MEMSQL FOR MEMEX: AN ANALYSIS OF LARGE THROUGHPUT DATA TRANSACTIONS OF PARCEL SCANNING INFORMATION

September 14, 2015

Noel Carrascal
235 15th Ave
San Francisco, CA 94118
noelcarrascal@gmail.com

Overview

MemEx is a growing company that delivers a large number of parcels per day. Approximately fifty million parcels are moved daily, and each parcel is scanned for tracking up to seven times. This operation requires a reliable data infrastructure that can handle the daily workload, an increase in the number of parcels and the analysis of scanning information for operation optimization and package tracking. For these reasons, MemEX needs reliable database transactions of parcel information with easy horizontal scalability of its computational distributed system.

MemSQL is the world's fastest in memory database that specializes in the type of services that MemEx demands. In this report, the capabilities of MemSQL are demonstrated by timing high throughput data transactions using a two-node-cluster hosted by Amazon Web Services.

Introduction

The essence of MemEx operations are extracted in order to write a program that generates parcel data at a rate that is equal or higher than MemEx daily database transaction numbers. The specifications of MemEx operation gives a minimum of three hundred fifty million database transactions per day, or fifty million parcels scanned seven times. The same specifications can also be stated in terms of database transactions per second. This conversion provides numbers that are easier to grasp and compare during benchmarking. Fifty million parcels scanned seven times per day is equivalent to four thousand and fifty one scans per second.

The generation of simulated scanning data is done in a two-node cluster. Each node has a CPU with eight cores and two sockets. Each of the nodes in the cluster performs different MemSQL tasks for optimizing performance. One node is called the aggregator, and it receives parcel information that transmits to other nodes in a way that maximizes cluster performance. The nodes that receive and process the queries are called leaf nodes. This aggregator-leaf node architecture has been designed around the development of MemSQL database code in order to provide smooth and reliable data manipulation in large scale operations of companies such as MemEx.

The database environment presented in this report is put to the test by timing a large number of simulated transactions. The underlying hardware and the MemSQL software are tested rigorously until they reach their maximum performance. This test even exceeds the transaction volume of MemEx in real time from the back-end point of view.[1] At the end of this report, results are presented to demonstrate that MemSQL performance exceeds MemEX data transaction demands.

Data Specification and Generation

Each parcel is given a unique, alphanumeric SCAN_HASH value representing a consistent hash of the meta data associated with the postage stamp on that parcel. Using SCAN_HASH, MemEx can check all instances where a parcel has been scanned. There are seven instances for each unique SCAN_HASH in the database table(Listing 1.). Each identical SCAN_HASH in the table is distinguished from the others by a unique SCAN_ID associated to every parcel scan. Each parcel scanning entry is inserted one row at a time. Software was developed in python to insert and read rows from a table in the database that has the following columns.

Listing 1: SQL code used to generate a table for tracking parcels

```

CREATE TABLE SCANS(
  SCAN_ID                    BIGINT AUTO_INCREMENT NOT NULL,
```

[1]The benchmarking software also does a small amount of additional processing that would not be required in an actual implementation of MemSQL for MemEx. See hardware and software considerations for details.

```
  SCAN_HASH               VARCHAR(11) NOT NULL,
  SCAN_TYPE               VARCHAR(3),
  SCAN_COUNT              INT,
  MACHINE_TYPE            VARCHAR(10),
  SEQUENCE_CODE           VARCHAR(5),
  LOAD_DATE               TIMESTAMP,
  PRIMARY KEY (SCAN_ID)
)
```

Performance tests

Several different MemSQL tests are performed in order to find the maximum number of inserts and reads that MemSQL and the two-node cluster can handle. The tests are:

- Number of single row inserts per second.
- Number of reads per second using either a SCAN_ID or SCAN_HASH selection criteria.
- A comparison between SCAN_HASH and SCAN_ID for reading data from the database.
- Number of combined inserts and reads per second (using SCAN_HASH).
- Maximum amount of data, and individual rows of scanning information, that MemSQL can handle.
- A comparison between loading data from a file and individual inserts.

Hardware and Software Considerations

In an actual deployment of a scanning system for MemEx parcels, data would be generated remotely and sent to a server for storage and analysis. Data would be obtained from parcel information and scanning equipment, so this information is encoded, encrypted and transmitted at no additional computational cost to the database.

To compensate for the lack of data generating equipment, the benchmarking software needs to create data for SCAN_HASH, SCAN_TYPE, MACHINE_TYPE and SEQUENCE_CODE. The benchmarking code also needs to ensure that there are up to seven SCAN_IDs for each SCAN_HASH. Additionally, each of the seven scans generated are shuffled with scans from other parcels so that they reflect the random nature in which scanning data is received.

The specifics of the scanning information transmitted is irrelevant for the purpose of evaluating MemSQL's performance, with the exception of SCAN_HASH and SCAN_ID, which are used in inserting and retrieving data. In an actual MemEx transaction, MemSQL needs to assign only the appropriate values for SCAN_ID, SCAN_COUNT and

LOAD_DATE, and the benchmarking software would do the same plus the additional burden of creating a unique SCAN_HASH for each parcel.

Generating and submitting data sequentially into a database one single row at time is not a realistic representation of how data is received for storage in the MemEx scenario. These strictly sequential, single thread, type of communications between aggregator and leaf nodes are slowed down by I/O delays. In order to emulate the concurrent arrival of scanning information to a MemSQL aggregator node, a series of threads that submit a set of SQL transactions is created. This is a considerably faster way to transact queries between aggregator and leaf nodes. It is also how enough queries are generated to simulate the transaction volume of four thousand and fifty parcel scans per second at MemEx. In this way the software used for benchmarking creates a similar scenario to that used in an actual implementation of MemSQL for MemEx.

Benchmarking software specifications

The benchmarking software to test MemSQL for MemEx is written in Python, and it inserts and reads data into MemSQL. The script requires the following arguments from the user: the number of inserts/read operations and the number of threads to simulate concurrency. When the software reads data from the database, it is necessary to include either SCAN_HASH or SCAN_ID in the arguments as selection criteria. When the software inserts data into the database, it is necessary to include in the arguments the number of parcels that have been entered to avoid having parcels with more than seven scans.

The software is then run multiple times while increasing the number of SQL transactions and threads. The goal is to find the number of SQL transactions that attains the maximum number of reads and inserts per second.

Maximizing performance with parallel computing

Because MemSQL does not query directly from the disk, the limit of transactions per second is bounded by I/O limitations between the nodes in the cluster. This is because the python interpreter is awaiting the result of a function call that is manipulating data from a remote source such as a network address. This hurdle is overcome by dividing the SQL workload among the threads that optimize I/O communications. But threads in Python are inherently a single process under the disguise of concurrency and parallelism.

Even when the inserts and reads per second limit is reached for a single core and its socket, the underlying multi-core architecture of the CPU is leveraged to reach an even higher number of SQL transactions per second. The aggregator runs on an eight core CPU with two sockets; thus, taking advantage of an additional core and socket doubles performance. This performance increase is obtained with a bash script that submits two of the same benchmarking python programs to the operating system to run in parallel and independently.

Listing 2: `Bash script used to run mix.py in parallel. The & at the end of the script makes the first line of code run in the background, and it proceeds immediately to the next line. The operating system takes care of running the two line on different cores and using the two available sockets.`

```
#! /bin/bash

time python memex.py 3 5 100 0 &
time python memex.py 3 5 100 0 &
```

Timing runs

Time is measured by preceding the python software with UNIX's *time* command. The *time* command returns three numbers that are labeled *real*, *user* and *sys*. *Real* is the actual length of time used by the program. This number is not useful for benchmarking because it accounts for when the script is waiting idle to run. Adding *user* and *sys* gives reliable times for the benchmarking software. This translates to how much CPU time is used by the user's software. This is how transactions per second are calculated in this proof of concept of MemSQL for MemEx.

Results

Tests were run up to the maximum number of transactions per second possible without getting error handling exceptions caused by excessively large query loads. For these reasons, larger query loads became impossible even as the number of SQL transactions kept going up in the last stable runs. This maximum number of SQL transactions is slightly larger than other stable query loads that did not produce exceptions. From the tables below, it is concluded that not much is gained by getting close to the point where large query loads cause error handling exceptions.

Benchmarking results for reading data from the database using a core and a socket. Using SCAN_HASH as the selection criteria for retrieving scanning information from the database is faster than SCAN_ID (table 1).

Table 1: Retrieving information from the database.

	Using SCAN_HASH		Using SCAN_ID	
Number of Reads	Time	Reads per second	Time	Reads per second
10,000	3.05	3278.69	3.74	2673.79
12,500	3.78	3306.88	4.81	2598.75
15,000	4.52	3318.58	5.58	2688.17
20,000	6.02	3322.25	7.54	2652.51
25,000	7.50	3333.33	9.23	2708.56

Inserting data into the database. Inserting data into the database reaches a maximum below 27,000 insertions per second (Table 2).

Table 2: Inserting scanning information into the database.

Number of Inserts	Time	Reads per second
700	0.32	2215.19
1400	0.56	2477.88
2100	0.81	2576.69
2800	1.07	2621.72
4200	1.57	2668.36
6300	2.35	2678.57
8400	3.12	2688.00
10500	3.89	2699.23
12600	4.69	2688.29
14700	5.47	2685.42
16800	6.25	2689.29
18900	7.00	2698.07
21000	7.84	2679.94
23100	8.68	2662.21
25200	9.48	2659.35
27300	10.33	2642.79

Reading and inserting data at the same time. Results are comparable to inserts and reads done separately. For this test, the number of threads used were divided evenly among inserting and reading for this test (Table 3).

Table 3: Inserting and reading parcel scanning information.

Time	Inserts	Reads	Total transactions	Transactions per second
1.71	350	5000	5350	3123.18
2.10	1400	5250	6650	3165.16
2.38	2100	5500	7600	3190.60
2.77	3150	5750	8900	3211.84
3.47	5250	6000	11250	3242.07
3.95	6300	6500	12800	3242.15
4.41	7350	7000	14350	3250.28
4.84	8400	7500	15900	3287.16
5.28	9450	8000	17450	3304.30

Benchmarking results using two cores and two sockets. Timing using two cores and sockets (Table 4)

Table 4: Parallel inserting and reading parcel scanning information.

	Time 1	Time 2	Ave. Time	Transactions	Transactions per second
Reads	3.10	3.09	3.09	10,000 x 2	6,453.69
Writes	3.14	3.13	3.13	10,500 x 2	6,694.29
Reads & Writes	3.45	3.43	3.44	11,250 x 2	6,540.70

Maximum Data Storage

The amount of free RAM memory was obtained with the UNIX command *top*. The aggregator node showed 30.7 gigabytes of free memory. In order to find out how much data can be handled by MemSQL, the database with the parcel scanned table was dumped into a SQL file. From the size of the table and the number of rows, a number was calculated for the maximum capacity of the two-node cluster in this test. A total of 474'601,526 rows can be stored in this two-node cluster. This is equivalent to 1.36 days of scanning parcels at MemEx at current volumes before running out of RAM capacity.

Loading data from a file

Data that was produced concurrently with the benchmarking software took several minutes to load at maximum transactions per second. When the same amount of data in the database was dumped to a SQL file, it only took a few seconds to load it again. This is an indication that in this test, the database transaction limits of scanning information is due to the two-node cluster. MemSQL can handle larger data loads if the information is packed and streamed more efficiently using files. It is the management of many single queries at the time that cause a performance bottleneck at the hardware level.

Conclusions

Limitations due to I/O and hardware are easily overcome by using the right number of aggregator and leaf nodes. This number is calculated so that the maximum SQL transaction load that causes handling exceptions is never reached. This two tier system allows for easy expansion of the cluster to meet MemEx needs. The amount of RAM memory that MemEx can handle is large enough for handling a day of operations at MemEx. This RAM capacity can also be safely managed if sufficient number of aggregators and leaf nodes are used. This hardware redundancy will guarantee a flawless MemEx daily operation with MemSQL.

To anticipate future expansion of MemEx operations, Shard Keys can be used by to increase the number of parcel scans per second if more than one leaf node is present in the cluster. Shard Keys divide the table among leaf nodes so that all instances of a package are found in the same leaf node. This allows the aggregator to work more efficiently by not having to search all leaf nodes for a SCAN_HASH request or an insert.

This also allows for queries to be done in parallel in a way that maximizes CPU socket usage and minimizes network traffic.

Compared to an actual deployment of MemSQL for MemEx, the additional lines of simulation code used by the benchmarking software increase the aggregator's computational burden. Despite of this, MemSQL surpasses the transaction volume at MemEx even with the smallest possible two-node cluster, and without using shard keys. When tests are performed in parallel, MemSQL can load up to 6,540.70 mixed reads and inserts per second. This is 62% higher than the 4051 transactions per second that MemEx handles daily.