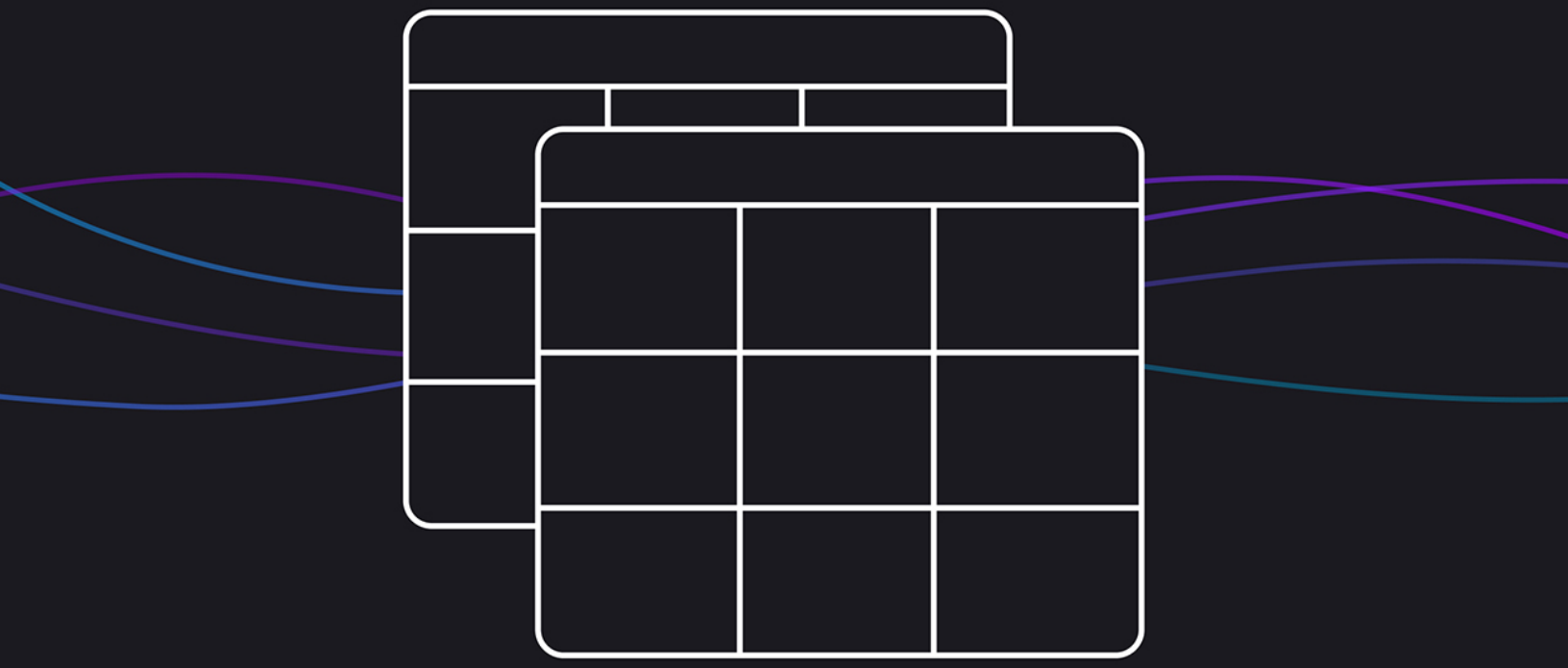
RESEARCH PAPER

A Column Store Engine for Real-Time Streaming Analytics



WRITTEN BY
Alex Skidanov, Anders J. Papito, Adam Prout

SingleStore

A Column Store Engine for Real-Time Streaming Analytics

Alex Skidanov, Anders J. Papito, Adam Prout
MemSQL
San Francisco, CA
{alex,anders,adam}@memsql.com

***Abstract*— This paper describes novel aspects of the column store implemented in the MemSQL database engine and describes the design choices made to support real-time streaming workloads. Column stores have traditionally been restricted to data warehouse scenarios where low latency queries are a secondary goal, and where restricting data ingestion to be offline, batched, append-only, or some combination thereof is acceptable. In contrast, the MemSQL column store implementation treats low latency queries and ongoing writes as first class citizens, with a focus on avoiding interference between read, ingest, update, and storage optimization workloads through the use of fragmented snapshot transactions and optimistic storage reordering. This implementation broadens the range of serviceable column store workloads to include those with more stringent demands on query and data latency, such as those backing operational systems used by adtech, financial services, fraud detection and other real-time or data streaming applications.**

Keywords—columnstore, columnar, realtime, memsql, tpch, analytics, concurrency, streaming

I. Introduction

Many modern analytical databases are based on column store engines. Column stores have many advantages for analytical queries, such as highly optimized compression and fast table scans. Relative to row stores, they tend to suffer on workloads that require selective filters, as column stores cannot leverage index seeks. Column stores are primarily optimized for queries that scan most or all of a table. Compression and column pruning vastly reduce disk IO, and processing can be optimized with vectorized execution, code generation, and by running operations directly on compressed data [5] [11].

With the addition of a sort order, often called a sorted projection, several new patterns of optimized queries are enabled. Operations on small key ranges, such as SQL BETWEEN queries, can be executed with low variance latencies. In many cases, after examination of memory-pinned file metadata, only a single disk IO will be required to service a query. Heavier queries can also be significantly improved - for example, two large tables could be joined with a merge join algorithm, or a distributed grouping operation could be performed in a streaming manner and with constant space overhead.

However, these improvements in latency tend to be most required by systems that have a transactional write aspect to them - for example, providing an analytical dashboard over a live stream of data. To support such workloads, static sorted projections are of no use. Allowing online write operations is a requirement. Despite this, popular column store databases either do not support sorted projections or fail to efficiently maintain them in the presence of continuous data ingestion, which restricts their use to data warehouse scenarios.

In this paper we describe an implementation of column store projections which treats low latency queries as a first class citizen alongside batch processed analytics. It enables transactional and analytical queries to leverage fast selective filters alongside concurrent, continuous data ingestion and modification, with asynchronous storage reordering allowing data to be accessible to reads immediately after ingestion.

The approach differs from those of other state-of-the-art column store engines in several key aspects. It maintains groups of rows in an order that allows for fast seeks and can be efficiently maintained during concurrent data ingestion. It leverages snapshot isolation level to reorder rows in a background thread with constant disk and memory overhead, and without interfering with concurrent read queries. It uses an optimistic reordering technique that does not interfere with concurrent updates and deletes. And finally, it enables a wide range of workloads which depend on data being visible to read queries immediately after ingest, and which have traditionally been unattainable for column store engines. We describe these techniques in detail, and provide an experimental performance comparison with two state-of-the-art column store databases.

A. MemSQL Architecture

MemSQL is a distributed SQL engine with a two tiered architecture consisting of scheduler nodes and execution nodes. It contains two primary backing data formats: an in-memory skip list-backed rows store and a disk/SSD-backed column store, each of which can be selected on a per-table basis and the latter of which is the main focus of this paper. MemSQL makes use of an advanced distributed query planner and optimizer. However the details of distributed query execution are orthogonal to the concerns of low-latency processing that we discuss in this paper, and in fact for the most part the distributed component of our query optimizer does not distinguish between row store and column

store tables. MemSQL also makes use of other advanced implementation techniques including runtime code generation, automatic parameter detection and distributed transactions. These features are largely unrelated to our treatment of a low latency column store engine, and we do not discuss them further.

B. *Paper Structure*

In section 2, we review the relevant storage concepts for column store systems. In section 3, we introduce our treatment of segment elimination. In section 4, we introduce our treatment of sorted runs, which we expand with projection maintenance concerns in section 5. In section 6 we discuss real-world practical considerations of our system. Section 7 contains experimental results, which section 8 fleshes out with a discussion of related systems.

II. BACKGROUND CONCEPTS

For an overview of the architecture and implementation of modern column stores, see [3]. Here we review the concepts most relevant to building a column store system that handles operational queries in the presence of a continual, online write workload. We focus mainly on the organization of rows into larger units, namely segments, sorted runs, and projections.

A. *Segments*

In a column store, each column of the source table is stored separately, to allow independent compression and query access. To allow flexibility in implementation, as well as amortization of costs such as disk seeks, tables are also partitioned horizontally into subsets of rows which we call segments. Each row belongs to exactly one segment. Each segment will typically contain between tens of thousands and millions of rows, depending on the details of the workload.

For each segment, several items are maintained. There is a file on disk per column of the table, storing the appropriate values. There is also a metadata representation of the segment, held in a durable in-memory data structure (in fact, a MemSQL row-store table). The metadata representation stores the file locations and a bit vector marking logically deleted rows. It also stores optimization-related information, including encoding schemes and the maximum and minimum projection key values represented in the segment.

With this representation, inserts and deletes are straightforward. Inserts always create a new segment (or multiple new segments, for large inserts), while deletes update the metadata representation to mark rows as logically deleted. Updates are implemented as deletes followed by inserts (within a single transaction).

B. *Projections*

Projections have a purpose analogous to that of a row store table's indexes. They allow seeking to a particular key value or key range in order to optimize query performance, though only to the granularity of a single segment. They can also be leveraged to provide ordered iteration of rows, although unless care is taken, this property degrades easily in the presence of an ongoing write workload.

As with row store indexes, it can be valuable to have multiple projections on the same table, allowing efficient access along several keys. In this paper we consider only maintaining a single projection. Maintaining multiple projections is an orthogonal problem, and is described in detail in [1].

Query optimization and execution techniques can leverage the projection sort order both internally within a single segment, and externally across multiple segments. On the left of Figure 1 is an example of two segments sorted internally but not externally - the ranges [1, 108] and [3, 72] overlap. On the right of Figure 1 is an example of two segments sorted both internally and externally.

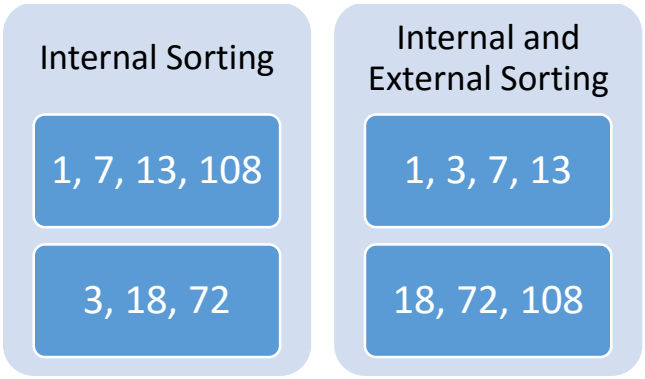


Fig. 1. Two approaches to sorting data in segments.

Internal sorting is supported by majority of column store databases. It is easy to maintain as the data is inserted or modified, but it also provides only a limited value to the optimizer and the query execution engine. In many cases the cost of going to disk and decompressing overshadows any savings from intra-segment seeking. External sorting, and, in particular, maintaining all the rows in a perfectly sorted order, is a harder problem that is the main focus of the remainder of this paper.

III. SEGMENT ELIMINATION

In some cases, in particular when we have external sorting, queries with selective filters can employ a technique that we call segment elimination. Segment elimination is a procedure that avoids opening a segment if that segment is guaranteed not to contain rows that match the filter. For example, if the segment has only values in the interval [1, 100], and the query predicate only keeps values greater than 150, we do not need to open or decompress this segment at all. To enable segment elimination, we store the minimum and maximum key value in the per-segment metadata. Note that it is not necessary to maintain these values in the presence of deletes (and thus updates) - segment elimination is a safe heuristic.

A complementary approach, which we term full enclosure, is also available. If a query predicate can be seen

to be true for all values in a segment, certain aggregates can be computed directly from metadata. For example, the following query

```
SELECT COUNT(col)
FROM   table
WHERE  col > 100 AND col < 300;
```

when run over four segments with intervals [0, 150], [150, 250], [250, 400] and [400, 500], can leverage both segment elimination and full enclosure. Only the first and the third segments need to be opened. The second segment is fully enclosed by the filter, so its minimum value can be taken from the metadata, and the forth segment can eliminated, since its interval is outside of the range of the filter.

Note that segment elimination can be similarly performed on raw column values, without explicitly maintaining a sort order. However projections maximize the usefulness of the technique.

IV. SORTED RUNS

Sorted runs are the foundation upon which we implement projections. A sorted run is an ordered set of segments such that the minimum value of every segment in the set is larger or equal to the maximum element of the preceding segment in the set (we say also that segments in the run are externally sorted). For instance, if the key of the projection is a single integer column, then the segments that are presented on Figure 2 on the left can be combined into sorted runs in several ways, as shown on Figure 2 on the right. The smallest number R of sorted runs into which a given set of segments can be partitioned is an important value, because in case of a seek for a specific value v at most $2R + k$ segments must be opened, where k is number of segments whose interval is [v, v].

To see this, we first consider all k segments with interval [v, v]. Clearly these must be opened, so we add k to our count and remove them from consideration. For each run, the value v must then be either in the interval of no segments, in the interval of a single segment, or on the boundary of at most two adjacent segments.

It is desirable to limit the number of sorted runs that the segments form. The smaller the number of sorted runs, the fewer segments that need to be opened to process a query that seeks for a single value, or a query with a selective filter in general.

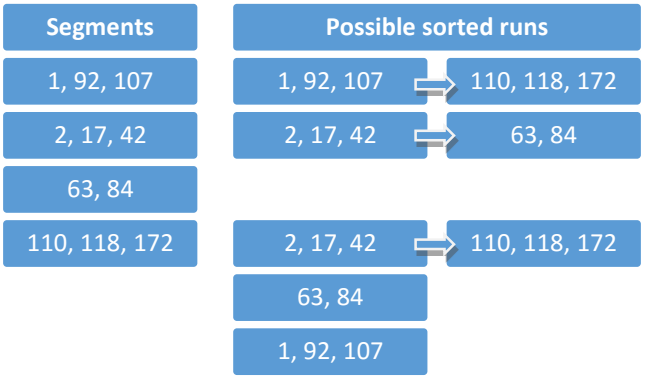


Fig. 2. Sorted runs.

Certain kinds of data are naturally distributed in such a way that the number of sorted runs will be very low. Most commonly, if the projection is sorted on a time-based column, and rows are created and inserted in roughly chronological order, segments will partition into very few runs even without reordering.

However, for most of the columns the distribution does not naturally align with the order in which the data is inserted, and there will be almost as many sorted runs as there are individual segments, making it impossible to effectively perform segment elimination.

In order, then, to partition a set of segments into the fewest sorted runs, we use the following algorithm, costing $n \log n$ time:

1. Initialize the set of sorted runs S to an empty set

2. For every segment in the order of increasing minimum value:

2.1. If there's a sorted run in S, that has the maximum value in it smaller than the minimum value of the current segment, add the current segment to that sorted run.

2.2. Otherwise, add a new sorted run to S, consisting only of the current segment.

3. Set S contains the smallest number of sorted runs that the input set of segments can be split into.

A. *Exponentially Decaying Sorted Runs*

Ideally we would want all rows in the table to form a single sorted run. With such an ordering, the overhead of any equality seek or range scan is at most two segments' worth, if we consider overhead to be work spent scanning rows that do not satisfy the predicate. However, in a presence of continuous data ingestion this is not a feasible approach because every insert or load could require resorting the entire table. Note that except in the case of naturally clustered data (as with a time-based projection key) this is not only a worst case, but rather a common case, because we expect fresh data to be fairly well distributed throughout our key range. Since one of our goals is for reads to see new data immediately after getting loaded, we discard this approach. Note that the

problem still exists even if the new batch is internally ordered.

An alternative approach, used in the core of the MemSQL column store engine, is to maintain one large sorted run that has at least half the rows in the table, another sorted run that has at least half of the remaining rows, and so forth. With this structuring, the number of sorted runs does not exceed $\log s$, where s is the total number of segments. We also have the property that ingesting a new batch of rows will only require amortized $k \log s$ time to update the order, where the new batch comprises k total segments. This idea is similar to that of LSM trees [10].

In practice, we have found it optimal to use a larger decay ratio. The MemSQL column store engine uses a constant of 8, so that the biggest sorted run has at least $\frac{7}{8}$ of all the segments, the second biggest sorted run has at least $\frac{7}{8}$ of the remaining segments, and so forth. There is a tradeoff between the constant factor in the data ingestion performance and constant factor in performance of reads that can leverage segment elimination. Higher multiplier makes data ingestion slower, but at the same time improves the performance of the reads.

V. MAINTENANCE OF SORTED RUNS

The addition of new segments due to inserts and updates disturbs the invariant of exponentially decaying sorted runs. To restore the invariant, it becomes necessary to reorder and repartition rows across segments. Because segments are sorted internally as well as externally, merge sort is a natural way to redistribute rows across segments. The algorithm to redistribute rows across multiple segments is the following:

1. Split the segments into the smallest number of sorted runs

2. While number of sorted runs is greater than 1:

2.1. Group sorted runs into pairs of similar size

2.2. For each pair, merge two sorted runs in that pair into one

The body of the loop at step 2 is a very expensive operation that rewrites all the segments in the segment group. It is being executed $\log_2 n$ times, where n is the number of input sorted runs. Because of this, in practice it makes sense to merge more than two segments at a time so that the base of the logarithm is higher and the number of iterations is smaller.

We also revisit the merge step of the merge sort algorithm, modified to work properly with sorted runs. We consider the merge of two sorted runs, but it can be easily extended to arbitrarily many sorted runs:

1. For each sorted run, initialize an iterator at the first row of the first segment in that sorted run.

2. Initialize a new empty segment.

3. While neither of the iterators is saturated:

3.1. Read current row from both iterators.

3.2. Write the smaller of the two rows into the new segment.

3.3. If the new segment is bigger than the segment size threshold, write it to disk, and initialize a new segment.

3.4. Advance the iterator that had the smaller row.

4. For each row in the sorted run that was not saturated, move it to the new segment, writing it to disk and resetting when necessary.

5. Delete all the input segment files.

In order to maintain the exponentially decaying sorted runs, we perform several steps on insert. Each time a new batch of data is inserted it is fully sorted internally before the transaction completes. Due to this, the newly introduced rows form at most one additional sorted run, even if they correspond to multiple segments. Then if that run is larger than $\frac{1}{8}$ of the smallest sorted run they are merged together. If the resulting sorted run is larger than $\frac{1}{8}$ of the next smallest sorted run, then they are merged together as well.

With care in the implementation, it is possible to closely estimate how many segments will result from merging sorted runs. Our implementation inspects the segment metadata for present and deleted row counts to determine the length of the resulting run. This information is used to collapse multiple iterations of merging into a single round in cases where a small trailing run would be produced and then immediately enter the merging process again - instead we merge with the additional run immediately.

A. *Reads During Sorting*

In order to allow reads while merging is underway, it is important that the reads have a consistent view of the data. In particular, new segments produced by the merge cannot be immediately made visible to concurrent reads. If a concurrent read sees the new segment, it may either encounter duplicate rows (from another segment that was partially consumed by the merge to produce the new segment), or may miss rows if the partially consumed segment is relocated in the metadata index and thus missed entirely. These issues are similar to those that could be observed due to a concurrent update in a READ COMMITTED isolation level transaction - and indeed, one way to look at merges is as an UPDATE that happens to not change any column values. However because these update-like operations are generated internally by the engine's background process, instead of by explicit queries in the user workload, it is important that they cause no anomalies.

To address this issue, queries against the column store engine are served with SNAPSHOT isolation level. With SNAPSHOT isolation, the background sorting process is entirely transparent to read queries.

However, wrapping merges in a big transaction has an additional downside. While the merge is underway, disk

usage of the affected rows is doubled. If we are merging the largest segments in the table, this can mean almost doubling the disk usage of the whole server.

The approach that we use in MemSQL column store engine is to commit immediately as a new segment is written, and mark all the rows in the input segments that were already moved to the new segments as deleted as part of the same transaction. If some segment was completely saturated as part of a merge step, it is deleted from the metadata entirely, and all its files are erased from disk, as part of the transaction that writes the new segment onto disk. Because reads are served with SNAPSHOT isolation, each query will see the correct set of rows.

This approach has an important properly that it will only use a constant extra disk space. If the merge sort algorithm merges *k* segments at a time, then at most *k* extra segments can exist on disk simultaneously, assuming that all the segments have the same number of rows. The particular scenario in which case this worse case occurs is when all the iterators of the merge sort are pointing to the last rows of their corresponding segments.

B. Writes During Sorting

The merge sort algorithm introduced above needs to properly handle concurrent write queries. There are two challenges to consider:

1. After the smallest set of sorted runs is found, but before the sorting finishes, segments may be removed or introduced by concurrent writes. This can result in the smallest set of sorted runs being changed.

2. As we process segments and write new segments, concurrent writes may want to change those same segments. The merger must either lock the segments it processes or employ some optimistic concurrency control technique.

1) Handling changes in sorted runs

To address the set of sorted runs being changed as we sort it, first we claim that the process of building the sorted runs requires amount of time negligibly small compared to the actual merging algorithm. This is because it is a metadata-only operation with a very low time complexity, while merging involves intensive disk IO. Thus, we can recompute the sorted runs periodically - every so many merges, or after writing every so many new segment. In the common case (over half of the time) when the merge involves only a few segments in several small runs, concurrent modifications can be handled by aborting and rolling back the entire merge operation and trying again - in this case the lost work is not significant. When merging a larger number of segments where losing work becomes expensive, we simply skip over any missing segments, and ignore any new segments. Note that removing any number of segments from a sorted run does not destroy the ordered property of the remaining segments. Therefore, regardless of how many segments are removed concurrently, the merge will produce precisely one sorted run as output, and take time proportional to the length of this output run. Therefore the background merging process will always make reasonable progress.

The type of writes that interferes with merges the least is deletes. If a segment has been partially or entirely deleted, the merge will simply skip that segment. It results in a smaller sorted run being produced than anticipated. However, the criteria to choose which sorted runs to merge was such that the resulting sorted runs is no larger than some fraction of the next biggest sorted run. Clearly, skipping input segments cannot violate this constraint.

Inserts can interact with sorted runs in very unintuitive ways. In particular, introducing new segments can change the way existing segments are distributed among sorted runs. To see why, consider an example of two segments [1-50] and [150-200]. Clearly, they form a single sorted run. However, if two more segments were introduced, with ranges [1-110] and [90-200], then the smallest set of sorted runs is ([1-50] [90-200]) and ([1-110] [150-200]), in other words the segments that were previously in the same sorted run are now belong to two different sorted runs. Thus, an insert can result in a segment moving from a small sorted run which was due for merging to a much larger run which is not due for merging, after recomputation of the ideal partitioning of segments into sorted runs. The most straightforward way to address this is to not replan which merges to perform until all currently planned merges have been completed. Therefore, even though the execution of the merges will itself use many small snapshot transactions, the planning and scheduling of these merges is based on a single up-front snapshot of metadata which is explicitly not updated as execution proceeds. We discuss this further in the section on practical considerations below.

Updates in general can be considered as deletes followed by inserts within a transaction. As long as both deletes and inserts are handled properly, updates will be as well. However, for some classes of updates, operation behavior and performance of the system can be improved significantly by playing nicely with the background merging process. In particular, if an update modifies all rows in a segment but does not change the segments key interval (for example, by modifying only columns which are not part of the projection key), then the merging process can substitute in the updated segment rather than skipping it. This causes the background process to converge to a stable state sooner, since the state of the engine more closely resembles the state for which the preplanned schedule was derived.

2) Handling concurrent access to a single segment

To handle writes attempting to change a segment which is concurrently being read by the background merging process, a way of resolving concurrent accesses is needed.

One way would be for merging to lock all the segments it is currently reading from. The maximum number of segments being merged at a time is bounded by the exponential decay and merge fan-in ratio (8 in the case of the MemSQL engine). However such an approach would increase the latency of concurrent writes waiting to acquire the same locks. In

particular, the worst case scenario involves merging a single segment whose key interval is large with a long run. For example, Figure 3 shows an example where a sorted run consisting of a segment with minimum value of 1 and maximum value of one thousand is being merged with a sorted run of segments, each containing a small range of values.

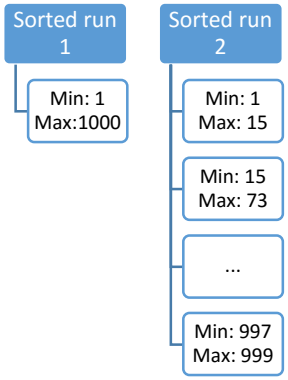


Fig. 3. Worst case for pessimistic locking.

In this worst case scenario the segment that spans a large range of values will stay locked for the entire merge operation (taking time proportional to the sum length of all sorted runs involved in the merge) because it has rows that need to be merged with segments at the both ends of the other sorted runs.

Unfortunately, such a scenario is very likely to happen in practice because the existing segments that currently form long sorted runs usually span a very small range of values while new segments that are inserted tend to span almost the entire range of possible values.

In order to avoid this pitfall, we use optimistic concurrency control. Merging does not take lock on segments as it processes them. Locks are taken only when a transaction is ready to be committed. Every time the merger is ready to write a new segment to disk it verifies that all the input segments that contributed to that segment still exist and that all the rows that were merged in have not been deleted by concurrent writes. If this is not the case, it rolls back all iterators to the positions they were before the last output segment was populated, and discard the new segment. If any input segment that the iterators are currently pointing at was completely deleted, the corresponding iterator simply advances to the next segment in its sorted run. If a subset of the rows in a segment were deleted then the new output segment can be regenerated, skipping the deleted rows on the second pass.

We consider the impact on the performance of the merger of using the optimistic approach. We assume that writes proceed at approximately the same speed as the merge. Clearly, new data being ingested does not interfere with the merger because the merger operates on a snapshot of the metadata. Updates which affect many consecutive rows have a low probability of interfering with the merger because the time it takes to update one segment's worth of rows is comparable to time it takes for merger to produce a new segment. Thus the probability of such an update to change the segment that is being merged has an upper bound of $\frac{1}{n}$, where n is the total number of segments. A more precise estimation for the case when the merger is slower than the writes is derived below.

However, deletes and updates of individual or well-distributed rows need to be handled separately. In the presence of queries that update few rows across many segments, the chance of modifying one of the segments that is being merged is very high and this can result in starvation of the merger if it retries each time. To address the problem we optimize this case to track exactly which rows were merged into the new output segment. As we commit we check if any of them were deleted. If any were concurrently deleted then in place of discarding the segment and starting over, we mark those rows as deleted in the output segment before writing it to disk. Newly created segments can therefore already contain tombstoned rows. We only trigger this optimization if the fraction of concurrently deleted rows falls below a certain threshold because introducing segments with a significant fraction of deleted rows negatively impacts both disk usage and query performance. The value of the threshold is a tradeoff. Higher values will result in higher disk usage and some penalty on select but will provide faster merges since fewer restarts will be necessary. Lower values will result in lower disk usage but will slow down merges.

To get a better sense of how the merger is affected by the write queries in practice where the performance of the merger is not necessarily equal to the performance of the writes, we derive the effect of write queries on a concurrent merge more precisely for several cases. We will consider separately two scenarios: when the updates are modifying rows that are located consecutively within the segment, and when the updates modify the rows randomly with respect to the projection key.

Let us examine how likely it is for such an update to modify a segment that is being processed by the merger.

We assume that the ratio between the throughput of the updates and the throughput of the merger in terms of the number of rows written per unit time is r, which for simplicity we will assume to be an integer, and the total number of segments is n, with all segments containing a similar number of rows. We will assume that there are on average s rows per segment. We will also assume that the merger processes m segments at a time, and that it will only start the work over if one of the input segments was completely removed or if the fraction of the rows that were merged into the new segment that were deleted concurrently was higher than d. Recall that the merger commits a transaction every time it has a full segment's worth of data ready to be written to disk. Thus it will conflict with an update only if the update managed to change one of the input segments in the time duration it took the merger to produce one output segment. That time is sufficient for the update to produce r segments, which, due to

the fact that the rows were read consecutively, implies that at most $r + 1$ segments were affected. Either $r - 1$ segments were completely deleted and two segments partially deleted, or r segments were completely deleted. The probability that one of these $r - 1$ (or r) segments was being merged is

$$1 - \frac{\frac{s-1}{s}\binom{n-r+1}{m} + \frac{1}{s}\binom{n-r}{m}}{\binom{n}{m}}$$

The two remaining segments, which were only partially affected, would only make the merger restart if it read from one or both of them, and between them processed more than ds rows that were deleted concurrently. We do not provide a tight bound for this probability, but it does not exceed $\frac{1}{n}$ for the cases when $d > \frac{1}{m}$.

For the case of updates that modify a random set of rows the derivation is simpler. It is the probability that out of rs rows that were changed more than ds were among the s rows we read. The probability of that is

$$1 - \sum_{i=1}^{ds} \frac{\binom{ns-s}{rs-ds+i}\binom{s}{ds-i}}{\binom{ns}{rs}}$$

Knowing the expected values of n and s, we can derive the values of m and d that give the best value of r and an acceptable likelihood of restarting the merge. For instance, we consider a system that stores 10,000 rows per segment, has around 1000 segments worth of data, expects writes to be 5 times as fast as the merger, merges 8 segments at a time, and does not roll back the merged segment if number of deleted rows does not exceed 12.5% of the total number of rows merged. In such a system the probability of a random update to interfere with the merge is negligibly small, and the probability of an update that updates consecutive rows to interfere with the merger is 4%. Increasing expected number of segments to 10000 or decreasing the ratio between writes and merger performance to 2, brings that probability down to 1%. Changing number of rows per segment does not have a significant effect.

C. Concurrent Merges

In certain scenarios, some of which we consider below, it is desirable for multiple merges to occur simultaneously on the same set of segments. For example, while two very large sorted runs are being merged, it might be desirable to merge several smaller sorted runs to get immediate performance benefits. If two merges operate on two disjoint sets of sorted runs, then they will not interfere with each other in any way, which allows the introduction of concurrent merges.

One way to ensure that concurrent merges never attempt to merge the same sorted run, or sorted runs that share a segment (the same segment at different moments of time can belong to different sorted runs) is for the merger that started earlier to mark all the segments it is planning to merge, and for any concurrent merge to disregard such marked segments during its planning stage.

VI. PRACTICAL CONSIDERATIONS

In practice it is not always feasible for merges to be as fast as the concurrent inserts. If they were as fast, it would mean that in their absence twice as must data could have been ingested, assuming that the disk is the bottleneck, and reducing the peak ingestion speed by a factor of two to maintain the projections is not always an acceptable tradeoff.

In this section we present several techniques that make projections usable in practice, when concurrent data ingestion and other write queries outperform merging. In particular, we aim for projection maintenance to be as asynchronous as possible so that for time-varying workloads we have minimal resource requirements at times of peak workload intensity.

A. Fast and Slow Mergers

A critical observations is that merging two large sorted runs and merging two small sorted runs result in the same expected improvement in performance of the read queries that can leverage sorted order. Merging either of the pairs would result in reducing the number of sorted runs by one, and the worst case complexity of the read queries with high selectivity is proportional to the number of sorted runs. With only a single background merging process, if it becomes occupied merging two very large sorted runs, and several smaller sorted runs were inserted concurrently, all read queries will suffer from poor performance, while a very cheap merge operation could have improved it. To address this, we have two background mergers - the Fast Merger and the Slow Merger, which are distinguished by which size runs they process. The Fast Merger processes only runs below a certain small cutoff (in MemSQL, this bound is set to the length of ten segments), while the Slow Merger exclusively processes runs at or above that threshold. Since the fast merger only merges small sorted runs it never gets stuck on an expensive operation. This way when a new batch of data is inserted, if a very small sorted run is created that can be quickly merged into another existing sorted run, the fast merger will be available to do that with very low latency. The slow merger, on the other hand, can work on merging large sorted runs for a long time without any regression in performance of concurrent queries. Note that while the engine endeavors to keep segments at a certain size (e.g. 100,000 rows each) the user workload may produce vastly smaller segments (a handful of rows) as a result of inserts. It is especially vital that these anomalously small segments are cleaned up as quickly as possible.

B. Manual Merging

Techniques such as optimistic merging and splitting dividing between fast and slow merges allow for maintenance of the projections in the presence of concurrent writes. However, in certain cases it is desirable to improve the read performance even more by reducing number of sorted runs at an accelerated rate, even at the price of temporarily reduced write performance. One common such case occurs after an initial bulk load when a project or workload is first kicked off. To address such use cases, we introduce two manual commands that a user can run. The first command allows the

user to run the algorithm that restores the invariant of exponentially decaying sorted runs using a pessimistic merger. The pessimistic merger is significantly faster than the optimistic merger if there are concurrent writes because it will never roll back and lose work, but in the worst case some writes might get stuck for a considerable amount of time waiting for a lock on a segment. This way the user can restore the invariant faster and increase read performance by temporarily sacrificing latency of write queries. It is important to note than overall time complexity is not sacrificed - if the user invokes the pessimistic merger each time a new batch of data is loaded, the pessimistic merger will still be doing amortized $k \log n$ operations, where k is the number of segments that were inserted. The only difference is that now it will be taking locks on the segments it processes. Thus, if one wants to increase read performance by sacrificing update performance, it is a viable approach to run the pessimistic merger after each batch is loaded.

Another command allows the user to entirely sort all the segments into one large sorted run. This operation will reduce the time taken by all the reads that follow by a factor of $\log n$, which for the case of several thousand segments means at least an improvement by a factor of 10. However, it will take time proportional to $n \log n$, so it is not viable to use after ingesting each new batch of data. It is very valuable in practice when workloads have a range of read and data freshness requirements. Often this command is used only for some tables, or in preparation for running intensive batch workloads which benefit from reducing the overhead of sorted iteration.

C. Sorting New Batches

When a new batch of data is inserted, unless the new rows are tightly clustered with respect to the sort key, then it is unlikely that any of the segments this batch introduces will naturally fit into any of the existing sorted runs. It is also unlikely that any two of the new segments will form a sorted run on their own. A new batch consisting of n segments is likely to introduce n new sorted runs, which will immediately be picked up by the fast merger. Until the fast merger processes them, the performance of read queries with high selectivity will be degraded significantly. For example, if the new batch introduced 10 new segments, and the number of sorted runs is 8 ($\log_8 n$ for n of around 100 millions), then the performance of the selects will degrade by a factor of two until the fast merger finishes with them.

Because of these two considerations, we sort the entire new batch of data before the transaction commits and the data is made visible.

D. Streaming

In our analysis above we assumed that write queries always insert enough rows to produce full segments (one hundred thousand rows per segment is the default in MemSQL). Since column stores were originally designed with bulk loading of data in mind, such an assumption was reasonable. However, our experience indicates that there are many practical use cases for column store where it is desirable to insert small chunks of data at a time, with batch sizes as low as a single row. In the context of a distributed database system, this challenge can be significantly amplified. If rows are hashed and distributed among many partitions on different machines, then each individual partition may receive input batch sizes two or three orders of magnitude smaller than the application initially provided, which are negligibly small compared to the full segment size.

The naïve approach is to create a new segment for each new batch of rows, even if there are not enough to provide a full segment. Such an approach invalidates assumptions we rely on above. In particular, our analysis of how quickly a merger can merge segments in the presence of concurrent writes assumes that the segments produced by writes and those produced by the merger have approximately the same size. However, if the write queries produce significantly smaller segments, the merger will fall behind, which will result in a lot of very small sorted runs. A high number of small sorted runs can in turn cripple read performance.

Another approach is to employ a buffer that accumulates rows as they are ingested, and only writes a full segment when enough rows are accumulated. For example, C-store, a predecessor of Vertica, describes a write-optimized storage designed for the same purpose [1]. The key challenge is that read queries must incorporate these buffered rows, so that they are visible immediately when the write query commits.

In our implementation we use our row store data structure, namely a lock-free skip list, in front of the column store. Each time a write query attempts to insert less than a full segment of rows, those rows are inserted into that skip list instead. As soon as the skip list has enough rows for a full segment, that segment is written to disk and the rows are deleted from the skip list in a single transaction. From the perspective of the read queries such a skip list is just an extra segment, indistinguishable from a segment that is actually stored on disk. The skip list naturally stores data sorted, so this extra segment is also always sorted.

It is important to note that some algorithms can leverage the internal representation of data in the column store. For example, since the data for each column is stored consecutively, it is possible to process multiple values in a single SIMD instruction. Moreover, some operations can be done on compressed data, without even restoring the original values. A write optimized data structure, such as skip list, will not allow for either of these two optimizations. In practice losing these optimizations is not important, because the amount of data in the skip list is insignificant relative to the amount of data on disk, and the performance degradation due to using less optimized algorithms to process data in the skip list is negligible compared to total time spent processing all the remaining data.

VII. EXPERIMENTAL RESULTS

In this section we present two use cases, compared against two state-of-the-art column store databases that are widely used in production. We will refer to the competing databases as A and B throughout this section. The first use case is TPC-

H benchmark. TPC-H provides a specification [8] for refresh functions that need to be executed as the workload is running. However the frequency of those functions can be chosen arbitrarily by the test sponsor. We are running the TPC-H benchmark against MemSQL and the two other databases with a very high refresh rate. The second benchmark is the adtech scenario that will be described in detail below.

A. TPC-H Benchmark

The TPC-H benchmark defines refresh functions that insert new orders and lineitems, as well as delete old orders. The functions in the definition insert and delete single elements. We run benchmark both executing such queries, and also executing batched loads and deletes, to see how MemSQL and the two other databases handle these two different cases.

Our first results are for vanilla TPC-H with a scale factor of 100, with refresh functions as declared in the specification running as frequently as each database allows. We verified that MemSQL has at least as high a throughput as both A and B.

We run all three databases on comparable clusters. MemSQL and A use 4 execution nodes, each with 8 cores, while B uses an 8 node compute cluster with each node having 4 cores.

To get a baseline, we first run all the TPC-H queries immediately after the initial load but before starting the concurrent write workload. For that we get the performance numbers that are shown on figure 4. There are 22 TPC-H queries on the X-axis of the chart, with time taken to execute the query on Y-axis, in seconds.

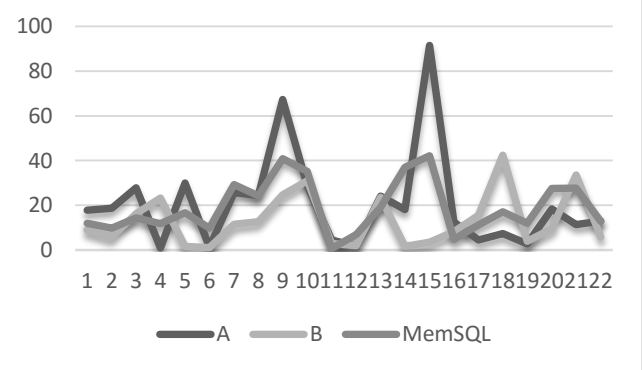


Fig. 4. TPC-H comparison without concurrent write queries.

As expected, different databases perform well on different queries, due to differences in query optimization and execution. The fact that the performance of B is on average better than that of A or MemSQL is largely due to the fact that its implementation has automatically performed a full sort after the initial load (at the expense of load time), while MemSQL has data in exponentially decaying sorted runs and A has data only sorted within segments.

We then start a write workload that conforms to the TPC-H specification, and consists of singleton inserts and deletes with an equality predicate. We evaluate the read query performance 5 minutes after the write workload started, without stopping the write workload. The performance degrades slightly for MemSQL, and noticeably for A and for B. The performance in the presence of the write workload is shown on Figure 5. For several queries, performance did not noticeably degrade for A or for B - this corresponds to those queries which cannot make effective use of a projection sort order, for example because they scan the full table with no filter. For the majority of the queries, however, both A and B performed noticeably worse with concurrent writes because the row ordering was no longer optimal and in some cases may have even forced a significant change in query plan, such as using hash grouping in place of a streaming grouping operation that makes use of sorted runs.

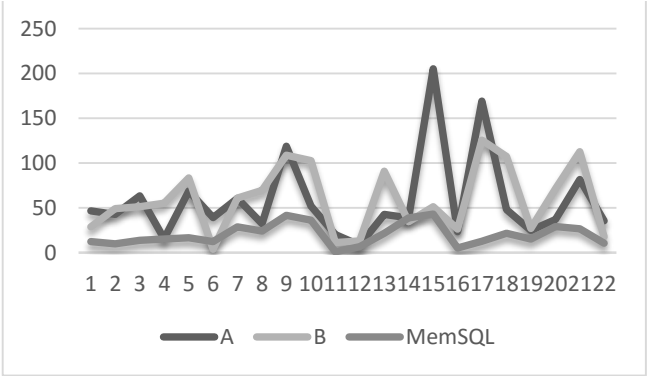


Fig. 5. TPC-H comparison with concurrent write queries running.

To emphasize the difference, consider the change in performance for each of the three databases separately. Note how for the database A the overall performance degraded slightly on most of the queries, and significantly on two of them. For the database B performance on several queries degraded significantly as well. MemSQL's performance stayed almost the same in the presence of the write queries.

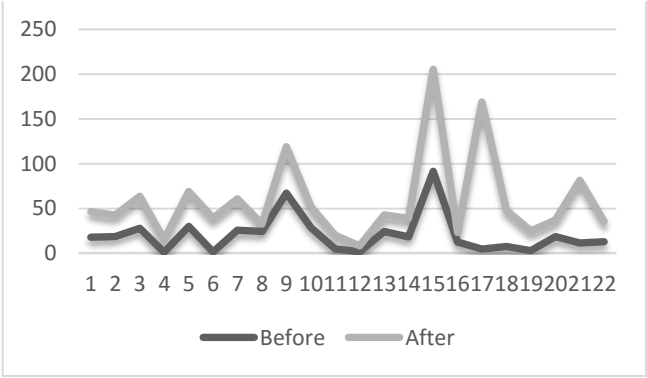


Fig. 6. Change in performance for database A

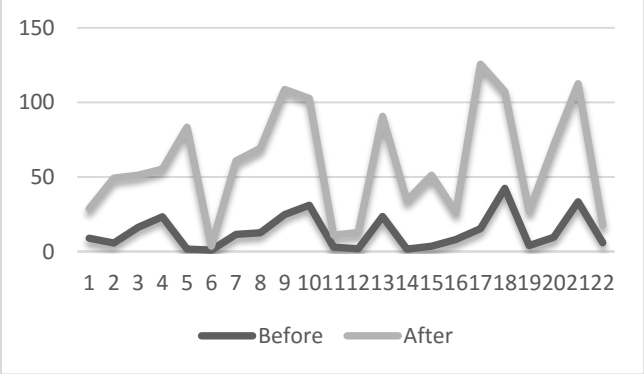


Fig. 7. Change in performance for database B

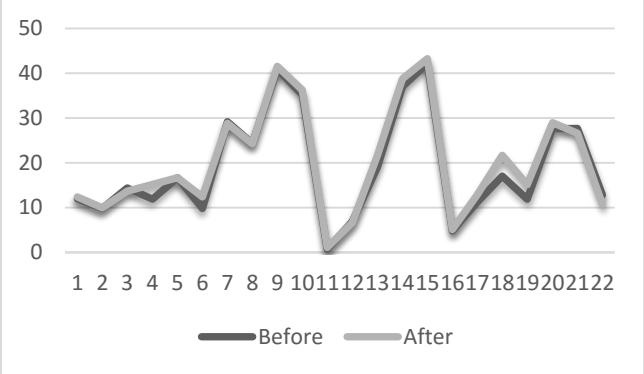


Fig. 8. Change in performance for MemSQL

B. Real World AdTech benchmark

The second use case we consider is an adtech workload centered on the following schema

```
CREATE TABLE users_categories
    (user_id BIGINT NOT NULL,
     category_id BIGINT NOT NULL,
     observation_time BIGINT NOT NULL,
     load_id BIGINT NOT NULL);
```

The table contains observations on which users have been observed to be in which cohorts. A category corresponds to an interesting criteria - for example, having read an ICDE paper. The observation time records when the user was observed to belong to the category - observations older than a cutoff date (e.g. 30 days in the past) are discarded for all analyses.

The first set of queries which the workload requires are overlap queries, which have variations of the following form

```
SELECT COUNT(*) FROM (SELECT
    user_id,
    (MAX(category_id = 1234) AND
     MIN(category_id != 2345)) AS
relevant
  FROM users_categories
  WHERE category_id = 1234
     OR category_id = 2345
  GROUP BY user_id
  HAVING relevant = TRUE)
    AS distinct_relevant_users;
```

This query produces a count of the distinct users who have been observed to belong to category 1234 and have never been observed to belong to category 2345. In general this family of queries finds counts of users satisfying an arbitrary set of membership constraints over a small constant number of categories. The family of queries powers both human-visible dashboards and automatic systems. In the former case it must have consistent response times within a second; in the latter, response times must be within a couple tens of milliseconds. Note that the filter can be served very well by segment elimination when we have a projection on the category_id column - the query will spend minimal time scanning rows that are not immediately relevant to the particular query instance.

The second set of queries which the workload requires are inserts (typically in the form of LOAD DATA for bulk loading). Data is loaded immediately and continually as it is received from external services. Typical ingest batches range from tens of rows (efficiently handling such excessively small batches is an important property) to tens or hundreds of millions of rows. Such rows must immediately become visible to read queries.

The third set of queries are deletes. Data beyond the cutoff date is deleted. Note that by this point, the rows will be physically spread through the storage layer - many or all segments will be modified by a deletion.

The fourth set of queries are updates, or more specifically batch reloads. A regular event is for previously ingested data to be found to be inaccurate or incomplete. In this case previously loaded rows must be removed and corrected versions submitted, transactionally and without interruption to the read workload. Such jobs take the form of

```
BEGIN TRANSACTION;
DELETE FROM users_categories
  WHERE load_id = 3456;
LOAD DATA INFILE …
  INTO TABLE users_categories (…)
  SET load_id = 4567;
COMMIT;
```

Because the projection key does not align with the load_id (which is used to tag ingest batches for this express purpose), the affected rows are typically well spread through the system. A reload is expected to affect most or all of the current set of segments.

The final query is a batch processing job, which is used to compute pairwise overlap between categories.

```
SELECT t1.category_id, t2.category_id,
  COUNT(DISTINCT t1.user_id)
FROM users_categories t1,
     users_categories t2
WHERE t1.user_id = t2.user_id
  AND t1.category_id < t2.category_id
GROUP BY t1.category_id, t2.category_id
```

For this query, we use a secondary projection (see [1] for a discussion of multiple projections) ordered along the user_id column in order to facilitate a streaming merge join. Note that this enables the critical optimization of converting the COUNT(DISTINCT t1.user_id) into a COUNT(*) after pushing the DISTINCT down into the table scan. The hash table required to service the GROUP BY already extends to tens or hundreds of millions of buckets - if each bucket were itself to take space proportional to the distinct user count, timely query execution would be infeasible. Ordered iteration of the projection key, even in the face of continual inserts, is key to enabling this query.

The workload for this section consists of a family of twenty queries described above, with 200 billion rows loaded into the users_categories table. Ingest and deletion rate are each 100,000 rows per second, with reloads accounting for an average of 5,000 rows per second. For MemSQL we use a cluster of 8 nodes in the public cloud, each equipped with 8 cores. For A we use this same hardware, while for B we use a cluster of 16 machines with 4 cores each.

Each query has selective predicates over the projection key, which allows an optimized implementation to scan a small fraction of the total row count. We continually ran the queries one at a time against each system over a twenty minute period and recorded response times.

For the MemSQL column store engine the average query execution time was 0.028 seconds, with first and third quartile response times at 0.015 and 0.034 seconds. The implementation is extremely resilient to a mixed and continuous write workload, and is able to provide responsiveness capable of driving automatic as well as human-facing systems.

For A the average query execution time was 2.11 seconds, with first and third quartile response times at 0.86 and 2.86 seconds. We believe that the penalty in performance is primarily due to two effects. First, segment elimination is performed with coarser granularity, causing system A to process many more rows than the query semantics and data set require. Second, write operations and background storage optimization in this system can cause concurrent reads to block while heavy storage optimization operations proceed. However, the system does maintain relatively good variance in query latency.

For B the average query execution time was 8.02 seconds, with first and third quartile response times at 4.27 and 14.23 seconds. As noted in the discussion of the TPC-H benchmark, B automatically performs a full sort after initial data loads. In the presence of an ongoing write workload it does not maintain this full sort order, which in many cases causes it to resort to scanning large fractions of the table.

VIII. RELATED SYSTEMS

There are many column store databases that are widely used in production environments today, as well as hybrid column- and row-store databases, such as DB2 BLU [5] and SAP HANA [6] [9]. Two particular column store databases that are known to leverage sorted order on the data are Amazon RedShift [7] and HP Vertica [4]

Amazon RedShift does not maintain sorted order in the presence of concurrent writes. It sorts the data after the initial load, but subsequent writes populate their rows in a separate unsorted region [2]. A manual VACUUM command is necessary to bring the data into a sorted order again. The high overhead imposed by this command (which must be run to allow performant reads) makes the system unsuitable for analytical workloads in the presence of continuous data ingestion.

Vertica maintains exponentially growing segments (ROS container in their terminology), that they maintain efficiently in the presence of concurrent writes [4]. However, large segments do not allow for efficient segment elimination unless an efficient mechanism exists to perform elimination on subsections of a single segment. To the best of our knowledge there is no literature on such an implementation involving Vertica ROS containers.

IX. CONCLUSIONS

In this paper we have presented the MemSQL column store engine, with an architecture designed to provide scalable, low latency queries over a vast and continually changing data set. We believe that existing column store systems leave much on the table by restricting themselves to bulky and awkward ingestion schemes. An integrated design that considers both storage maintenance and constantly evolving data will not just improve performance along the traditional strengths of column store systems. It will also enable their strengths to be leveraged against problem domains that require promptness and analytical capabilities over large data sets, such as adtech, financial services, fraud detection, and real-time analytics applications.

REFERENCES

[1] M. Stonebreaker et al, “C-Store: A Column-oriented DMBS”, *VLDB, pages 553-564*, 2005.

[2] Vacuuming Tables, Amazon Redshift Database Developer Guide, http://docs.aws.amazon.com/redshift/latest/dg/t_Reclaiming_storage_space202.html

[3] Daniel Abadi, Peter Boncz, Stavros Harizopoulos, Stratos Idreos and Samuel Madden, "The Design and Implementation of Modern Column-Oriented Database Systems", *Foundations and Trends® in Databases: Vol. 5: No. 3, pp 197-280*, 2013.

[4] Andrew Lamb, Matt Fuller, Ramakrishna Varadarajan, Nga Tran, Ben Vandiver, Lyric Doshi, Chuck Bear, "The Vertica Analytic Database: C-Store 7 Years Later", *Proceedings of the VLDB Endowment* 5.12 (2012): 1790-1801

[5] Raman, Vijayshankar, et al. "DB2 with BLU acceleration: So much more than just a column store." *Proceedings of the VLDB Endowment* 6.11 (2013): 1080-1091.

[6] Färber, Franz, et al. "SAP HANA database: data management for modern business applications." *ACM Sigmod Record* 40.4 (2012): 45-51.

[7] Mathew, Sajee, and J. Varia. "Overview of amazon web services." Amazon Whitepapers (2013).

[8] Council, Transaction Processing Performance. "TPC-H benchmark specification." Published at http://www.tcp.org/hspec.html (2008).

[9] Sikka, Vishal, et al. "Efficient transaction processing in SAP HANA database: the end of a column store myth." *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. ACM, 2012.

[10] O'Neil, Patrick, et al. "The log-structured merge-tree (LSM-tree)." *Acta Informatica* 33.4 (1996): 351-385.

[11] Boncz, Peter A., Marcin Zukowski, and Niels Nes. "MonetDB/X100: Hyper-Pipelining Query Execution." *CIDR*. Vol. 5. 2005.