In-Memory Database and MemSQL

Dwi Prasetyo Adi Nugroho (dnugroho@ulb.ac.be)

Haydar Ali Ismail (haismail@ulb.ac.be)

I. Introduction

The development of technology dimension in information system allow more data to be collected, stored, and used to support decision making. For some businesses, time become a critical aspect of their data driven decision support. For instance in Pinterest, the customer data they gathered for A/B testing today may not be relevant to be used tomorrow. Therefore, analysis should be done right away in real-time manner [1]. Other company that also need a real-time analysis is Teespring. With more than 300 million events happen a day, Teespring could use it's buyer activity to create an analysis to help the sellers to increase their lead to purchases [2].

The need of this real-time analysis require a database system that could perform analysis on the whole data while new data being inserted and be added to the analysis. The standard batch mode data warehouse is not suitable for this kind of task since the data usually staged in batch before being inserted together and used in analytic. Therefore, the utilization of main memory to store database, which often called in memory database, become inevitable.

To this end, there are several in-memory database management system on the market. However, in this paper we are interested to have an in-deep study on developing real-time application using MemSQL. One of the reasons to choose MemSQL is that its suitability to perform both OLTP and OLAP tasks. Another reason is in the usage of SQL, a standardized query language that proven to work well for decades, as interface to interact with the data. Finally, MemSQL is designed to be a distributed system. This means that it can be easily scaled horizontally by adding more nodes to the system.

We organize this paper into seven chapters. The first chapter is introduction to the problem that will be discussed in the paper. The second and third chapter is explanations of basic knowledge of the technology we are using, that is in memory database and MemSQL respectively. In the fourth chapter, we present the requirement analysis of the system that is going to be developed. We then detail the implementation in chapter five. In chapter six, we discuss several test and experiment to the developed systems. Finally, we conclude our study as well as providing some recommendations in chapter seven.

II. In-Memory Database

A. Motivation behind and development of In-Memory Database

Idea of In-Memory Databases already initiated since 1970s. It started as a prototype as system memories were generally small and expensive. As memories getting cheaper, people start to use in-memory database as a secondary database to cope with frequent selection so that it does not need to fetch the data from the primary database every time. Today we finally got into the point where average RAM size available in the market already enough for most application databases, making it possible for developers to load the whole data into the memory instead of having to load from disk back and forth.

In-memory databases primarily rely on main memory for data storage in contrast with database management systems that use disk-based storage mechanisms. In-memory databases are faster than disk-optimized databases because memory access is faster than disk access, the internal optimizations are simpler in memory and they need to execute fewer CPU instructions because of less overhead.

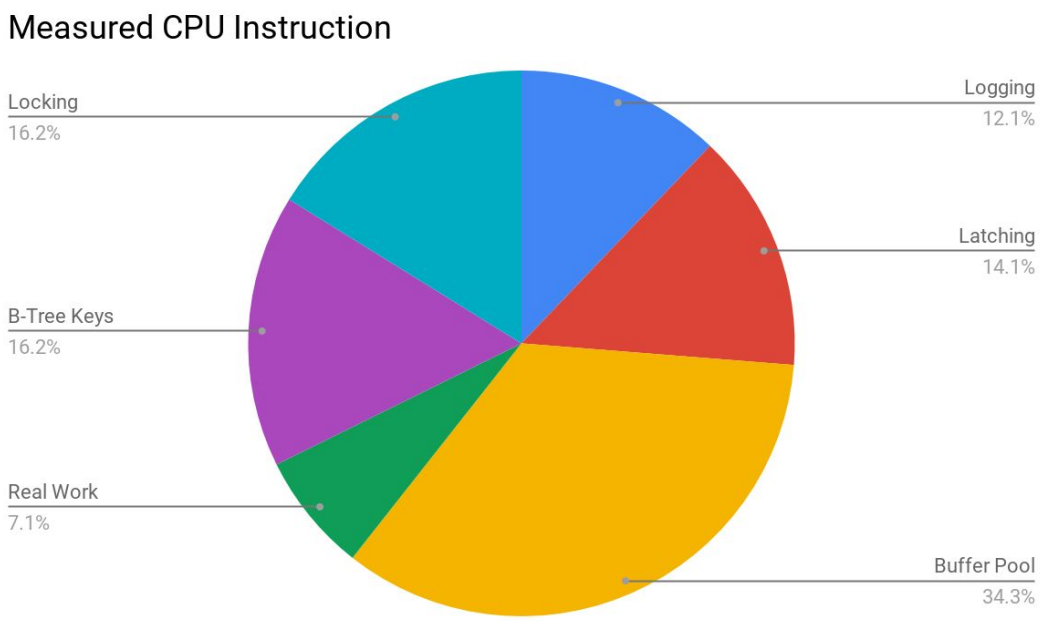

Figure 1. CPU instructions when using disk based databases [3]

The chart above shows the measured CPU instruction of disk based databases. In which shows the overheads that the CPU have to endure when working with disk oriented databases. In fact only about 7% of the CPU instructions that is used to execute the real database work, the other 93% is what the disk oriented database need the CPU to do in order to make the disk based databases works.

In-memory databases find applications where response time is critical, like those running mobile advertising networks or telecommunications network equipment. Due to multi-core

processors that can address large memory and less expensive RAM, main-memory databases have gained a lot of traction in the past two decade or so.

A potential shortcoming of in-memory data storage is the volatility of RAM, that is, in the event of a power loss, the data stored in the memory is lost. The introduction of non-volatile RAM technology will enable in-memory databases to maintain data in the event of power failure while running at high speeds.

B. How In-Memory Database Works

Much of the existing DBMS in the last few decades, system applications relies on disk-based databases. This is happened due to limitation of existing hardwares in the past which includes the unavailability of multi-core processors and the availability of RAM. Due to this issue, in the original development of DBMS people are forced and only able to utilize disks hence developers in the past tend to developed disk oriented databases. The main issue with disk oriented databases is that the data is stored in a disk, which is widely known for its slowness compared to system memory.

The primary storage location of the database in disk-base databases is on a non-volatile storage. Then as the system requires to look up for data in the database, it will use an in-memory (volatile) buffer pool to cache blocks from the disk. So basically the data will be transferred back and forth between disk and RAM as required. The DBMS will keep those blocks that it fetched from disk in the memory as long as it still have enough space. The motivation to do is that if the same data going to be fetched again, the DBMS will just have to lookup into the system memory and if it is exist in memory then it can just simply reuse it. In cases of no more free buffer frames, the DBMS have to decide which blocks of data need to be removed.

However, available hardware in the market nowadays is much different when the development of DBMS just started. Memory modules nowadays are getting bigger and can even surpassed databases size. This invoke the popularity of in-memory databases, basically because of the speed. However as I/O operations not being the biggest bottleneck anymore in in-memory database systems, other existing issues escalated such as pointer chasing or cache line misses. As in-memory DBMS does not necessarily need to store the database in slotted pages the way disk-based DBMS do, most of them still organize tuples in blocks. The main difference is that now the pointer in the block of memory can be referenced directly to the block of memory corresponding to that data instead of having DBMS to get the data from disk and load it into memory first.

Another different property is that in memory database avoid the use of buffer pool. By this means that the system don’t need to perform a hash table lookup the actual memory block for a given tuples. It also don’t need to calculate offset to get to the tupple’s pointer on the memory block. Furthermore, latch contingency (the CPU operation of locking and releasing a memory resource) can also be eliminated.

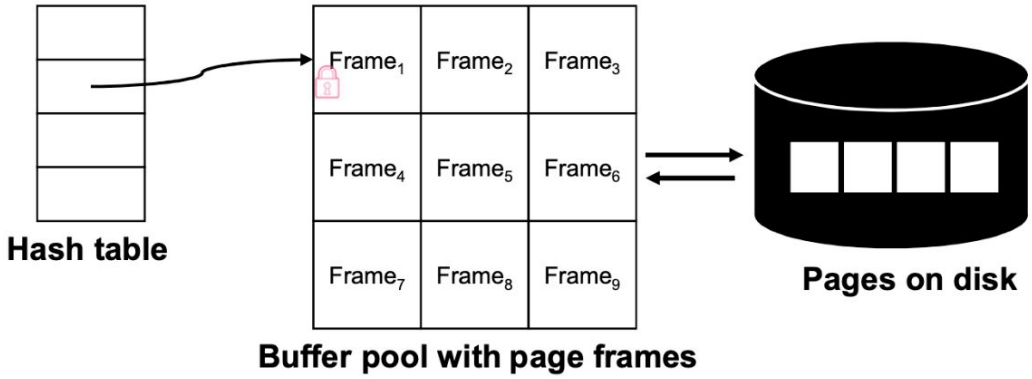

Figure 2. Buffer Pool in Disk Oriented Database[5]

In memory databases also arrange index in different way with the disk-oriented databases. Similar to the tuple data organization, it avoid the use of memory buffer. Hence, it gains similar advantages as mentioned before. The index is also pointing directly to the tuple's pointer. It also not necessary to arrange the index in clustered index mechanism since access to any index will have the same cost in either sequential or random access. Moreover, as the index not arranged in a specific page, the dbms can allocate memory resource as much as the index needed. This will help to avoid underutilized memory block that happen in disk-oriented databases.

In term of query processing, in memory databases avoid using the iterator model to get next tuple (the getNext() call). Instead, it will compile the transactions and queries directly into machine code. This method boost in memory databases performance since in disk oriented database, the getNext() may be called many times to execute a query. In every execution of getNext(), the function may chain another function call such as interpreting bytes to be processed that will increase the number of CPU instruction to be executed. The getNext() function may also be called virtually (using function pointer), that is slower than direct call.

An in-memory database is a kind of database that keep the whole information in system memories. By keeping in mind the volatility character of RAM, creators of in-memory databases able to create databases that loaded in memory but still able to persist new changes even if there are events that will remove the content of RAM such as a power loss.

The way in-memory database with persistent works is that for each changes operation, it will create a transaction log to a disk. Note that even it is running those operations on a disk that is considerably slow compared to memory, this is not going to be a problem as it will be done in queue-manner and disk are fast enough to do this because the disk does not have to do random access and only need to write to disk sequentially. As for accessing the data, the database will use the content that is stored in memory to do it as fast as possible.

Sometimes these transaction logs can be very long, thus it can actually delete outdated translation logs. This is called snapshotting where basically it will make a copy of the whole

dataset and the last transaction log. These information are enough to recover your database to the current state on system startup or even on power loss scenario.

C. In-Memory Database Softwares

There are variety of in-memory database systems in the market such as Oracle TimesTen, IBM solidDB, VoltDB, MemSQL, etc [4]. Even tho4gh they all use the same concept as to keep loaded data in their memory, they are not necessarily using the same technique to handle everything. For example, the way Oracle TimesTen and VoltDB handle concurrency control are different and it might have great performance difference in certain cases which makes each of the database applications unique on its own.

III. MemSQL

A. How MemSQL Works

MemSQL is a distributed relational database management system that combines in-memory row-based storage and disk-based column-oriented storage. It uses code generation to compile Structured Query Language into machine code.

On April 23, 2013, MemSQL Inc. first, introduce the initial release of MemSQL database. MemSQL is mainly developed to tackle real-time data warehousing. The vision of MemSQL is to enable analytic capability of business needs to be functioning in real-time. To achieve this, MemSQL is designed to make it possible to run an application on a data warehouse while maintaining fast data ingestion. This is implemented by shifting away from the traditional batch process to continuously load data to the system.

B. Features

1. Storage mechanism

As other in-memory databases, MemSQL store its data primarily on the main memory. To cope with the volatility of main memory, MemSQL write logs of its transaction to the disk. These logs are then compressed into snapshots. When the system needs to recover data, these snapshots and logs are then used to load the data back to main memory. MemSQL offers two kinds of storage mechanism: rowstore and columnstore and explained as follow.

a. Rowstore

The rowstore method store the data completely in memory. Similar to other DBMS, a row is used as a unit to store a tuple. This method is suitable for an operational database that performs frequent insert, for example, the stock price database where the stock price is updated in every small timeframe.

b. Columnstore

This method takes advantage of secondary memory to store the database. A column is used as a unit to store data. Data in one column will be stored in one place. This will speed up aggregation operations such as sum, count, etc as it will only need to look up at a single place. In the row store method, this will require a full scan of each tuple and get only the column that we are interested in. This speed up made it suitable to be used in the data warehouse projects. Aside from using the secondary memory, columnstore also store some metadata of a column in the main memory to increase query performance. To cope with disk write bottleneck when inserting data to columnstore, MemSQL use a kind of staging area in the main memory to temporarily store newly inserted data, and periodically store data from staging area to the disk.

2. Relational

MemSQL belongs to the group of relational databases. This kind of database can be accessed by the user using the SQL interface. MemSQL uses the ANSI SQL to access the database.

3. Connection

MemSQL uses the MySQL wire protocol to connect the client to the MemSQL server. This means that the user can use any SQL client that works well with MySQL to connect to a MemSQL server and perform SQL queries. Another advantage from using wire protocol from the world's most popular open source DBMS is that existing application that previously connects to MySQL can easily shift to connect to MemSQL without complicated configuration.

4. Distributed

MemSQL is designed to be a distributed system. The distributed system means that instead of installing it on one host, MemSQL can be installed on several hosts that work together, enabling the user to scale out MemSQL by adding more host to the system. The concept also enables the user to install MemSQL on a commodity hardware.

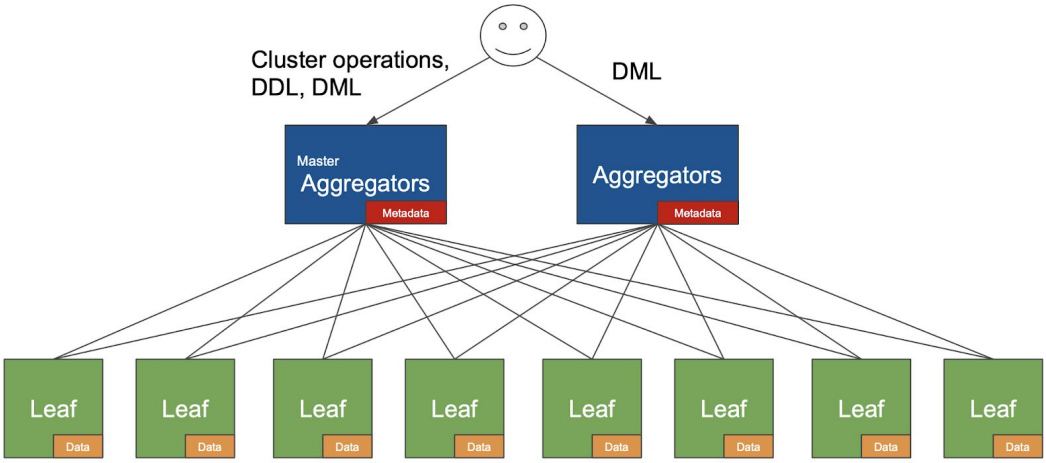

Figure 3. Illustration of nodes relationships in MemSQL

MemSQL cluster consists of multiple nodes. These nodes can be classified into aggregation nodes and leaf nodes. Aggregation nodes then can be further classified into a master aggregator and child aggregators. A cluster consists of at least one master aggregator node and one leaf node. There should be only one master aggregator in a cluster. All aggregator nodes expose an SQL interface to interact with the user, while leaf nodes hold the data. The master aggregator has an additional capability of running both DDL and DML, while child aggregators can only run DML. To load data, user will fire a query through the aggregator. In each aggregator, it have metadatas about leaf nodes and which data each of them correspond. When it receive the query, it will look in its metadata and will forward the query to the leaf node where the data is stored in according to the metadata.

IV. Requirement Analysis

A. Problem Definition

In today's industry, being able to intelligently make a decision based on on the data is essential to many companies. It now become a new field for companies to compete with their competitors. Retails is one of the sectors where the utilization of data can decide whether they can keep their customer buying their products, or even acquiring new customer from the competitors.
There are various ways of utilizing the data that can be done by retails companies to get insights. One of them is by having a dashboard that can give overviews to managers on what is going on and provide hints on how to react with the condition. However, in order to react correctly, the dashboard need to provide most actual data. Aggregations and visualization must be done on the current data to avoid wrong interpretation.

The problem might look trivial at the beginning. However, as the rate of the data coming to the system increased, the requirement of doing near real-time dashboard update become difficult to fulfilled with standard disk-based database system.

Looking at this opportunity, we decided to implement near real-time dashboard application using MemSQL that can perform aggregation and visualization while new data is being inputted. The following section detailed the requirement that need to be implemented in the system.

In order to simulate the data application, we use retails data from TPC-C OLTP Benchmark as our case study. TPC-C perform its benchmark by simulating retail transaction which perform insert and update to the target database. While the benchmark is being run, we run analytical queries that and used the result to create dashboard visualization.

B. System Requirements

1. User

The dashboard application will be used by managers. This user will perform analytical operation such as selecting aggregation by different dimensions and putting it in the visualization.

2. Functionality

a) Latest sales
The first element in our dashboard shows total sales over last periods of time. We set the time frame to 'overall', which shows yearly sales of all years, 'year' which show total monthly sales of this year, 'month' which show total daily sales of this month, and 'day' which show total hourly sales of today.

b) Top k Entities
This element show top entities based on the sales it made. We decided to choose top cities, top item, top customer, and top selling month.

c) Summary
This element shows statistic of current time. In this element we show total sales of this year, average monthly sales of this year, total sales of this month, and average daily sales of this month.

C. System Design

1. System Architecture

We separate the application into front-end modules, back-end modules and the data ingestion module. Front-end module will take care of presenting data to the user either by providing tables or visualization as well as receiving user's input such as how the data will be aggregated. Backend modules will parse user input into SQL queries and call the query to the database. It will also receive the result from the database and transform the result into a format that is easily parsed by the front-end module. The data ingestion module work separately with the two other modules. It will have its own connection to the database and will simulate the data ingestion to the database.

2. Simulating Data Ingestion

TPC-C is originally designed to measure OLTP throughput of a database. It use retail database as a model for an OLTP database as shown in the figure below. The content of reference tables are initially loaded to the database. After that, sales transaction is simulated by inserting data to several tables. Beside can be used as a benchmark, the implementation of TPC-C can also be used to simulate an OLTP workload. In our architecture, we put this implementation as data ingestion part and will be run simultaneously with the analytical query.

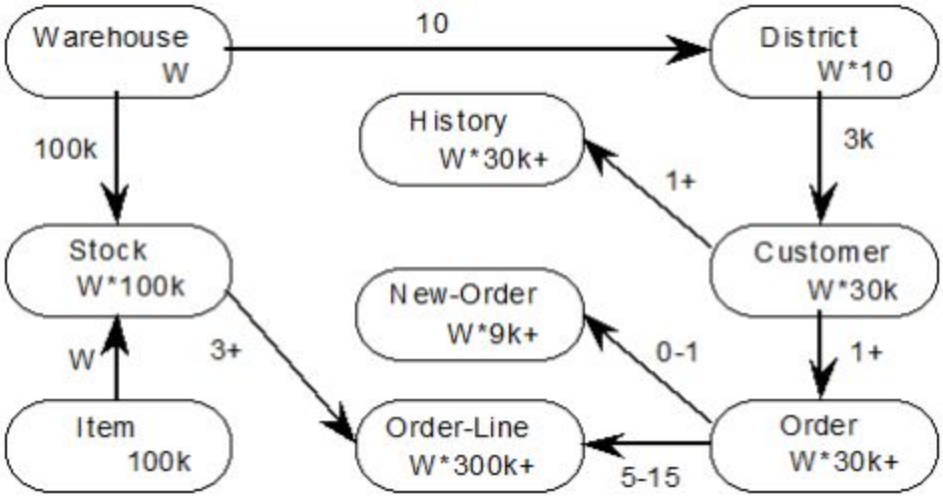

Figure 4. Retail database

V. Implementation

A. MemSQL Installation

MemSQL can only run natively on 64-bit Linux operating systems, but you can use MemSQL in other operating systems through docker. In this case study we will install it on Ubuntu 16.04. MemSQL minimum requirement is 4 cores and 8GB of RAM, so make sure the machine you are using have enough resources.

1. Download MemSQL tar file

   ```
   $> wget
   http://download.MemSQL.com/MemSQL-ops-6.5.11/MemSQL-ops-6.5.11.tar.gz
   ```

2. Extract tar file

   ```
   $> tar zxvf MemSQL-ops-6.5.11.tar.gz
   ```

3. Run the installer

   a. Move to the directory where it was extracted

   ```
   $> cd MemSQL-ops-6.5.11
   ```

   b. Install MemSQL as a simple cluster with using the **install.sh** file

   ```
   $> sudo ./install.sh
   ```

   c. Type "N" when prompted to install MemSQL on this host only

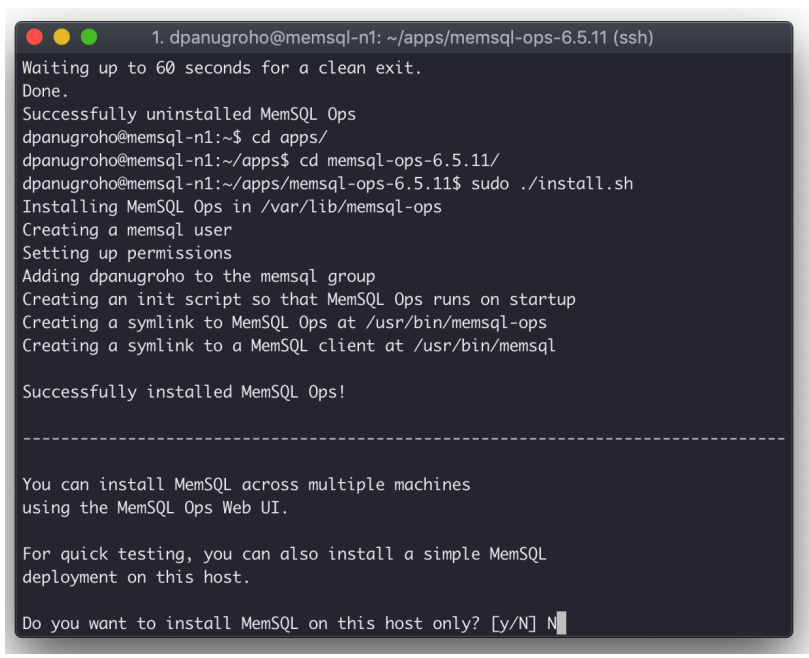

Figure 5. MemSQL installation using Ops

   d. Open the server's IP in the browser using port 9000. Make sure this port is set to open in the firewall rule. If it is installed in localhost, open following address:
   http://127.0.0.1:9000

   e. Now we can use this web UI to configure the cluster. Select Developer Edition of MemSQL and cluster installation on single

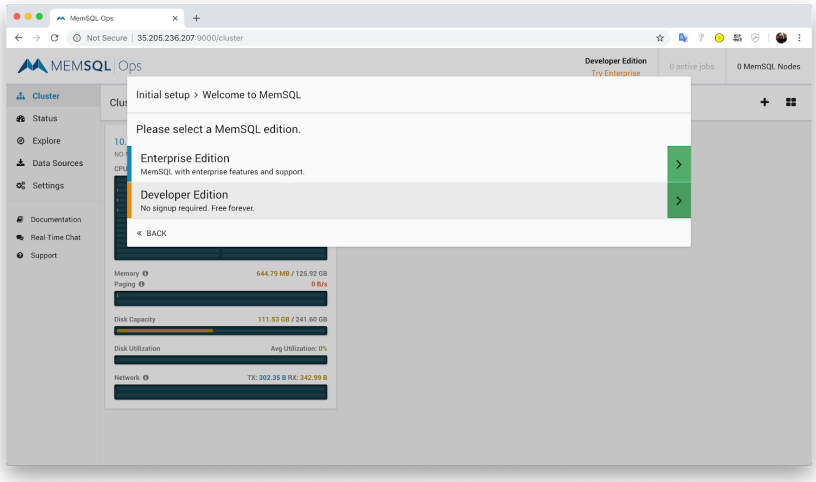

Figure 6. Choosing MemSQL edition

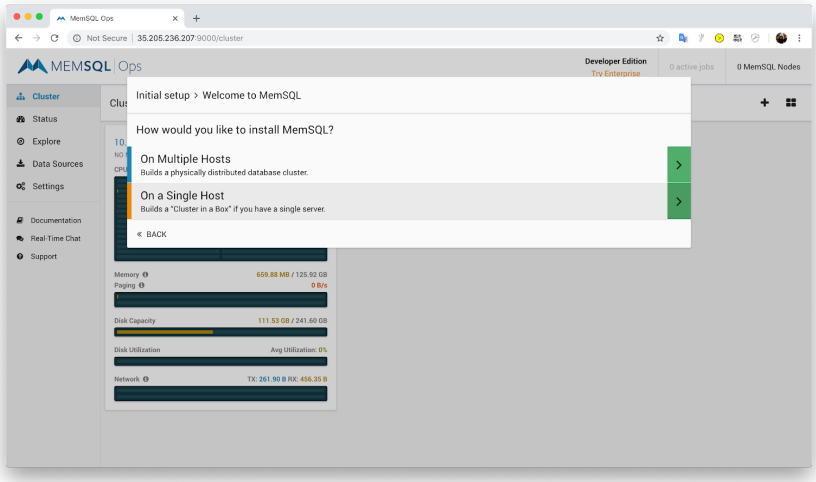

Figure 7. Choosing cluster configuration

f. Add nodes to the cluster. On this experiment, we are using 1 Master aggregator, 2 child aggregators, and 4 leaves

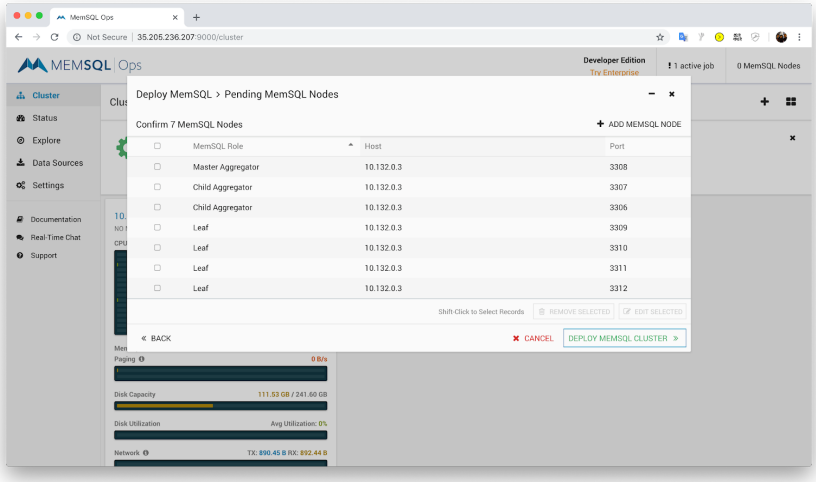

Figure 8. Choosing nodes

g. Click on Deploy cluster when finish configuring the node. MemSQL Ops will automatically download and install MemSQL on each node. We can start accessing the cluster using the master aggregator

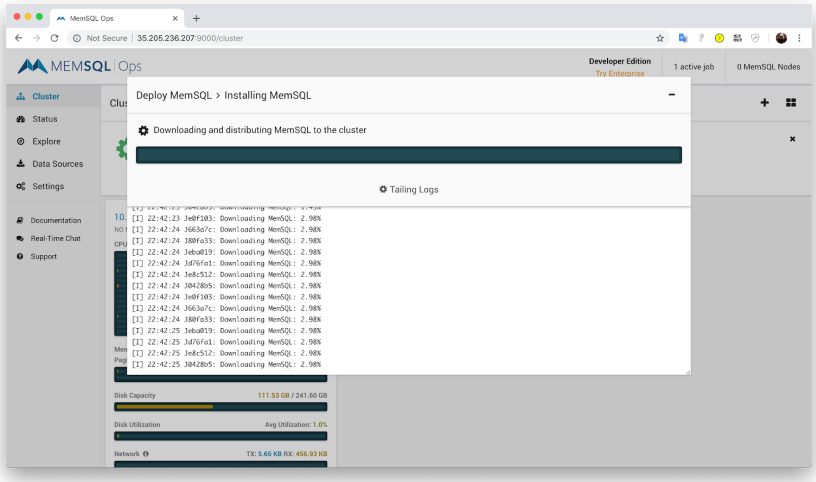

Figure 9. Installing MemSQL on each node

B. Implementing Data Warehouse Backend

We used flask framework on python to implement our backend service. This backend service will serve endpoints for each of the functionality mentioned before. Each endpoint will invoke an execution to an sql query. The result is then converted to json and returned. The detail of each endpoint is described in the table below.

URL	Descrition
/sales/last/overall/	Lists total yearly sales from the first date of transaction up to current date.

/sales/last/year/	Lists total monthly sales of current year.
/sales/last/month/	Lists total daily sales of the current month.
/sales/last/day/	Lists total hourly sales of current day.
/sales/summary/month/avg	Gives average daily sales of current month.
/sales/summary/month/sum	Gives total sales of current month.
/sales/summary/year/avg	Gives average monthly sales of current year.
/sales/summary/year/sum	Gives total sales of current year.
/sales/top/customer/<n>	Lists top <n> customer have highest purchase.
/sales/top/city/<n>	Lists top <n> cities that have highest purchase.
/sales/top/item/<n>	Lists top <n> selling items.
/sales/top/month/<n>	List top <n> month and its year that have highest sales.

C. Implementing Data Ingestion Simulation

In this implementation, we extend the TPC-C benchmark implementation that is publicly available at https://github.com/apavlo/py-tpcc. The implementation is written in python and already provide an abstract class to implement TPC-C benchmark in any database. We extend this implementation by adding SQL queries and TPC-C transaction functionality for MemSQL. Most of our implementation is based on the sqlite implementation that is also available on the repository.

Some notable changes that we made from the base sqlite implementations are:

- DDL. MemSQL doesn't support foreign key constraint, hence it should be removed from the DDL script.
- Connection object. We need to change the connection object to be able to connect to our MemSQL database. As mentioned before, MemSQL is compatible with any MySQL connector. In this case, we used the official MySQL connector and use the standard connection configuration.
- String formatting should be change from using question mark '?' to '%s'.

D. User Interface Implementation

To provide user with a friendly interface, we create a front-end client that consume the backend api and then display it in the form of charts and tables to the user. We use flask framework to structure our front-end application. We also use Bokeh library to display interactive charts. Finally, we use Bootstrap css library to style our application. The screenshot of our front-end application is as follows:

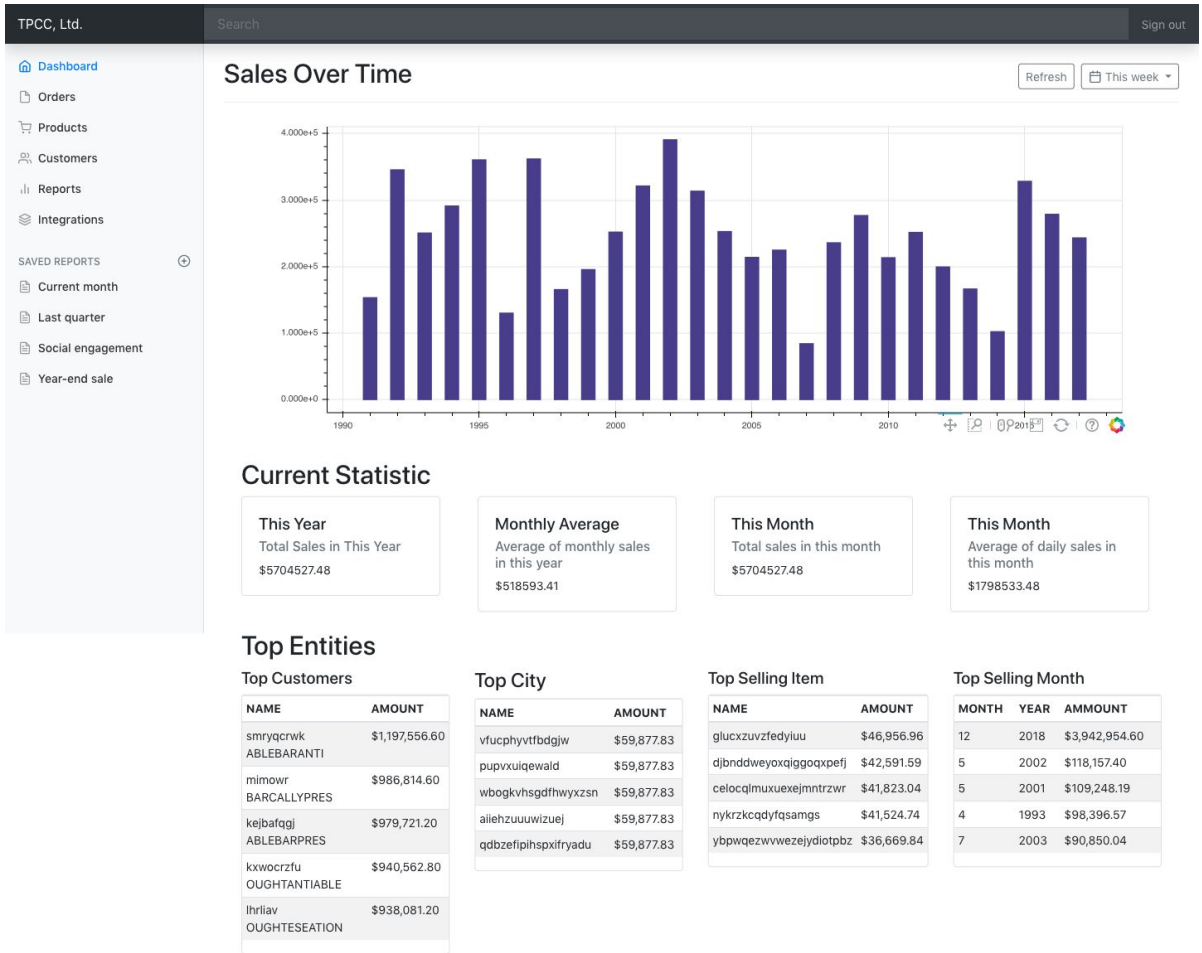

Figure 10. Statistic dashboard

VI. Performance Test and Experiments

In this chapter, we report performance tests that is performed to the system. The test is done by varying the scale factor of TPC-C data and running additional analytical queries. The TPC-C insertion rate is then measured and checked to observe its change due to larger data and heavier analytical load.

We follow analytical query that is propossed by Cole et al in [6] as follows:

Query	File	Description
Query 1	/additionalquery/query1.sql	Return total amount and quantity of all shipped orderlines given by a specific time period. It also informas about average amount and quantity plus the total count of all these orderlines ordered by the orderline number.

Query 2	/additionalquery/query2.sql	Return unshipped orders with the highest price amount for a customer within a given state and with orders newer than a specific timestamp. Sorted by descending amount.
Query 3	/additionalquery/query3.sql	Return all orders with orderlines or just parts of them that are shipped after the entry date of their booking.
Query 4	/additionalquery/query4.sql	Return all total amount of archived revenue from orderlines which were delivered in a specific period and a specific quantity.
Query 5	/additionalquery/query5.sql	Return the count of amount of orders grouped by the number of orderlines in each order attending the number of orders which are shipped with a higher or lower order priority.
Query 6	/additionalquery/query6.sql	Return the percentage of revenue in a period of time which have been realized from promotional campaigns.
Query 7	/additionalquery/query7.sql	Return yearly loss in revenue if orders with a quantity of more than the average quantity of all orders in the system would be taken and shipped to customers.
Query 8	/additionalquery/query8.sql	Return rank of all customers who have ordered for more than a specific amount of money.
Query 9	/additionalquery/query9.sql	Return the revenue achieved by some specific attributes, as the price, the detailed information of the item and the quantity of the ordered amount of them.

Each scale factor will be tested with four different number of additional queries that run simultaneously. The queries will be randomly chosen from the available queries. Each test will be run five times and average TPC-C insertion rate will be reported.

The test result is as follows:

Number of Warehouses	Number of Simultaneously Executed Analytical Query	Average TPC-C Rate (Transaction/second)	Performance Degradation
1	0	64.296	
1	5	61.37	4.55%
1	9	62.366	3.00%
1	17	57.596	10.42%

5	0	61.464	
5	5	55.772	9.26%
5	9	54.426	11.45%
5	17	34.738	43.48%

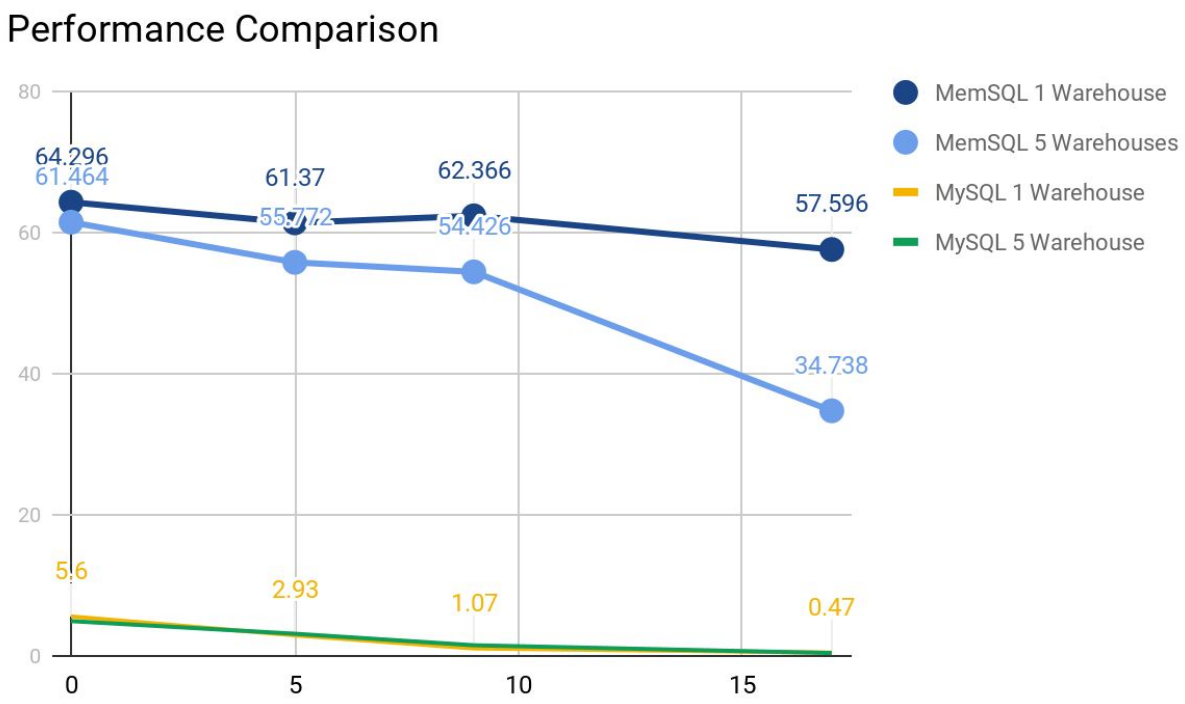

Figure 11. Performance comparison between MemSQL and MySQL

From the result, we can see that although there's a noticeable performance degradation on larger number of warehouse and analytical query, both insertion/OLTP and OLAP operation still can be handled by the system. This means that the analytical query can be answered right away and does not need to wait to be collected and ingested in batch.

We also compare the transaction rate with MySQL. We can see that even without the analytical query, MySQL can only get around 5.6 transaction per second. This emphasis the benefit of in memory database as we mentioned before.

VII. Conclusion

The requirement of having having real time dashboard that perform aggregation, summarization, and other analytical operation can be done using In-Memory database with help of development tools that is available for MySQL. This kind of database uses main memory as main storage for its data. This

makes the operation can be performed faster compared to the disk-based database. One of In-Memory databases that available on the market is MemSQL. The mixed test of TPC-C OLTP workload and OLAP workload from analytical query shows that the system still able to give answer to analytical query while being OLTP transaction being ingested into it. Although companies still have to consider the pricing and the technology maturity, we believe that investing in using in-memory database for could give benefit for companies in the future.

VIII. References

[1] https://www.MemSQL.com/blog/pinterest-experiments-in-real-time/

[2] https://www.MemSQL.com/blog/teespring-real-time-analytics/

[3] Harizopoulos, Stavros, et al. "OLTP through the looking glass, and what we found there." *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. ACM, 2008.

[4] Lomet, D. “Bulletin of the technical committee on data engineering”. IEEE, 2013.

[5] Faerber, Franz, et al. "Main memory database systems." *Foundations and Trends® in Databases* 8.1-2 (2017): 1-130.

[6] Cole, Richard, et al. "The mixed workload CH-benCHmark." *Proceedings of the Fourth International Workshop on Testing Database Systems*. ACM, 2011.