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

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

ABSTRACT

Large-scale internet services aim to remain highly available and responsive in the presence of unexpected failures. Providing this service often requires monitoring and analyzing tens of millions of measurements per second across a large number of systems, and one particularly effective solution is to store and query such measurements in a time series database (TSDB).

A key challenge in the design of TSDBs is how to strike the right balance between efficiency, scalability, and reliability. In this paper we introduce Gorilla, Facebook's in-memory TSDB. Our insight is that users of monitoring systems do not place much emphasis on individual data points but rather on aggregate analysis, and recent data points are of much higher value than older points to quickly detect and diagnose the root cause of an ongoing problem. Gorilla optimizes for remaining highly available for writes and reads, even in the face of failures, at the expense of possibly dropping small amounts of data on the write path. To improve query efficiency, we aggressively leverage compression techniques such as delta-of-delta timestamps and XOR'd floating point values to reduce Gorilla's storage footprint by 10x. This allows us to store Gorilla's data in memory, reducing query latency by 73x and improving query throughput by 14x when compared to a traditional database (HBase)-backed time series data. This performance improvement has unlocked new monitoring and debugging tools, such as time series correlation search and more dense visualization tools. Gorilla also gracefully handles failures from a single-node to entire regions with little to no operational overhead.

1. INTRODUCTION

Large-scale internet services aim to remain highly-available and responsive for their users even in the presence of unexpected failures. As these services have grown to support a global audience, they have scaled beyond a few systems running on hundreds of machines to thousands of individual systems running on many thousands of machines, often across multiple geo-replicated datacenters.

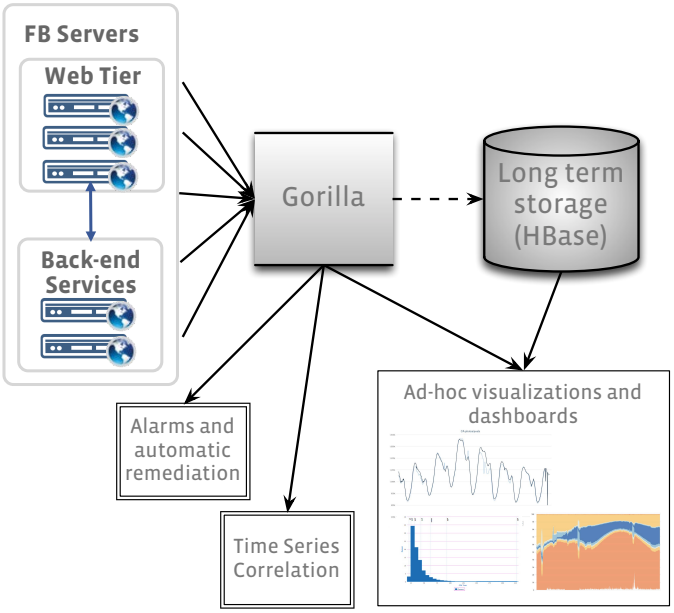

Figure 1: High level overview of the ODS monitoring and alerting system, showing Gorilla as a write-through cache of the most recent 26 hours of time series data.

An important requirement to operating these large scale services is to accurately monitor the health and performance of the underlying system and quickly identify and diagnose problems as they arise. Facebook uses a time series database (TSDB) to store system measuring data points and provides quick query functionalities on top. We next specify some of the constraints that we need to satisy for monitoring and operating Facebook and then describe Gorilla, our new in-memory TSDB that can store tens of millions of datapoints (e.g., CPU load, error rate, latency etc.) every second and respond queries over this data within milliseconds.

Writes dominate. Our primary requirement for a TSDB is that it should always be available to take writes. As we have hundreds of systems exposing multiple data items, the write rate might easily exceed tens of millions of data points each second. In constrast, the read rate is usually a couple orders of magnitude lower as it is primarily from automated systems watching 'important' time series, data

 Articles from this volume were invited to present their results at the 41st International Conference on Very Large Data Bases, August 31st - September 4th 2015, Kohala Coast, Hawaii.
Proceedings of the VLDB Endowment, Vol. 8, No. 12

visualization systems presenting dashboards for human consumption, or from human operators wishing to diagnose an observed problem.

State transitions. We wish to identify issues that emerge from a new software release, an unexpected side effect of a configuration change, a network cut and other issues that result in a significant state transition. Thus, we wish for our TSDB to support fine-grained aggregations over short-time windows. The ability to display state transitions within tens of seconds is particularly prized as it allows automation to quickly remediate problems before they become wide spread.

High availability. Even if a network partition or other failure leads to disconnection between different datacenters, systems operating within any given datacenter ought to be able to write data to local TSDB machines and be able to retrieve this data on demand.

Fault tolerance. We wish to replicate all writes to multiple regions so we can survive the loss of any given datacenter or geographic region due to a disaster.

Gorilla is Facebook's new TSDB that satisfies these constraints. Gorilla functions as a write-through cache of the most recent data entering the monitoring system. We aim to ensure that most queries run within 10's of milliseconds.

The insight in Gorilla's design is that users of monitoring systems do not place much emphasis on individual data points but rather on aggregate analysis. Additionally, these systems do not store any user data so traditional ACID guarantees are not a core requirement for TSDBs. However, a high percentage of writes must succeed at all times, even in the face of disasters that might render entire datacenters unreachable. Additionally, recent data points are of higher value than older points given the intuition that knowing if a particular system or service is broken *right now* is more valuable to an operations engineer than knowing if it was broken an hour ago. Gorilla optimizes for remaining highly available for writes and reads, even in the face of failures, at the expense of possibly dropping small amounts of data on the write path.

The challenge then arises from high data insertion rate, total data quantity, real-time aggregation, and reliability requirements. We addressed each of these in turn. To address the first couple requirements, we analyzed the Operational Data Store (ODS) TSDB, an older monitoring system that was widely used at Facebook. We noticed that at least 85% of all queries to ODS was for data collected in the past 26 hours. Further analysis allowed us to determine that we might be able to serve our users best if we could replace a disk-based database with an in-memory database. Further, by treating this in-memory database as a cache of the persistent disk-based store, we could achieve the insertion speed of an in-memory system with the persistence of a disk based database.

As of Spring 2015, Facebook's monitoring systems generate more than 2 billion unique time series of counters, with about 12 million data points added per second. This represents over 1 trillion points per day. At 16 bytes per point, the resulting 16TB of RAM would be too resource intensive for practical deployment. We addressed this by repurposing an existing XOR based floating point compression scheme to work in a streaming manner that allows us to compress time series to an average of 1.37 bytes per point, a 12x reduction in size.

We addressed the reliability requirements by running multiple instances of Gorilla in different datacenter regions and streaming data to each without attempting to guarantee consistency. Read queries are directed at the closest available Gorilla instance. Note that this design leverages our observation that individual data points can be lost without compromising data aggregation unless there's significant discrepancy between the Gorilla instances.

Gorilla is currently running in production at Facebook and is used daily by engineers for real-time firefighting and debugging in conjunction with other monitoring and analysis systems like Hive [27] and Scuba [3] to detect and diagnose problems.

2. BACKGROUND & REQUIREMENTS

2.1 Operational Data Store (ODS)

Operating and managing Facebook's large infrastructure comprised of hundreds of systems distributed across multiple data centers would be very difficult without a monitoring system that can track their health and performance. The Operational Data Store (ODS) is an important portion of the monitoring system at Facebook. ODS comprises of a time series database (TSDB), a query service, and a detection and alerting system. ODS's TSDB is built atop the HBase storage system as described in [26]. Figure 1 represents a high-level view of how ODS is organized. Time series data from services running on Facebook hosts is collected by the ODS write service and written to HBase.

There are two consumers of ODS time series data. The first consumers are engineers who rely on a charting system that generates graphs and other visual representations of time series data from ODS for interactive analysis. The second consumer is our automated alerting system that read counters off ODS, compares them to preset thresholds for health, performance and diagnostic metrics and fires alarms to oncall engineers and automated remediation systems.

2.1.1 Monitoring system read performance issues

In early 2013, Facebook's monitoring team realized that its HBase time series storage system couldn't scale handle future read loads. While the average read latency was acceptable for interactive charts, the 90^{th} percentile query time had increased to multiple seconds blocking our automation. Additionally, users were self-censoring their usage as interactive analysis of even medium-sized queries of a few thousand time series took tens of seconds to execute. Larger queries executing over sparse datasets would timeout as the HBase data store was tuned to prioritize writes. While our HBase-based TSDB was inefficient, we quickly rejected wholesale replacement of the storage system as ODS's HBase store held about 2 PB of data [5]. Facebook's data warehouse solution, Hive, was also unsuitable due to its already orders of magnitude higher query latency comparing to ODS, and query latency and efficiency were our main concerns [27].

We next turned our attention to in-memory caching. ODS already used a simple read-through cache but it was primarily targeted at charting systems where multiple dashboards shared the same time series. A particularly difficult scenario was when dashboards queried for the most recent data point, missed in the cache, and then issued requests

directly to the HBase data store. We also considered a separate Memcache [20] based write-through cache but rejected it as appending new data to an existing time series would require a read/write cycle, causing extremely high traffic to the memcache server. We needed a more efficient solution.

2.2 Gorilla requirements

With these considerations, we determined the following requirements for a new service:

- 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.

After a brief comparison with other TSDB systems in Section 3, we detail the implementation of Gorilla in Section 4, first discussing its new time stamp and data value compression schemes in Section 4.1. We then describe how Gorilla remains highly available despite single node failures and region-wide disasters in Section 4.4. We describe how Gorilla has enabled new tools in Section 5. We close out by describing our experience developing and deploying Gorilla in Section 6.

3. COMPARISON WITH TSDB SYSTEMS

There are a number of publications detailing data mining techniques to search, classify, and cluster enormous amounts of time series data efficiently [8, 23, 24]. These systems demonstrate many of the uses of examining time series data, from clustering and classifying [8, 23] to anomaly detection [10, 16] to indexing the time series [9, 12, 24]. However, there are fewer examples detailing systems able to gather and store massive amounts of time series data in real-time. Gorilla's design, focusing on reliable real-time monitoring of production systems, makes stand out compared to other TSDBs. Gorilla occupies an interesting design space, where being available for reads and writes in the face of failures prioritized over availability of any older data.

Since Gorilla was designed from the beginning to store all data in memory, its in-memory structure is also different from existing TSDBs. However, if one views Gorilla as an intermediate store for in-memory storage of time series data in front of another on-disk TSDB, then Gorilla could be used as a write through cache for any TSDB (with relatively simple modifications). Gorilla's focus on speed of ingest and horizontal scaling is similar to existing solutions.

3.1 OpenTSDB

OpenTSDB is based on HBase [28], and very closely resembles the ODS HBase storage layer we use for long term data. Both systems rely on similar table structures, and have come to similar conclusions for optimization and horizontal scalability [26, 28]. However, we had found that supporting the volume of queries necessary to build advanced monitoring tools required faster queries than a disk based store can support.

Unlike OpenTSDB, the ODS HBase layer does do time roll up aggregation for older data to save space. This results in older, archived data having lower time granularity compared to more recent data in ODS, while OpenTSDB will keep the full resolution data forever. We have found that cheaper long time period queries and space savings are worth the loss of precision.

OpenTSDB also has a richer data model for identifying time series. Each time series is identified by a set of arbitrary key-value pairs, called tags [28]. Gorilla identifies time series with a single string key and relies on higher level tools to extract and identify time series meta data.

3.2 Whisper (Graphite)

Graphite stores time series data on local disk in the Whisper format, a Round Robin Database (RRD) style database [1]. This file format expects time series data to be timestamped at regular intervals, and does not support jitter in the time series. While Gorilla does work more efficiently if data are timestamped at regular intervals, it can handle arbitrary and changing intervals. With Whisper, each time series is stored in a separate file, and new samples overwrite old ones after a certain amount of time [1]. Gorilla works in a similar fashion, only holding the most recent day of data in memory. However, with its focus on on-disk storage, query latency using Graphite/Whisper is not fast enough to match the requirements for Gorilla.

3.3 InfluxDB

InfluxDB is a new open-source time series database, with an even richer data model than OpenTSDB. Each event in a time series can have a full set of meta data. While this flexibility does allow for rich data, it necessarily results in larger disk usage than schemes that only store time series within the database [2].

InfluxDB also contains the code to build it as a distributed storage cluster, allowing users to scale horizontally without the overhead of managing an HBase/Hadoop cluster [2]. At Facebook, we already have dedicated teams to support our HBase installations, so using it for ODS did not involve a large extra investment in resources. Like other systems, InfluxDB keeps data on-disk, leading to slower queries than if data are kept in memory.

4. GORILLA ARCHITECTURE

Gorilla is an in-memory TSDB that functions as a write-through cache for monitoring data written to an HBase data store. The monitoring data stored in Gorilla is a simple 3-tuple of a string key, a 64 bit time stamp integer and a double precision floating point value. Gorilla incorporates a new time series compression algorithm that allows us to compress each by series down from 16 bytes to an average of 1.37 bytes, a 12x reduction in size. Further, we have arranged Gorilla's in-memory data structures to allow fast

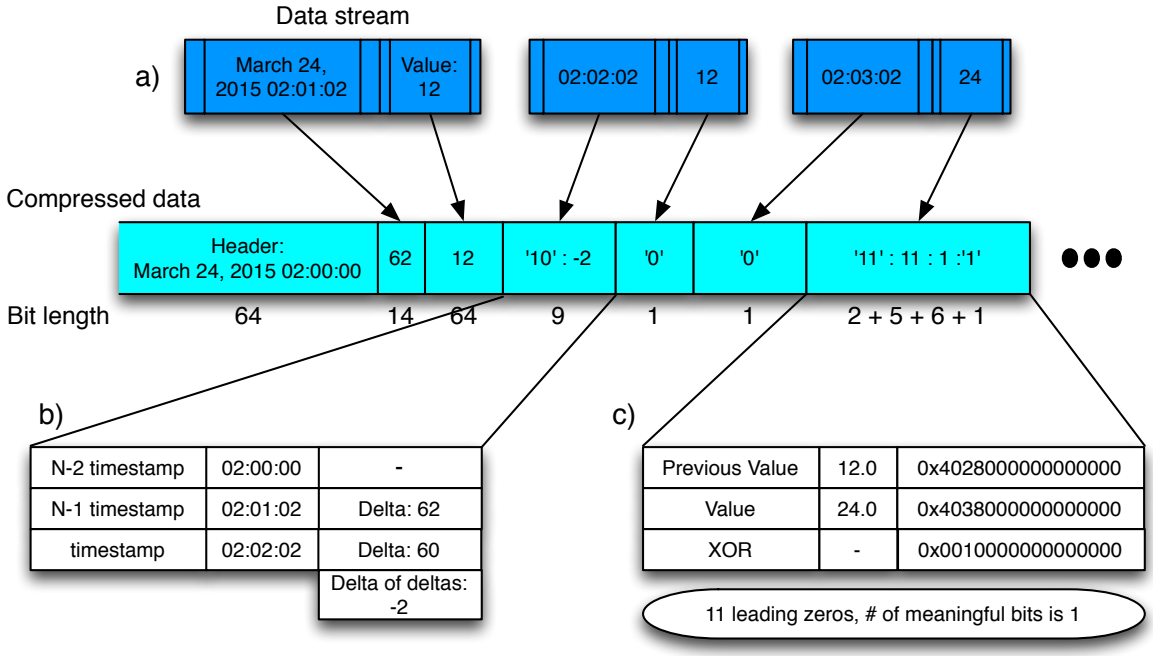

Figure 2: Visualizing the entire compression algorithm. For this example, 48 bytes of values and time stamps are compressed to just under 21 bytes/167 bits.

and efficient scans of all data while maintaining constant time lookup of individual time series.

The key specified in the monitoring data is used to uniquely identify a time series. By sharding all monitoring data based on these unique string keys, each time series dataset can be mapped to a single Gorilla host. Thus, we can scale Gorilla by simply adding new hosts and tuning the sharding function to map new time series data to the expanded set of hosts. When Gorilla was launched to production 18 months ago, our dataset of all time series data inserted in the past 26 hours fit into 1.3TB of RAM evenly distributed across 20 machines. Since then, we have had to double the size of the clusters twice due to data growth, and are now running on 80 machines within each Gorilla cluster. This process was simple due to the share-nothing architecture and focus on horizontal scalability.

Gorilla tolerates single node failures, network cuts, and entire datacenter failures by writing each time series value to two hosts in separate geographic regions. On detecting a failure, all read queries are failed over to the alternate region ensuring that users do not experience any disruption.

4.1 Time series compression

In evaluating the feasibility of building an in-memory time series database, we considered several existing compression schemes to reduce the storage overhead. We identified techniques that applied solely to integer data which didn't meet our requirement of storing double precision floating point values. Other techniques operated on a complete dataset but did not support compression over a stream of data as was stored in Gorilla [7, 13]. We also identified lossy time series approximation techniques used in data mining to make the problem set more easily fit within memory [15, 11], but Gorilla is focused on keeping the full resolution representation of data.

Our work was inspired by a compression scheme for floating point data derived in scientific computation. This scheme leveraged XOR comparison with previous values to generate a delta encoding [25, 17].

Gorilla compresses data points within a time series with no additional compression used across time series. Each data point is a pair of 64 bit values representing the time stamp and value at that time. Timestamps and values are compressed separately using information about previous values. The overall compression scheme is visualized in Figure 2, showing how time stamps and values are interleaved in the compressed block.

Figure 2.a illustrates the time series data as a stream consisting of pairs of measurements (values) and time stamps. Gorilla compresses this data stream into blocks, partitioned by time. After a simple header with an aligned time stamp (starting at 2 am, in this example) and storing the first value in a less compressed format, Figure 2.b shows that timestamps are compressed using delta-of-delta compression, described in more detail in Section 4.1.1. As shown in Figure 2.b the time stamp delta of delta is -2. This is stored with a two bit header ('10'), and the value is stored in seven bits, for a total size of just 9 bits. Figure 2.c shows floating-point values are compressed using XOR compression, described in more detail in Section 4.1.2. By XORing the floating point value with the previous value, we find that there is only a single meaningful bit in the XOR. This is then encoded with a two bit header ('11'), encoding that there are eleven leading zeros, a single meaningful bit, and the actual value ('1'). This is stored in fourteen total bits.

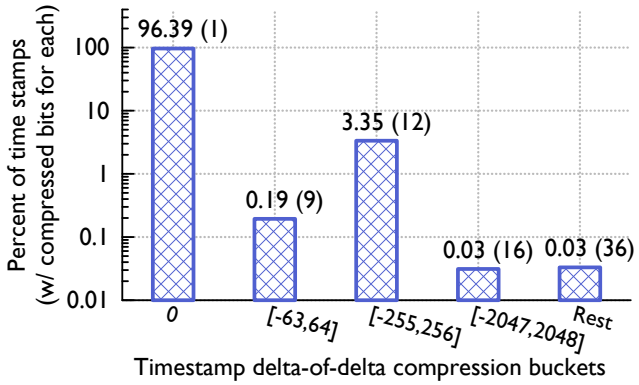

Figure 3: Distribution of time stamp compression across different ranged buckets. Taken from a sample of 440,000 real time stamps in Gorilla.

Decimal	Double Representation	XOR with previous
12	0x4028000000000000	
24	0x4038000000000000	0x0010000000000000
15	0x402e000000000000	0x0016000000000000
12	0x4028000000000000	0x0006000000000000
35	0x4041800000000000	0x0069800000000000

Decimal	Double Representation	XOR with previous
15.5	0x402f000000000000	
14.0625	0x402c200000000000	0x0003200000000000
3.25	0x400a000000000000	0x0026200000000000
8.625	0x4021400000000000	0x002b400000000000
13.1	0x402a333333333333	0x000b733333333333

Figure 4: Visualizing how XOR with the previous value often has leading and trailing zeros, and for many series, non-zero elements are clustered.

4.1.1 *Compressing time stamps*

We analyzed the time series data stored in ODS so we could optimize the compression scheme implemented in Gorilla. We noticed that the vast majority of ODS data points arrived at a fixed interval. For instance, it is common for a time series to log a single point every 60 seconds. Occasionally, the point may have a time stamp that is 1 second early or late, but the window is usually constrained.

Rather than storing timestamps in their entirety, we store an efficient delta of deltas. If the delta between time stamps for subsequent data points in a time series are 60, 60, 59 and 61 respectively, the delta of deltas is computed by subtracting the current time stamp value from the previous one which gives us 0, -1 and 2. An example of how this works is shown in Figure 2.

We next encode the delta of deltas using variable length encoding with the following algorithm:

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. [1]
2. For subsequent time stamps, t_n:
   (a) Calculate the delta of delta: $D = (t_n - t_{n-1}) - (t_{n-1} - t_{n-2})$
   (b) If D is zero, then store a single '0' bit
   (c) If D is between [-63, 64], store '10' followed by the value (7 bits)
   (d) If D is between [-255, 256], store '110' followed by the value (9 bits)
   (e) if D is between [-2047, 2048], store '1110' followed by the value (12 bits)
   (f) Otherwise store '1111' followed by D using 32 bits

The limits for the different ranges were selected by sampling a set of real time series from the production system and selecting the ones that gave the best compression ratio. A time series might have data points missing but the existing points likely arrived at fixed intervals. For example if there's one missing data point the deltas could be 60, 60, 121 and 59. The deltas of deltas would be 0, 61 and -62. Both 61 and -62 fit inside the smallest range and fewer bits can be used to encode these values. The next smallest range [-255, 256] is useful because a lot of the data points come in every 4 minutes and a single data point missing still uses that range.

Figure 3 show the results of time stamp compression in Gorilla. We have found that about 96% of all time stamps can be compressed to a single bit.

[1]The first time stamp delta is sized at 14 bits, because that size is enough to span a bit more than 4 hours (16,384 seconds), If one chose a Gorilla block larger than 4 hours, this size would increase.

4.1.2 *Compressing values*

In addition to the time stamp compression, Gorilla also compresses data values. Gorilla restricts the value element in its tuple to a double floating point type. We use a compression scheme similar to existing floating point compression algorithms, like the ones described in [17] and [25].

From analyzing our ODS data, we discovered that the value in most time series does not change significantly when compared to its neighboring data points. Further, many data sources only store integers into ODS. This allowed us to tune the expensive prediction scheme in [25] to a simpler implementation that merely compares the current value to the previous value. If values are close together the sign, exponent, and first few bits of the mantissa will be identical. We leverage this to compute a simple XOR of the current and previous values rather than employing a delta encoding scheme.

We then encode these XOR'd values with the following variable length encoding scheme:

1. The first value is stored with no compression
2. If XOR with the previous is zero (same value), store single '0' bit
3. When XOR is non-zero, calculate the number of leading and trailing zeros in the XOR, store bit '1' followed by either a) or b):
   (a) (Control bit '0') If the block of meaningful bits falls within the block of previous meaningful bits, i.e., there are at least as many leading zeros and as many trailing zeros as with the previous value,

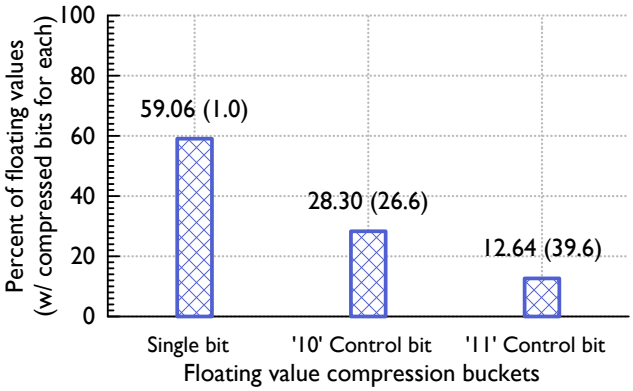

Figure 5: Distribution of values compressed across different XOR buckets. Taken from a sample of 1.6 million real values in Gorilla.

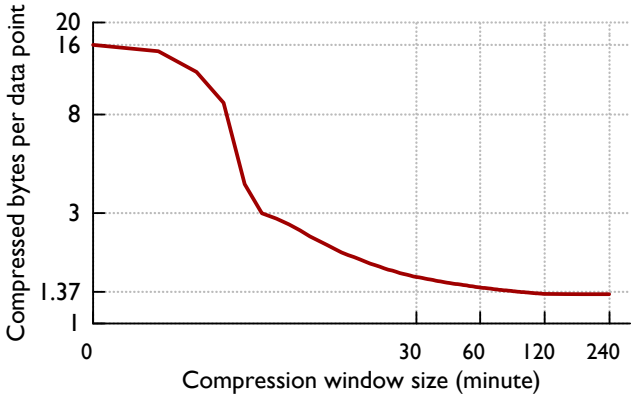

Figure 6: Average bytes used for each ODS data point as the compression bucket is varied from 0 (no compression) to 240 minutes. Bucket size larger than two hours do not give significant additional compression for our dataset. This is across the entire production Gorilla data set (approximately 2 billion time series).

use that information for the block position and just store the meaningful XORed value.

(b) (Control bit '1') Store the length of the number of leading zeros in the next 5 bits, then store the length of the meaningful XORed value in the next 6 bits. Finally store the meaningful bits of the XORed value.

The overall compression scheme is visualized in Figure 2 which depicts how our XOR encoding can store the values in a time series efficiently.

Figure 5 shows the distribution of actual values in Gorilla. Roughly 51% of all values are compressed to a single bit since the current and previous values are identical. About 30% of the values are compressed with the control bits '10' (case b), with an average compressed size of 26.6 bits. The remaining 19% are compressed with control bits '11', with an average size of 36.9 bits, due to the extra 13 bits of overhead required to encode the length of leading zero bits and meaningful bits.

This compression algorithm uses both the previous floating point value and the previous XORed value. This results in an additional compression factor because a sequence of XORed values often have a very similar number of leading and trailing zeros, as visualized in Figure 4. Integer values compress especially well because the location of the one bits after the XOR operation is often the same for the whole time series, meaning most values have the same number of trailing zeros.

One trade-off that is inherent in our encoding scheme is the time span over which the compression algorithm operates. Using the same encoding scheme over larger time periods allows us to achieve better compression ratios. However, queries that wish to read data over a short time range might need to expend additional computational resources on decoding data. Figure 6 shows the average compression ratio for the time series stored in ODS as we change the block size. One can see that blocks that extend longer than two hours provide diminishing returns for compressed size. A two-hour block allows us to achieve a compression ratio of 1.37 bytes per data point.

4.2 In-memory data structures

The primary data structure in Gorilla's implementation is a *Timeseries Map* (TSmap). Figure 7 provides an overview of this data structure. TSmap consists of a vector of C++ standard library shared-pointers to time series and a case-insensitive, case-preserving map from time series names to the same. The vector allows for efficient paged scans through all the data, while the map enables constant time lookups of particular time series. Constant time lookup is necessary to achieve the design requirement for fast reads while still allowing for efficient data scans.

The use of C++ shared-pointers enables scans to copy the vector (or pages thereof) in a few microseconds, avoiding lengthy critical sections that would impact the flow of incoming data. On deletion of a time series, a vector entry is tombstoned, and the index is placed in a free pool which is re-used when new time series are created. Tombstoneing a section of memory marks it as 'dead', and ready to be reused, without actually freeing it to the underlying system.

Concurrency is attained with a single read-write spin lock protecting map and vector access and 1-byte spin lock on each time series. Since each individual time series has a relatively low write throughput, the spin lock has very low contention between reads and writes.

As illustrated in Figure 7, the mapping of shard identifier (shardId) to TSmap, named ShardMap, is maintained with a vector of pointers to the TSmaps. Mapping of a time series name to a shard is done using the same case-insensitive hash in the TSmap, mapping it to an ID between $[0, NumberOfShards)$. Since the total number of shards in the system is constant and numbering in the low thousands, the additional overhead of storing null pointers in the ShardMap is negligible. Like the TSmaps, concurrent access to the ShardMap is managed with a read-write spin lock.

Since the data are already partitioned by shard, individual maps remain sufficiently small (about 1 million entries), the C++ standard library unordered-map has sufficient performance, and there have been no issues with lock contention.

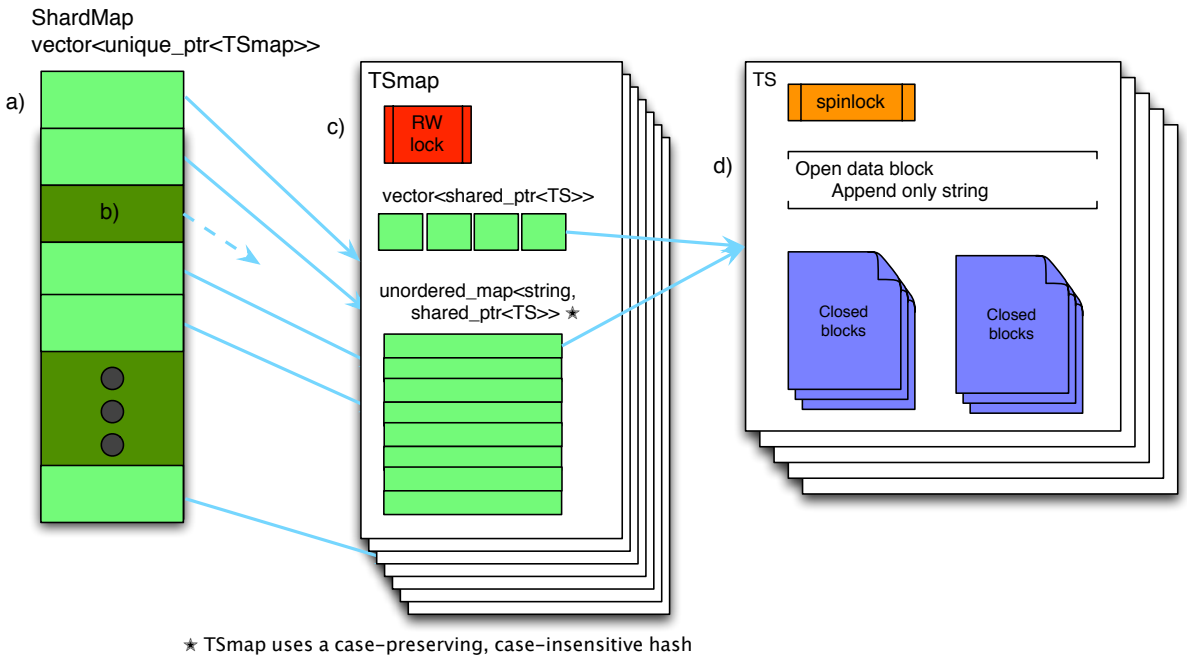

Figure 7: Gorilla in-memory data structure. On a query, first a) the TSmap pointer is examined. If b) the pointer is null, it means this Gorilla host does not own the shard. If non-null, then c) the TSmap is read-locked, and the pointer to the time series structure (TS) is found in the unordered_map and copied out. At this point, both RW locks can be unlocked. Next, d) the TS spinlock is locked, and data for the query time range can be directly copied out of the TS.

A time series data structure is composed of a sequence of closed blocks for data older than two hours and a single open data block that holds the most recent data. The open data block is an append-only string, to which new, compressed time stamps and values are appended. Since each block holds two hours of compressed data, the open block is closed once it is full. Once a block is closed, it is never changed until it is deleted out of memory. Upon closing, a block is copied to memory allocated from large slabs to reduce fragmentation. While the open block is often reallocated as it changes sizes, we find that the copy process reduces the overall fragmentation within Gorilla.

Data is read out by copying the data blocks that could contain data for the queried time range directly into the output remote procedure call structure. The entire data block is returned to the client, leaving the decompression step to be done outside Gorilla.

4.3 On disk structures

One of our goals for Gorilla is to survive single host failures. Gorilla achieves persistence by storing data in GlusterFS, a POSIX-compliant, distributed file system [4] with 3x replication. HDFS or other distributed file systems would have sufficed just as easily. We also considered single host databases such as MySQL and RocksDB but decided against these because our persistency use case did not require a database query language.

A Gorilla host will own multiple shards of data, and it maintains a single directory per shard. Each directory contains four types of files: Key lists, append-only logs, complete block files, and checkpoint files.

The key list is simply a map of the time series string key to an integer identifier. This integer identifier is the index into the in-memory vector. New keys are append to the current key list, and Gorilla periodically scans all the keys for each shard in order to re-write the file.

As data points are streamed to Gorilla, they are stored in a log file. The time stamps and values are compressed using the format described in Section 4.1. However, there is only one append-only log per shard, so values within a shard are interleaved across time series. This difference from the in memory encoding means that each compressed time stamp-value pair is also marked with it's 32-bit integer ID, adding significant storage overhead to the per-shard log file.

Gorilla does not offer ACID guarantees and as such, the log file is not a write-ahead-log. Data is buffered up to 64kB, usually comprising one or two seconds worth of data, before being flushed. While the buffer is flushed on a clean shutdown, a crash might result in the loss of small amounts of data. We found this trade-off to be worth the data loss, as it allowed higher data rate to be pushed to disk and higher availability for writes compared with a traditional write-ahead log.

Every two hours, Gorilla copies the compressed block data to disk, as this format is much smaller than the log files. There is one complete block file for every two hours worth of data. It has two sections: a set of consecutive 64kB slabs of data blocks, copied directly as they appear in memory, and a list of <time series ID, data block pointer> pairs. Once a block file is complete, Gorilla touches a checkpoint file and deletes the corresponding logs. The checkpoint file is used to mark when a complete block file is flushed to disk. If a block file was not successfully flushed to disk when it on a process crash, when the new process starts up, the

checkpoint file will not exist, so the new process knows it cannot trust the block file and will instead read from the log file only.

4.4 Handling failures

For fault tolerance, we chose to prioritize tolerating single node, temporary failures with zero observable downtime and large scale, and localized failures (such as a network cut to an entire region). We did this because single node failures happen quite often, and large scale, localized failures become a concern at Facebook's scale to allow the ability to operate under natural (or human-caused) disasters. There is the added benefit that one can model rolling software upgrades as a set of controlled, single node failures, so optimizing for this case means hassle-free and frequent code pushes. For all other failures, we chose trade-offs that, when they do cause data-loss, will prioritize the availability of more recent data over older data. This is because we can rely on the existing HBase TSDB system to handle historical data queries, and automated systems that detect level changes in time series are still useful with partial data, as long as has the most recent data.

Gorilla ensures that it remains highly available to data center faults or network partitions by maintaining two completely independent instances in separate data center regions. On a write, data is streamed to each Gorilla instance, with no attempt to guarantee consistency. This makes large-scale failures easy to handle. When an entire region fails, queries are directed at the other until the first has been back up for 26 hours. This is important to handle large scale disaster events, whether actual or simulated [21]. For example, when the Gorilla instance in region A completely fails, both reads and writes to that region will also fail. The read failures will be transparently retried to the Gorilla instance in the healthy region B. If the event lasts long enough (more than one minute), data will be dropped from region A, and requests will not be retried. When this happens, all reads can be turned off from region A until the cluster has been healthy for at least 26 hours. This remediation can be performed either manually or automated.

Within each region, a Paxos-based [6, 14] system called ShardManager assigns shards to nodes. When a node fails, ShardManager distributes its shards among other nodes in the cluster. During shard movement, write clients buffer their incoming data. The buffers are sized to hold 1 minute of data, and points older than a minute are discarded to make room for newer data. We have found that this time period is sufficient to allow shard reassignment in most situations, but for extended outages, it prioritizes the most recent data, as more recent data is intuitively more useful for driving automated detection systems. When a Gorilla host α in region A crashes or goes down for any reason, writes are buffered for at least 1 minute as the Gorilla cluster tries to resurrect the host. If the rest of the cluster is healthy, shard movement happens in thirty seconds or less, resulting in no data loss. If the movement does not occur fast enough reads can be pointed to the Gorilla instance in region B, either in a manual or automated process.

When shards are added to a host, it will read all the data from GlusterFS. These shards may have been owned by the same host before the restart or another. A host can read and process all the data it needs to be fully functional in about 5 minutes from GlusterFS. Because of the number of shards in the system and the total data stored, each shard represents about 16GB of on-disk storage. This can be read from GlusterFS in just a few minutes, as the files are spread across several physical hosts. While the host is reading the data, it will accept new incoming data points and put them into a queue to be processed at the earliest possible time. When shards are reassigned, clients immediately drain their buffers by writing to the new node. Going back to the Gorilla host α in region A crashing example: when α crashes, the shards are reassigned to host β in the same Gorilla instance. As soon as host β is assigned the shards, it begins accepting streaming writes, so no data loss occurs for in-flight data. If Gorilla host α is going down in a more controlled manner, it flushes all data to disk before exiting, so no data is lost for software upgrades.

In our example, if host α crashes before successfully flushing its buffers to disk, that data will be lost. In practice, this happens very rarely, and only a few seconds of data is actually lost. We make this trade-off to accept a higher throughput of writes and to allow accepting more recent writes sooner after an outage. Also, we monitor this situation, and are able to point reads at the more healthy region.

Note that after a node failure, shards will be partially unavailable for reads until the new nodes hosting these shards read the data from disk. Queries will return partial data (blocks are read most recent to least recent) and will mark the results as partial.

When the read client library receives a partial result from its query to the Gorilla instance in region A, it will retry fetching the affected time series from region B and keep those results if they are not partial. If both region A and region B return partial results, both partial results are returned to the caller with a flag set that some error caused incomplete data. The caller can then decide if it has enough information to continue processing the request or if it should fail outright. We make this choice because Gorilla is most often used by automated systems to detect level changes in time series. These systems can function well with only partial data, as long as it is the most recent data.

Automatic forwarding of reads from an unhealthy host to a healthy one means that users are protected from restarts and software upgrades. We find that upgrading the version of software causes zero data drops, and all reads continue to be served successfully with no manual intervention. This also allows Gorilla to transparently serve reads across server failures ranging from a single node to an entire region [21].

Finally, we still use our HBase TSDB for long-term storage of data. If all in-memory copies of data are lost, our engineers can still query the more durable storage system to do their analysis and drive ad-hoc queries, and Gorilla can still drive real-time detection of level changes, once it is restarted and accepting new writes.

5. NEW TOOLS ON GORILLA

Gorilla's low latency query processing has enabled the creation of new analysis tools.

5.1 Correlation engine

The first is a time series correlation engine that runs within Gorilla. Correlation search allows users to perform interactive, brute-force search on many time series, currently limited to 1 million at a time.

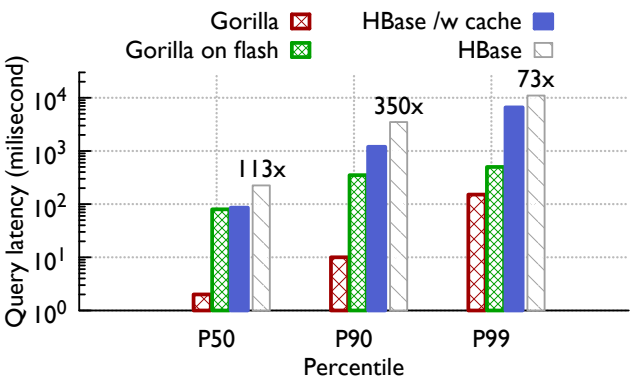

Figure 8: Total query latency breakdown with different TSDB solutions for ODS. Comparing to HBase, Gorilla has provided between 73x and 350x improvement, depending on the query size. This plot also includes preliminary results of two other options: Gorilla using flash to store data older than 26 hours, and HBase with ODS cache.

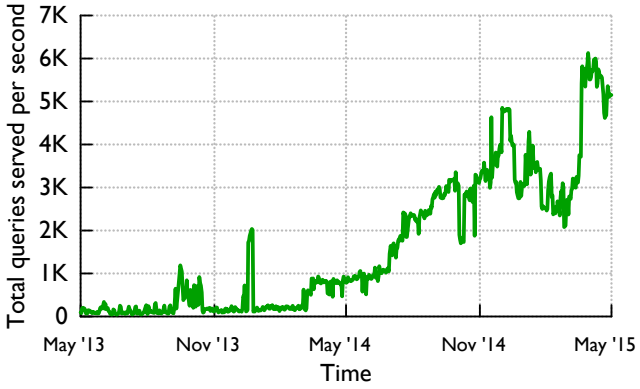

Figure 9: Growth of the total query volume since Gorilla's introduction to ease data exploration and develop new analysis tools.

The correlation engine calculates the Pearson Product-Moment Correlation Coefficient (PPMCC) which compares a test time series to a large set of time series [22]. We find that PPMCC's ability to find correlation between similarly shaped time series, regardless of scale, greatly helps automate root-cause analysis and answer the question "What happened around the time my service broke?". We found that this approach gives satisfactory answers to our question and was simpler to implement than similarly focused approaches described in the literature[10, 18, 16].

To compute PPMCC, the test time series is distributed to each Gorilla host along with all of the time series keys. Then, each host independently calculates the top N correlated time series, ordered by the absolute value of the PPMCC compared to the needle, and returning the time series values. In the future, we hope that Gorilla enables more advanced data mining techniques on our monitoring time series data, such as those described in the literature for clustering and anomaly detection [10, 11, 16].

5.2 Charting

Low latency queries have also enabled higher query volume tools. As an example, engineers unrelated to the monitoring team have created a new data visualization which will show large sets of horizon charts, which themselves are reductions across many time series. This visualization allows users to quickly visually scan across large sets of data to see outliers and time-correlated anomalies.

5.3 Aggregations

Recently, we moved the roll up background process from a set of map-reduce jobs to running directly against Gorilla. Recall that ODS performs time-based aggregations (or roll up) compression on old data, which is a lossy compression that reduces data granularity [26], similar to the format used by Whisper [1]. Before Gorilla, map-reduce jobs were run against the HBase cluster, which would read all data for the past hour and output values for a new, lower granularity table. Now, a background process periodically scans all completed buckets every two hours, generating the new values for the lower granularity table. Because scanning all data in Gorilla is very efficient, this move has reduced load on the HBase cluster, as we no longer need to write all the high granularity data to disk and do expensive full table scans on HBase.

6. EXPERIENCE

6.1 Fault tolerance

We next describe several planned and unplanned events that occurred over the past 6 months that affected some portion of Facebook's site availability. We restrict ourselves to discussing the impact of these events on Gorilla as other issues are beyond the scope of this paper.

Network cuts. 3 unplanned events resembling network cuts/outages to some portion of machines. The cuts were automatically detected and Gorilla automatically shifted reads to the unaffected coast without any service disruption.

Disaster readiness. 1 planned major fire drill simulating total network cut to one storage back end. As above, Gorilla switched reads to the unaffected coast. Once the downed region was restored, Gorilla in the down region was manually remediated to pull in logs from the firedrill time frame so dashboards served out of the down region would display the expected data to end users.

Configuration changes and code pushes. There were 6 configuration changes and 6 code releases that required restarting Gorilla in a given region.

Bug. A release with a major bug was pushed to a single coast. Gorilla immediately shifted load to the other region and continued serving uses until the bug was fixed. There was minimal correctness issues in the served data.

Single node failures. There were 5 single machine failures (unassociated with the major bug), causing no lost data and no remediation needed

There were zero events in Gorilla in the last 6 months that caused anomaly detection and alerting issues. Since Gorilla launched, there has only been 1 event that disrupted real-time monitoring. In all cases, the long-term storage was able to act as a backup for all monitoring-related queries.

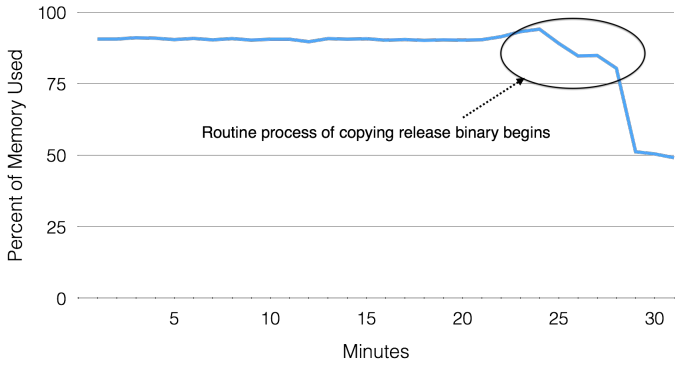

Figure 10: When searching for the root cause for a site-wide error rate increase, Gorilla's time series correlation found anomalous events that were correlated in time, namely a drop in memory used when copying a newly released binary.

6.2 Site wide error rate debugging

For an example of how Facebook uses time series data to drive our monitoring, one can look at a recent issue that was detected quickly and fixed due to monitoring data, first described externally at SREcon15 [19].

A mysterious problem resulted in a spike in the site wide error rate. This error rate was visible in Gorilla a few minutes after the error rate spike and raised an alert which notified the appropriate team a few minutes later [19]. Then, the hard work began. As one set of engineers mitigated the issue, others began the hunt for a root cause. Using tools built on Gorilla, including a new time series correlation search described in Section 5, they were able to find that the routine process of copying the release binary to Facebook's web servers caused an anomalous drop in memory used across the site, as illustrated in Figure 10. The detection of the problem, various debugging efforts and root cause analysis, depended on time series analysis tools enabled by Gorilla's high performance query engine.

Since launching about 18 months ago, Gorilla has helped Facebook engineers identify and debug several such production issues. By reducing the 90^{th} percentile Gorilla query time to 10ms, Gorilla has also improved developer productivity. Further by serving 85% of all monitoring data from Gorilla, very few queries must hit the HBase TSDB [26], resulting in a lower load on the HBase cluster.

6.3 Lessons learned

Prioritize recent data over historical data. Gorilla occupies an interesting optimization and design niche. While it must be very reliable, it does not require ACID data guarantees. In fact, we have found that it is more important for the most recent data to be available than any previous data point. This led to interesting design trade-offs, such as making a Gorilla host available for reads before older data is read off disk.

Read latency matters. The efficient use of compression and in-memory data structures has allowed for extremely fast reads and resulted in a significant usage increase. While ODS served 450 queries per second when Gorilla launched, Gorilla soon overtook it and currently handles more than 5,000 steady state queries per second, peaking at one point to 40,000 peak queries per second, as seen in Figure 9. Low latency reads have encouraged our users to build advanced data analysis tools on top of Gorilla as described in Section 5.

High availability trumps resource efficiency. Fault tolerance was an important design goal for Gorilla. It needed to be able to withstand single host failures with no interruption in data availability. Additionally, the service must be able to withstand disaster events that may impact an entire region. For this reason, we keep two redundant copies of data in memory despite the efficiency hit.

We found that building a reliable, fault tolerant system was the most time consuming part of the project. While the team prototyped a high performance, compressed, in-memory TSDB in a very short period of time, it took several more months of hard work to make it fault tolerant. However, the advantages of fault tolerance were visible when the system successfully survived both real and simulated failures [21]. We also benefited from a system that we can safely restart, upgrade, and add new nodes to whenever we need to. This has allowed us to scale Gorilla effectively with low operational overhead while providing a highly reliable service to our customers.

7. FUTURE WORK

We wish to extend Gorilla in several ways. One effort is to add a second, larger data store between in-memory Gorilla and HBase based on flash storage. This store has been built to hold the compressed two hour chunks but for a longer period than 26 hours. We have found that flash storage allows us to store about two weeks of full resolution, Gorilla compressed data. This will extend the amount of time full resolution data is available to engineers to debug problems. Preliminary performance results are included in Figure 8.

Before building Gorilla, ODS relied on the HBase backing store to be a real-time data store: very shortly after data was sent to ODS for storage, it needed to be available to read operations placing a significant burden on HBase's disk I/O. Now that Gorilla is acting as a write-through cache for the most recent data, we have at least a 26 hour window after data is sent to ODS before they will be read from HBase. We are exploiting this property by rewriting our write path to wait longer before writing to HBase. This optimization should be much more efficient on HBase, but the effort is too new to report results.

8. CONCLUSION

Gorilla is a new in-memory times series database that we have developed and deployed at Facebook. Gorilla functions as a write through cache for the past 26 hours of monitoring data gathered across all of Facebook's systems. In this paper, we have described a new compression scheme that allows us to efficiently store monitoring data comprising of over 700 million points per minute. Further, Gorilla has allowed us to reduce our production query latency by over 70x when compared to our previous on-disk TSDB. Gorilla has enabled new monitoring tools including alerts, automated remediation and an online anomaly checker. Gorilla has been in deployment for the past 18 months and has successfully doubled in size twice in this period without much operational effort demonstrating the scalability of our solution. We have also verified Gorilla's fault tolerance capabilities via

several large scale simulated failures as well as actual disaster situations—Gorilla remained highly available for both writes and reads through these events aiding site recovery.

9. ACKNOWLEDGEMENTS

Lots of thanks to Janet Wiener, Vinod Venkataraman and the others who reviewed early drafts of this paper to find typos and incorrect information.

Huge thanks to Sanjeev Kumar and Nathan Bronson, who had great insights into framing the paper to make it read better.

Thank you to Mike Nugent, who had the brilliant idea to use PPMCC to find root causes and effects caused by interesting time series, and hacked a prototype together so quickly.

Of course, thanks to the current ODS team (Alex Bakhturin, Scott Franklin, Ostap Korkuna, Wojciech Lopata, Jason Obenberger, and Oleksandr Voietsa), and ODS alumnus (Tuomas Pelkonen and Charles Thayer) who have made monitoring Facebook's infrastructure so much fun over the last few years. You guys are awesome!

10. REFERENCES

[1] Graphite - Scalable Realtime Graphing. http://graphite.wikidot.com/. Accessed March 20, 2015.

[2] Influxdb.com: InfluxDB - Open Source Time Series, Metrics, and Analytics Database. http://influxdb.com/. Accessed March 20, 2015.

[3] L. Abraham, J. Allen, O. Barykin, V. R. Borkar, B. Chopra, C. Gerea, D. Merl, J. Metzler, D. Reiss, S. Subramanian, J. L. Wiener, and O. Zed. Scuba: Diving into Data at Facebook. *PVLDB*, 6(11):1057–1067, 2013.

[4] E. B. Boyer, M. C. Broomfield, and T. A. Perrotti. GlusterFS One Storage Server to Rule Them All. Technical report, Los Alamos National Laboratory (LANL), 2012.

[5] N. Bronson, T. Lento, and J. L. Wiener. Open Data Challenges at Facebook. In *Workshops Proceedings of the 31st International Conference on Data Engineering Workshops, ICDE Seoul, Korea*. IEEE, 2015.

[6] T. D. Chandra, R. Griesemer, and J. Redstone. Paxos Made Live: An Engineering Perspective. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, pages 398–407. ACM, 2007.

[7] H. Chen, J. Li, and P. Mohapatra. RACE: Time Series Compression with Rate Adaptivity and Error Bound for Sensor Networks. In *Mobile Ad-hoc and Sensor Systems, 2004 IEEE International Conference on*, pages 124–133. IEEE, 2004.

[8] B. Hu, Y. Chen, and E. J. Keogh. Time Series Classification under More Realistic Assumptions. In *SDM*, pages 578–586, 2013.

[9] E. Keogh, K. Chakrabarti, M. Pazzani, and S. Mehrotra. Locally Adaptive Dimensionality Reduction for Indexing Large Time Series Databases. *ACM SIGMOD Record*, 30(2):151–162, 2001.

[10] E. Keogh, S. Lonardi, and B.-c. Chiu. Finding Surprising Patterns in a Time Series Database in Linear Time and Space. In *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 550–556. ACM, 2002.

[11] E. Keogh, S. Lonardi, and C. A. Ratanamahatana. Towards Parameter-Free Data Mining. In *Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 206–215. ACM, 2004.

[12] E. Keogh and C. A. Ratanamahatana. Exact Indexing of Dynamic Time Warping. *Knowledge and information systems*, 7(3):358–386, 2005.

[13] I. Lazaridis and S. Mehrotra. Capturing Sensor-Generated Time Series with Quality Guarantees. In *Data Engineering, 2003. Proceedings. 19th International Conference on*, pages 429–440. IEEE, 2003.

[14] Leslie Lamport. Paxos Made Simple. *SIGACT News*, 32(4):51–58, December 2001.

[15] J. Lin, E. Keogh, S. Lonardi, and B. Chiu. A Symbolic Representation of Time Series, with Implications for Streaming Algorithms. In *Proceedings of the 8th ACM SIGMOD workshop on Research issues in data mining and knowledge discovery*, pages 2–11. ACM, 2003.

[16] J. Lin, E. Keogh, S. Lonardi, J. P. Lankford, and D. M. Nystrom. Visually Mining and Monitoring Massive Time Series. In *Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 460–469. ACM, 2004.

[17] P. Lindstrom and M. Isenburg. Fast and Efficient Compression of Floating-Point Data. *Visualization and Computer Graphics, IEEE Transactions on*, 12(5):1245–1250, 2006.

[18] A. Mueen, S. Nath, and J. Liu. Fast Approximate Correlation for Massive Time-Series Data. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 171–182. ACM, 2010.

[19] R. Nishtala. Learning from Mistakes and Outages. Presented at SREcon, Santa Clara, CA, March 2015.

[20] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, et al. Scaling Memcache at Facebook. In *nsdi*, volume 13, pages 385–398, 2013.

[21] J. Parikh. Keynote speech. Presented at @Scale Conference, San Francisco, CA, September 2014.

[22] K. Pearson. Note on regression and inheritance in the case of two parents. *Proceedings of the Royal Society of London*, 58(347-352):240–242, 1895.

[23] F. Petitjean, G. Forestier, G. Webb, A. Nicholson, Y. Chen, and E. Keogh. Dynamic Time Warping Averaging of Time Series Allows Faster and More Accurate Classification. In *IEEE International Conference on Data Mining*, 2014.

[24] T. Rakthanmanon, B. Campana, A. Mueen, G. Batista, B. Westover, Q. Zhu, J. Zakaria, and E. Keogh. Searching and Mining Trillions of Time Series Subsequences Under Dynamic Time Warping. In *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 262–270. ACM, 2012.

[25] P. Ratanaworabhan, J. Ke, and M. Burtscher. Fast Lossless Compression of Scientific Floating-Point Data. In *DCC*, pages 133–142. IEEE Computer Society, 2006.

[26] L. Tang, V. Venkataraman, and C. Thayer. Facebook's Large Scale Monitoring System Built on HBase. Presented at Strata Conference, New York, 2012.

[27] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: A Warehousing Solution Over a Map-Reduce Framework. *PVLDB*, 2(2):1626–1629, 2009.

[28] T. W. Wlodarczyk. Overview of Time Series Storage and Processing in a Cloud Environment. In *Proceedings of the 2012 IEEE 4th International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 625–628. IEEE Computer Society, 2012.