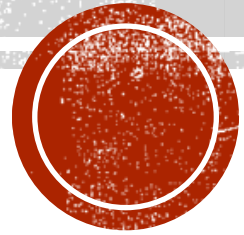
Gorilla: A Fast, Scalable, In-memory Time Series Database

Authors -
Tuomas Pelkonen, Scott Franklin, Paul Cavallaro, Qi Huang, Justin Meza,
Justin Teller, Kaushik Veeraraghavan, Facebook Inc, Menlo Park, CA



Presentation by -

Aniket Kulkarni

University of California, Santa Cruz

(amkulkar@ucsc.edu)

Agenda[1]

- Motivation
- Introduction
- Constraints and Challenges
- Comparison with existing databases
- Gorilla Architecture
- On Disk Structures
- Handling Failures
- New Tools on Gorilla
- Experience
- Takeaway
- Future Scope
- Conclusion

Motivation[1]

- ‘A’ from ‘CAP’ theorem. A stands for Availability. Being a large-scale internet service Facebook needs to ensure high availability in the event of unexpected failures.
- Not only high availability but the service must be responsive to their users.
- Facebook needed a system to monitor health and performance of the underlying systems and detect and diagnose problems.
- Facebook deals with a huge amount of data at scale i.e. data associated with thousands of machines geo-replicated across many geo-replicated datacenters.

At Scale: From Facebook Perspective[3]

- Facebook has 6 datacenters around the world.
- 2 datacenters are in United States.
- Facebook is continuously growing and thousands of servers are being added to Facebook's data centers on the daily basis.
- They have over 100 Petabytes of photos and videos that are growing every single day.

Introduction[1]

- Facebook used Operational Data Store (ODS) TSDB previously.
- To ensure high availability and to monitor health and performance of the underlying system at scale, Facebook engineers developed Gorilla, an in-memory Time Series Database (TSDB).
- Gorilla can store tens of millions of data points every second and respond to queries associated with a huge scale data within milliseconds.

Introduction[1]

- Why Time Series Databases ?

To offer the large-scale internet service, Facebook requires monitoring and analyzing millions of measurements per second across thousands of systems. Time series data is a sequence of data points collected at regular intervals over a period of time.

According to Facebook, from the monitoring perspective, aggregate analysis is important rather than individual data points. Recent data points are much more valuable than the older ones in order to detect and diagnose the root cause of problem.

Introduction[1]

- Traditional databases vs. In-memory databases

Traditional databases move data from disk to memory in a cache or buffer pool when it is accessed.

A constant need to move data can cause performance issues.

In-memory databases reside data in the memory and do not require moving of data from disk to memory.

Facebook relies on in-memory technique which allows them to provide quick query functionality. Query results are returned faster because of in-memory technique.

Constraints By Facebook[1][2]

- Writes Dominate
- State Transitions
- High Availability
- Fault Tolerance

Writes Dominate[1]

- Database should be always ready to perform write operation.
- Since Facebook deals with hundreds of systems handling multiple data points, it is a write-heavy system.
- Read operation rate is lower as compared to write operation rate. Most of the read operations arise from automated systems checking time series data and from various types of visualizations visited by users or human operators to diagnose the problem.

State Transitions[1]

- Aim is to detect the unexpected effects due to software release, network issue or configuration change.

- This ability to quickly detect effects of state transition is very useful since it helps to rectify the problem in the early stages without wide-spreading that problem in the network.

High Availability[1]

- In case of network partition or unexpected failures, the system is expected to function correctly.

- Even if the datacenters face the problem of disconnection, systems belonging to a particular datacenter must be able to write data to local time series database and must be able to access that if queried.

Fault Tolerance[1]

- Facebook requires their systems to be fault tolerant because in case of datacenter failure or natural disaster, Facebook should be able to survive this loss and required to be functional.

Insights / Assumptions[1]

- From the perspective of monitoring systems, aggregate analysis is more important rather than the individual data points.
- Monitoring systems do not store any user data. Therefore, ensuring traditional ACID properties is not the primary requirement for a time series database.
- Recent data points have higher importance as that of older data points.

Challenges[1]

- High Data Insertion Rate

Facebook found out that 85% of their queries to ODS were based on the data collected in the last 26 hours.

To achieve high data insertion rate, replacing disk-based database with in-memory database turned out to be best idea.

Facebook went an extra mile and found out that using in-memory database as a cache of persistent disk-based store they are able to achieve in-memory insertion speed with disk-based database.

Challenges[1]

- Total Data Quantity

As we have seen in the earlier slide, Facebook is continuously growing day by day. Storing a huge amount of time series data is practically too resource intensive.

To solve this problem, Facebook re-structured the existing XOR based floating point compression scheme to work with streaming data points.

Results obtained from this compression schemes were excellent!

Reduction in size: 12 times.

From 16 bytes per data point to 1.37 bytes per data point.

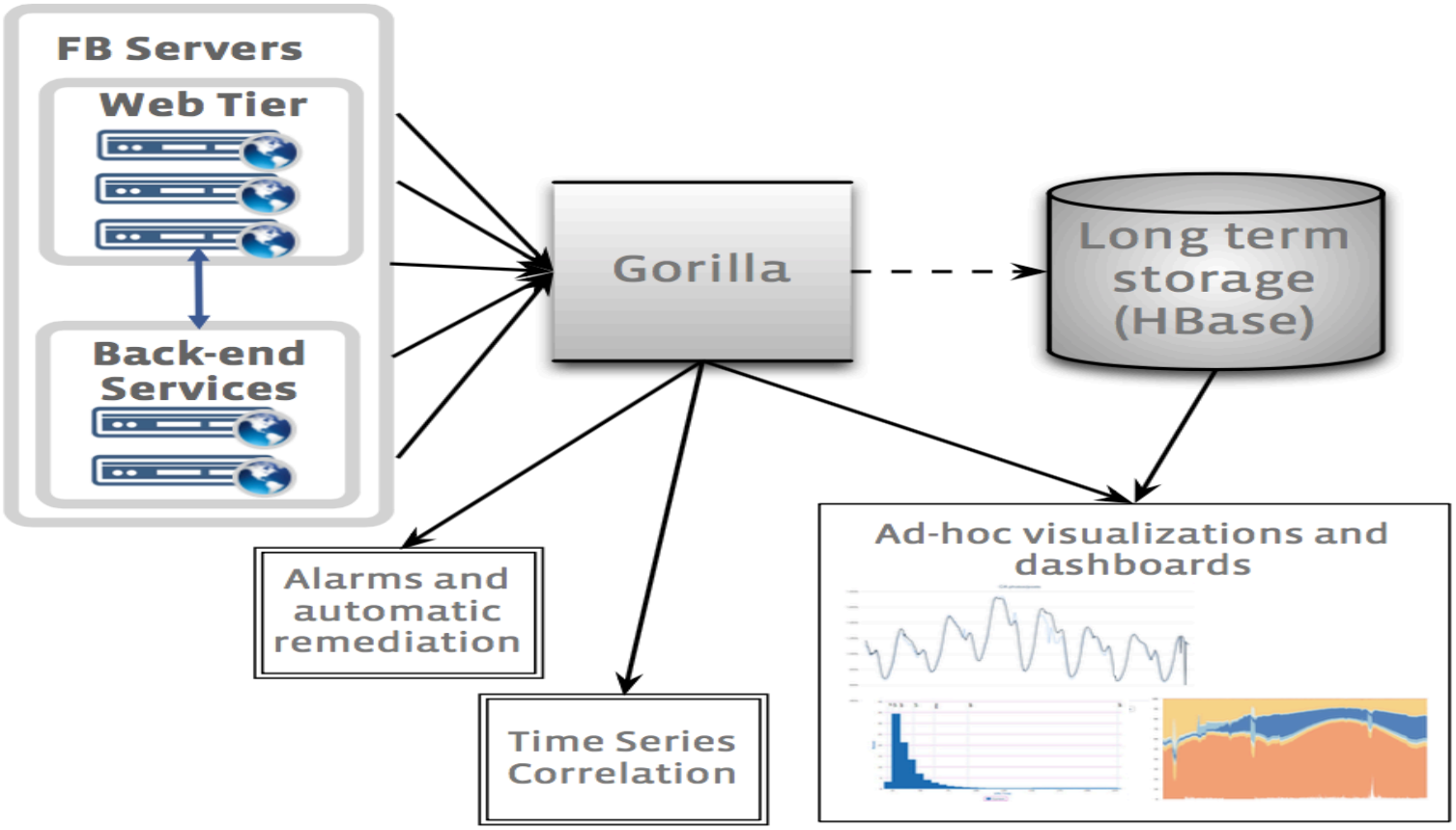
Challenges[1]

- Reliability Requirements

To address this issue, multiple instances of Gorilla are executed in different data centers regions and streaming data to each data center without having strict consistency.

Locality is the king. Read queries are directed to the nearest functional Gorilla instance.

Operational Data Store Overview[1]



Operational Data Store (ODS)[1]

- ODS time series data is used by:

1] Engineers who are dependent on a charting system which generates graphs and visualizations of time series data

2] Automated alerting system which reads statistics from ODS and compare the results with thresholds for health and performance metrics and alerts the engineers in case of anomaly.

Performance Issues With HBase[1]

- In early 2013, Facebook realized that HBase was no longer capable of scaling with respect to a large amount of read loads.
- Large number of medium sized queries started taking more time.
- Large queries executing over sparse datasets would time-out since writes always have high priority.

In-memory Caching: Not a good idea!

- Facebook explored in memory caching to solve the problem of Hbase.
- ODS was already using read-through cache but it was primarily focused on a charting system in which same time series data was shared amongst many dashboards.
- One issue with this read-through cache is that when the dashboards used to request for the most recent data point, misses in cache used to result in querying HBase for that data.
- Facebook also considered Memcache based write-through but decided to drop this idea because appending new data to previous time series data would result into read/write cycles traffic at Memcache server.

Gorilla Requirements[1]

- 2 billion unique time series identified by a string key.
- 700 million data points (time stamp and value) added per minute.
- Store data for 26 hours.
- More than 40,000 queries per second at peak.
- Reads succeed in under one millisecond.
- Support time series with 15 second granularity (4 points per minute per time series).
- Two in-memory, not co-located replicas (for disaster recovery capacity).
- Always serve reads even when a single server crashes.
- Ability to quickly scan over all in memory data.
- Support at least 2x growth per year.

Comparison With Existing Time Series Databases[1]

- OpenTSDB[3]

OpenTSDB is based on HBase. ODS HBase performs aggregation on older data to save the space. Because of this archived data has lower time granularity as compared to the most recent data. OpenTSDB keeps the full resolution of data. In OpenTSDB, time series is identified by <K,V> pairs i.e. tags. In Gorilla, a single key is enough to identify time series data.

Comparison With Existing Time Series Databases[1]

- Whisper (Graphite)[4]

Graphite stores time series data on local disk in whisper format i.e. a round robin database (RRD) style database. Whisper does not support irregularities in time series while Gorilla can handle irregularities in time series (e.g. arbitrary changing intervals)

Graphite is not fast enough as compared to Gorilla since it relies on disk-based local storage which has slower query performance as compared to in-memory database.

Comparison With Existing Time Series Databases[1]

- InfluxDB[5]

It provides richer model than OpenTSDB. InfluxDB offers code to perform horizontal scaling without the overhead of managing HBase or Hadoop cluster. Facebook can make use of InfluxDB because they already have dedicated teams for HBase. However, since it keeps data on disk, query execution is slower as compared to in-memory database.

Gorilla Architecture[1]

- Monitoring Data

It consists of:

- 3 tuple of a string key
- 64-bit time-stamp integer
- Double precision floating-point value

Gorilla Architecture[1]

- Time Series Compression Algorithm

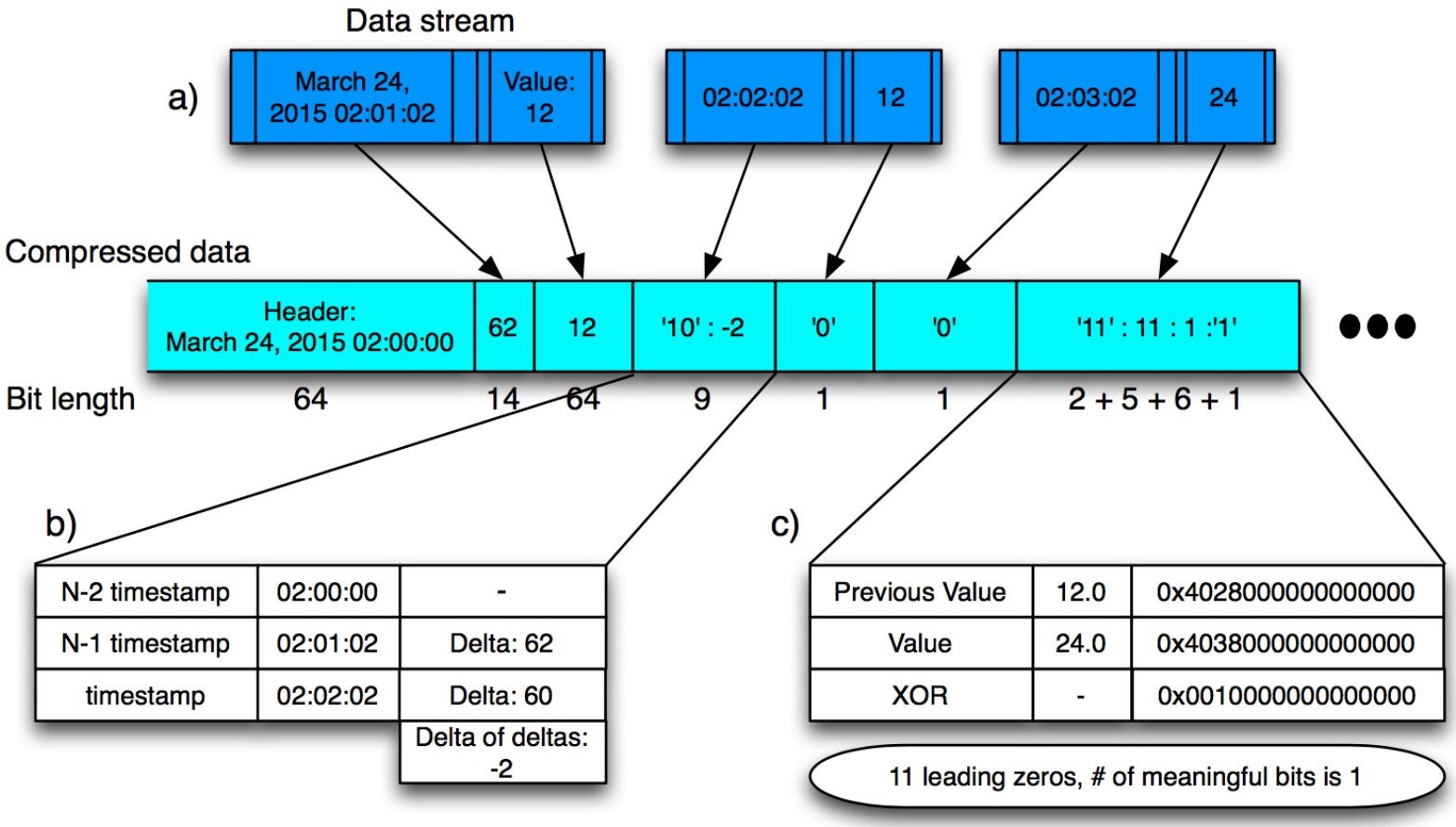
1] Compressing time-stamp

2] Compressing values

This algorithm is similar to a compression algorithm for floating-point data. This new algorithm uses the concept of XOR comparison between current value and previous value.

This novel compression algorithm allows Facebook to reduce the size of each time series from 16 bytes to an average 1.37 bytes. i.e. 12 times reduction in size.

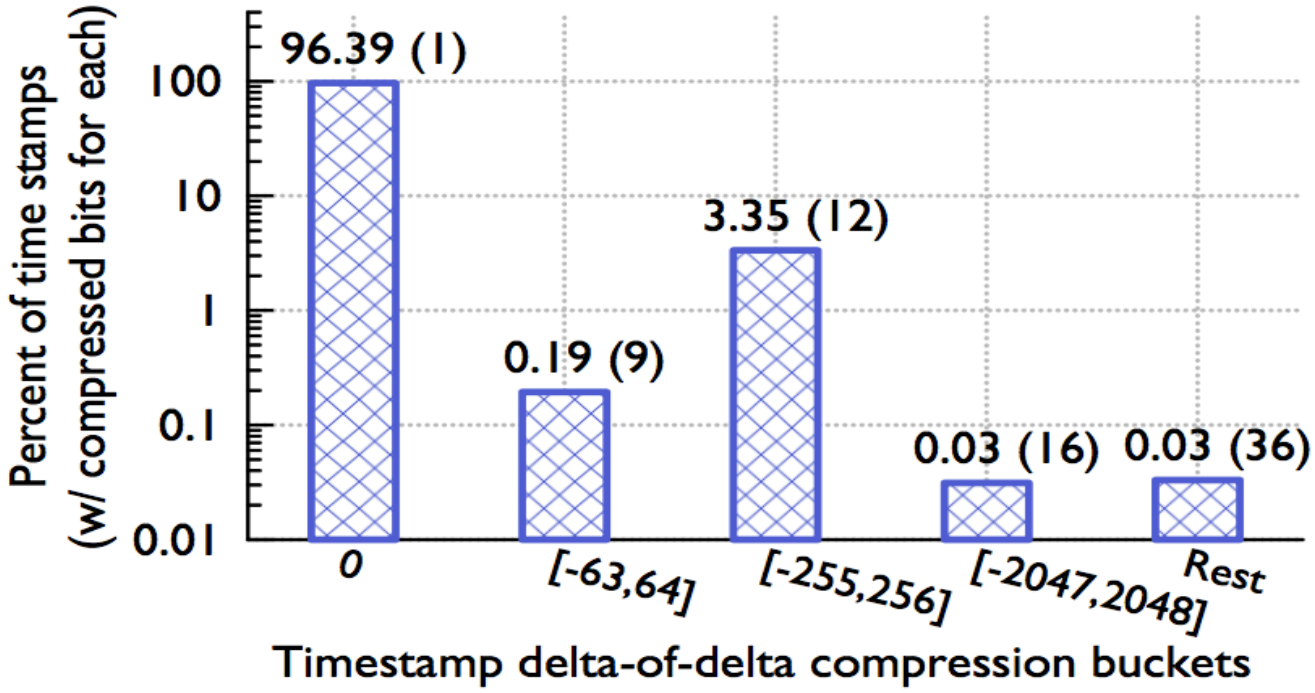
Time Series Compression Algorithm[1]



Compressing Time-Stamps[1]

1) The block header stores the starting time stamp t_{-1}, which is aligned to a two hour window. The first time-stamp t_0 in the block is stored as a delta from t_{-1} in 14 bits.
2) For subsequent time stamps, t_n:
    1) Calculate the delta of delta:
    2) $D = (t_n - t_{n-1})-(t_{n-1} - t_{n-2})$
    3) If D is zero, then store a single '0' bit
    4) If D is between [-63, 64], store '10' followed by the value (7 bits)
    5) If D is between [-255, 256], store '110' followed by the value (9 bits)
    6) If D is between [-2047, 2048], store '1110' followed by the value (12 bits)
    7) Otherwise store '1111' followed by D using 32 bits

Compressing Time-Stamps[1]



96% of all time-stamps can be compressed to a single bit.

Compressing Values[1]

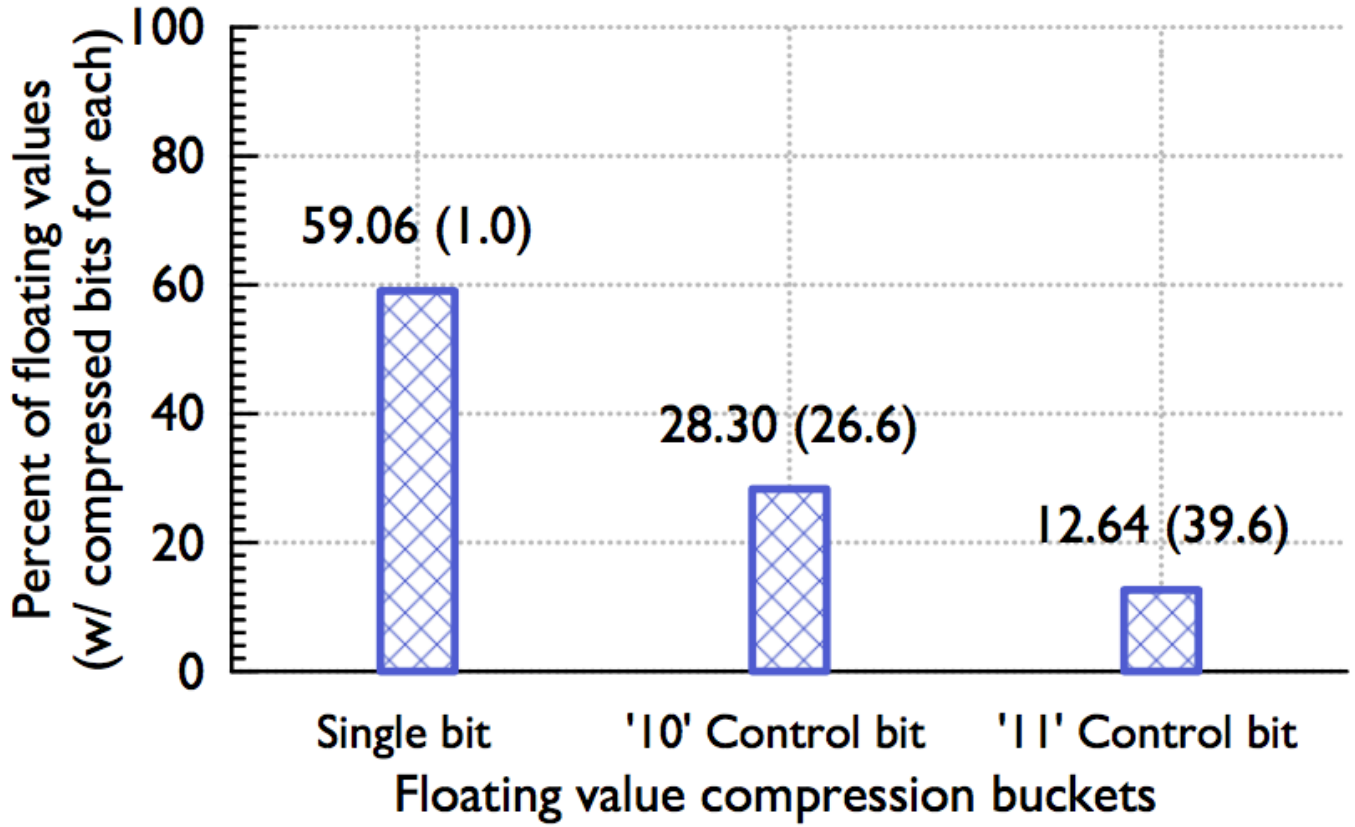
- According to Facebook, value in most time series does not change significantly when compared to its neighboring data points.

- At Facebook, Many of data sources only store integers into ODS. Hence, they have used this opportunity to fine tune the system by using a simple implementation in which current value is simply compared with the previous value.

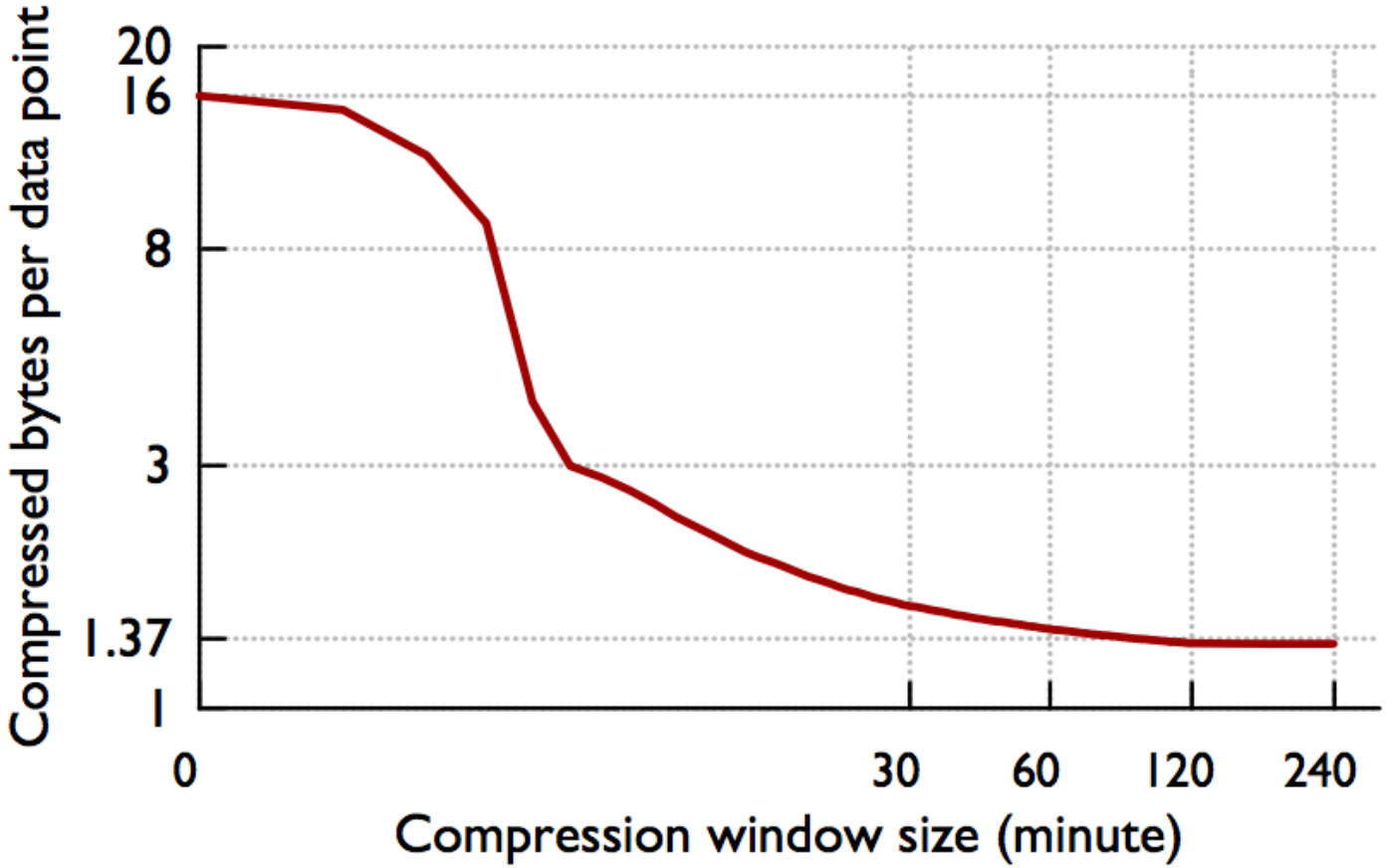
Compressing Values[1]

1) The first value is stored without compression.
2) Perform XOR operation with previous value. If both bits are identical then XORing should return 0.
3) When XORing results in a non-zero value, calculate the number of leading and trailing zeros in the XOR, store bit '1' followed by either a) or b)
   a) If meaningful bits of current value are matching with meaningful bits of of previous value then, the result should contain at least as many leading zeros and as many trailing zeros as with the previous value.
   b) Store the length of the number of leading zeros in the next 5 bits then, store the length of meaningful XORed value in the next 6 bits. In the end, store the meaningful bits of XORed value.

Compressing Values[1]



From 16 Bytes To 1.37 Bytes[1]



In-memory Data Structures[1]

- Timeseries Map (TSmap)

The primary data structure in Gorilla's implementation is a Timeseries Map (TSmap).

It consists of:

1) A vector from C++ standard library shared-pointers to time series (TS)
2) Case-preserving map from time series (TS) to TSmap.

In-memory Data Structures[1]

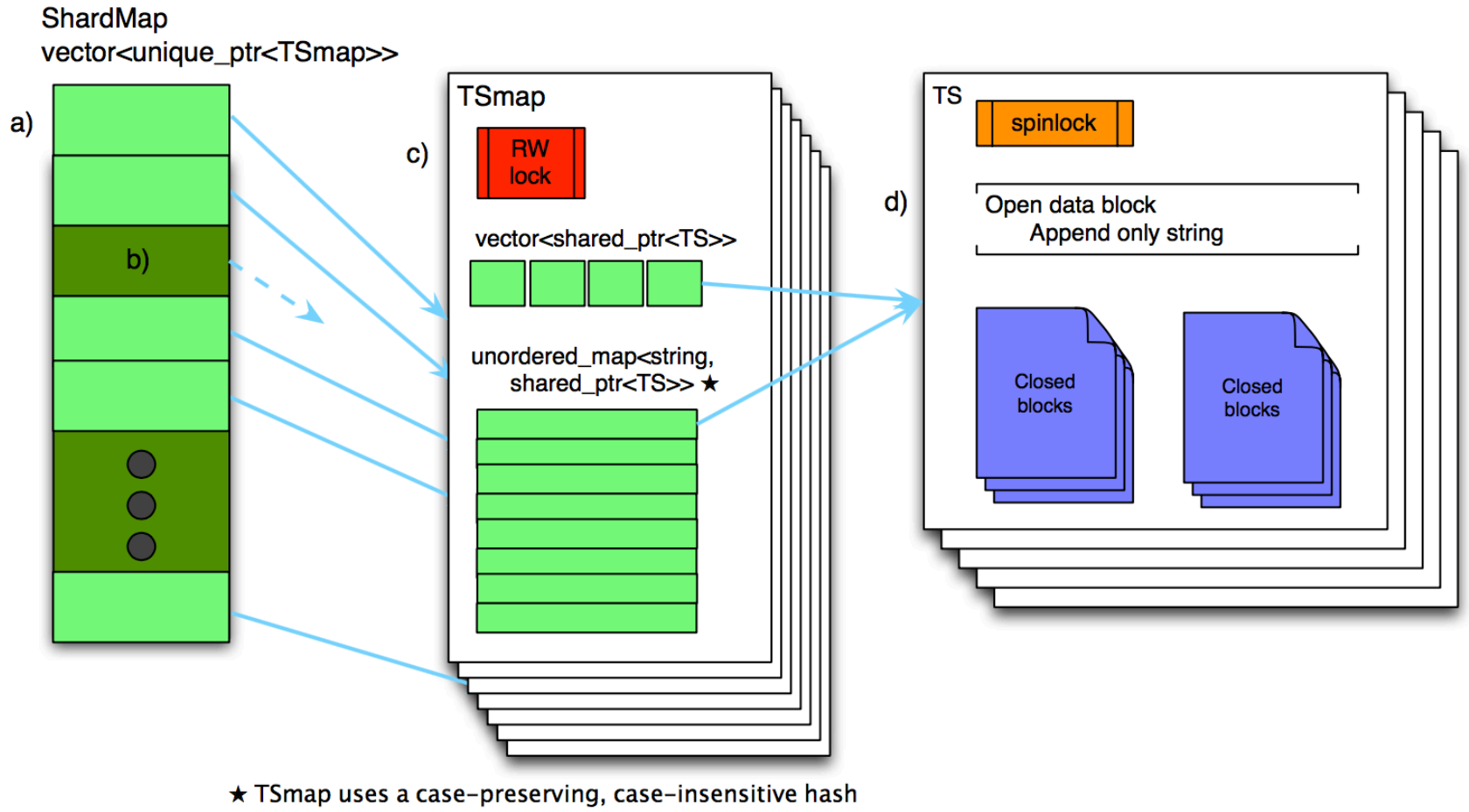
- ShardMap

It is used to match shard identifier (ShardId) to TSmap.

It consists of a vector of pointers to the TSmap.

Case-insensitive hash is used to map time series name to a shard. To achieve concurrent execution, ShardMap is managed with a read-write spin lock.

In-memory Data Structures[1]



On Disk Structures[1]

- Gorilla ensures persistence by incorporating GlusterFS with 3 times replication.
- Other distributed file systems can also be used instead of GlusterFS.
- Facebook thought of using databases like MySQL and RocksDB but database query language was not required by their persistence use case.
- Gorilla host consists of multiple shards of data.
- A single directory per shard

On Disk Structures[1]

- Each directory consists of:

1] Key Lists

- Mapping of time series string key to an integer id which represents index into the in-memory vector,

- New keys are appended to the current key list.

Gorilla scans all the keys belonging to each shard to do re-write operation on the file.

2] Append-Only Logs

- when data pointers are fed to Gorilla, they are store in a log file.

- There is only one append-only log per shard.

On Disk Structures[1]

3] Complete Block Files

- Every 2 hours, Gorilla copies a complete block of file to the disk. A complete block file consists of: A set of consecutive 64KB data blocks and list<TS_ID, data block pointer>

4] Checkpoint Files

- Once the block is complete, Gorilla marks a checkpoint in the checkpoint file and deletes the corresponding log. The checkpoint file is used to keep mark milestones when a complete block file is flushed to the disk. If a block file was not successfully flushed to disk in the event of process crash then, the new process is initiated and it reads data from log file since checkpoint file won't be there and block file is likely to be incomplete.

Trade-offs[1]

- Gorilla does not provide ACID properties
- Log is append-only log. No write-ahead-log (WAL).
- Data can be buffered up to 64KB (consists of data worth of 1 or 2 seconds) before being flushed.
- When the buffer is flushed the disk in case of shut-down, a crash might result in the loss of small amounts of data.
- Facebook acknowledges this trade-off because higher data insertion rate and high availability can be achieved at its cost.

Handling Failures[1]

- Single Node Failures and localized failures

- At Facebook, single node failures and localized failures (like network cuts to an entire region) are taken very serious since such situations can affect their business at the large scale.

- Handling single node failures has an additional benefit. When there are software upgrades or code pushes, they can model single node failures in their systems in order to perform well-organized software upgrades without disrupting the whole system and without any hassle.

Handling Failures[1]

- When a write operation is performed to Gorilla instance, high availability is guaranteed and consistency is not guaranteed.
- This makes failure handling a little bit easy. When one region fails then, queries will be directed to other healthy region until the failed region becomes alive for 26 hours.
- Dealing with failed region –

Let's say region A fails. Reads will be transparently retried to another healthy region B. If failure of A lasts for more than 60 seconds, data will be discarded from region A and retrying is stopped. In this case, all read operations from region A can be turned off until that region becomes healthy for at least 26 hours. This can be done automatically as well as manually.

Handling Failures[1]

- Dealing with shard re-assignment –

According to Facebook, 60 seconds are enough to do shard re-assignment. The buffers can hold the data points for 60 seconds and data points older than a minute will be discarded and most recent data will be loaded into the buffer since most recent data is more valuable as compared to the older data.

Let's say region A crashes during shard re-assignment. Writes are maintained for at least 1 minute as the Gorilla cluster tries to repair the region A. If region A comes up within 30 seconds or less then there is no data loss. If recovery does not happen quickly then reads can be directed to the other healthy region. This can be done manually as well as automatically.

Role Of GlusterFS and Host

- When shards are added to a host, it will read all the data from GlusterFS.
- A host is capable of reading all data that it needs to be fully functional within 5 minutes from GlusterFS.
- If the host is reading data and new data points come in then those data points are stored in the queue and processed as earlier as possible.
- When shards are re-assigned, the clients will immediately empty their buffers and put new nodes in the buffers.
- If the gorilla host is failing then it tries to flush all the data to the disk before dying so that no data is lost for software upgrades.

Returning Partial Results[1]

- When the region A is affected then, read operation can receive partial results from the Gorilla instance in region A. In this case, the read client library tries to fetch affected time-stamp data from region B which is a healthy region.
- It keeps those results if they are not partial.
- If the region A and region B, both return partial results then partial results are returned to the client with a flag informing that it is an incomplete data.
- To use that partial data or not, it is up to client. Facebook provides this option of delivering partial data because many automated systems can function well with only partial data as long as it is most recent data.
- Automatic read forwarding from an unhealthy region to healthy region protects the users from restarting the service and also avoid software upgrades.

Do we still need HBase? Yes!

- Facebook uses HBase as a log-term storage repository.
- If a particular data in not available in the memory then, HBase is queried for that data.
- Facebook engineers can still do their analysis and ad-hoc queries and Gorilla can still capture real-time detection of level changes after restarting it.
- HBase comes handy when all in-memory copies of data are lost.

New Tools On Gorilla[1]

Because of low latency query processing, Gorilla can be used to create new analysis tools.

- Correlation Engine

- It allows users to perform interactive search on many time series. This is currently limited to 1 million at a time.

- Correlation engine calculates Pearson Product-Moment Correlation Coefficient (PPMCC) which helps in finding co-relation between similarly shaped time-series at scale and helps to automate root-cause analysis.

New Tools On Gorilla[1]

- Charting

- Because of low latency query processing, it is possible to perform queries on higher volume datasets and develop charting tools.

- Such visualizations help users to detect time-series anomalies.

New Tools On Gorilla[1]

- Aggregations

- Before Gorilla, map-reduce jobs used to run against HBase cluster which would read all data for the 1 hour and return values.
- After Gorilla deployment, map-reduce jobs were eliminated and all completed buckets are scanned every 2 hours.
- Scanning in Gorilla is efficient and hence, load on HBase cluster was reduced. There is no need to write all data to disk and do expensive scans on HBase.

Experience[1]

- Fault-tolerance

- Gorilla survived 3 unplanned network cuts. It performed automatic read forwarding to healthy region without any issue.

- During fire-drill which simulated total-network cut to one storage back-end, Gorilla forwarded read to unaffected region and dashboards at the end users were showing expected data.

- Gorilla survived 6 configuration changes and 6 code pushes which involved restarting Gorilla in that region.

Experience[1]

- Bug Fixing –

Worked as expected during bug fixing event.

- Single Node Failures –

There were 5 single machine failures. No data is lost.

EXPERIENCE[1]

- Site Wide Error Rate Debugging

- Gorilla detected this error and generated an alert.
- One team of engineers mitigated the problem and another team found out the root cause of this error.
- 85% of all monitoring data is served by Gorilla. Hence, very few queries are required to be fetched from HBase which reduces load on HBase cluster.

Takeaway

- Most recent data is more valuable than any older data point.
- Low latency reads can open the doors for data analysis tools.
- When you want highly available and fault-tolerant system, there are trade-offs.

Future Work[1]

- Add second layer larger data store between Gorilla and Hbase based on flash storage.
- This store can be built to store data for more than 26 hours.
- At present, Gorilla is acting as a write-through cache for most recent data. That means there is a window of at least 26 hours before reading the data from HBase. One optimization would be increasing this period since most recent data will reside in Gorilla for a longer period of time and accessing HBase is discouraged as far as possible.

Conclusion

- Gorilla has reduced Facebook's production query latency by more than 70 times as compared to their previous on-disk TSDB.
- When this paper was published, Gorilla had been in production since 18 months and Facebook was able to double its size without much operational cost.
- Gorilla provides ensures scalability.
- Because of Fault tolerance capability, Gorilla can stay highly available for both read and writes in the event of unexpected failures.

Recent Work

- Beringei[7] is the open-source representation of the ideas presented in this paper.

- "Beringei currently stores up to 10 billion unique time series and serves 18 million queries per minute, powering most of the performance and health monitoring at Facebook while enabling our engineers and analysts to make decisions quickly with accurate, real-time data"[8].

References

[1] http://www.vldb.org/pvldb/vol8/p1816-teller.pdf

[2] https://blog.acolyer.org/2016/05/03/gorilla-a-fast-scalable-in-memory-time-series-database/

[3] https://www.youtube.com/watch?v=_r97qdyQtIk

[4] http://opentsdb.net

[5] https://www.influxdata.com

[6] https://github.com/graphite-project/whisper

[7] https://github.com/facebookincubator/beringei

[8] https://code.facebook.com/posts/952820474848503/beringei-a-high-performance-time-series-storage-engine/

Thanks!
Questions?