IT4BI - Université Libre de Bruxelles

In-Memory Databases MemSQL

Gabby Nikolova
Thao Ha

Contents

Database management systems are the heart of any well-functioning software and applications. Traditionally, database systems keep the data on disk where it is non-volatile, but retrieving it from there takes time. That's where in-memory databases (also called as main memory databases) come into play. In-memory databases are widely used for time-sensitive applications. Whether they are providing directions to a hungry user looking for the nearest pizza place or live information to a data scientist trying to gain insight and improve their corporation, fast performance is essential. In today's world people don't have the patience to wait for information to load, we want instantaneous results. We think, "the faster the better" when it comes to software performance. In-memory databases differ from regular database management systems because the data resides in the main memory and not on disk. When the data is stored in the main memory the access time is magnitude faster than from disk therefore increasing the response time and transaction throughput. That means faster access to the data and a higher number of transactions for the system. In-memory databases are usually used for real-time applications, big data analytics, and monitoring and detection, just to name a few. In this report, we would like to present our general analysis about in-memory database and one of its representations in market place – MemSQL to know more about how in-memory database comes from theory to real life.

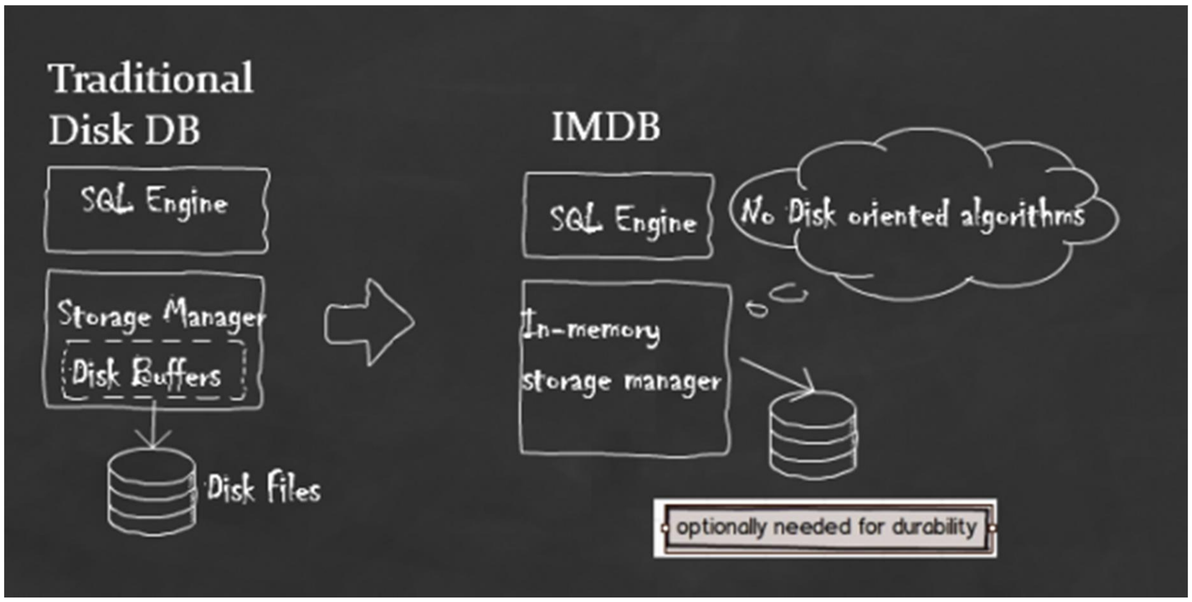

Traditional Disk DB
SQL Engine
Storage Manager
Disk Buffers
Disk Files
IMDB
SQL Engine
No Disk oriented algorithms
In-memory storage manager
optionally needed for durability

Figure 1: Difference in architecture between a traditional DB and in-memory DB
http://opensourceforu.efytimes.com/2012/01/importance-of-in-memory-databases/

I. In-memory Databases

1. Concept:

When talking about in-memory databases we can assume that the whole database will be kept in memory. In recent years this has become more possible because of the increased size of main memory and the decrease in price. However, in really large applications it is possible to find different classifications of data and partitioning of data into a few databases. There is hot data which is accessed frequently and cold data which is accessed less and keeps more historical data. Naturally, the database which contains the hot data will be kept in memory.

In traditional database management systems the primary performance goal is to limit the amount of disk accesses. In a regular database management system the main memory is usually used as a cache for some data to increase the database performance. Caching the frequently accessed records in memory may speed up reading the data, but when writing the data must still be written to disk, so it is not as efficient as in-memory databases. Another way people believe they can get the same performance as an in-memory database is by putting an entire disk database on RAM. However, the system is still designed for disk storage, has to compute the disk address, and then has to go through a buffer manager to check if the data block is in memory. After the block is found the tuple will be loaded into the application buffer where it will be used. It will be far more efficient to use the memory address instead if the tuple is already in memory. The in-memory database primary goal is to efficiently use the CPU cycles and memory storage. The algorithms, data structures, query processing, concurrency control and recovery must be reworked in order to achieve this.

2. Indexing:

Indexing in memory and indexing structures on disk have completely different purposes. When storing data in main memory the goal is to reduce the overall computation time and minimize the amount of memory used.

The disk uses blocks to store and access data. The disk is always going to read/write entire blocks even if the request is for smaller amounts of data. These read/write operations have a fixed cost, thus the goal is to limit disk accesses. Data in memory is stored with pointers to the tuples. The pointers provide access to the data value in the tuple and the tuple itself. Following the pointers is fast so accessing and querying the data becomes easier because there is no extra time allocated for getting it from disk. There are two main types of data structures for storing data in memory. These are structures that have a natural ordering and those that randomize the data.

a. AVL Tree:

One structure is the AVL tree. It is good because it uses binary search, which is fast. The tree is susceptible to becoming imbalanced (becomes too deep and there are too many nodes on one side) so it needs to be rebalanced with a rotation operation when nodes are inserted or deleted. One notable disadvantage is that it has poor storage utilization. Each node holds one data item, control information, and two pointers.

b. **B-Tree and B+ Tree:**

B-trees and B+ trees are used for storing data on disk because they are shallow and allow for a few node accesses to retrieve the data. These only keep data values on the leaves of the tree. For in-memory data storage a B-tree is better than a B+ tress because keeping data only in the leaves is a waste of space. Searching a B tree is quick because it uses binary search, storage utilization is good because the leaf nodes only hold data (without pointers) and they form a large portion of the tree.

c. **T-Tree:**

For ordered data there is another structure called the T-tree which gives the overall the best performance for inserts and deletes while maintaining a low cost for storage. It is a binary tree with many elements in a node.

- ***Internal node****:* t-node that has two subtrees
- ***Half-leaf node:*** one null child pointer, one none-null child pointer
- ***Leaf node:*** two null child pointers

Each internal node contains a number of items. It also has a leaf or half-leaf node that holds the predecessor of its smallest data item called the "greatest lower bound" and one that contains the successor of its greatest data item called "the least upper bound".

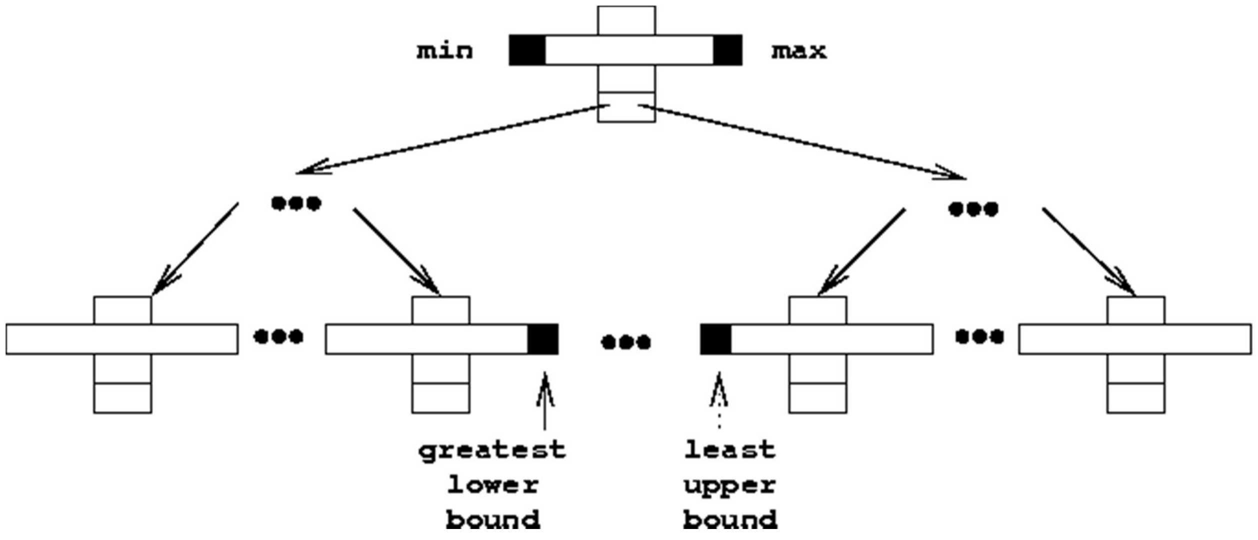


Figure 2: T-tree structure
https://en.wikipedia.org/wiki/T-tree#/media/File:T-tree-2.png

The minimum and maximum data items usually differ by a small amount but that is enough to notably decrease the number of rotations needed. Occupancy of internal nodes is flexible so that aids to limit the number of rebalancing rotations necessary when inserting or deleting. The rebalancing of a T-tree is similar to that of an AVL tree and the balance is checked whenever a leaf is added or deleted. If two subtrees of a node differ in depth by more than one level, then a rotation needs to be performed. Overall it has the good storage usage of a B Tree and requires fewer rotations on inserts/deletes which is the reason why it is a good option to use when trying to limit the space used in memory and improve query performance.

d. **Modified Linear Hashing:**

Modified linear hashing is a storage structure for keeping unordered data in memory. It is similar to linear hashing except optimized to be used in memory. Modified linear hashing users a dynamic hash table that grows linearly (unlike extendible hashing) with chained single item nodes. The splitting criteria are based on the average length of the hash chains rather than the storage utilization. The length of the chains provides a more direct way of managing the query speeds.

3. Querying In-Memory Database Systems:

Relational tuples can be represented as a set of pointers to data values. When querying in-memory sequential access is not significantly faster than random access. Using pointers is space efficient because actual values can be stored only once. An example of this is when we want to join a relation A with B on a common attribute. We do this by taking the smaller relation A and scanning all of the tuples. For each tuple we follow the pointer to the value of the common attribute and from there we follow the pointers to all of the tuples in B that use the common attribute and add them to the result. Additional storage might be needed to make sure we have all the pointers that lead to the tuples which have the common value in the relations. Temporary data structures can be created to hold results from each operation and consumed by the next operation. For example a temporary relation is created after a select. These temporary structures hold pointers referencing the relevant tuples. The temporary structures are compact since they no not hold the actual values. The structures speed up join processing. The performance of in-memory databases is evaluated mainly on the processing time because disk operations are conducted outside of transactions.

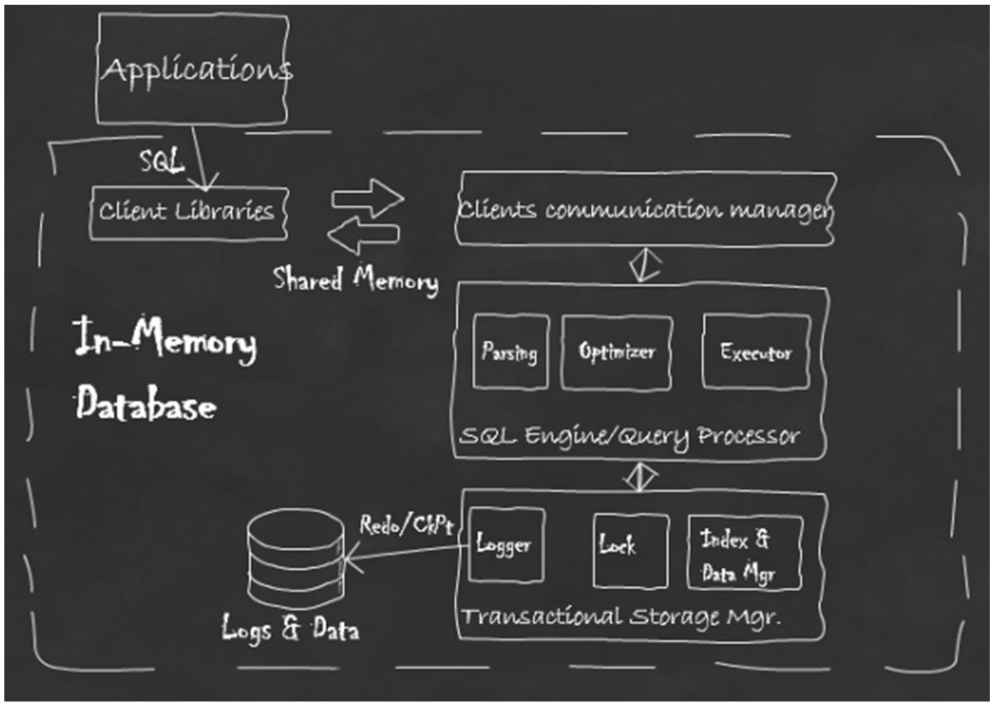


Figure 3: In-memory DB architecture
http://opensourceforu.efytimes.com/2012/01/importance-of-in-memory-databases/

4. Necessary Disk Access:

Although in-memory databases use the main memory as the data storage we cannot eliminate the use of disk entirely. The memory is usually accessible by the processors while disk is not. This makes the data in memory more vulnerable than on disk. Since the main memory is cleared on shutdown the data is susceptible to hardware failures, shutdowns, and crashes. That means there will need to be regular backups of the database stored on disk to limit data loss and be able to restore the database in case of failures. That will also affect the commits for each transactions logging. Committing each transaction to disk can affect the performance of the in-memory database system but at the same time logging to disk is essential to keep data integrity. One way to get around this issue to have a portion of the log in stable memory and then another process us responsible for writing the log from stable memory to disk. Another way to get around this problem is through group commits. This is where several transactions are allowed to gather in memory and then they are all written to disk in a single operation.

II. MemSQL

As the rising of new generation of database system which is used for specific purpose like temporal database, graph database, object database and etc., MemSQL is an in-memory database developed by MemSQL Inc. which was founded in 2011 and is a graduate of Y Combinator startup program. MemSQL Inc. officially launched their database to public on June 18, 2012. With notable implementation of columnstore structure, code generation and distributed architecture, MemSQL claimed itself as the fastest memory database. However, it is really hard to conclude if this is correct or not because database performance depends on various factors such as amount of data, usage purpose, queries, etc. Therefore, our group would like to use this report as an overview about MemSQL technology and terminology which helps to understand more about one representative product of in-memory database and easier to consider whether we should use MemSQL or not.

1. Architecture:

To provide high scalability and replication in database system, MemSQL implements two tiered clustered architecture. Each instance of MemSQL is called a node and there are two types of node in MemSQL (as described in Figure 4):

- ***Aggregator***: this node acts as a query router in clustered architecture. It stores only metadata and is responsible for querying leaf node, aggregating results and returning to client. A MemSQL cluster can have one or many aggregators but it can only have one master aggregator. Master aggregator is used for cluster monitoring and failover.
- ***Leaf node***: this node acts as a data storage node. It stores data into slices which is called partitions, for each partitions, a part of database is stored.

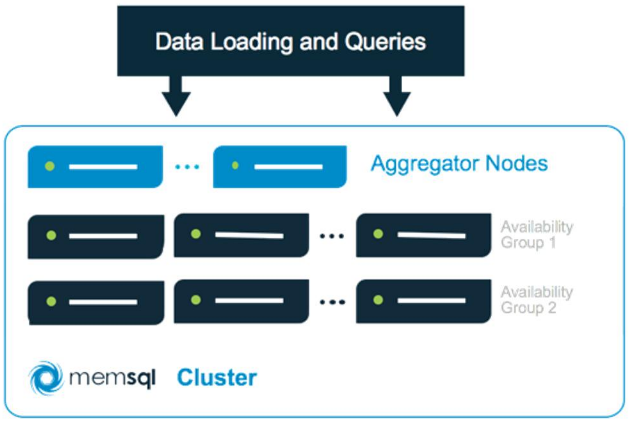
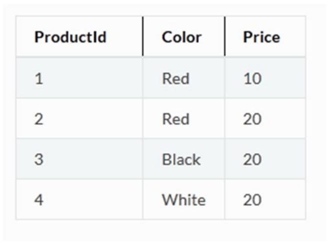


Figure 4: MemSQL architecture
http://docs.memsql.com/latest/intro/

Thanks to two tiered architecture, MemSQL allows user restore data and create new node easily. It also allows managing cluster without restarting database server.

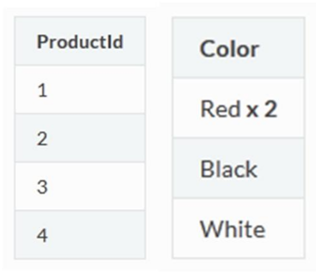
2. Data storage:

In addition to in-memory rowstore storage as traditional database system, MemSQL also supports disk backend columnstore storage. Column – oriented stores treat each column as a unit and store segment of each column in the same physical location. The MemSQL rowstore and columnstore differ both in storage format (row, column) and storage location (RAM, disk). Compared to rowstore storage, columnstore is more efficient for retrieving data individually from column and allows using compression for duplicate data in columns (We can see in Figure 6: x2 means Red color happens twice). Therefore, we often use columnstore for aggregation queries, applications where the data set does not fit in memory or where cost-efficiency does not make using the in-memory rowstore possible. Meanwhile, rowstore is typically used for highly concurrent OLTP and mixed OLTP/analytical workloads.



ProductId	Color	Price
1	Red	10
2	Red	20
3	Black	20
4	White	20

Figure 5: Rowstore
http://docs.memsql.com/latest/concepts/columnstore/#choosing-a-columnstore-or-rowstore



ProductId
1
2
3
4

Color
Red x 2
Black
White

Figure 6: Columnstore

http://docs.memsql.com/latest/concepts/columnstore/#choosing-a-columnstore-or-rowstore

The column store is enabled by adding a CLUSTERED COLUMNSTORE index to a table. When we define this index, the table itself is physically stored using columnstore.

3. Code generation:

MemSQL is an in-memory database and it could be not denied that the obvious benefit of in-memory database is fast processing speed of RAM and throwing away I/O cost. However, MemSQL also uses C++ for their infrastructures which partly help to boost the system performance. Moreover, instead of caching query results as traditional databases, MemSQL uses parameterized query as a key figure in its code generation system.

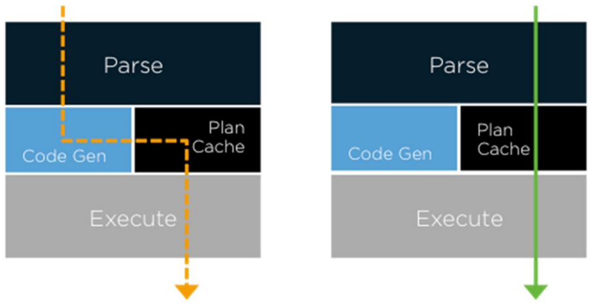


Figure 7: Code generation flow
http://www.memsql.com/content/architecture/

As described in above figure, the first time a query is executed; all constants are pulled out of the query and replaced with parameters. Then MemSQL turns the resulting parameterized query into a C++ program and compiles it into a shared object (.so file). Subsequent executions of the same parameterized query will skip the code generation phase and reuse this shared object to execute later. Therefore, the first time query is always slower than subsequent executions.

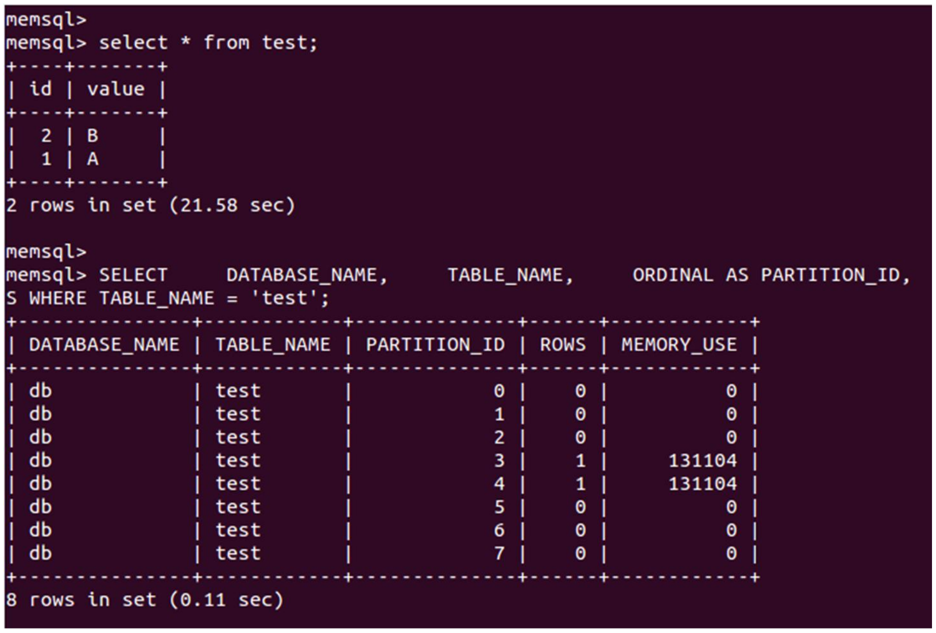
4. Distribution:

a. Tables:

There are two types of tables in MemSQL: reference table and shared table.

- ***Reference tables:*** are small tables that do not need to be distributed so they are replicated on every node in the cluster. User interacts with reference table through master aggregator then the action is reflected on every machine.

- ***Sharded table:*** every database in a distributed system is split into a number of partitions. Each shared table is split with a hash partitioning over the table's primary key (also called as shard key) and part of the table will be stored in each partitions of database. Partitions are stored on leaf node and considered as a database of that leaf. For example, partition 3 of database Sample is stored as database Sample_3 on leaf node. For every shared table created inside Sample, a portion of its data will reside in Sample_3 on that leaf. In the case that queries match the shard key exactly, an aggregator will route it directly to a single leaf. If the queries do not match, the aggregator will send the query across all leaves and aggregate results from those leaves. Those executions will be explained in detail in the next part. The below figure shows partitions of table Test. We can see that there are 2 rows in table Test which are stored on separate partitions.



```
memsql>
memsql> select * from test;
+----+-------+
| id | value |
+----+-------+
|  2 | B     |
|  1 | A     |
+----+-------+
2 rows in set (21.58 sec)

memsql>
memsql> SELECT     DATABASE_NAME,     TABLE_NAME,     ORDINAL AS PARTITION_ID,
S WHERE TABLE_NAME = 'test';
+---------------+------------+--------------+------+------------+
| DATABASE_NAME | TABLE_NAME | PARTITION_ID | ROWS | MEMORY_USE |
+---------------+------------+--------------+------+------------+
| db            | test       |            0 |    0 |          0 |
| db            | test       |            1 |    0 |          0 |
| db            | test       |            2 |    0 |          0 |
| db            | test       |            3 |    1 |     131104 |
| db            | test       |            4 |    1 |     131104 |
| db            | test       |            5 |    0 |          0 |
| db            | test       |            6 |    0 |          0 |
| db            | test       |            7 |    0 |          0 |
+---------------+------------+--------------+------+------------+
8 rows in set (0.11 sec)
```

Figure 8: Sharded table

MemSQL also implements availability group in distribution system. "*An availability group is a set of leaves which store data redundantly to ensure high availability*" (*http://docs.memsql.com/4.1/concepts/distributed_architecture/*). Each availability group contains a copy of every partition. Therefore, each leaf in the availability group has a corresponding pair node in the other group. This means a leaf and its pair share the same partitions and in the event of failure, MemSQL will use partitions from the leaf's pair to recover data. Currently, MemSQL supports up to two availability groups but you can reset this via the redundancy_level variable.

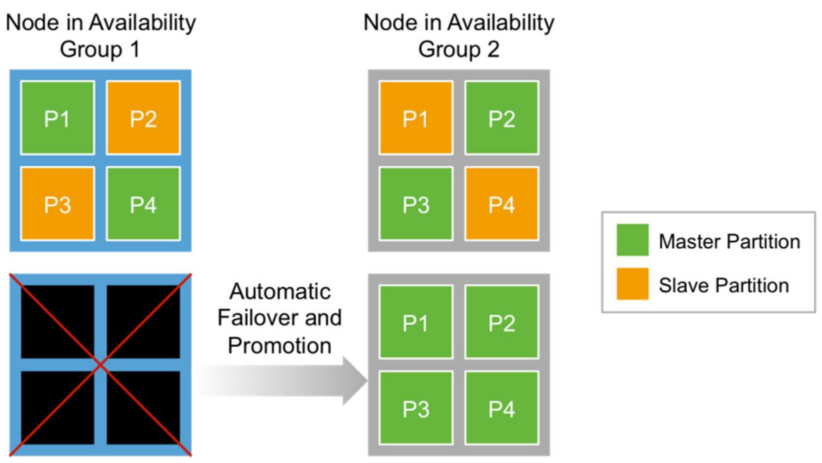

Node in Availability Group 1
Node in Availability Group 2
P1
P2
P3
P4
Automatic Failover and Promotion
Master Partition
Slave Partition

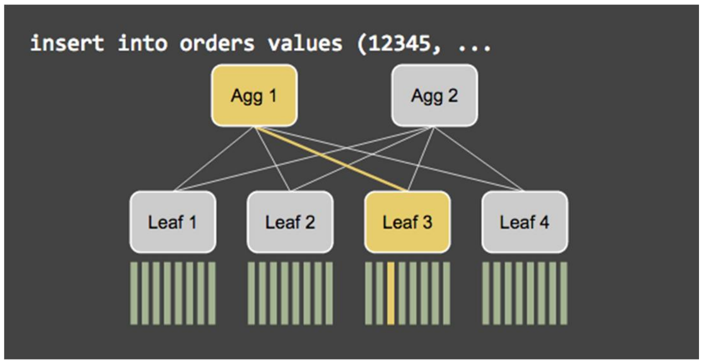
Figure 9: Availability group
http://www.memsql.com/content/durability/

b. **Query execution architecture:**

MemSQL allows user interaction with an aggregator as if it were a database, running queries and updating data like normal via SQL commands. But in fact, the aggregator queries the leaf, then gets back the result, aggregates it and sends back the result to the client. All communication between aggregators and leaves is also implemented as SQL statements.

As indicated in a previous part, data is shared across leaves into partitions. Each partition will be named as <databasename_N> with N – ordinal number of leaf. By default, MemSQL will create one partition per CPU core on the leaves for maximum parallelism. Below is the query execution mechanism implemented in MemSQL:

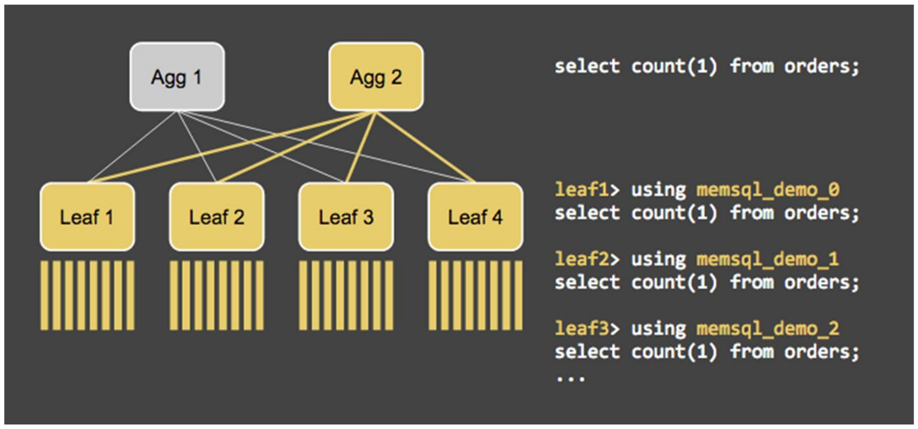
- When user runs an INSERT query, the aggregator computes the hash value of shard key in the distributed table, it does a modulo operation to get a partition number and sends the INSERT query to the appropriate partition on a leaf machine.



```
insert into orders values (12345, ...
```

Figure 10: INSERT query
http://docs.memsql.com/latest/concepts/distributed_sql/

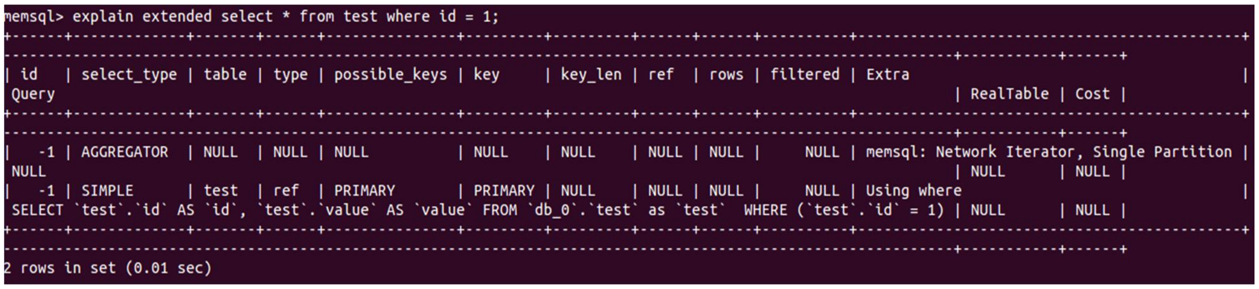
- When user runs a SELECT query, if the query matches the whole shard key, aggregator will directly send queries to the appropriate leaf. If the query does not match the aggregator then sends the query to all leafs(like pictured in figure 11).



```
select count(1) from orders;

leaf1> using memsql_demo_0
select count(1) from orders;

leaf2> using memsql_demo_1
select count(1) from orders;

leaf3> using memsql_demo_2
select count(1) from orders;
...
```

Figure 11: SELECT query
http://docs.memsql.com/latest/concepts/distributed_sql/

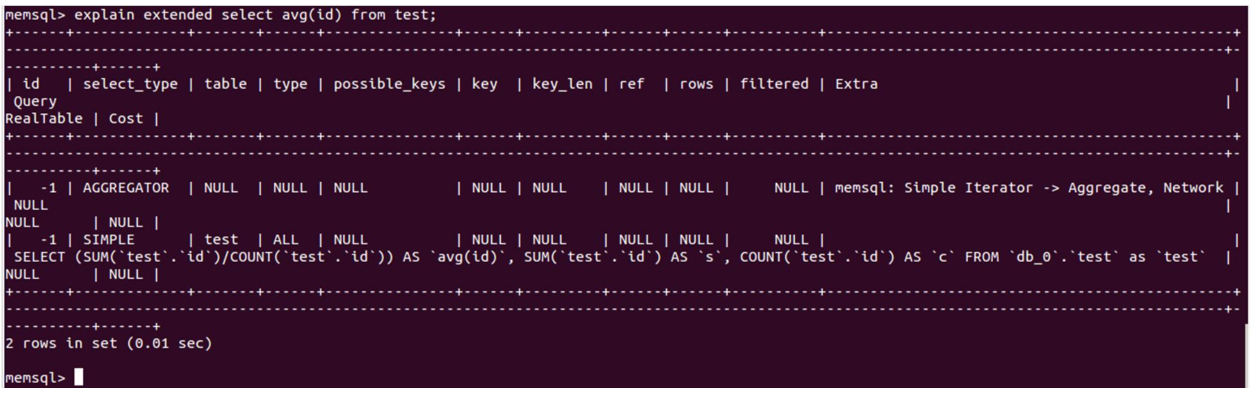
Below is the screenshot which shows the translated query which aggregator sends to leaf node. We can see that aggregator uses “Test_0” (which is “db_0.test”) as a database in a leaf node.



```
memsql> explain extended select * from test where id = 1;
+------+-------------+-------+------+---------------+---------+---------+------+------+----------+---------------------------------------------
---------------------------------------------------------------------------------------------------------+-----------+------+
| id   | select_type | table | type | possible_keys | key     | key_len | ref  | rows | filtered | Extra                                      |
 Query                                                                                                   | RealTable | Cost |
+------+-------------+-------+------+---------------+---------+---------+------+------+----------+---------------------------------------------
---------------------------------------------------------------------------------------------------------+-----------+------+
|   -1 | AGGREGATOR  | NULL  | NULL | NULL          | NULL    | NULL    | NULL | NULL |     NULL | memsql: Network Iterator, Single Partition |
 NULL                                                                                                    | NULL      | NULL |
|   -1 | SIMPLE      | test  | ref  | PRIMARY       | PRIMARY | NULL    | NULL | NULL |     NULL | Using where                                |
 SELECT `test`.`id` AS `id`, `test`.`value` AS `value` FROM `db_0`.`test` as `test`  WHERE (`test`.`id` = 1) | NULL      | NULL |
+------+-------------+-------+------+---------------+---------+---------+------+------+----------+---------------------------------------------
---------------------------------------------------------------------------------------------------------+-----------+------+
2 rows in set (0.01 sec)
```

Figure 12: Sample of translated query

When user runs an aggregate query, the query will be forwarded to one or many partitions, then the result will be sent back and merged on the aggregator node to compute the final result. Each calculation will be converted into an expression that is associative which can computed in each leaf. For example, *AVG(expr)* is converted to *SUM(expr) / COUNT(expr)* as shown in below screenshot



```
memsql> explain extended select avg(id) from test;
+------+-------------+-------+------+---------------+------+---------+------+------+----------+-------------------------------------------+
-------------------------------------------------------------------------------------------------------------------------------------------+-
---------+------+
| id   | select_type | table | type | possible_keys | key  | key_len | ref  | rows | filtered | Extra                                     |
 Query                                                                                                                                    |
RealTable | Cost |
+------+-------------+-------+------+---------------+------+---------+------+------+----------+-------------------------------------------+
-------------------------------------------------------------------------------------------------------------------------------------------+-
---------+------+
|   -1 | AGGREGATOR  | NULL  | NULL | NULL          | NULL | NULL    | NULL | NULL |     NULL | memsql: Simple Iterator -> Aggregate, Network |
 NULL                                                                                                                                     |
NULL      | NULL |
|   -1 | SIMPLE      | test  | ALL  | NULL          | NULL | NULL    | NULL | NULL |     NULL |                                               |
 SELECT (SUM(`test`.`id`)/COUNT(`test`.`id`)) AS `avg(id)`, SUM(`test`.`id`) AS `s`, COUNT(`test`.`id`) AS `c` FROM `db_0`.`test` as `test`  |
NULL      | NULL |
+------+-------------+-------+------+---------------+------+---------+------+------+----------+-------------------------------------------+
-------------------------------------------------------------------------------------------------------------------------------------------+-
---------+------+
2 rows in set (0.01 sec)

memsql> █
```

Figure 13: Sample of translated query

5. Lock-free data structures and multiversion concurrency control:

Unlike traditional databases which manage concurrency with locks, MemSQL achieves high throughput using lock-free data structures and multiversion concurrency control, which allows the database to avoid locking on reads and writes. Every time a transaction modifies a row, MemSQL creates a new version which stays on top of the existing one. This version is only visible to the transaction that made the modification. Read queries which access the same row can only see the old version of this row. MemSQL only takes a lock in case of a write – write conflict on the same row.

6. Index:

MemSQL supports two index types: skiplist index and clustered columnstore index.

The basic idea of skiplist is a sorted linked list. "*A skiplist is made up of elements attached to towers. Each tower in a skiplist is linked at each level of the tower to the next tower at the same height forming a group of linked lists, one for each level of the skiplist. When an element is inserted into the skiplist, its tower height is determined randomly via successive coin flips (a tower with height n occurs once in 2^n times). The element is linked into the linked lists at each level of the skiplist once its height has been determined. The towers support binary searching by starting at the highest tower and working towards the bottom, using the tower links to check when one should move forward in the list or down the tower to a lower level.*" (*http://blog.memsql.com/the-story-behind-memsqls-skiplist-indexes/*)

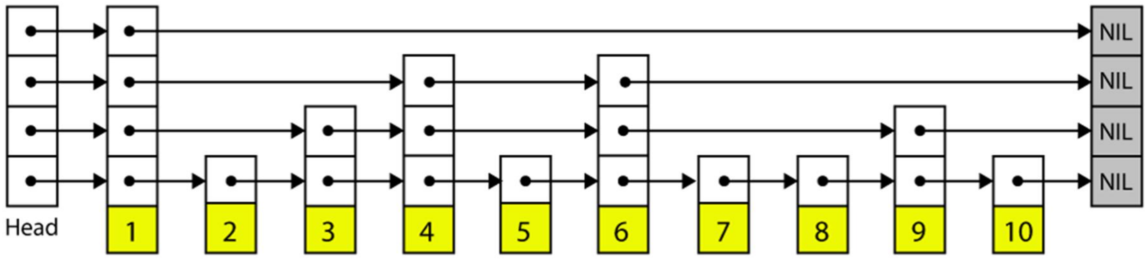


Figure 14: Skiplist index
http://blog.memsql.com/the-story-behind-memsqls-skiplist-indexes/

Skiplist is optimized to run in memory because it can be implemented lock free and offer faster insert performance. Meanwhile, column store indexes provide significant data compression, are backed by disk, and are very useful for analytical workload. Currently, column store indexes cannot be combined with in-memory row store indexes on the same table.

7. Durability:

MemSQL uses persistent logs and snapshots to ensure durability for database. Transactions written in memory are serialized into a transaction buffer. After that, a background process will pull groups of transaction and logs them on disk. If transaction buffer size is set to 0, database will run with disk-synchronous durability which means every transaction is committed to disk before the query is acknowledged. MemSQL also periodically takes full database snapshot and saves to disk. To restore a database, MemSQL will load the most recent snapshot and replays remaining transactions from the log.

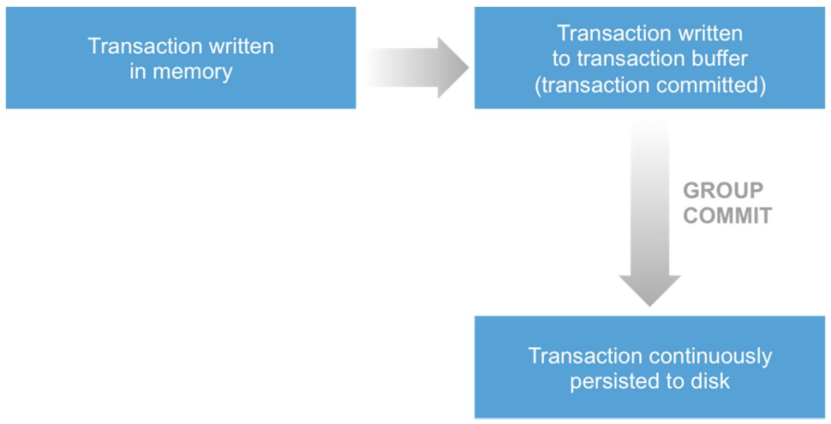

Transaction written in memory
Transaction written to transaction buffer (transaction committed)
GROUP COMMIT
Transaction continuously persisted to disk

Figure 15: Transaction persistent flow
http://www.memsql.com/content/durability/

8. Integration:

One of the main features which effects the popularity of a database system is the integration with other system. MemSQL can connect with Spark by MemSQL Spark Connector. It allows transferring data in parallel between MemSQL and Spark cluster. Moreover, a user can use MemSQL Loader as a utility for loading data from HDFS and Amazon S3. MemSQL can use SQL to query JSON which allow BI tools to report on relational and JSON data together.

III. Performance analysis between MemSQL and SQL Server

1. Environment:

For testing performance, we used a virtual machine with a Linux operating system because MemSQL requires it. We set up an Ubuntu VM using VMware and installed MemSQL on it. For both MemSQL and SQL Server we used machines with 4GB RAM. However, on virtual machines the virtual RAM actually uses physical disk on its host machine instead of using RAM. MemSQL performance will be slightly hindered because it is set up on a VM.

In this test environment, we use SQL Server 2014 and MemSQL 4.1. We did not use distribution architecture in MemSQL. Because one host can have only one leaf, we can only configure one master node and one leaf node.

2. Data:

For testing purposes, we decided to use the employee sample data from MySQL (Source https://dev.mysql.com/doc/employee/en/). We used data of departments, employees, and salaries tables with the schema as shown in Figure 16. We made two versions of a script. One is SQL server script. Another is MemSQL. We put data into 4 tables: department, employees (300 thousand rows), dept_emp (500 thousand rows), salaries (1 million rows).

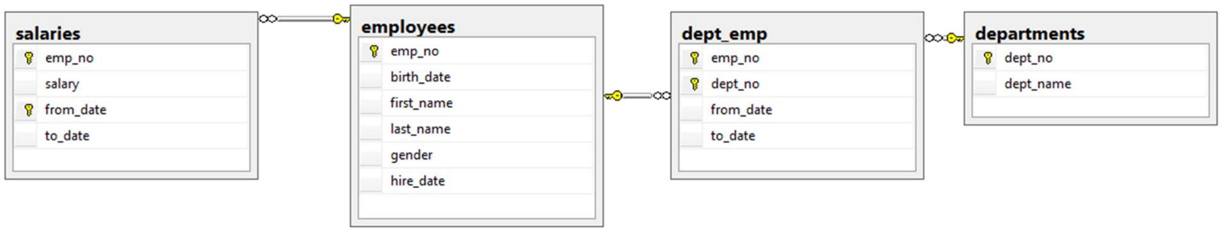


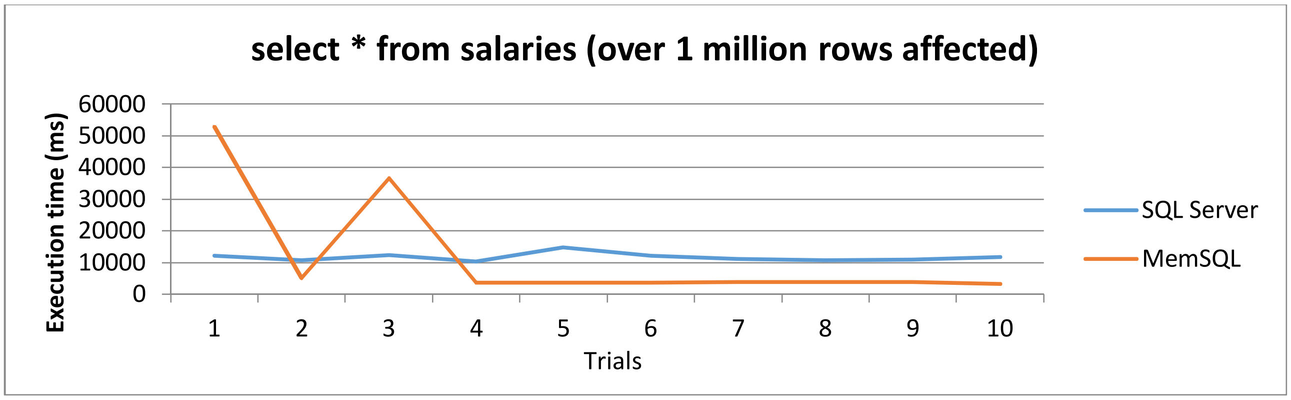
Figure 16: Database schema

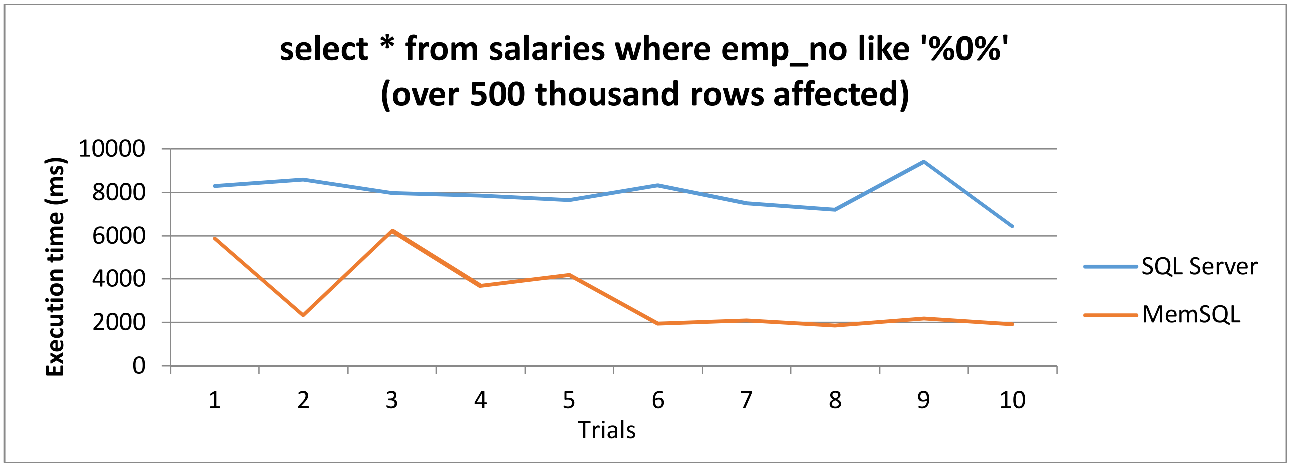
3. Queries:

a. SELECT

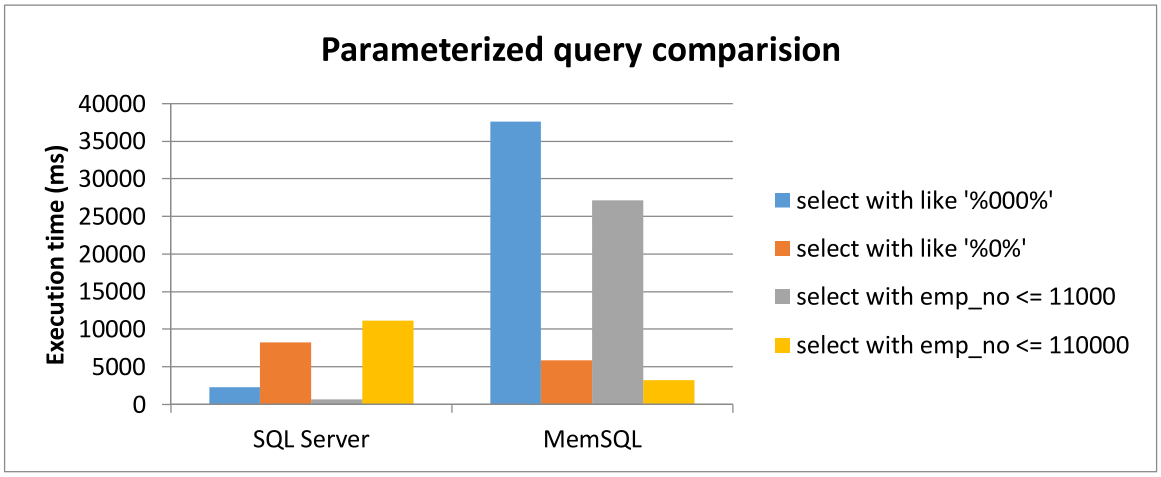
For simple SELECT statement, we run three types of queries:

- Search all with select * from salaries
- Search string with select * from salaries where emp_no like '%0%'
- Search number with select * from salaries where emp_no <= 110000





Comparing with SQL Server, MemSQL always has greater execution time at the first run. Then it will decrease after 4 trials and be more stable. You can see in the above graph, when the performance reach stability, the speed in MemSQL can be equivalent to SQL Server or even faster than SQL Server in second graph.



Additionally, we also put a test with parameterized query between MemSQL and SQL Server. For the first time we run with like '%000%', we have only 3 thousand rows affected then we change like expression by '%0%' to increase the result to 500 thousand rows. In this case, SQL Server will increase their processing time due to the larger number of rows returned. Meanwhile, MemSQL decreases their execution time for the second query even with larger number of rows. And we have the same result when testing with emp_no <= 11000. The reason here is that SQL Server uses memory to cache the result, while MemSQL use code generation mechanism to compile shared object caching query, when we change the query SQL Server needs to calculate another execution plan whereas MemSQL only needs to replace the constants in subsequent query with parameters in shared object.

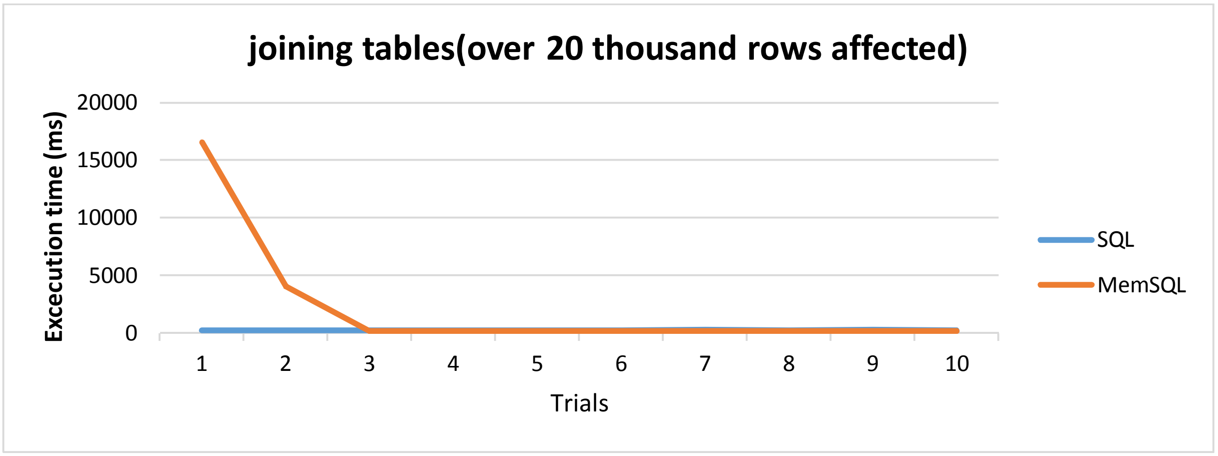
b. **SELECT with JOIN**

Here we will test the performance of queries that join one or multiple tables. We will test how long it takes for MemSQL and SQL server to execute the queries and compare the performance.

- Select the current employees who have a salary higher than 80 thousand dollars.

-
  ```
  Select e.first_name, e.last_name, s.salary, s.from_date, s.to_date
  from employees e, salaries s
  where e.emp_no = s.emp_no
  and s.salary > 80000
  and s.to_date > GETDATE()
  ```

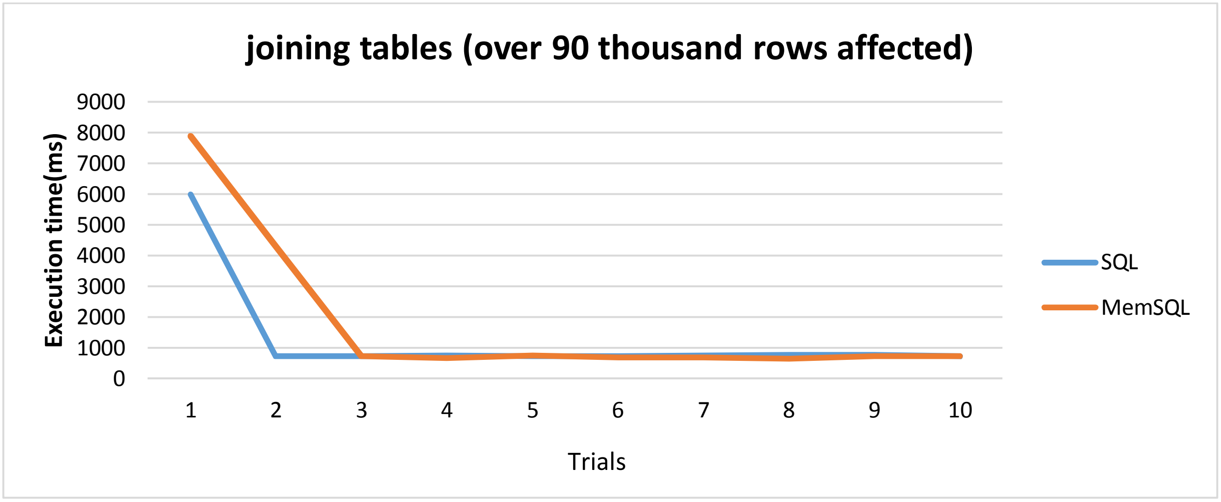
Rows affected: 26725



- Select current employees, their salary, and current department they work for.

- Select e.first_name, e.last_name, s.salary, d.dept_name
  From employees e, salaries s, dept_emp de, departments d
  Where e.emp_no = s.emp_no
  And s.to_date > getDate()
  And s.emp_no=de.emp_no
  And de.to_date > getdate()
  And de.dept_no = d.dept_no

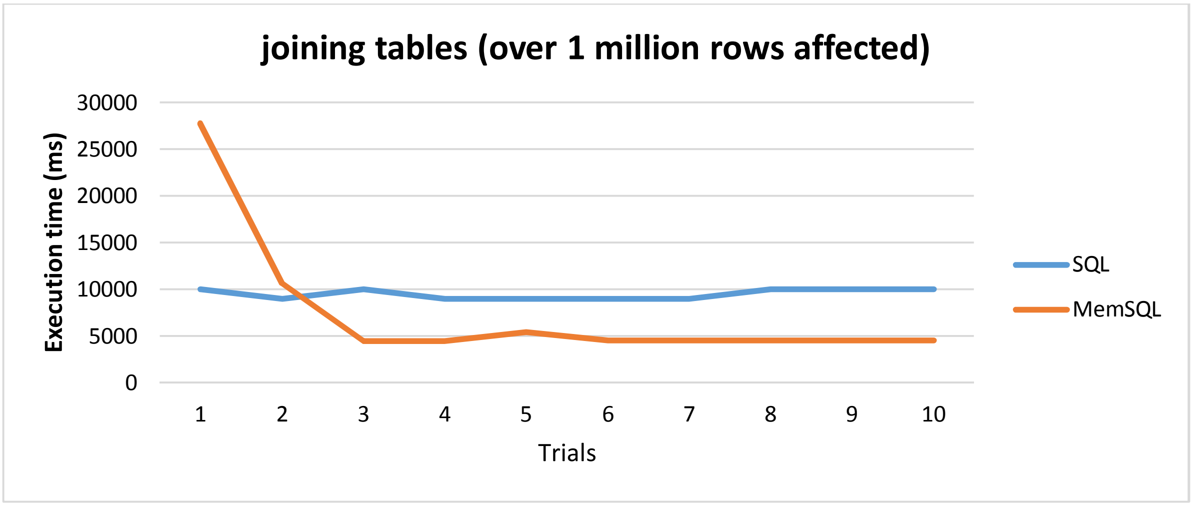
Rows affected: 92147



- Select everything from table employees and salaries joined on emp_no. Check performance on simple join with more than 1 million rows

- Select * from salaries s, employees e
  Where s.emp_no = e.emp_no

Rows affected: 1091938

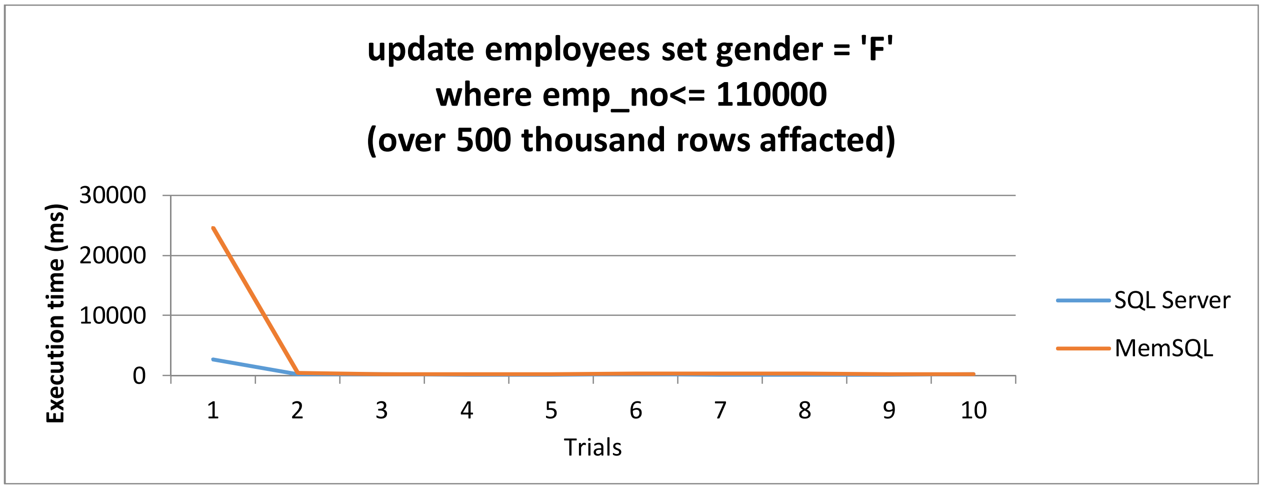


After MemSQL stabilized on all queries that incorporated a join of tables we noticed that it performed as well or better than SQL. We really start noticing a difference when the number of affected rows reaches 1 million or higher. MemSQL takes a few runs to stabilize but once it does it performs really well while SQL has a stable performance each time the query is executed. SQL stays at around 10 seconds to execute a query with over 1 million affected rows but MemSQL takes half that time after it is stabilized.

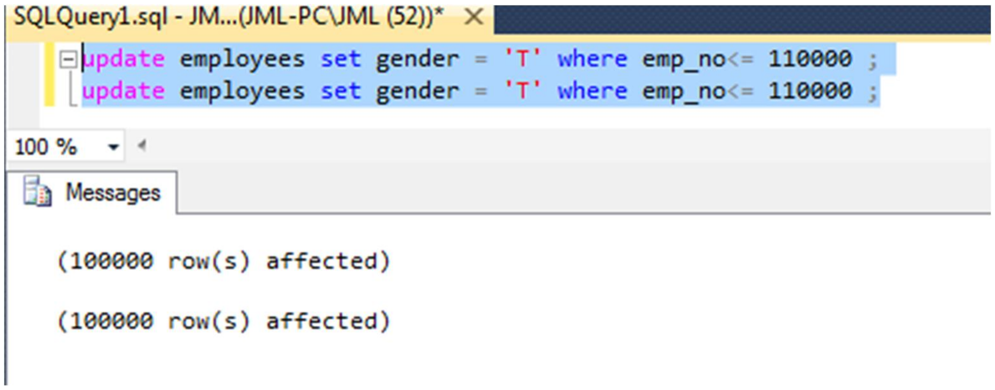
c. **UPDATE**

For testing Update statement, we need to switch between two statements below in MemSQL to guarantee the number of affected rows in MemSQL is always 100 thousand rows for comparing with SQL Server

- update employees set gender = 'F' where emp_no <= 110000;
- update employees set gender = 'T' where emp_no <= 110000;

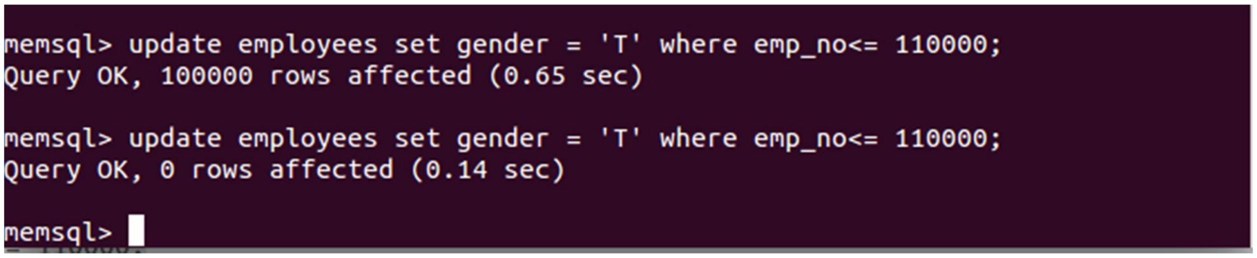


Even MemSQL takes longer time for updating than SQL Server at the first run, MemSQL is better because when updating with the same value, SQL Server will change values in all returned rows, while MemSQL will get only different values to update (this is the reason why we need to switch between two update statements in MemSQL). We can see in the picture below when running the same update statement for two times, SQL Server update for 100 thousand rows in the second run, while MemSQL updates no rows, which will get better performance with only 0.14 sec.



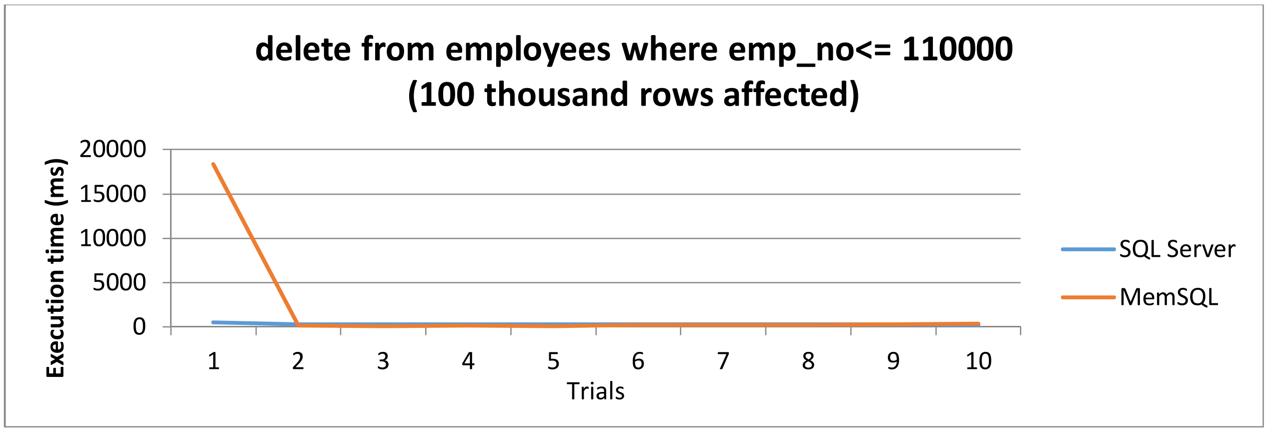
```
update employees set gender = 'T' where emp_no<= 110000 ;
update employees set gender = 'T' where emp_no<= 110000 ;
```

```
(100000 row(s) affected)

(100000 row(s) affected)
```



```
memsql> update employees set gender = 'T' where emp_no<= 110000;
Query OK, 100000 rows affected (0.65 sec)

memsql> update employees set gender = 'T' where emp_no<= 110000;
Query OK, 0 rows affected (0.14 sec)

memsql> 
```

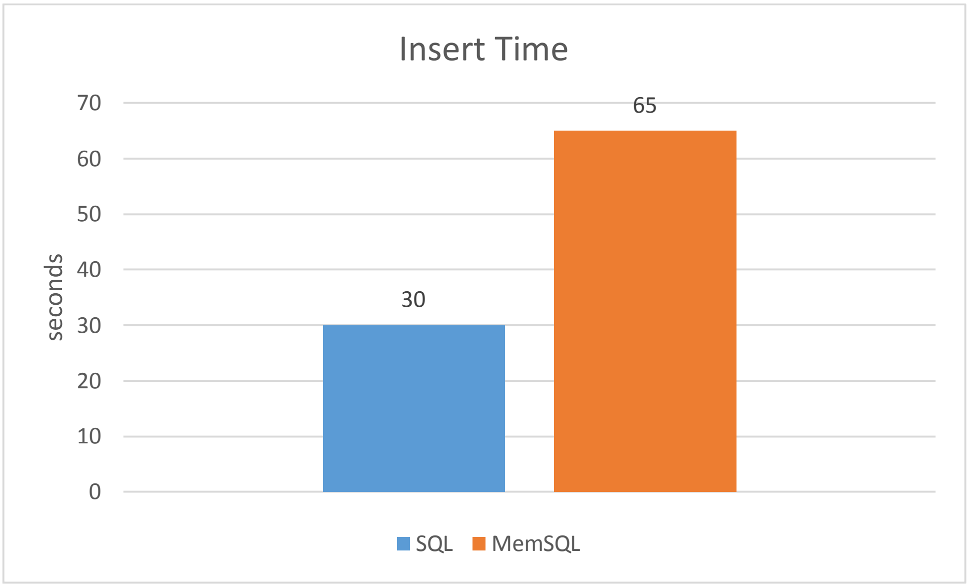
d. **DELETE**



With Delete statement, MemSQL gets the same result with Update statement. The first run always takes longer for code generation in MemSQL, then the subsequent runs get accepted time which is nearly equal to SQL Server.

e. **INSERT**

Here we tested the performance of inserting 107378 rows into the employees table. To test this we created a new database in MemSQL and SQL with the same tables and timed the insertion of 107378 rows.



Notice that SQL takes about half the amount of time MemSQL does to insert over 100 thousand rows. This is surprising because SQL needs to write these new rows all the way to disk while MemSQL writes them only to RAM. We believe that this is due to the performance of the VM.

We believe if this was being executed on a physical server running Linux the MemSQL query would be a few times faster than SQL.

IV. Conclusion

Overall, the MemSQL performance against SQL was descent. Before doing the performance test, we expected the performance of MemSQL would be better than SQL Server because SQL Server needs to write to disk while MemSQL keeps the data in memory and that should give it a big advantage over SQL Server. Our results only partly reflect MemSQL performance. There are a few reasons why we did not get the expected results. We believe our results could be due to the use of a VM. VMs tend to a much slower than physical servers. The same queries on VMs can be a few times slower than on a real server. Another reason behind our results could be that we were only able to use one leaf node instead of distributing the data. If the data were split up between a few leaf nodes then each one would query only a part of the data and then the results would be put together by the aggregate, increasing the performance. Even though it still lacks lots of features which are important for complex databases such as foreign key, triggers, stored procedure, etc. we think MemSQL is a good foundation for a new generation of databases. Especially, with DML statement like insert, update, delete and code generation mechanism applied for parameterized queries, MemSQL gained good performance comparing to SQL Server. It can be highly effective for large data sets with millions of rows. This is apparent in our results where we can start to see big differences in execution time for queries which involve more than 1 million affected rows. The difference in query execution time between SQL Server and MemSQL is likely to increase as the size of the data increases more than this. Lastly, query execution time improved and steadied as the number of trials increased. This will be convenient for applications which use the same queries often because these queries will be highly performant. Therefore, we think MemSQL is good for analytics applications and real-time applications, large datasets, simple databases, and queries which are executed often and require good response time.

Resources:

http://opensourceforu.efytimes.com/2012/01/importance-of-in-memory-databases/

http://pages.cs.wisc.edu/~jhuang/qual/main-memory-db-overview.pdf

http://go.sap.com/docs/download/2015/08/4481ad9e-3a7c-0010-82c7-eda71af511fa.pdf

https://www.quora.com/How-does-a-relational-DBMS-internally-store-its-data

http://www.mcobject.com/in_memory_database

http://www.vldb.org/conf/1986/P294.PDF

http://www-cs.ccny.cuny.edu/~jzhang/papers/nnsp_tr.pdf

http://docs.memsql.com

http://www.memsql.com/content/durability/

http://blog.memsql.com/the-story-behind-memsqls-skiplist-indexes/

http://www.memsql.com/content/architecture/

http://highscalability.com/blog/2012/8/14/memsql-architecture-the-fast-mvcc-inmem-lockfree-codegen-and.html

http://www.zdnet.com/article/a-look-at-memsqls-memory-first-database-software/