Planning Guide

Abstract

This book provides recommendations for those considering the use of VoltDB.

V3.0

Planning Guide

V3.0

This document was generated on January 18, 2013.

Table of Contents

List of Figures

List of Tables

Preface

This book provides information to help those considering the use of VoltDB. Choosing the appropriate technology is more than deciding between one set of features and another. It is important to understand how the product fits within the existing technology landscape and what it requires in terms of systems, support, etc. This books provides guidelines for evaluating VoltDB, including sizing hardware, memory, and disks based on the specific requirements of your application and the VoltDB features you plan to use.

1. Organization of this Manual

This book is divided into 5 chapters:

- Chapter 1, *The Planning Process*
- Chapter 2, *Proof of Concept*
- Chapter 3, *Choosing Hardware*
- Chapter 4, *Sizing Memory*
- Chapter 5, *Benchmarking*

2. Other Resources

This book assumes you are already familiar with the VoltDB feature set. The choice of features, especially those related to availability and durability, will impact sizing. Therefore, if you are new to VoltDB, we encourage you to visit the VoltDB web site to familiarize yourself with the product options and features.

You may also find it useful to review the following books to better understand the process of designing, developing, and managing VoltDB applications:

- *Getting Started with VoltDB*, a quick introduction to the product and is recommended for new users
- *Using VoltDB*, a complete reference to the features and functions of the VoltDB product
- *VoltDB Management Guide*, information on managing VoltDB clusters, with a focus on the VoltDB Enterprise Manager and REST API

These books and more resources are available on the web from `http://www.voltdb.com/`.

Chapter 1. The Planning Process

Welcome to VoltDB, a best-in-class database designed specifically for high volume transactional applications. Since you are reading this book, we assume you are considering the use of VoltDB for an existing or planned database application. The goal of this book is to help you understand the impact of such a decision on your computing environment, with a particular focus on hardware requirements.

Technology evaluation normally involves several related but separate activities:

- **Feature Evaluation**

  The goal of the feature evaluation is to determine how well a product's features match up to the needs of your application. For VoltDB, we strongly encourage you to visit our website and review the available product overviews and technical whitepapers to see if VoltDB is right for you. If you need additional information, please feel free to contact us directly.

- **Proof of Concept**

  The proof of concept, or POC, is usually a small application that emulates the key business requirements using the proposed technology. The goal of the POC is to verify, on a small scale, that the technology performs as expected for the target usage.

- **Hardware Planning**

  Once you determine that VoltDB is a viable candidate for your application, the next step is to determine what hardware environment is needed to run it. Hardware sizing requires an understanding of the requirements of the business application (volume, throughput, and availability needs) as well as the technology. The primary goal of this book is to provide the necessary information about VoltDB to help you perform this sizing exercise against the needs of your specific application.

- **Benchmarking**

  Having determined the feasibility of the technology, the final activity is to perform benchmarking to evaluate its performance against the expected workload. Benchmarking is often performed against the proof of concept or a similar prototype application. Benchmarking can help validate and refine the hardware sizing calculations.

Let's assume you have already performed a feature evaluation, which is why you are reading this book. You are now ready to take the next step. The following chapters provide practical advice when building a proof of concept, sizing hardware, and benchmarking a solution with VoltDB.

Note that this book does *not* help with the detailed application design itself. For recommendations on application design we recommend the other books about VoltDB. In particular, *Using VoltDB*.

Chapter 2. Proof of Concept

A proof of concept (POC) is a small application that tests the key requirements of the proposed solution. For database applications, the POC usually focuses on a few critical transactions, verifying that the database can support the proposed schema, queries, and, ultimately, the expected volume and throughput. (More on this in the chapter on Benchmarking.)

A POC is *not* a full prototype. Instead, it is just enough code to validate that the technology meets the need. Depending upon the specific business requirements, each POC emphasizes different database functionality. Some may focus primarily on capacity, some on scalability, some on throughput, etc.

Whatever the business requirements, there are two key aspects of VoltDB that must be designed correctly to guarantee a truly effective proof of concept. The following sections discuss the use of partitioning and stored procedures in POCs.

2.1. Effective Partitioning

VoltDB is a distributed database. The data is partitioned automatically, based on a partitioning column you, as the application developer, specify. You do not need to determine where each record goes — VoltDB does that for you.

However, to be effective, you much choose your partitioning columns carefully. The best partitioning column is not always the most obvious one.

The important thing to keep in mind is that VoltDB partitions both the data *and* the work. For best performance you want to partition the database tables and associated queries so that the most common transactions can be run in parallel. That is, the transactions are, in VoltDB parlance, "single-partitioned".

To be single-partitioned, a transaction must only contain queries that access tables based on a specific value for the partitioning column. In other words, if a transaction is partitioned on the EmpID column of the Employee table (and that is the partitioning column for the table), all queries in the transaction accessing the Employee table must include the constraint `WHERE Employee.EmpID = {value}`.

To make single-partitioned transactions easier to create, not all tables have to be partitioned. Tables that are not updated frequently can be replicated, meaning they can be accessed in any single-partitioned transaction, no matter what the partitioning key value.

When planning the partitioning schema for your database, the important questions to ask yourself are:

- Which are the critical, most frequent queries? (These are the transactions you want to be single-partitioned.)
- For each critical query, what database tables does it access and using what column?
- Can any of those tables be replicated? (Replicating smaller, less frequently updated tables makes joins in single-partitioned procedures easier.)

2.2. Designing the Stored Procedures

Designing the schema and transactions to be single-partitioned is one thing. It is equally important to make sure that the stored procedures operate in a way that lets VoltDB do its job effectively.

The first step is to write the transactions as stored procedures that are compiled into the application catalog. Do *not* write critical transactions as ad hoc queries to the database. VoltDB provides the @AdHoc system procedure for executing arbitrary SQL queries, which can be helpful when building early prototypes to test queries or occasionally validate the content of the database. But @AdHoc queries often run as multi-partitioned transactions and, therefore, should not be used for critical or repetitive transactions.

The next step is to ensure that single-partitioned stored procedures are correctly identified as such in the schema using the PARTITION PROCEDURE statement and specifying the appropriate partitioning column.

Finally, when designing for maximum throughput, use asynchronous calls to invoke the stored procedures from within the POC application. Asynchronous calls allow VoltDB to queue the transactions (and their responses), avoiding any delays between when a transaction completes, the results are returned to the POC application, and the next procedure is invoked.

Chapter 5, *Benchmarking* later in this book provides additional suggestions for effectively designing and testing proof of concept applications.

Chapter 3. Choosing Hardware

VoltDB is designed to provide world class throughput on commodity hardware. You do not need the latest or most expensive hardware to achieve outstanding performance. In fact, a few low- to mid-range servers can easily outperform a single high end server, since VoltDB throughput tends to scale linearly with the number of servers in the cluster.

People often ask us at VoltDB "what type of servers should we use and how many?" The good news is that VoltDB is very flexible. It works well on a variety of configurations. The bad news is that the true answer to the question is "it depends." There is no one configuration that is perfect for all situations.

Like any technology question, there are trade offs to be made when choosing the "right" hardware for your application. This chapter explains what those trade offs are and provides some general rules of thumb that can help when choosing hardware for running a VoltDB database.

3.1. The Dimensions of Hardware Sizing

There are three key dimensions to sizing individual servers: the number and speed of the processors, the total amount of memory, and the size and type of disks available. When sizing hardware for a distributed database such as VoltDB, there is a fourth dimension to consider: the number of servers to use.

Each of these dimensions impacts different aspects of VoltDB performance. The number of processors affects how many partitions can be run on each server and, as a result, throughput. The available memory obviously impacts capacity, or the volume of data that can be stored. The size, number, and type of disks impacts the performance of availability features such as snapshots and command logging.

However, they also interact. The more memory per server, the longer it takes to write a snapshot or for a node to rejoin after a failure. So increasing the number of servers but reducing the amount of memory per server may reduce the impact of durability on overall database performance. These are the sorts of trade offs that need to be considered.

The following sections discuss hardware sizing for three key aspects of a VoltDB application:

- Throughput
- Capacity
- Durability

3.2. Sizing for Throughput

The minimum hardware requirements for running a VoltDB database server is a 64-bit machine with two or more processor cores. The more cores the server has, the more VoltDB partitions can potentially run on that server. The more unique partitions, the more throughput is possible with a well partitioned workload.

However, the number of processor cores is not the only constraint on throughput. Different aspects of the server configuration impact different characteristics of the database process.

For example, although the more physical cores a server has increases the number of partitions that server can potentially handle, at some point the number of transactions being received and data being returned exceeds the capacity of the network port for that server. As a consequence, going beyond 12-16 cores on a single machine may provide little value to a VoltDB database, since the server's network adapter becomes the gating factor.

Rule of Thumb

VoltDB runs best on servers with between 4 and 16 cores.

It should be noted that the preceding discussion refers to physical processor cores. Some servers support hyperthreading. *Hyperthreading* is essentially the virtualization of processor cores, doubling the reported number of cores. For example, a system with 4 cores and hyperthreading acts like an 8 core machine. These virtualized cores can improve VoltDB performance, particularly for servers with a small (2 or 4) number of physical cores. However, the more physical cores the system has, the less improvement is seen in VoltDB performance. Therefore, hyperthreading is not recommended for VoltDB servers with more than 8 physical cores.

The alternative to adding processors to an individual server for improving throughput is to add more servers. If a single 4-core server can handle 3 partitions, two such servers can handle 6 partitions, three can handle 9, etc. This is how VoltDB provides essentially linear scaling in throughput.

But again, there are limits. For peak performance it is key that network latency and disruption between the cluster nodes be kept to a minimum. In other words, all nodes of the cluster should be on the same network switch. Obviously, the capacity of the network switch constrains the number of nodes that it can support. (A 32 port switch is not uncommon.)

Rule of Thumb

Best performance is achieved with clusters of 2-32 servers connected to the same network switch.

It is possible to run a VoltDB cluster across switches. (For example, this is almost always the case in cloud environments.) However, latency between the cluster nodes will have a negative impact on performance and may ultimately limit overall throughput. In these situations, it is best to benchmark different configurations to determine exactly what performance can be expected.

Finally, it should be noted that the speed of the processor cores may not have a significant impact on overall throughput. Processor speed affects the time it takes to execute individual transactions, which may be only a small percentage of overall throughput. For workloads with very compute-intensive transactions, faster processors can improve overall performance. But for many small or simple transactions, improved processor speed will have little or no impact.

3.3. Sizing for Capacity

The second aspect of database sizing is capacity. Capacity describes the maximum volume of data that the database can hold.

Since VoltDB is an in-memory database, the capacity is constrained by the total memory of all of the nodes in the cluster. Of course, one can never size servers too exactly. It is important to allow for growth over time and to account for other parts of the database server that use memory.

Chapter 4, *Sizing Memory* explains in detail how memory is assigned by the VoltDB server for database content. Use that chapter to perform accurate sizing when you have a known schema. However, as a rough estimate, you can use the following table to approximate the space required for each column. By adding up the columns for each table and index (including index pointers) and then multiplying by the expected number of rows, you can determine the total amount of memory required to store the database contents.

Table 3.1. Quick Estimates for Memory Usage By Datatype

Datatype	Bytes in Table	Bytes in Index
TINYINT	1	1
SMALLINT	2	2
INTEGER	4	4
BIGINT	8	8
DOUBLE	8	8
DECIMAL	16	16
TIMESTAMP	8	8
VARCHAR or VARBINARY (less than 64 bytes)	length + 1	length + 1
VARCHAR or VARBINARY (64 bytes or greater)	length	8
index pointers	n/a	40 or 48*

*40 bytes for tree indexes, 48 for hash indexes.

Having calculated the total storage required for the database content, you can estimate the memory required for the server overall by adding the memory required by the server process itself (one gigabyte is recommended) and adding 30% as a buffer.

Server memory = (total content size + 1GB) + 30%

When sizing for a cluster, where the content is distributed across the servers, the calculation for the memory required for content on each server is the total content size divided by the number of servers, plus some percentage for replicated tables. For example, if 20% of the tables are replicated, a rough estimate of the space required for each server is given by the following equation:

Per server memory = ((total content size / number of servers) + 20% + 1GB) + 30%

When sizing memory for VoltDB servers, it is important to keep in mind the following points:

- Memory usage includes not only storage for the data, but also temporary storage for processing transactions, managing queues, and the server processes themselves.

- Even in the best partitioning schemes, partitioning is never completely balanced. Make allowances for variations in load across the servers.

- If memory usage exceeds approximately 70% of total memory, the operating system can start paging and swapping, severely impacting performance.

Rule of Thumb

Keep memory usage per server within 50-70% of total memory.

Memory technology and density is advancing so rapidly, (similar to the increase in processor cores per server), it is feasible to configure a small number of servers with extremely large memory capacities that provide capacity and performance equivalent to a larger number of smaller servers. However, the amount of memory in use can impact the performance of other aspects of database management, such as snapshots and failure recovery. The next section discusses some of the trade offs to consider when sizing for these features.

3.4. Sizing for Durability

Durability refers to the ability of a database to withstand — or recover from — unexpected events. VoltDB has several features that increase the durability of the database, including K-Safety, snapshots, command logging, and database replication

K-Safety replicates partitions to provide redundancy as a protection against server failure. Note that when you enable K-Safety, you are replicating the unique partitions across the available hardware. So the hardware resources — particularly servers and memory — for any one copy are being reduced. The easiest way to size hardware for a K-Safe cluster is to size the initial instance of the database, based on projected throughput and capacity, then multiply the number of servers by the number of replicas you desire (that is, the K-Safety value plus one).

Rule of Thumb

When using K-Safety, configure the number of cluster nodes as a whole multiple of the number of copies of the database (that is, K+1).

K-Safety has no real performance impact under normal conditions. However, the cluster configuration can affect performance when recovering from a failure. In a K-Safe cluster, when a failed server rejoins, it gets copies of all of its partitions from the other members of the cluster. The larger (in size of memory) the partitions are, the longer they can take to be restored. Since it is possible for the restore action to block database transactions, it is important to consider the trade off of a few large servers that are easier to manage against more small servers that can recover in less time.

Two of the other durability features — snapshots and command logs — have only a minimal impact on memory and processing power. However, these features do require persistent storage on disk.

Most VoltDB disk-based features, such as snapshots, export overflow, network partitions, and so on, can be supported on standard disk technology, such as SATA drives. They can also share space on a single disk, assuming the disk has sufficient capacity, since disk I/O is interleaved with other work.

Command logging, on the other hand, is time dependent and must keep up with the transactions on the server. The chapter on command logging in *Using VoltDB* discusses in detail the trade offs between asynchronous and synchronous logging and the appropriate hardware to use for each. But to summarize:

- Use fast disks (such as battery-backed cache drives) for synchronous logging

- Use SATA or other commodity drives for asynchronous logging. However, it is still a good idea to use a dedicated drive for the command logs to avoid concurrency issues between the logs and other disk activity.

Rule of Thumb

When using command logging, whether synchronous or asynchronous, use a dedicated drive for the command logs. Other disk activity (including command log snapshots) can share a separate drive.

Finally, database replication (DR) does not impact the sizing for memory or processing power of the initial, master cluster. But it does require a duplicate of the original cluster hardware to use as a replica. In other words, when using database replication, you should double the estimated number of servers — one copy for the master cluster and one for the replica. In addition, one extra server is recommended to act as the DR agent.

Rule of Thumb

When using database replication, double the number of servers, to account for both the master and replica clusters. Also add one server to be used as the DR agent.

Chapter 4. Sizing Memory

An important aspect of system sizing is planning for the memory required to support the application. Because VoltDB is an in-memory database, allocating sufficient memory is vital.

Section 3.3, “Sizing for Capacity” provides some simple equations for estimating the memory requirements of a prospective application. If you are already at the stage where the database schema is well-defined and want more precise measurements of potential memory usage, this chapter provides details about how memory gets allocated.

For VoltDB databases, there are two aspects of memory sizing: understanding how much memory is required to store the data itself, including indexes, and the memory required for server processes, including queues and temporary storage. The following sections discuss these two aspects separately.

4.1. Planning for Database Capacity

To plan effectively for database capacity, you must know in advance both the structure of the data and the projected volume. This means you must have at least a preliminary database schema, including tables, columns, and indexes, as well as the expected number of rows for each table.

It is often useful to write this initial sizing information down, for example in a spreadsheet. Your planning may even allow for growth, assigning values for both the initial volume and projected long-term growth. For example, here is a simplified example of a spreadsheet for a database supporting a flight reservation system:

Name	Type	Size	Initial Volume	Future Volume
Flight	Replicated table		5,000	20,000
- FlightByID	Hash Index		5,000	20,000
- FlightByDepartTime	Tree Index		5,000	20,000
Airport	Replicated table		10,000	10,000
- AirportByCode	Tree Index		10,000	10,000
Reservation	Table		100,000	200,000
- ReservByFlight	Hash Index		100,000	200,000
Customer	Table		200,000	1,000,000
- CustomerByID	Hash Index		200,000	1,000,000
- CustomerByName	Tree Index		200,000	1,000,000

Using the database schema, it is possible to calculate the size of the individual table records and indexes, which when multiplied by the volume projections gives a good estimate the the total memory needed to store the database contents. The following sections explain how to calculate the size column for individual table rows and indexes.

4.1.1. Sizing Database Tables

The size of individual table rows depends on the number and datatype of the columns in the table. For fixed-size datatypes, such as INTEGER and TIMESTAMP, the column is stored inline in the table record using the specified number of bytes. Table 4.1, “Memory Requirements For Tables By Datatype” specifies the length (in bytes) of fixed size datatypes.

For variable length datatypes, such as VARCHAR and VARBINARY, how the data is stored and, consequently, how much space it requires, depends on both the actual length of the data and the maximum possible length. If the maximum length is less than 64, the data is stored inline in the tuple as fixed-length data consuming the maximum number of bytes plus one for the length. So, for example, a VARCHAR(32) column takes up 33 bytes, no matter how long the actual data is.

If the maximum length is 64 bytes or more, the data is stored in pooled memory rather than inline. To do this, there is an 8-byte pointer stored inline in the tuple, a 24-byte string reference object, and the space required to store the data itself in the pool. Within the pool, the data is stored as a 4-byte length, an 8-byte reverse pointer to the string reference object, and the data.

To complicate the calculation somewhat, data stored in pooled memory is not stored as arbitrary lengths. Instead, data is incremented to the smallest appropriate "pool size", where pool sizes are powers of 2 and intermediary values. In other words, pool sizes include 2, 4, 6 (2+4), 8, 12 (8+4), 16, 24 (8+16), 32 and so on up to a maximum of 1 megabyte for data plus 12 bytes for the pointer and length. For example, if the LastName column in the Customer table is defined as VARCHAR(128) and the actual content is 95 bytes, the column consumes 160 bytes:

```
              8   Inline pointer
             24   String reference object
        4         Data length
        8         Reverse pointer
       95         Data
      107   128   Pool total / incremented to next pool size
            160   Total
```

Note that if a variable length column is defined with a maximum length greater than or equal to 64, it is not stored inline, even if the actual contents is less than 64 bytes. Variable length columns are stored inline only if the maximum length is less than 64.

Table 4.1, “Memory Requirements For Tables By Datatype” summarizes the memory requirements for each datatype.

Table 4.1. Memory Requirements For Tables By Datatype

Datatype	Size (in bytes)	Notes
TINYINT	1	
SMALLINT	2	
INTEGER	4	
BIGINT	8	
DOUBLE	8	
DECIMAL	16	
TIMESTAMP	8	
VARCHAR (<64)	maximum size + 1	Stored inline
VARBINARY (<64)	maximum size + 1	Stored inline
VARCHAR (>=64)	32 + (actual size + 12 + padding)	Pooled resource. Total size includes an 8-byte inline pointer, a 24-byte reference pointer, plus the pooled resource itself.
VARBINARY (>=64)	32 + (actual size + 12 + padding)	Same as VARCHAR.

For tables with variable length columns less than 64 bytes, memory usage can be sized very accurately using the preceding table. However, for tables with variable length columns greater than 64 bytes, sizing is approximate at best. Besides the variability introduced by pooling, any sizing calculation must be based on an estimate of the *average length of data in the variable columns*.

For the safest and most conservative estimates, you can use the maximum length when calculating variable length columns. If, on the other hand, there are many variable length columns or you know the data will vary widely, you can use an estimated average or 90th percentile figure, to avoid over-estimating memory consumption.

4.1.2. Sizing Database Indexes

Indexes are sized in a way similar to tables, where the size and number of the index columns determine the size of the index. However, the calculations for tree and hash indexes are distinct.

For tree indexes, which are the default index type, you can calculate the size of the individual index entries by adding up the size for each column in the index plus 40 bytes for overhead (pointers and lengths). The size of the columns are identical to the sizes when sizing tables, as described in Table 4.1, “Memory Requirements For Tables By Datatype”, with the exception of non-inlined binary data. For variable length columns equal to or greater than 64 bytes in length, the index only contains an 8-byte pointer; the data itself is not replicated.

So, for example, the CustomerByName index on the Customer table, which is a tree index containing the VARCHAR(128) fields LastName and FirstName, has a length of 56 bytes for each entry:

```
  8   Pointer to LastName
  8   Pointer to FirstName
 40   Overhead
 56   Total
```

The following equation summarizes how to calculate the size of a tree index.

(sum-of-column-sizes + 8 + 32) * rowcount

For hash indexes, the total size of the index is not a direct multiple of individual index entries. Hash indexes are stored as a hash table, the size of which is determined both by the size of the content and the number of entries. The size of the individual entries are calculated by the size of the individual columns plus 32 bytes of overhead. (Hash indexes support integer columns only, so there are no variable length keys.) But in addition to the individual entries, the index requires additional space for the hash, based on the total number of entries. This space is calculated (in bytes) as two times the number of rows, plus one and multiplied by eight.

The following equation summarizes how to calculate the size of a hash index.

(((2 * rowcount) + 1) * 8) + ((sum-of-column-sizes + 32) * rowcount)

Note that hash indexes do not grow linearly. Incrementally allocating small blocks of memory is not productive. Instead, when the index needs more space (because additional entries have been added), the index doubles in size all at one time. Similarly, the index does not shrink immediately if entries are removed.

In general, the space allocated to hash indexes doubles in size whenever the index usage reaches 75% of capacity and the memory allocation is cut in half if usage drops below 15%.

4.1.3. An Example of Database Sizing

Using the preceding formulas it is possible to size the sample flight database mentioned earlier. For example, it is possible to size the individual rows of the Flight table based on the schema columns and datatypes. The following table demonstrates the sizing of the Flight table.

Column	**Datatype**	**Size in Bytes**
FlightID	INTEGER	4
Carrier	VARCHAR(128)	160
DepartTime	TIMESTAMP	8
ArrivalTime	TIMESTAMP	8
Origin	VARCHAR(3)	4
Destination	VARCHAR(3)	4
Destination	VARCHAR(3)	4
Total:		**192**

The same calculations can be done for the other tables and indexes. When combined with the expected volumes (described in Section 4.1, “Planning for Database Capacity”), you get a estimate for the total memory required for storing the database content of approximately 500 megabytes, as shown in the following table.

Name	**Type**	**Size**	**Final Volume**	**Total Size**
Flight	Replicated table	184	20,000	3,840,000
- FlightByID	Hash Index	*36	20,000	1,040,008
- FlightByDepartTime	Tree Index	48	20,000	960,000
Airport	Replicated Table	484	10,000	4,840,000
- AirportByCode	Tree Index	44	10,000	440,000
Reservation	Table	243	200,000	48,600,000
- ReservByFlight	Hash Index	*36	200,000	10,400,008
Customer	Table	324	1,000,000	324,000,000
- CustomerByID	Hash Index	*36	1,000,000	52,000,008
- CustomerByName	Tree Index	56	1,000,000	56,000,000
Total:				**502,120,024**

*For hash indexes, size specifies the individual entry size only. The total size includes the sum of the entries plus the hash itself.

4.2. Planning for Server Distribution

The preceding sections explain how to calculate the total memory required for storing the database content and indexes. However, at least four other aspects of your configuration contribute to how you calculate memory usage:

- The server process memory requirements
- The number of servers in the cluster
- K-Safety

- Special buffers for rejoin and overflow

4.2.1. Server Process Memory Usage

On a single server, the memory for a VoltDB database is the memory required for the content plus the memory required for the server process itself. The majority of the memory required for the Java server process is defined by the Java max heap that is set using the `-Xmx` argument when starting the server process (or defined in the Database Configuration dialog in the VoltDB Enterprise Manager).

Memory per server = java max heap + database content

Using the recommended max heap of one gigabyte, and our estimated database volume of 500 megabytes, our example Flight database would require a minimum of 1.5 gigabytes of memory. Keeping in mind that memory usage should be kept under 75% of total memory, this means that at least 2 gigabytes should be allowed for running the VoltDB database (beyond what the operating system and other processes may require).

4.2.2. Memory Usage in Clusters

Both capacity and throughput can be increased by running a VoltDB database on a cluster. Although it is tempting to simply divide the total memory consumption by the number of servers, this is not an accurate formula for two reasons:

- Not all data is partitioned. Replicated tables (and their indexes) appear on all servers.
- Few if any partitioning schemes provide perfectly even distribution. It is important to account for some variation in the distribution.

To accurately calculate the memory usage per server in a cluster, you must account for the server process, all replicated tables and indexes, and the server's portion of the partitioned tables and indexes.

Memory per server = java max heap + replicated tables + (partitioned tables/number of servers)

Using the sample sizing for the Flight database described in Section 4.1.3, “An Example of Database Sizing”, the total memory required for the replicated tables and indexes (for the Flight and Airport tables) is only approximately 12 megabytes. The memory required for the remaining partitioned tables and indexes is approximately 490 megabytes. Assuming the database is run on two servers, the total memory required for each server is approximately 1.25 gigabytes:

```
         1,000   Java max heap (in MB)
            12   Replicated data
    2            Number of servers
  490      245   Paritioned data total / per server
         1,257   Total
```

4.2.3. K-Safety

The features you plan to use with a VoltDB database can also impact capacity planning, most notably K-Safety. K-Safety improves availability by replicating data across the cluster, allowing the database to survive individual node failures.

Because K-Safety involves replication, it also increases the memory requirements for storing the replicated data. Perhaps the easiest way to size a K-Safe cluster is to size a non-replicated cluster, then multiply by the K-Safety value plus one.

For example, let's assume you plan to run a database with a K-Safety value of 2 (in other words, three copies) on a 6-node cluster. The easiest way to determine the required memory capacity per server is to calculate the memory requirements for a 2-node (non K-Safe) cluster, then create three copies of that hardware and memory configuration.

4.2.4. Special Buffers for Rejoin and Overflow

The previous sections describe memory usage under normal operating conditions. In special cases where nodes fail and rejoin or network problems cause other processes to buffer their output, VoltDB must accommodate this additional memory usage. Special conditions that can affect memory usage in the Java heap include the following:

- **Rejoin** — When a node fails and rejoins a high availability cluster, the rejoining node uses additional memory to buffer the incoming data for the partitions it must recreate. The rejoining node may receive data from one or two other nodes in the cluster. In this situation, the rejoining node will use 128MB per partition, buffering content from up to two other nodes at any given time. In other words, it may require up to 128MB * 2 * sitesperhost in additional memory during the rejoin process.

- **Export** — If network issues or slow response from the export target result in the export process not keeping up with the queued export data, VoltDB first buffers the output in memory before overflowing to disk. Export buffering can use up to 4MB per partition.

- **Data Replication** — If the database is being replicated (that is, it is a *master* that a DR agent is connected to) and either the replica fails, the DR agent stops, of network issues cause the replication process to not keep up, the cluster will buffer DR data in memory. The database can buffer up to 128MB per partition before data replication is stopped on the master.

This means that if any of these features are enabled, you must account for the the overflow buffers in your memory calculations. In particular, memory used for rejoin and for data replication overflow consumes space in the Java heap. You must account for this variability to avoid running out of memory in the Java process.

For example, if you are using data replication on a master cluster with 8 partitions per node (that is sitesperhost=8) and replication stops, it is possible that DR overflow can consume up to a gigabyte (8X128MB) of the Java heap before replication queueing stops. It is also possible that, if the master cluster is K-safe, a node might fail and then rejoin the cluster. In that case, the rejoining node may require up to 2 gigabytes (8 X 2 X 128MB) of the Java heap. However, any single node would never perform these two actions together. So you must only account for the higher value. (In this instance, the 2GB required for rejoin.)

This overflow memory should be added to your calculations of both total memory for the server and for the sizing of the maximum Java heap size.

Chapter 5. Benchmarking

Benchmarking is the process of evaluating performance against a known baseline. For database applications, you can benchmark against other technologies, against a target metric (such as a specific number of transactions per second), or you can use the same application on different hardware configurations to determine which produces the best results.

For VoltDB applications, benchmarking is useful for establishing metrics with regards to:

- Optimum throughput
- Optimum number of sites per host

5.1. Benchmarking for Performance

When establishing performance criteria for database applications, there are usually two key metrics:

- **Throughput** — how many transactions can be completed at one time, usually measured in transactions per second, or TPS
- **Latency** — how long each individual transaction takes, usually measured as the average or percentile of the latency for a sample number of transactions

Note that neither of these metrics is exact. For many databases, throughput can vary depending upon the type of transaction; whether it is a read or write operation. One of the advantages of VoltDB is that throughput does not change significantly for write versus read operations. However, VoltDB throughput *does* change when there are many multi-partitioned transactions versus a single-partitioned transaction workload. This is why it is important to design your schema and stored procedures correctly when benchmarking a VoltDB database.

Similarly, latency can vary, both in how it is measured and what impacts it. You can measure latency as the time from when the client issues a transaction request until the response is received, or from when the database server receives the request until it queues the response.

The former measurement, latency from the client application's perspective, is perhaps the most accurate "real world" metric. However, this metric includes both database latency and any network latency between the client and the server. The latter measurement, latency from the database perspective, is a more accurate measurement of the technology's capability. This metric includes the time required to process the transaction itself (that is, the stored procedures and the database queries it contains) as well as time the request spends in the queue waiting to be executed.

5.1.1. Designing Your Benchmark Application

There is a relationship between throughput, latency, and system configuration. Throughput is a combination of the amount of time it takes to execute a transaction (which is part of latency) and the number of transactions that can be run in parallel (that is, the percentage of single-partitioned transactions plus the number of unique partitions, which is a combination of sites per host and number of servers).

This is why it is important that benchmarking for performance be done in conjunction with benchmarking for server configuration (as discussed later). Different configurations will result in different values for throughput and latency.

Your benchmark results are also affected by the design of the application itself.

5.1.1.1. Good Application Design

As mentioned before, throughput and latency are not abstract numbers. They are a consequence of the schema, stored procedures, application, and server configuration in use. As with any VoltDB solution, to affectively benchmark performance you must start with a well-designed application. Specifically:

- Partition all tables (except those that are small and primarily read-only)
- Make all frequent transactions single-partitioned
- Use asynchronous procedure calls

See Chapter 2, *Proof of Concept* for more information on writing effective VoltDB applications.

It is common practice to start benchmarking with the proof of concept, since the POC includes an initial schema and the key transactions required by the solution. If you decide to proceed with development, it is also a good idea to benchmark the application periodically throughout the process to make sure you understand how the additional functions impact your overall performance targets.

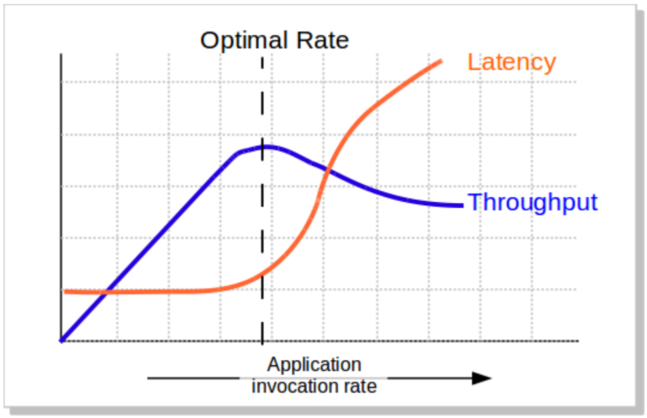
However, before benchmarking the POC, it is important to decide how you are going to measure performance. The following sections provide useful information for determining what and how to measure performance in a VoltDB application.

5.1.1.2. Rate Limiting

Latency and throughput are related. To measure throughput, your test application can make stored procedure calls as fast as it can and then measure how many are completed in a second (TPS). However, if the test application invokes more transactions that the server can complete in that time frame, the additional invocations must wait in the queue before being processed. The time these transactions wait in the queue will result in the latency measurement suddenly spiking.

There is an optimal throughput for any given application and system configuration. As the rate of invocations from the client application increases, there is a direct increase in TPS while latency stays relatively flat. However, as the invocation rate approaches and surpasses the optimal throughput (the limit of what the current configuration can process), latency increases dramatically and TPS may even drop, as shown in Figure 5.1, “Determining Optimal Throughput and Latency”.

Figure 5.1. Determining Optimal Throughput and Latency



If your test application "fire hoses" the database server — that is, it sends invocations as fast as it can — all you can measure is the misleading throughput and latency on the right side of the preceding chart. To determine the optimal rate, you need to be able control the rate at which your benchmark application submits transaction requests. This process is called *rating limiting*.

At its simplest, rate limiting is constraining the number of invocations issued by the application. For example, the following program loop, constrains the application to invoking the SignIn stored procedure a maximum of 10 times per millisecond, or 10,000 times per second.

```
boolean done = false;
long maxtxns = 10;
while (!done) {
  long txns = 0;
  long millisecs = System.currentTimeMillis();
  while (millisecs == System.currentTimeMillis()) {
      if ( txns++ < maxtxns } {
         myClient.callProcedure(new SignInCallback(),
                                "SignIn",
                                id, millisecs);
      }
  }
}
```

You could use a command line argument to parameterize the rate limit value maxtxns and then use multiple runs of the application to create a graph similar to Figure 5.1, “Determining Optimal Throughput and Latency”. An even better approach is to use a limit on latency to automatically control the invocation rate and let the application close in on the optimal throughput value.

How rate limiting based on latency is done is to have a variable for the target latency as well as a variable for maximum allowable throughput (such as maxtxns in the preceding example). The application measures both the average throughput and latency for a set period (every 1 to 5 seconds, for example). If the average latency exceeeds the goal, reduce the maximum transactions per second, then repeat. After the same period, if the latency still exceeds the goal, reduce the maximum transaction rate again. If the latency *does* meet the goal, incrementally *increase* the maximum transaction rate.

By using this mix of rate limiting and automated adjustment based on a latency goal, the test application will eventually settle on an optimal throughput rate for the current configuration. This is the method used by the sample applications (such as voter and Voltkv) that are provided with the VoltDB software.

5.1.2. How to Measure Throughput

Normally for benchmarking it is necessary to "instrument" the application. That is, add code to measure and report on the benchmark data. Although it is possible (and easy) to instrument a VoltDB application, it is not necessary.

The easiest way to measure throughput for a VoltDB database is to monitor the database while the benchmark application is running. You can monitor a database using either the VoltDB Web Studio or the VoltDB Enterprise Manager (if you are using the Enterprise Edition). Both tools provide a graphical display of the overall throughput for the database cluster.

For more detailed information, you can instrument your application by using a variable to track the number of completed transactions (incrementing the variable in the asynchronous procedure callback). You then periodically report the average TPS by dividing the number of transactions by the number of seconds since the last report. This approach lets you configure the reporting to whatever increment you choose.

See the voter sample application for an example of an instrumented benchmarking application. The README for the voter application explains how to customize the reporting through the use of command line arguments.

5.1.3. How to Measure Latency

It is also possible to get a sense of the overall latency without instrumenting your application using the VoltDB Web Studio and Enterprise Manager. However, it is important to understand what each tool is telling you.

The latency graph provided by Web Studio shows the average latency, measured every second, as reported by the server. This value is very dynamic, resulting in a dramatic sawtooth graph with extreme highs and lows. The best way to get a sense of the general latency the application sees with Web Studio is to imagine a line drawn across the high points of the graph.

The latency graph shown in the VoltDB Enterprise Manager, on the other hand, shows latency to the 99th percentile since the application began. The latency is also measured in "buckets", resulting in more of a stair step graph, showing you the high watermark or worst case latency over time. Since latency is measured from the beginning of the database instance, the 99th percentile report also tends to stay high after a spike in actual latency. In other words, the graph does not come down until latency has stayed consistency low for an extended period of time.

Both graphs can provide you with a rough estimate of latency, either as latency-over-time or worst case high points. For a more accurate benchmark, you can use latency metrics built into the VoltDB system procedures and Java client interface.

To instrument your client application, you can use the ClientResponse interface returned by stored procedures invocations to measure latency. Part of the ClientResponse class are the getClientRoundTrip() and getClusterRoundTrip() methods. Both methods return an integer value representing the number of milliseconds required to process the transaction. The getClientRoundTrip() method reports total latency, including the network round trip from the client application to the server. The getClusterRoundTrip() method reports an estimate of the latency associated with the database only.

It is easy to use these methods to collect and report custom latency information. For example, the following code fragment captures information about the minimum, maximum, and combined latency during each procedure callback.

```
static class PocCallback implements ProcedureCallback {
   @Override
   public void clientCallback(ClientResponse response) {

      txncount++;
      int latency = response.getClusterRoundTrip();
      if (latency < minlatency) minlatency = latency;
      if (latency > maxlatency) maxlatency = latency;
      sumlatency += latency;
              .
              .
              .
```

The benchmarking application can then use this information to periodically report specific latency values, like so:

```
if (System.currentTimeMillis() >= nextreport ) {
      // report latency
  printf("Min latency: %d\n" +
         "Max latency: %d\n" +
         "Average latency: %d\n",
          minlatency, maxlatency, sumlatency/txncount);
      // reset variables
  txncount=0;
  minlatency = 5000;
  maxlatency = 0;
  sumlatency = 0;
      // report every 5 secsonds
  nextreport = System.currentTimeMillis() + 5000;
}
```

5.2. Determining Sites Per Host

Another important goal for benchmarking is determining the optimal number of sites per host. Each VoltDB server can host multiple partitions, or "sites". The ideal number of sites per host is related to the number of processor cores available on the server. However, it is not an exact one-to-one relationship. Usually, the number of sites is slightly lower than the number of cores.

The equation becomes even more complex with hyperthreading, which "virtualizes" multiple processors for each physical core. Hyperthreading can improve the number of sites per host that a VoltDB server can support. But again, not in direct proportion to a non-hyperthreaded server.

The easiest way to determine the optimal number of sites per host is by testing, or benchmarking, against the target application. The process for determining the correct number of sites for a specific hardware configuration is as follows:

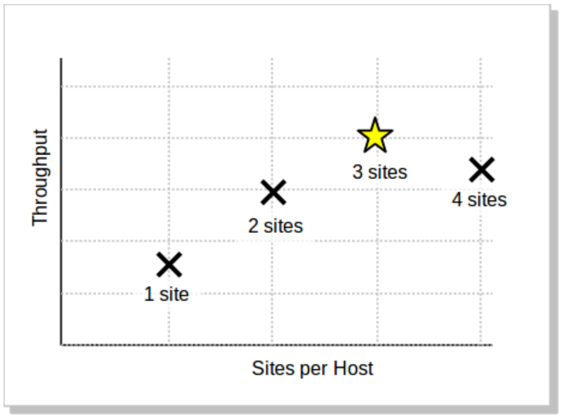
1. Create a benchmark application that measures the optimal throughput, as described in Section 5.1, “Benchmarking for Performance”.

2. Run the benchmark application multiple times, each time increasing the number of sites per host for the database.

3. Make note of the optimal throughput for each run and graph the optimal TPS against the number of sites per host.

As the number of sites per host increases, the optimal throughput increases as well. However, at some point, the number of sites exceeds the number of threads that the hardware can support, at which point the throughput levels out, or even decreases, as contention occurs in the processor. Figure 5.2, “Determining Optimal Sites Per Host” shows the results graph of a hypothetical benchmark of sites per host for a quad core server. In this case, the optimal number of sites per host turned out to be three.

Figure 5.2. Determining Optimal Sites Per Host



By graphing the relationship between throughput and partitions using a benchmark application, it is possible to maximize database performance for the specific hardware configuration you will be using.