MVTO for isolation

We implement two databases, one with MVTO and another without and compare the two implementations. Note that the database without MVTO has no isolation for it's transactions while the MVTO version does.

We compare the following parameters:

1. Memory consumed for the same amount of samples ingested.
2. Throughput of write transactions and average transaction time.
3. Throughput of read transactions and average transaction time.

We implement garbage collection also with the following rule:

1. Remove any version that is older than the least active version.
2. Keep the most recent version.
3. Further, while we can do GC periodically, we do it at append time, essentially continuously cleaning up versions.

Further, a full implementation has been done in Go here:
https://github.com/prometheus/tsdb/pull/306

This is a production ready implementation with tests, benchmarks and everything, done **during course period**. We re-implemented a smaller part of it in C++ because Go is not acceptable in assignments.

Architecture

This database is essentially a "mini" time-series database. A time-series is a sorted set of (timestamp, value) pairs, sorted by timestamp. For example,
[(10, 50.2), (12, 2), (15, 11)]

Now this time-series database can have millions of independent-series, each labelled by a seriesID, for example:

1 → [(10, 50.2), (12, 2), (15, 11)]
2 → [(101, 32), (121, 223), (156, 1123)]
3 → [(1, 2), (2, 3), (3, 233)]
.
.
.

Now we can have both read and read-write transactions. Now ReadQuery looks like

```
q := db.NewQuerier(mint, maxt)
data := q.Select(<seriesIDs>)
```

Essentially data will now have all the samples for the seriesIDs that fall in the range [mint, maxt].

Write Queries can read data and also write data. An example of a WriteQuery:

```
app := db.NewAppender()
data := app.Read(<seriesIDs>, mint, maxt)

for s in samples:
  app.Add(s.seriesID, s.t, s.v)

app.Commit()
```

Now we propose two algorithms to do this, and then compare the two.

One needs to understand the following, a transaction can touch hundreds of thousands of series, be it read or read-write. Now how do we ensure a read transaction succeeds fast in the face several write transactions?

We cannot have big lock on everything, as this will cause large delays. A lock might be held for several secs and in the face of heavy throughput the latencies will spike.

Instead, we could use a lock per series, and when writing to it, WLock the series, write and WUnlock. When reading, we RLock the series, read the values and RUnlock. Now this works great, but there is no isolation.

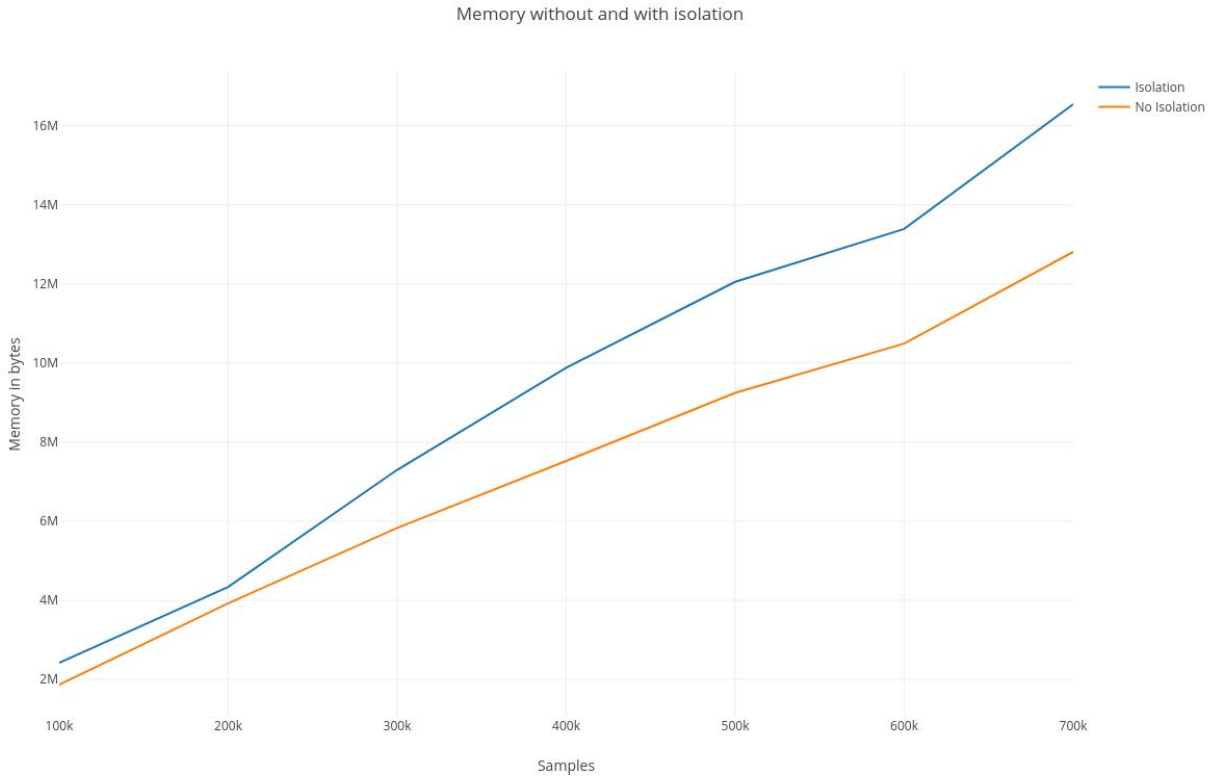
For example, we have write transaction, that is writing to series, (1, 3, 5, 7, 9) and so far it has written to (1, 3, 5), and we have read that is also reading the same series, and it starts now and reads (1, 3, 5, 7, 9), before the write transaction could write to (7, 9). This means the read sees a partial commit.

Now we could solve it by using a transaction manager but lets face it, it could be done much simpler with MVCC. We just need to version the items and then only read the right versions. We compare 3 things, the memory utilised, the amount of time it takes for transactions to finish.

We also do GC while doing a write: For each series that is being written to, we see if we can clean up any of the versions and if yes, we go ahead and clean up those versions.

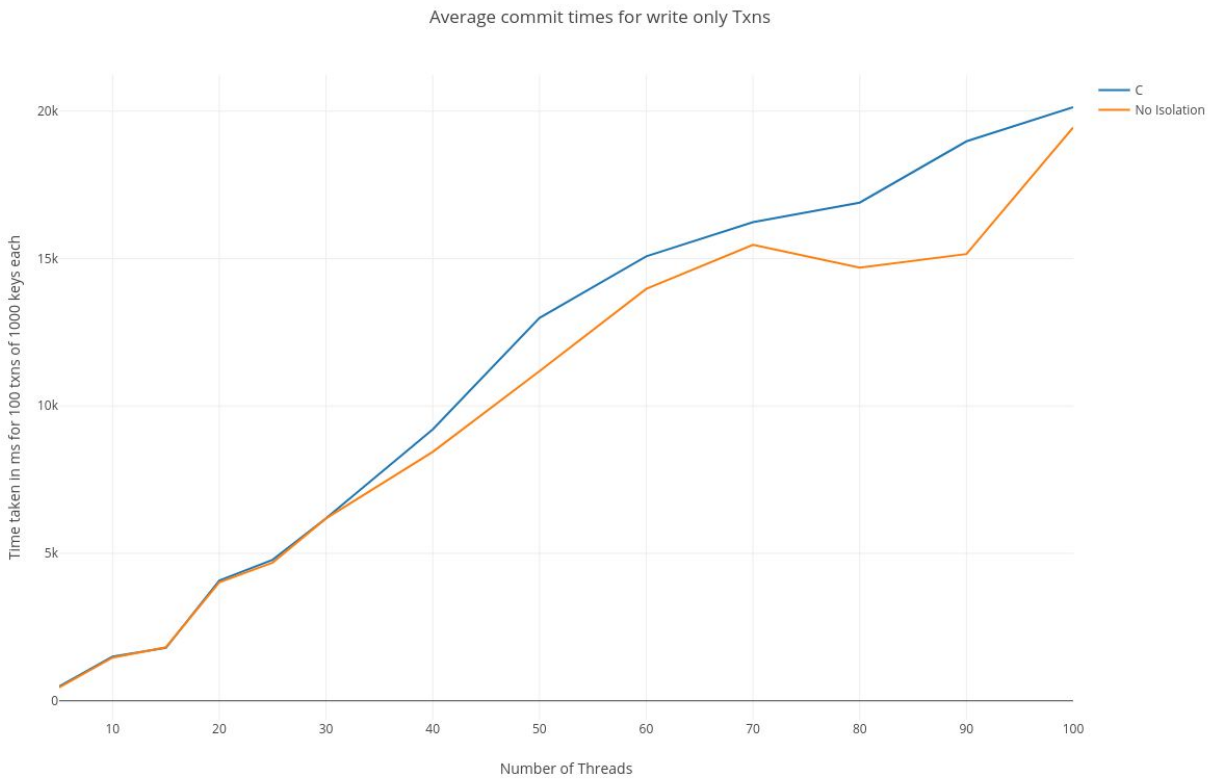
Memory Consumed

Overall, the MVCC version takes around 30% more memory compared to the non-MVCC version because we are storing one more number with each sample, but GC usually helps cutting it down. This graph is with GC switched off.



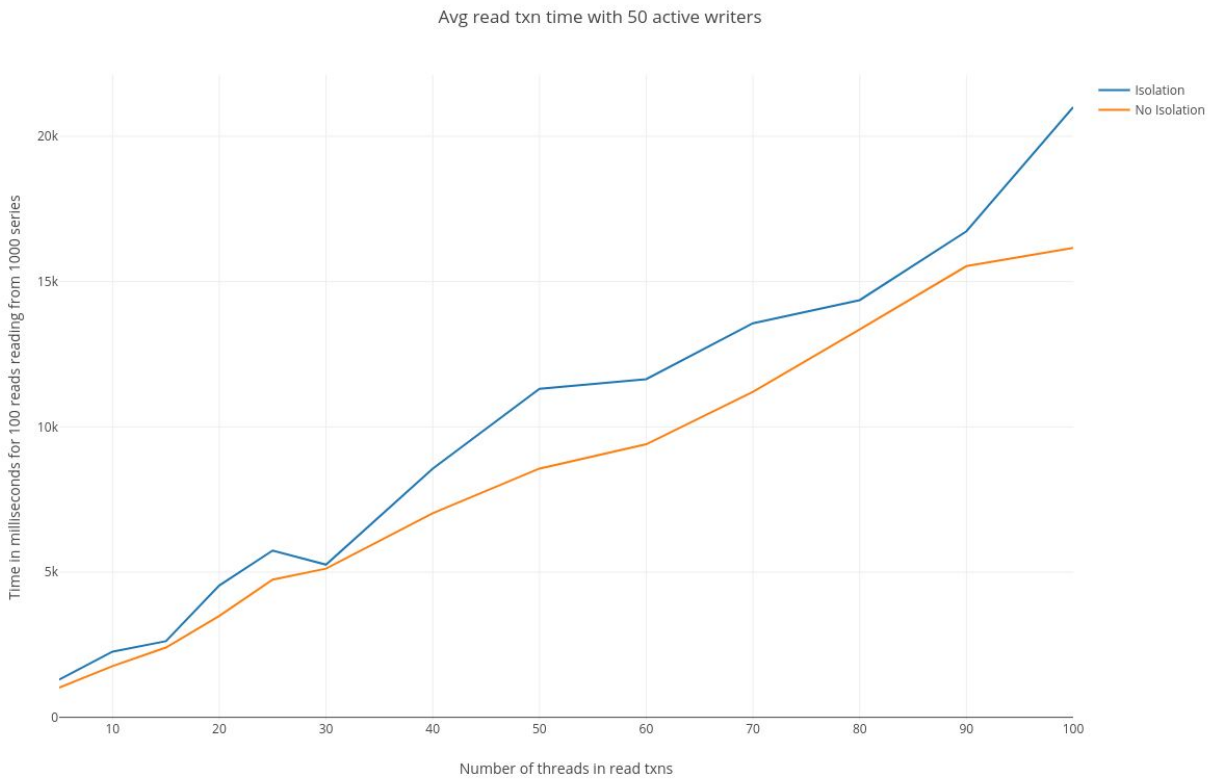
Throughput of Write Transactions

Well, we observed that there is not much of a change in the write transactions, as we are still getting the same amount of locks and there is the same amount of contention.



Throughput of Read Transactions

Same as above. We still need to get the read lock to read the version. It is a little slower though because we need to read several versions instead of one.



While overall, the results seem bad, it should be noted however the version we compared with did not have isolation and is essentially “not correct”. Further a lot of this could be just optimised away with better data-structures on the read-path of isolation.