the operating system as the database volume changes. At the same time, storage for hash indexes and strings greater than 64 bytes in length is being managed more effectively as a pool of resources. However, free memory in the pool for hash indexes and strings is not currently being returned to the operating system. In other words, VoltDB holds and reuses memory that is allocated but unused for these objects.

The consequence of these changes is that when you delete rows, the allocated memory for VoltDB (as shown by RSS) may go up during the delete operation (to allow for the undo buffer), but then it will go down — by differing amounts — based on the type of content that is deleted. Memory for tuples not containing large strings or hash indexes will be returned to the operating system quickly. Memory for large strings and hash indexes will not be returned but wlll be held for later reuse.

In other words, the pool size for strings and hash indexes will tend to reach a maximum size (based on the maximum required for your application workload) and then stabilize. Whereas memory for tree indexes as well as numeric and short string data will oscillate as your application needs vary.

To help you understand these changes, new keywords have been added to the @Statistics system procedure — as described later in this white paper — to tell you how much memory VoltDB is using and how much unused memory

is being held for each type of content. These statistics provide a more accurate view of actual memory usage than the lump sum value of system RSS.

In an upcoming release, all persistent data (including has indexes and strings) will be compacted and deallocated. At that point, the operating system RSS will provide a more accurate view of VoltDB's overall memory use and reflect the variations in persistent and semi-persistent memory use.

How Memory is Allocated and Deallocated

Technically, persistent and semi-persistent memory within VoltDB is managed using code written in C++. Temporary memory is managed using code written in Java. What language the source code is written in is not usually relevant, except in the case of memory, because different languages manage memory differently. C++ uses the traditional explicit allocation and deallocation of memory, where the application code controls exactly how and when memory is assigned and deassigned. In Java, memory is not explicitly allocated and deallocated. Instead, Java uses what is called "garbage collection" to free up memory that is not in use.

To complicate matters, the language libraries themselves do some performance optimizations to avoid allocating and deallocating memory from the operating system too frequently. So even if VoltDB explicitly frees memory in persistent or semi-persistent storage, that memory may not be immediately returned to the operating system or alter the process's perceived RSS value.

For temporary storage (which is managed in Java), VoltDB cannot explicitly control memory allocation and deallocation and relies on the Java virtual machine (JVM) to manage memory appropriately. The JVM decides when and how to collect free space from unused objects. This means that the VoltDB server cannot directly control if and when the memory associated with temporary storage is returned to the operating system.

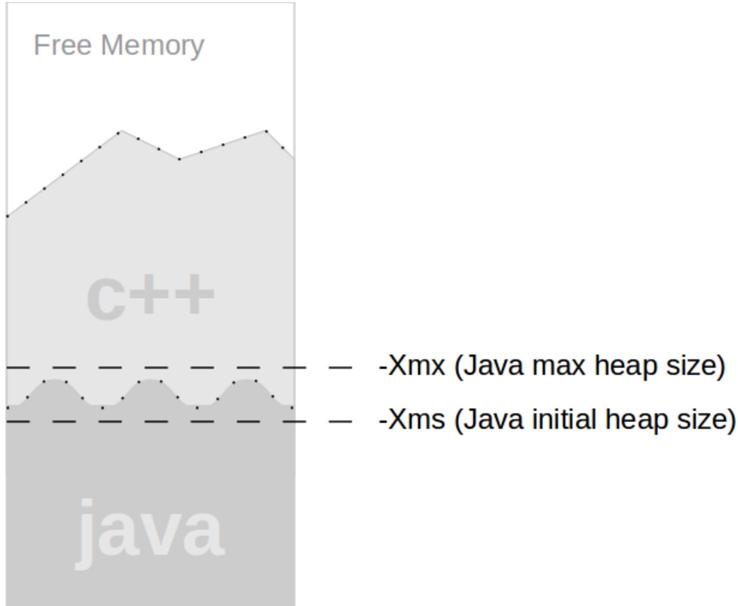
Controlling How Memory is Allocated

Despite the fact that you as a developer or database administrator cannot control *when* temporary storage is allocated and freed, you can control *how much* memory is used. Java provides a way to specify the size of the heap, the portion of memory the JVM uses to store runtime data such as class instances, arrays, etc. The `-Xms` and `-Xmx` arguments to the `java` command specify the initial and maximum heap size, respectively.

By setting the `-Xmx` argument for the maximum heap size, you can control the maximum amount of memory that will be used for temporary storage. In general, VoltDB does not recommend setting the maximum heap size greater than 2 gigabytes for the VoltDB server — any additional space over 2 gigabytes is likely to go unused. If your servers are constrained, you can set a lower maximum heap size to allow more memory for persistent and semi-persistent storage.

By setting both the `-Xmx` and `-Xms` arguments, you can control not only the maximum amount of memory used, but also the amount of fluctuation that can occur. Figure 3, “Controlling the Java Heap Size” illustrates how the `-Xms` and `-Xmx` arguments can be used to control the overall size of temporary storage.

Figure 3. Controlling the Java Heap Size



However, you must be careful when setting the values for the Java heap size, since the JVM will not exceed the value you set as a maximum. It is possible, under some conditions, to force a Java out-of-memory error if the maximum heap size is not large enough for the temporary storage VoltDB requires.

Remember, temporary storage is used to queue the procedure requests and responses. If you are using synchronous procedures calls (and therefore little or no queuing on the server) a small heap size is acceptable. Also, if the size of the procedure invocations (in terms of the arguments passed into the procedures) and the return values are small, a lower heap size is acceptable. But if you are invoking procedures asynchronously with large argument lists or return values, be very careful before setting a low maximum heap size.

Understanding Memory Usage for Specific Applications

To help understand the memory usage for a specific VoltDB database, support for a new keyword, "MEMORY:", has been added to the @Statistics system procedure.[1] The "MEMORY" keyword returns a separate row of data for each server in the cluster, with columns providing information about the different aspects of memory usage, as described in the following table.

Column	Type of Storage	Description
JAVAUSED	Temporary	The amount of memory currently in use for temporary storage.
JAVAUNUSED	Temporary	The maximum amount of Java heap allowed but not currently in use.
TUPLECOUNT	Persistent	The number of tuples currently being held in memory.
TUPLEDATA	Persistent	The approximate amount of memory in use to store tables.

[1]The features described in this section are new as of VoltDB version 1.2.1.

Column	Type of Storage	Description
TUPLEALLOCATED	Persistent	The approximate amount of memory allocated for table storage. This includes the amount in use and any free space currently being held by VoltDB.
INDEXMEMORY	Persistent	The approximate amount of memory in use to store indexes.
STRINGMEMORY	Persistent	The approximate amount of memory in use for string storage.
POOLEDMEMORY	Persistent	The total amount allocated to pooled memory, including the memory assigned for storing strings, hash indexes, free lists, and metadata associated with tuple storage.
RSS	All	The resident set size for the VoltDB server process.

You can use periodic calls to the @Statistics system procedure with the "MEMORY" keyword to track your database cluster's memory usage in detail. But if you are only looking for an overall picture, VoltDB provides simple graphs at runtime.

Connect to a VoltDB server's HTTP port (by default, http://<server-name>:8080) to see graphs of basic memory usage for that server, including total resident set size (RSS), used Java heap and unused Java heap. Memory statistics are collected and displayed over three different time frames: 2 minutes, 30 minutes, and 24 hours. Click on the web browser's refresh button to update the charts.