Time Series Databases

Background

In many applications, there is a pattern of “write-once, read many times” data inserts corresponding to a range of time - things like logs, sensors, or any other consistent data being ingested from a stream follows this rule.

As a result, many databases specifically tailored for this use case, such as InfluxDB, TimescaleDB, and Apache Druid have been created. They allow both great throughput for reads and writes on time series data.

Time Series Operations

In order to optimize for time series data, let’s consider the access patterns used by applications that might be dealing with it:

- Writes
  - Primarily inserts to a recent time interval, adjacent values likely similar
  - Generally using a compound index like a timestamp in conjunction with some data source ID
- Reads
  - Generally from the same timestamp/data source combo over one column of data
  - All from relatively small time interval (probably the most recent one)
- Deletes
  - Common scenario is to delete time data older than some date prior to the present

Optimizing Writes

- Should likely build a compound index first sorted on source ID, and then within that sorted on timestamp
  - This allows writes from the same data source over similar intervals of time to be on the same node

Optimizing Reads

- Storing data in a column oriented manner is much faster
  - Generally speaking all timestamps should be sequential and similar and should be able to be compressed a ton
  - The same applies for certain metrics which are similar in value to one another as time progresses
  - Additionally, having all column values in one file reduces disk I/O and allows faster aggregations of column data, generally we only need one or two columns at a time for graphing

Optimizing Reads Continued

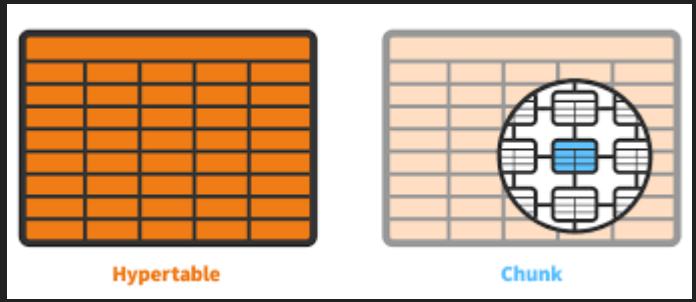
- Even within one node, it is better to treat each (source, time interval) tuple as its own chunk table (abstract these away via a large table)
  - Since most writes are going to only a couple of these we can achieve much better performance by caching their entire index in memory
  - Otherwise we would have many more relevant index pages that would occasionally have to be swapped in and out of memory which incurs significant processing overhead

Chunk1:
Sensor1
1/1/22

Chunk2:
Sensor2
1/1/22

Chunk3:
Sensor1
1/1/22

Chunk4:
Sensor2
1/2/22



Optimizing Deletes

- By breaking the main table into many smaller chunk tables, we also greatly optimize on the performance of deletes
  - If using an LSM tree based architecture, each delete of data is treated the same way as a write, and requires adding a tombstone to index files
  - If instead we want to just delete a bunch of old data all at once, we can just delete the corresponding chunk tables/index, as opposed to actually writing all of the deletes to index files and waiting for compaction
  - The same applies to B-trees, where we can just delete the appropriate index as opposed to going through the B-tree many times and setting a bunch of pointers in it to null

Conclusion

While storing time series data is a relatively niche topic, it turns out that using a database specifically made for it can allow for huge gains in performance.

Ultimately, they do so by:

- Creating separate indexes for each data source and time interval combination
- Column oriented storage both for fast aggregations as well as compression to reduce space significantly

While not every time series database works identically, it seems like across them these are the main features that allow for quick ingestion and processing of such types of data!