Speed up the Matrix SE

Currently, the Matrix SE suffers from two performance issues: read and write. Yes, both performance that an SE needs.

For read, the SE will pull all the payload that covers the requested range, even with a very high decimation factor which means many of the payloads is not needed at all. This is due to the payload metadata only store the start-end index of the dimension it is in.

For MongoDB SE this can somehow be solved. If we make a query that contains every index of each point, then only the payload covering these points will be read. The problem is if we are reading many points the query will be extreme and even not feasible to run. For Cassandra SE this is not possible since it does not support compare operation on some fields.

For write, to be specific: append. The SE will need to find the last payload of the dimension and either doing start=end+1 or append directly to the payload. This is a performance killer. The whole point we are using a NoSQL is that we can avoid reading will doing. If this is not solved the performance will never meet the requirement. Read to be more specific query is very heavy for any DB.

To solve the reading issue, the key is to know exactly which payload to fetch. This is achievable since the decimation is evenly done, so each sample we need is located on easily predictable position.

This is every simple to do. No need to scarify the flexibility of the original design. We can still have arbitrary sample in a different dimension. The only thing needs to be fixed is that every payload must contain the same number of samples.

The below picture show it.



The dim determins a row, if the every payload has the same number of samples then we know easily which payload to read. So we just need to give each payload a index.

**Index is a number that indecates the position of the payload in its dim.**

NOS=Number Of Samples in a payload

Index=Start/NOS

End=Start+NOS-1

DF=decimation factor

**Another thing to notice is that:**

**If DF is smaller than NOS, then do not bother index, every payload in the range is needed.**

**If DF is larger thean NOS then at most only one sample is read from a payload, and some are skiped.(1)**

**And moerover, the position of the sample in the payload has a rule, and is calculatable.**

**The above is ways to optimize reading.**

StartR=requested start sample index, you know EndR

Index=floor((StartR+n\*DF)/NOS), there are as many Indexes as the samples, that’s a problems, see (1).

Sample index in a payload=%(StartR+n\*DF)

The % and divide operation can be avoid, this left for you Liu Qiang.

Give you a tip: sample per payload=NOS/DF, skip factor=(DF-NOS)/NOS

Fore Write:

For above reading performance enhancement, the writing has some constrains to meet the design. The most important is that the payload has to be with the same number of samples. If the writer appends random number of samples to the DB, it is the DB’s responsibility to enforce the same NOS rule with requires reading existing payload and modify the existing payload. Which significantly reduces the write performance.

If the write itself enforces the same NOS rule it would be very simple, since if only need to buffer the sample in a buffer and upload them when the acuminates to a preset value.

The second thing is to make index for the new payload. The old way is to read the last payload and do end+1 remember? That is killing me. The very best way to do this is the write itself remembers the index of the last appended payload. It must remember every last index for each dim. This is very easy to implement. You create a Write object just like the Cursor. The write will query the last payload to get the Index, NOS, Start, End, and remembers it (NOS should be the same across the dims) when it first appends data to a dim (yes, using lazy pattern). Simple? Yes. One thing: this eliminate the possibility of concurrent write which is not possible for current design (append is not atomic). So the as soon as the write is created it must load the signal to prevent another writer to be created. And when finishes it will release the lock.