BTrDB: Optimizing Storage System Design for Timeseries Processing

Michael P Andersen and David E. Culler, ***University of California, Berkeley***

https://www.usenix.org/conference/fast16/technical-sessions/presentation/andersen

This paper is included in the Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST '16).

February 22–25, 2016 • Santa Clara, CA, USA

ISBN 978-1-931971-28-7

Open access to the Proceedings of the 14th USENIX Conference on File and Storage Technologies is sponsored by USENIX

BTrDB: Optimizing Storage System Design for Timeseries Processing

Michael P Andersen
University of California, Berkeley
m.andersen@cs.berkeley.edu

David E. Culler
University of California, Berkeley
culler@cs.berkeley.edu

Abstract

The increase in high-precision, high-sample-rate telemetry timeseries poses a problem for existing timeseries databases which can neither cope with the throughput demands of these streams nor provide the necessary primitives for effective analysis of them. We present a novel abstraction for telemetry timeseries data and a data structure for providing this abstraction: a time-partitioning version-annotated copy-on-write tree. An implementation in Go is shown to outperform existing solutions, demonstrating a throughput of 53 million inserted values per second and 119 million queried values per second on a four-node cluster. The system achieves a 2.9x compression ratio and satisfies statistical queries spanning a year of data in under 200ms, as demonstrated on a year-long production deployment storing 2.1 trillion data points. The principles and design of this database are generally applicable to a large variety of timeseries types and represent a significant advance in the development of technology for the Internet of Things.

1 Introduction

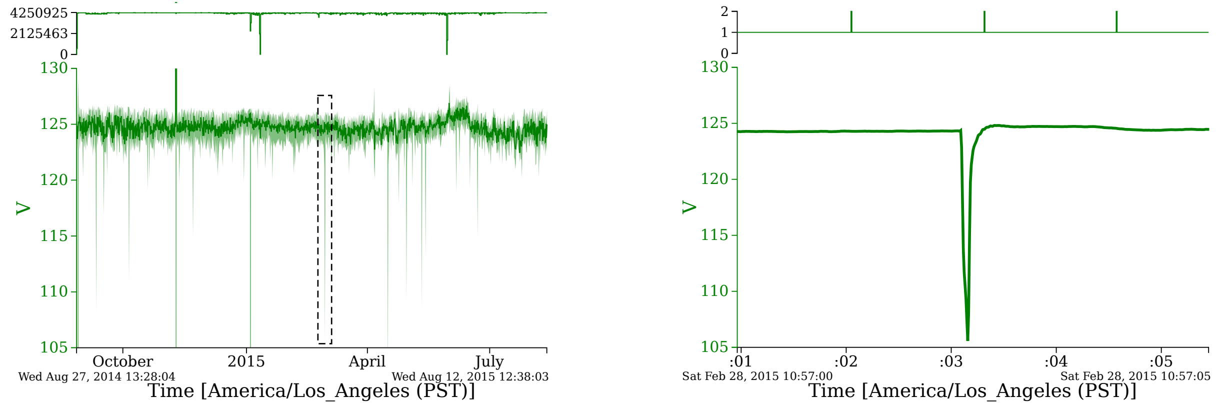
A new class of distributed system with unique storage requirements is becoming increasingly important with the rise of the Internet of Things. It involves collecting, distilling and analyzing – in near real-time and historically – time-correlated telemetry from a large number of high-precision networked sensors with fairly high sample rates. This scenario occurs in monitoring the internal dynamics of electric grids, building systems, industrial processes, vehicles, structural health, and so on. Often, it provides situational awareness of complex infrastructure. It has substantially different characteristics from either user-facing focus and click data, which is pervasive in modern web applications, smart metering data, which collects 15-minute interval data from many millions of meters, or one-shot dedicated instrument logging.

We focus on one such source of telemetry – microsynchophasors, or uPMUs. These are a new generation of small, comparatively cheap and extremely high-precision power meters that are to be deployed in the distribution tier of the electrical grid, possibly in the millions. In the distributed system shown in Figure 2, each device produces 12 streams of 120 Hz high-precision values with timestamps accurate to 100 ns (the limit of GPS). Motivated by the falling cost of such data sources, we set out to construct a system supporting more than 1000 of these devices per backing server – more than 1.4 million inserted points per second, and several times this in expected reads and writes from analytics. Furthermore, this telemetry frequently arrives out of order, delayed and duplicated. In the face of these characteristics, the storage system must guarantee the consistency of not only the raw streams, but all analytics derived from them. Additionally, fast response times are important for queries across time scales from years to milliseconds.

These demands exceed the capabilities of current timeseries data stores. Popular systems, such as KairosDB [15], OpenTSDB [20] or Druid [7], were designed for complex multi-dimensional data at low sample rates and, as such, suffer from inadequate throughput and timestamp resolution for these telemetry streams, which have comparatively simple data and queries based on time extents. These databases all advertise reads and writes of far less than 1 million values per second per server, often with order-of-arrival and duplication constraints, as detailed in Section 2.

As a solution to this problem, a novel, ground-up, use-inspired time-series database abstraction – BTrDB – was constructed to provide both higher sustained throughput for raw inserts and queries, as well as advanced primitives that accelerate the analysis of the expected 44 quadrillion datapoints per year *per server*.

The core of this solution is a new abstraction for time series telemetry data (Section 3) and a data structure that provides this abstraction: a *time-partitioning, multi-*



(a) Statistical summary of a year of voltage data to locate voltage sags, representing 50 billion readings, with min, mean, and max shown. The data density (the plot above the main plot) is 4.2 million points per pixel column.

(b) The voltage sag outlined in (a) plotted over 5 seconds. The data density (the plot above the main plot) is roughly one underlying data point per pixel column

Figure 1: Locating interesting events in typical uPMU telemetry streams using statistical summaries

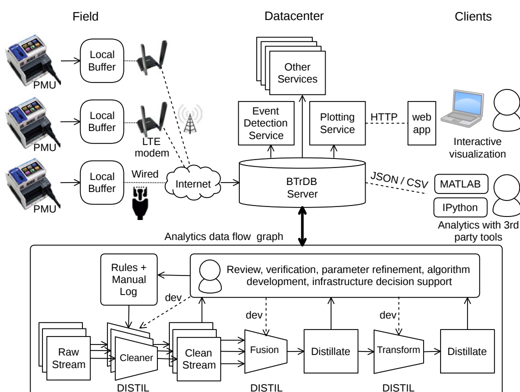


Figure 2: uPMU network storage and query processing system

resolution, version-annotated, copy-on-write tree, as detailed in Section 4. A design for a database using this data structure is presented in Section 5.

An open-source 4709-line Go implementation of BTrDB demonstrates the simplicity and efficacy of this method, achieving 53 million inserted values per second and 119 million queried values per second on a four node cluster with the necessary fault-tolerance and consistency guarantees. Furthermore, the novel analytical primitives allow the navigation of a year worth of data comprising billions of datapoints (e.g. Figure 1a) to locate and analyse a sub-second event (e.g. Figure 1b) using a sequence of statistical queries that complete in 100-200ms, a result not possible with current tools. This is discussed in Section 6.

2 Related work

Several databases support high-dimensionality time-series data, including OpenTSDB [20], InfluxDB [12], KairosDB [15] and Druid [7]. In terms of raw telemetry, these databases are all limited to millisecond-precision timestamps. This is insufficient to capture phase angle samples from uPMUs, which require sub-microsecond-precision timestamps. While all of these are capable of storing scalar values, they also support more advanced "event" data, and this comes at a cost. Druid advertises that "large production clusters" have reached "1M+ events per second" on "tens of thousands of cores." Published results for OpenTSDB show $<$ 1k operations per second per node [4][10]. MapR has shown OpenTSDB running on MapR-DB, modified to support batch inserts and demonstrated 27.5 million inserted values per second per node (bypassing parts of OpenTSDB) and 375 thousand reads per second per node [28][8]; unfortunately this performance is with 1 byte values and counter-derived second-precision timestamps, which somewhat undermines its utility [27].

A study evaluating OpenTSDB and KairosDB [10] with real PMU data showed that KairosDB significantly outperforms OpenTSDB, but only achieves 403,500 inserted values per second on a 36 node cluster. KairosDB gives an example of 133 k inserted values per second [14] using bulk insert in their documentation. Rabl et. al [24] performed an extensive benchmark comparing Project Voldemort [23], Redis [26], HBase [3], Cassandra [2], MySQL [21] and VoltDB [31][29]. Cassandra exhibited the highest throughput, inserting 230k records per second on a twelve node cluster. The records used were large (75 bytes), but even if optimistically normalised to the size of our records (16 bytes) it only yields roughly 1M inserts per second, or 89K inserts per second per node. The other five candidates exhibited lower throughput. Datastax [6] performed a simi-

lar benchmark [9] comparing MongoDB [19], Cassandra [2], HBase [3] and Couchbase [5]. Here too, Cassandra outperformed the competition obtaining 320k inserts/sec and 220k reads/sec on a 32 node cluster.

Recently, Facebook's in-memory Gorilla database [22] takes a similar approach to BTrDB - simplifying the data model to improve performance. Unfortunately, it has second-precision timestamps, does not permit out-of-order insertion and lacks accelerated aggregates.

In summary, we could find no databases capable of handling 1000 uPMUs per server node (1.4 million inserts/s per node and 5x that in reads), even without considering the requirements of the analytics. Even if existing databases *could* handle the raw throughput, and timestamp precision of the telemetry, they lack the ability to satisfy queries over large ranges of data efficiently. While many time series databases support aggregate queries, the computation requires on-the-fly iteration of the base data (e.g. OpenTSDB, Druid) - untenable at 50 billion samples per year per uPMU. Alternatively, some timeseries databases offer precomputed aggregates (e.g, InfluxDB, RespawnDB [4]), accelerating these queries, but they are unable to guarantee the consistency of the aggregates when data arrives out of order or is modified. The mechanisms to guarantee this consistency exist in most relational databases, but those fare far worse in terms of throughput.

Thus, we were motivated to investigate a clean slate design and implementation of a time-series database with the necessary capabilities – high throughput, fixed-response-time analytics irrespective of the underlying data size and eventual consistency in a graph of interdependent analytics despite out of order or duplicate data. This was approached in an integrated fashion from the block or file server on up. Our goal was to develop a multi-resolution storage and query engine for many higher bandwidth ($>$ 100 Hz) streams that provides the above functionality essentially "for free", in that it operates at the full line rate of the underlying network or storage infrastructure for affordable cluster sizes ($<$ 6 servers).

BTrDB has promising functionality and performance. On four large EC2 nodes it achieves over 119M queried values per second ($>$10GbE line rate) and over 53M inserted values per second of 8 byte time and 8 byte value pairs, while computing statistical aggregates. It returns results of 2K points summarizing anything from the raw values (9 ms) to 4 billion points (a year) in 100-250ms. It does this while maintaining the provenance of all computed values and consistency of a network of streams. The system storage overhead is negligible, with an all-included compression ratio of 2.9x – a significant improvement on existing compression techniques for synchrophasor data streams.

3 Time Series Data Abstraction

The fundamental abstraction provided by BTrDB is a consistent, write-once, ordered sequence of time-value pairs. Each stream is identified by a UUID. In typical uses, a substantial collection of metadata is associated with each stream. However, the nature of the metadata varies widely amongst uses and many good solutions exist for querying metadata to obtain a collection of streams. Thus, we separate the lookup (or directory) function entirely from the time series data store, identifying each stream solely by its UUID. All access is performed on a temporal segment of a version of a stream. All time stamps are in nanoseconds with no assumptions on sample regularity.

InsertValues(UUID, [(time, value)]) creates a new version of a stream with the given collection of (time,value) pairs inserted. Logically, the stream is maintained in time order. Most commonly, points are appended to the end of the stream, but this cannot be assumed: readings from a device may be delivered to the store out of order, duplicates may occur, holes may be backfilled and corrections may be made to old data – perhaps as a result of recalibration. These situations routinely occur in real world practice, but are rarely supported by timeseries databases. In BTrDB, each insertion of a collection of values creates a new version, leaving the old version unmodified. This allows new analyses to be performed on old versions of the data.

The most basic access method, **GetRange(UUID, StartTime, EndTime, Version) $\rightarrow$ (Version, [(Time, Value)])** retrieves all the data between two times in a given version of the stream. The 'latest' version can be indicated, thereby eliminating a call to **GetLatestVersion(UUID) $\rightarrow$ Version** to obtain the latest version for a stream prior to querying a range. The exact version number is returned along with the data to facilitate a repeatable query in future. BTrDB does not provide operations to resample the raw points in a stream on a particular schedule or to align raw samples across streams because performing these manipulations correctly ultimately depends on a semantic model of the data. Such operations are well supported by mathematical environments, such as Pandas [17], with appropriate control over interpolation methods and so on.

Although this operation is the only one provided by most historians, with trillions of points, it is of limited utility. It is used in the final step after having isolated an important window or in performing reports, such as disturbances over the past hour. Analyzing raw streams in their entirety is generally impractical; for example, each uPMU produces nearly 50 billion samples per year.

The following access methods are far more powerful for broad analytics and for incremental generation of computationally refined streams.

In visualizing or analyzing huge segments of data **GetStatisticalRange(UUID, StartTime, EndTime, Version, Resolution) $\rightarrow$ (Version, [(Time, Min, Mean, Max, Count)])** is used to retrieve statistical records between two times at a given temporal resolution. Each record covers $2^{resolution}$ nanoseconds. The start time and end time are on $2^{resolution}$ boundaries and result records are periodic in that time unit; thus summaries are aligned across streams. Unaligned windows can also be queried, with a marginal decrease in performance.

GetNearestValue(UUID, Time, Version, Direction) $\rightarrow$ (Version, (Time, Value)) locates the nearest point to a given time, either forwards or backwards. It is commonly used to obtain the 'current', or most recent to now, value of a stream of interest.

In practice, raw data streams feed into a graph of distillation processes in order to clean and filter the raw data and then combine the refined streams to produce useful data products, as illustrated in Figure 2. These distillers fire repeatedly, grab new data and compute output segments. In the presence of out of order arrival and loss, without support from the storage engine, it can be complex and costly to determine which input ranges have changed and which output extents need to be computed, or recomputed, to maintain consistency throughout the distillation pipeline.

To support this, **ComputeDiff(UUID, FromVersion, ToVersion, Resolution) $\rightarrow$ [(StartTime, EndTime)]** provides the time ranges that contain differences between the given versions. The size of the changeset returned can be limited by limiting the number of versions between **FromVersion** and **ToVersion** as each version has a maximum size. Each returned time range will be larger than $2^{resolution}$ nanoseconds, allowing the caller to optimize for batch size.

As utilities, **DeleteRange(UUID, StartTime, EndTime)**: create a new version of the stream with the given range deleted and **Flush(UUID)** ensure the given stream is flushed to replicated storage.

4 Time partitioned tree

To provide the abstraction described above, we use a *time-partitioning copy-on-write version-annotated k-ary tree*. As the primitives API provides queries based on time extents, the use of a tree that partitions time serves the role of an index by allowing rapid location of specific points in time. The base data points are stored in the leaves of the tree, and the depth of the tree is defined by the interval between data points. A uniformly sampled telemetry stream will have a fixed tree depth irrespective

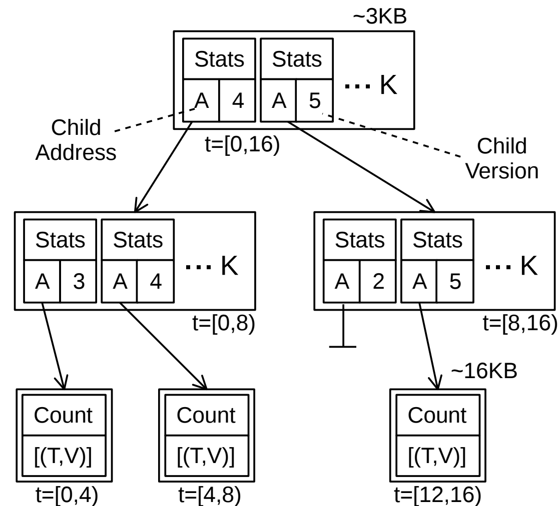


Figure 3: An example of a time-partitioning tree with version-annotated edges. Node sizes correspond to a K=64 implementation

of how much data is in the tree. All trees logically represent a huge range of time (from $-2^{60}ns$ to $3*2^{60}ns$ as measured from the Unix epoch, or approximately 1933 to 2079 with nanosecond precision) with big holes at the front and back ends and smaller holes between each of the points. Figure 3 illustrates a time-partitioning tree for 16 ns. Note the hole between 8 and 12 ns.

To retain historic data, the tree is *copy on write*: each insert into the tree forms an overlay on the previous tree accessible via a new root node. Providing historic data queries in this way ensures that all versions of the tree require equal effort to query – unlike log replay mechanisms which introduce overheads proportional to how much data has changed or how old is the version that is being queried. Using the storage structure as the index ensures that queries to any version of the stream have an index to use, and reduces network round trips.

Each link in the tree is annotated with the version of the tree that introduced that link, also shown in Figure 3. A null child pointer with a nonzero version annotation implies the version is a deletion. The time extents that were modified between two versions of the tree can be walked by loading the tree corresponding to the later version, and descending into all nodes annotated with the start version or higher. The tree need only be walked to the depth of the desired difference resolution, thus *ComputeDiff()* returns its results without reading the raw data. This mechanism allows consumers of a stream to query and process new data, regardless of where the changes were made, without a full scan and with only 8 bytes of state maintenance required - the 'last version processed'.

Each internal node holds scalar summaries of the subtrees below it, along with the links to the subtrees. Sta-

tistical aggregates are computed as nodes are updated, following the modification or insertion of a leaf node. The statistics currently supported are *min*, *mean*, *max* and *count*, but any operation that uses intermediate results from the child subtrees without requiring iteration over the raw data can be used. Any associative operation meets this requirement.

This approach has several advantages over conventional discrete rollups. The summary calculation is free in terms of IO operations – the most expensive part of a distributed storage system. All data for the calculation is already in memory, and the internal node needs to be copied anyway, since it contains new child addresses. Summaries do increase the size of internal nodes, but even so, internal nodes are a tiny fraction of the total footprint ($< 0.3\%$ for a single version of a $K = 64$ k-ary tree). Observable statistics are guaranteed to be consistent with the underlying data, because failure during their calculation would prevent the root node from being written and the entire overlay would be unreachable.

When querying a stream for statistical records, the tree need only be traversed to the depth corresponding to the desired resolution, thus the response time is proportional to the number of returned records describing the temporal extent, not the length of the extent nor the number of datapoints within it. Records from disparate streams are aligned in time, so time-correlated anaylsis can proceed directly. For queries requiring specific non-power-of-two windows, the operation is still dramatically accelerated by using the precomputed statistics to fill in the middle of each window, only requiring a "drill down" on the side of each window, so that the effort to generate a window is again proportional to the log of the length of time it covers, not linear in the underlying data.

Although conceptually a binary tree, an implementation may trade increased query-time computation for decreased storage and IO operations by using a k-ary tree and performing just-in time computation of the statistical metrics for windows that lie between actual levels of the tree. If k is too large, however, the on-the-fly computation impacts increases the variability of statistical query latencies, as discussed in Section 6.3.

To allow fetching nodes from the tree in a single IO operation, all addresses used in the tree are "native" in that they are directly resolvable by the storage layer without needing a translation step. If an indirect address were used it would require either a costly remote lookup in a central map, or complex machinery to synchronize a locally stored map. Multiple servers can execute reads on the same stream at a time, so all servers require an up-to-date view of this mapping. Native addresses remove this problem entirely, but they require care to maintain, as discussed below.

The internal blocks have a base size of $2 \times 8 \times K$ for the child addresses and child pointer versions. On top of that, the statistics require $4 \times 8 \times K$ for *min*, *mean*, *max* and *count* making them 3KB in size for $K = 64$. The leaf nodes require 16 bytes per (time, value) pair, and a 16 byte length value. For $N_{leaf} = 1024$ they are 16KB big. Both of these blocks are compressed, as discussed below.

5 System design

The overall system design of BTrDB, shown in Figure 4, is integrally tied to the multi-resolution COW tree data structure described above, but also represents a family of trade-offs between complexity, performance and reliability. This design prioritizes simplicity first, performance second and then reliability, although it does all three extremely well. The ordering is the natural evolution of developing a database that may require frequent changes to match a dynamically changing problem domain and workload (simplicity leads to an easily modifiable design). Performance requirements originate from the unavoidable demands placed by the devices we are deploying and, as this system is used in production, reliability needs to be as high as possible, without sacrificing the other two goals.

The design consists of several modules: request handling, transaction coalescence, COW tree construction and merge, generation link, block processing, and block storage. The system follows the SEDA [34] paradigm with processing occuring in three resource control stages – request, write and storage – with queues capable of exerting backpressure decoupling them.

5.1 Request processing stage

At the front end, flows of insertion and query requests are received over multiple sockets, either binary or HTTP. Each stream is identified by UUID. Operations on many streams may arrive on a single socket and those for a particular stream may be distributed over multiple sockets. Inserts are collections of time-value pairs, but need not be in order.

Insert and query paths are essentially separate. Read requests are comparatively lightweight and are handled in a thread of the session manager. These construct and traverse a partial view of the COW tree, as described above, requesting blocks from the block store. The block store in turn requests blocks from a reliable storage provider (Ceph in our implementation) and a cache of recently used blocks. Read throttling is achieved by the storage stage limiting how many storage handles are given to the session thread to load blocks. Requests hitting the cache are only throttled by the socket output.

On the insert path, incoming data is demultiplexed into per-stream coalescence buffers by UUID. Session man-

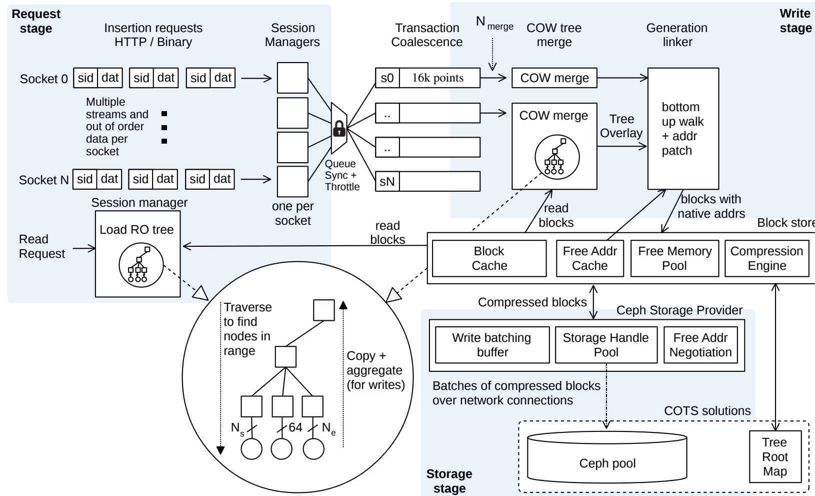


Figure 4: An overview of BTrDB showing the three SEDA-style stages, and composing modules

agers compete for a shortly held map lock and then grab a lock on the desired buffer. This sync point provides an important write-throttling mechanism, as discussed in Section 7.3. Each stream is buffered until either a certain time interval elapses or a certain number of points arrive, which triggers a commit by the write stage. These parameters can be adjusted according to target workload and platform, with the obvious trade-offs in stream update delay, number of streams, memory pressure, and ratio of tree and block overhead per version commit. These buffers need not be very big; we see excellent storage utilization in production where the buffers are configured for a maximum commit of 5 seconds or 16k points and the average commit size is 14400 points.

5.2 COW merge

A set of threads in the write stage pick up buffers awaiting commit and build a writable tree. This process is similar to the tree build done by a read request, except that all traversed nodes are modified as part of the merge, so must remain in memory. Copying existing nodes or creating new nodes requires a chunk of memory which is obtained from a **free pool** in the block store. At this point the newly created blocks have temporary addresses – these will be resolved by the linker later to obtain the index-free native addressing.

5.3 Block store

The block store allocates empty blocks, stores new blocks and fetches stored blocks. It also provides compression/decompression of storage blocks, and a cache. Empty blocks, used in tree merges, are satisfied primarily from the free pool to avoid allocations. After blocks are evicted from the block cache and are about to be garbage collected, they are inserted back into this pool.

Fields such as a block's address, UUID, resolution (tree depth) and time extent are useful for traversing the tree, but can be deduced from context when a block is read from disk, so are stripped before the block enters the compression engine.

The **block cache** holds all the blocks that pass through the block store with the least recently used blocks evicted first. It consumes a significant (tunable) portion of the memory footprint. Cache for a time series store may not seem an obvious win, other than for internal nodes in the COW tree, but it is extremely important for near-real-time analytics. As the majority of our read workload consists of processes waiting to consume any changes to a set of streams – data that just passed through the system – a cache of recently used blocks dramatically improves performance.

5.4 Compression engine

Part of the block store, the **compression engine** compresses the min, mean, max, count, address and version fields in internal nodes, as well as the time and value fields in leaf nodes. It uses a method we call *delta-delta* coding followed by Huffman coding using a fixed tree. Typical delta coding works by calculating the difference between every value in the sequence and storing that using variable-length symbols (as the delta is normally smaller than the absolute values [18]). Unfortunately with high-precision sensor data, this process does not work well because nanosecond timestamps produce very large deltas, and even linearly-changing values produce sequences of large, but similar, delta values.

In lower precision streams, long streams of identi-

cal deltas are typically removed with run-length encoding that removes sequences of identical deltas. Unfortunately noise in the lower bits of high precision sensor values prevents run-length encoding from successfully compacting the sequence of deltas. This noise, however, only adds a small jitter to the delta values. They are otherwise very similar. Delta-delta compression replaces run-length encoding and encodes each delta as the difference from the mean of a window of previous delta values. The result is a sequence of only the jitter values. Incidentally this works well for the addresses and version numbers, as they too are linearly increasing values with some jitter in the deltas. In the course of system development, we found that this algorithm produces better results, with a simpler implementation, than the residual coding in FLAC [13] which was the initial inspiration.

This method is lossless only if used with integers. To overcome this, the double floating point values are broken up into mantissa and exponent and delta-delta compressed as independent streams. As the exponent field rarely changes, it is elided if the delta-delta value is zero.

While a quantitative and comparative analysis of this compression algorithm is beyond the scope of this paper, its efficacy is shown in Section 6.

5.5 Generation linker

The generation linker receives a new tree overlay from the COW merge process, sorts the new tree nodes from deepest to shallowest and sends them to the block store individually, while resolving the temporary addresses to addresses native to the underlying storage provider. As nodes reference only nodes deeper than themselves, which have been written already, any temporary address encountered can be immediately resolved to a native address.

This stage is required because most efficient storage providers – such as append-only logs – can only write an object of arbitrary size to certain addresses. In the case of a simple file, arbitrarily sized objects can only be written to the tail, otherwise they overwrite existing data. Once the size of the object is known, such as after the linker sends the object to the block store and it is compressed, a new address can be derived from the previous one. The nature of the storage may limit how many addresses can be derived from a given initial address. For example, if the maximum file size is reached and a new file needs to be used.

For some storage providers, obtaining the first address is an expensive operation e.g. in a cluster operation, this could involve obtaining a distributed lock to ensure uniqueness of the generated addresses. For this reason the block store maintains a pool of pre-created initial addresses.

5.6 Root map

The *root map* is used before tree construction in both reads and writes. It resolves a UUID and a version to a storage "address." When the blocks for a new version have been acknowledged as durably persisted by the storage provider, a new mapping for the version is inserted into this root map. It is important that the map is fault tolerant as it represents a single point of failure. Without this mapping, no streams can be accessed. If the latest version entry for a stream is removed from the map, it is logically equivalent to rolling back the commit. Incidentally, as the storage costs of a small number of orphaned versions are low, this behaviour can be used deliberately to obtain cheap single-stream transaction semantics without requiring support code in the database.

The demands placed by these inserts / requests are much lower than those placed by the actual data, so many off-the-shelf solutions can provide this component of the design. We use MongoDB as it has easy-to-use replication.

One side effect of the choice of an external provider is that the latency in resolving this first lookup is present in all queries – even ones that hit the cache for the rest of the query. Due to the small size of this map, it would be reasonable to replicate the map on all BTrDB nodes and use a simpler storage solution to reduce this latency. All the records are the same size, and the version numbers increment sequentially, so a flat file indexed by offset would be acceptable.

5.7 Storage provider

The storage provider component wraps an underlying durable storage system and adds write batching, prefetching, and a pool of connection handles. In BTrDB, a tree commit can be done as a single write, as long as addresses can be generated for all the nodes in the commit without performing intermediate communication with the underlying storage. Throttling to the underlying storage is implemented here, for reasons described in Section 7.3.

As the performance of a storage system generally decreases with the richness of its features, BTrDB is designed to require only three very simple properties from the underlying storage:

1. It must be able to provide one or more free "addresses" that an arbitrarily large object can be written to later. Only a small finite number of these addresses need be outstanding at a time.

2. Clients must be able to derive another free "address" from the original address, and the size of the object that was written to it.

3. Clients must be able to read back data given just the "address" and a length.

Additional properties may be required based on the desired characteristics of the BTrDB deployment as a whole, for example distributed operation and durable writes. Sans these additional requirements, even a simple file is sufficient as a storage provider: (1) is the current size of the file, (2) is addition and (3) is a random read.

Note that as we are appending and reading with a file-like API, almost every distributed file system automatically qualifies as acceptable, such as HDFS, GlusterFS [25], CephFS [32], MapR-FS, etc.

Note also that if arbitrary but unique "addresses" are made up, then any database offering a key-value API would also work, e.g. , Cassandra, MongoDB, RADOS [33] (the object store under CephFS), HBase or BigTable. Most of these offer capabilities far beyond what is required by BTrDB, however, usually at a performance or space cost.

Although we support file-backed storage, we use Ceph RADOS in production. Initial addresses are read from a monotonically increasing integer stored in a RADOS object. Servers add a large increment to this integer while holding a distributed lock (provided by Ceph). The server then has a range of numbers it knows are unique. The high bits are used as a 16MB RADOS object identifier, while the low bits are used as an offset within that object. The address pool in the block store decouples the latency of this operation from write operations.

6 Quasi-production implementation

An implementation of BTrDB has been constructed using Go [11]. This language was chosen as it offers primitives that allow for rapid development of highly SMP-scalable programs in a SEDA [34] paradigm – namely channels and goroutines. As discussed above, one of the primary tenets of BTrDB is performance through simplicity: the entire implementation sans test code and auto-generated libraries is only 4709 lines.

Various versions of BTrDB have been used in a year-long deployment to capture data from roughly 35 microsynchrophasors deployed in the field, comprising 12 streams of 120 Hz data each. Data from these devices streams in over LTE and wired connections, as shown in Figure 2, leading to unpredictable delays, out-of-order chunk delivery and many duplicates (when the GPS-derived time synchronizes to different satellites). Many of the features present in BTrDB were developed to support the storage and analysis of this sensor data.

The hardware configuration for this deployment is shown in Figure 5. The compute server runs BTrDB (in a single node configuration). It also runs the DISTIL ana-

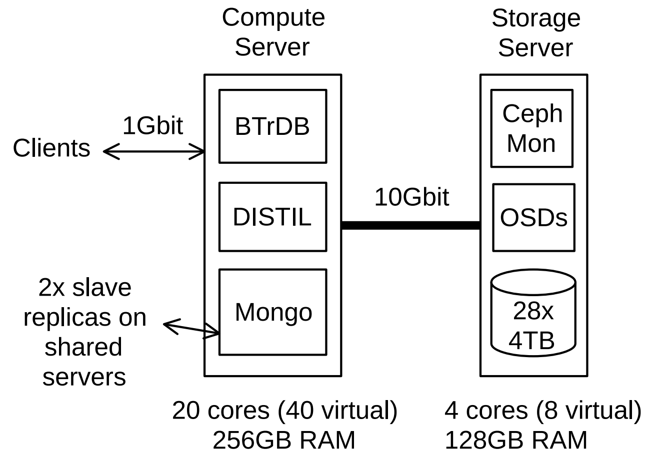


Figure 5: The architecture of our production system

lytics framework [1] and a MongoDB replica set master. The MongoDB database is used for the root map along with sundry metadata, such as engineering units for the streams and configuration parameters for DISTIL algorithms. The storage server is a single-socket server containing 28 commodity 4TB 5900 RPM spinning-metal drives. The IO capacity of this server may seem abnormally low for a high performance database, but it is typical for data warehousing applications. BTrDB's IO pattern was chosen with this type of server in mind: 1MB reads and writes with excellent data locality for the primary analytics workload.

6.1 Golang – the embodiment of SEDA

SEDA advocates constructing reliable, high-performance systems via decomposition into independent stages separated by queues with admission control. Although not explicitly referencing this paradigm, Go encourages the partitioning of complex systems into logical units of concurrency, connected by *channels*, a Go primitive roughly equal to a FIFO with atomic enqueue and dequeue operations. In addition, Goroutines – an extremely lightweight thread-like primitive with userland scheduling – allow for components of the system to be allocated pools of goroutines to handle events on the channels connecting a system in much the same way that SEDA advocates event dispatch. Unlike SEDA's Java implementation, however, Go is actively maintained, runs at near native speeds and can elegantly manipulate binary data.

6.2 Read throttling

As discussed below, carefully applied backpressure is necessary to obtain good write performance. In contrast, we have not yet found the need to explicitly throttle

reads, despite having a higher read load than write load. The number of blocks that are kept in memory to satisfy a read is fewer than for a write. If the nodes are not already in the block cache (of which 95% are), they are needed only while their subtree is traversed, and can be freed afterwards. This differs from a write, where all the traversed blocks will be copied and must therefore be kept in memory until the linker has patched them and written them to the storage provider. In addition, Go channels are used to stream query data directly to the socket as it is read. If the socket is too slow, the channel applies back pressure to the tree traversal so that nodes are not fetched until the data has somewhere to go. For this reason, even large queries do not place heavy memory pressure on the system.

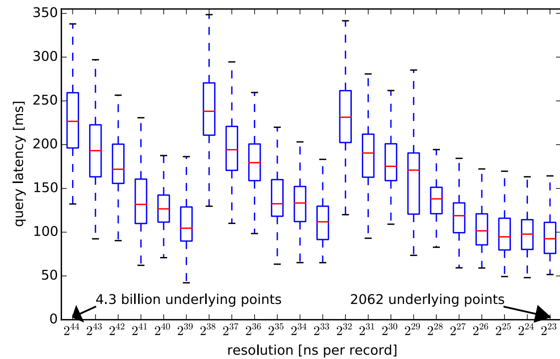


Figure 6: Query latencies for 2048 statistical records covering a varying time extent (1 year to 5 seconds), queried from a single node

6.3 Real-data quantitative evaluation

Although the version of BTrDB running on production is lacking the performance optimizations implemented on the version evaluated in Section 7, it can provide insight into the behavior of the database with large, real data sets. At the time of writing, we have accumulated more than 2.1 trillion data points over 823 streams, of which 500 billion are spread over 506 streams feeding from instruments deployed in the field. The remaining 1.6 trillion points were produced by the DISTIL analysis framework. Of this analysis data, roughly 1.1 trillion points are in extents that were invalidated due to algorithm changes, manual flagging or replaced data in input streams. This massive dataset allows us to assess several aspects of the design.

Compression: A concern with using a copy-on-write tree data structure with "heavyweight" internal nodes is that the storage overheads may be unacceptable. With real data, the compression more than compensates for this overhead. The total size of the instrument data in the production Ceph pool (not including replication) is 2.757 TB. Dividing this by the number of raw data points equates to 5.514 bytes per reading *including all statistical and historical overheads*. As the raw tuples are 16 bytes, we have a compression ratio of 2.9x despite the costs of the time-partitioning tree. Compression is highly data dependent, but this ratio is better than the results of in-depth parametric studies of compression on similar synchrophasor telemetry [16][30].

Statistical queries: As these queries come into play with larger data sets, they are best evaluated on months of real data, rather than the controlled study in Section 7. These queries are typically used in event detectors to locate areas of interest – the raw data is too big to navigate with ease – and for visualization. To emulate this workload, we query a year's worth of voltage data – the same data illustrated in Figure 1a – to locate a voltage sag (the dashed box) and then issue progressively finer-grained queries until we are querying a 5 second window (Figure 1b). Automated event detectors typically skip several levels of resolution between queries, but this pattern is typical of data exploration where a user is zooming in to the event interactively. This process is repeated 300 times, with pauses between each sequence to obtain distributions on the query response times. The results can be found in Figure 6. Typically these distributions would be tighter, but the production server is under heavy load.

Each query is for the same number of statistical records (2048), but the number of data points that these records represent grows exponentially as the resolution becomes coarser (right to left in Figure 6). In a typical on-the-fly rollup database, the query time would grow exponentially as well, but with BTrDB it remains roughly constant within a factor of three. The implementation's choice of K ($64 = 2^6$) is very visible in the query response times. The query can be satisfied directly from the internal nodes with no on-the-fly computation every 6 levels of resolution. In between these levels, BTrDB must perform a degree of aggregation – visible in the query latency – to return the statistical summaries, with the most work occurring just before the next tier of the tree (2^{44}, 2^{38}, 2^{32}). Below 2^{27} the data density is low enough ($<$ 16 points per pixel column) that the query is being satisfied from the leaves.

Cache hit ratios: Although cache behavior is workload dependent, our mostly-automated-analysis is likely representative of most use-cases. Over 22 days, the block cache has exhibited a 95.93% hit rate, and the Ceph read-behind prefetch cache exhibited a 95.22% hit rate.

6.4 Analysis pipeline

The raw data acquired from sensors in the field is eventually used for decision support; grid state estimation; is-

land detection and reverse power flow detection, to name a few examples. To obtain useful information the data must first go through a pipeline consisting of multiple transformation, fusion and synthesis stages, as shown in Figure 2.

All the stages of the pipeline are implemented with the same analysis framework and all consist of the same sequence of operations: find changes in the inputs, compute which ranges of the outputs need to be updated, fetch the data required for the computation, compute the outputs, and insert them. Finally, if this process completes successfully, the version numbers of the inputs that the distiller has now "caught up to" are written to durable storage (the same MongoDB replica set used for the root map). This architecture allows for fault tolerance without mechanisms, as each computation is idempotent: the output range corresponding to a given input range is deleted and replaced for each run of a DISTIL stage. If any error occurs, simply rerun the stage until it completes successfully, before updating the "input $\rightarrow$ last version" metadata records for the input streams.

This illustrates the power of the BTrDB *CalculateDiff()* primitive: an analysis stream can be "shelved," i.e., not kept up to date, and when it becomes necessary later it can be brought up-to date just-in-time with guaranteed consistency, even if the changes to the dependencies have occurred at random times throughout the stream. Furthermore the consumer obtains this with just 8 bytes of state per stream. The mechanism allows changes in a stream to propagate to all streams dependent on it, even if the process materializing the dependent stream is not online or known to the process making the change upstream. Achieving this level of consistency guarantee in existing systems typically requires a journal of outstanding operations that must be replayed on downstream consumers when they reappear.

7 Scalability Evaluation

To evaluate the design principles and implementation of BTrDB in a reproducible manner, we use a configuration of seven Amazon EC2 instances. There are four primary servers, one metadata server and two load generators. These machines are all c4.8xlarge instances. These were chosen as they are the only available instance type with both 10GbE network capabilities and EBS optimization. This combination allows the scalability of BTrDB to be established in the multiple-node configuration where network and disk bandwidth are the limiting factors.

Metric	Mean	Std. dev.
Write bandwidth [MB/s]	833	151
Write latency [ms]	34.3	40.0
Read bandwidth [MB/s]	1174	3.8
Read latency [ms]	22.0	18.7

Table 2: The underlying Ceph pool performance at max bandwidth

Ceph version 0.94.3 was used to provide the storage pool over 16 Object Store Daemons (OSDs). It was configured with a size (replication factor) of two. The bandwidth characteristics of the pool are shown in Table 2. It is important to note the latency of operations to Ceph, as this establishes a lower bound on cold query latencies, and interacts with the transaction coalescence backpressure mechanism. The disk bandwidth on a given BTrDB node to one of the OSD volumes measured using `dd` was approximately 175MB/s. This matched the performance of the OSD reported by `ceph tell osd.N bench`.

To keep these characteristics roughly constant, the number of Ceph nodes is kept at four, irrespective of how many of the servers are running BTrDB for a given experiment, although the bandwidth and latency of the pool does vary over time. As the Ceph CRUSH data placement rules are orthogonal to the BTrDB placement rules, the probability of a RADOS request hitting a local OSD is 0.25 for all experiments.

7.1 Throughput

The throughput of BTrDB in raw record tuples per second is measured for inserts, cold queries (after flushing the BTrDB cache) and warm queries (with a preheated BTrDB cache). Each tuple is an 8 byte time stamp and an 8 byte value. Warm and cold cache performance is characterized independently, because it allows an estimation of performance under different workloads after estimating the cache hit ratio.

#BTrDB	Streams	Total points	#Conn	Insert [mil/s]	Cold Query [mil/s]	Warm Query [mil/s]
1	50	500 mil	30	16.77	9.79	33.54
2	100	1000 mil	60	28.13	17.23	61.44
3	150	1500 mil	90	36.68	22.05	78.47
4	200	2000 mil	120	53.35	33.67	119.87

Table 1: Throughput evaluation as number of servers and size of load increases

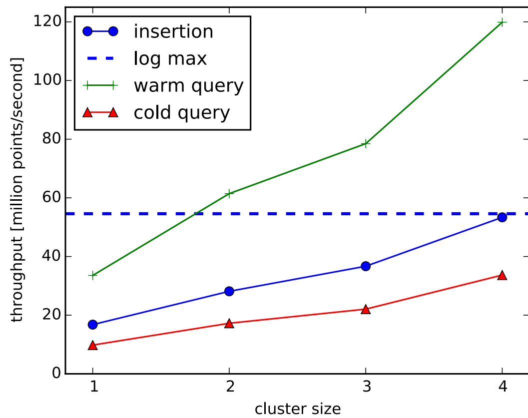


Figure 7: Throughput as the number of BTrDB nodes increases. The horizontal dashed line indicates the independently benchmarked write bandwidth of the underlying storage system.

	When insertion was	
Throughput [million pt/s] for	Chrono.	Random
Insert	28.12	27.73
Cold query in chrono. order	31.41	31.67
Cold query in same order	-	32.61
Cold query in random order	29.67	28.26
Warm query in chrono. order	114.1	116.2
Warm query in same order	-	119.0
Warm query in random order	113.7	117.2

Table 3: The influence of query/insert order on throughput

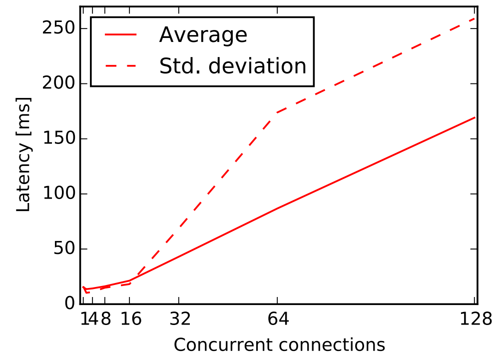
Inserts and queries are done in 10 kilorecord chunks, although there is no significant change in performance if this is decreased to 2 kilorecords.

Figure 7 shows that insert throughput scales linearly, to approximately 53 million records per second with four nodes. The horizontal dashed line is calculated as the maximum measured pool bandwidth (823MB/s) divided by the raw record size (16 bytes). This is the bandwidth that could be achieved by simply appending the records to a log in Ceph without any processing. This shows that despite the functionality that BTrDB offers, and the additional statistical values that must be stored, BTrDB performs on par with an ideal data logger.

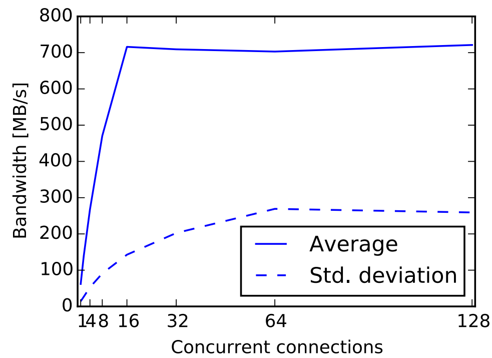
The warm query throughput of 119 million readings per second is typical for tailing-analytics workloads where distillers process recently changed data. This throughput equates to roughly 1815 MB/s of network traffic, or 907MB/s per load generator.

7.2 Data and operation ordering

BTrDB allows data to be inserted in arbitrary order, and queried in arbitrary order. To characterize the effect of insertion and query order on throughput, measurements with randomized operations were performed. The workload consists of two hundred thousand inserts/queries of 10k points each (2 billion points in total). Two datasets were constructed, one where the data was inserted chronologically and one where the data was inserted randomly. After this, the performance of cold and warm queries in chronological order and random order were tested on both datasets. For the case of random insert, queries in the same (non-chronological) order as the insert were also tested. Note that operations were randomized at the granularity of the requests; within each request the 10k points were still in order. The results are presented in Table 3. The differences in throughput are well within experimental noise and are largely insignificant. This out-of-order performance is an important result for a database offering insertion speeds near that of an in-order append-only log.



(a) Latency

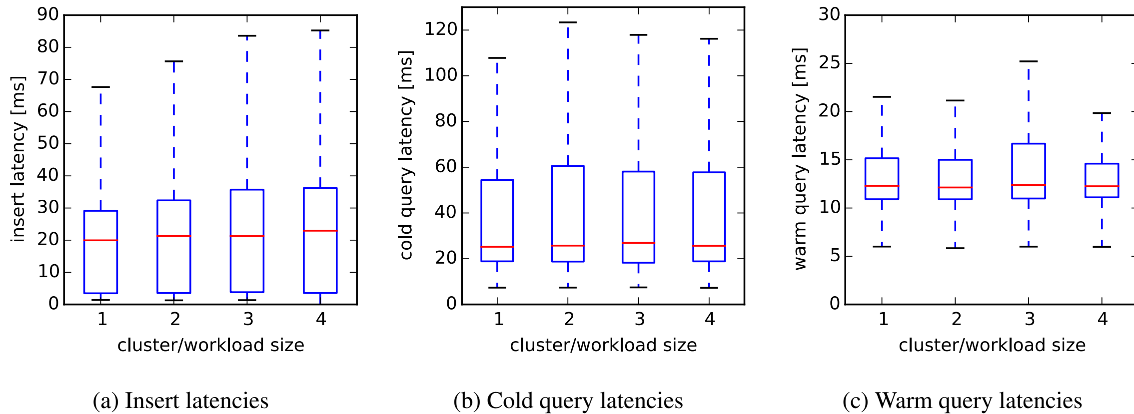


(b) Aggregate bandwidth

Figure 9: Ceph pool performance characteristics as the number of concurrent connections increases

7.3 Latency

Although BTrDB is designed to trade a small increase in latency for a large increase in throughput, latency is still an important metric for evaluation of performance under load. The load generators record the time taken for each



(a) Insert latencies (b) Cold query latencies (c) Warm query latencies

Figure 8: Operation latencies as server count and workload is increased linearly

insert or query operation. Figure 8 gives an overview of the latency of operations as the workload and number of servers grows. Ideally all four points would be equal indicating perfect scaling. The range of latencies seen for insert operations increases as the cluster approaches the maximum bandwidth of Ceph. This is entirely Ceph's latency being presented to the client as backpressure. When a transaction coalescence buffer is full or being committed, no data destined for that stream is admitted to the database. Furthermore, a fixed number of tree merges are allowed at a time, so some buffers may remain full for some time. Although this appears counter-intuitive, in fact it increases system performance. Applying this backpressure early prevents Ceph from reaching pathological latencies. Consider Figure 9a where it is apparent that not only does the Ceph operation latency increase with the number of concurrent write operations, but it develops a long fat tail, with the standard deviation exceeding the mean. Furthermore, this latency buys nothing, as Figure 9b shows that the aggregate bandwidth plateaus after the number of concurrent operations reaches 16 – the number of OSDs.

With Ceph's latency characteristics in mind, BTrDB's write latency under maximum load is remarkable. A four node cluster inserting more than 53 million points per second exhibits a third quartile latency of 35ms: less than one standard deviation above the raw pool's latency.

7.4 Limitations and future work

The tests on EC2 show that the throughput and latency characteristics of the system are defined primarily by the underlying storage system. This is the ideal place to be, as it renders most further optimization in the timeseries database tier irrelevant.

The exception to this is optimizations that reduce the number of IO operations. We have already optimized the write path to the point of one write operation per commit. Nevertheless, there are significant performance gains to be had by optimizing the read path. One such avenue is to improve the block cache policy, reducing read ops. At present, the cache evicts the least recently used blocks. More complex policies could yield improved cache utilization: for example, if clients query only the most recent version of a stream, then all originals of blocks that were copied during a tree merge operation could be evicted from the cache. If most clients are executing statistical queries, then leaf nodes (which are 5x bigger than internal nodes) can be prioritized for eviction. Furthermore, as blocks are immutable, a distributed cache would not be difficult to implement as no coherency algorithm is required. Querying from memory on a peer BTrDB server would be faster than hitting disk via Ceph.

8 Conclusion

BTrDB provides a novel set of primitives, especially fast difference computation and rapid, low-overhead statistical queries that enable analysis algorithms to locate subsecond transient events in data comprising billions of datapoints spanning months – all in a fraction of a second. These primitives are efficiently provided by a time-partitioning version-annotated copy-on-write tree, which is shown to be easily implementable. A Go implementation is shown to outperform existing timeseries databases, operating at 53 million inserted values per second, and 119 million queried values per second with a four node cluster. The principles underlying this database are potentially applicable to a wide range of telemetry timeseries, and with slight modification, are applicable to all timeseries for which statistical aggregate functions exist and which are indexed by time.

Acknowledgments

The authors would like to thank Amazon and the UC Berkeley AMPLab for providing computing resources. In addition, this research is sponsored in part by the U.S. Department of Energy ARPA-E program (DE-AR0000340), National Science Foundation CPS-1239552, and Fulbright Scholarship program.

References

[1] ANDERSEN, M. P., KUMAR, S., BROOKS, C., VON MEIER, A., AND CULLER, D. E. DISTIL: Design and Implementation of a Scalable Synchrophasor Data Processing System. *Smart Grid, IEEE Transactions on* (2015).

[2] APACHE SOFTWARE FOUNDATION. Apache Cassandra home page. `http://cassandra.apache.org/`, 9 2015.

[3] APACHE SOFTWARE FOUNDATION. Apache HBase home page. `http://hbase.apache.org/`, 9 2015.

[4] BUEVICH, M., WRIGHT, A., SARGENT, R., AND ROWE, A. Respawn: A distributed multi-resolution time-series datastore. In *Real-Time Systems Symposium (RTSS), 2013 IEEE 34th* (2013), IEEE, pp. 288–297.

[5] COUCHBASE. CouchBase home page. `http://www.couchbase.com/`, 9 2015.

[6] DATASTAX. DataStax home page. `http://www.datastax.com/`, 9 2015.

[7] DRUID. Druid.io. `http://druid.io/`, 9 2015.

[8] DUNNING, TED. MapR: High Performance Time Series Databases. `http://www.slideshare.net/NoSQLmatters/ted-dunning-very-high-bandwidth-time-series-database-implementation-nosql-matters-barcelona-2014`, 11 2014.

[9] ENDPOINT. Benchmarking Top NoSQL Databases. Tech. rep., Endpoint, apr 2015. `http://www.datastax.com/wp-content/themes/datastax-2014-08/files/NoSQL_Benchmarks_EndPoint.pdf`.

[10] GOLDSCHMIDT, T., JANSEN, A., KOZIOLEK, H., DOPPELHAMER, J., AND BREIVOLD, H. P. Scalability and robustness of time-series databases for cloud-native monitoring of industrial processes. In *Cloud Computing (CLOUD), 2014 IEEE 7th International Conference on* (2014), IEEE, pp. 602–609.

[11] GOOGLE. The Go Programming Language. `https://golang.org/`, 9 2015.

[12] INFLUXDB. InfluxDB home page. `https://influxdb.com/`, 9 2015.

[13] JOSH COALSON. Free Lossless Audio Codec. `https://xiph.org/flac/`, 9 2015.

[14] KAIROSDB. KairosDB import/export documentation. `https://kairosdb.github.io/kairosdocs/ImportExport.html`, 9 2014.

[15] KAIROSDB. KairosDB home page. `http://github.com/kairosdb/kairosdb`, 9 2015.

[16] KLUMP, R., AGARWAL, P., TATE, J. E., AND KHURANA, H. Lossless compression of synchronized phasor measurements. In *Power and Energy Society General Meeting, 2010 IEEE* (2010), IEEE, pp. 1–7.

[17] LAMBDA FOUNDRY. Pandas home page. `http://pandas.pydata.org/`, 9 2015.

[18] LELEWER, D. A., AND HIRSCHBERG, D. S. Data compression. *ACM Computing Surveys (CSUR) 19*, 3 (1987), 261–296.

[19] MONGODB INC. MongoDB home page. `https://www.mongodb.org/`, 9 2015.

[20] OPENTSDB. OpenTSDB home page. `http://opentsdb.net/`, 9 2015.

[21] ORACLE CORPORATION. MySQL home page. `https://www.mysql.com/`, 9 2015.

[22] PELKONEN, T., FRANKLIN, S., TELLER, J., CAVALLARO, P., HUANG, Q., MEZA, J., AND VEERARAGHAVAN, K. Gorilla: a fast, scalable, in-memory time series database. *Proceedings of the VLDB Endowment 8*, 12 (2015), 1816–1827.

[23] PROJECT VOLDEMORT. Project Voldemort home page. `http://www.project-voldemort.com/`, 9 2015.

[24] RABL, T., GÓMEZ-VILLAMOR, S., SADOGHI, M., MUNTÉS-MULERO, V., JACOBSEN, H.-A., AND MANKOVSKII, S. Solving big data challenges for enterprise application performance management. *Proceedings of the VLDB Endowment 5*, 12 (2012), 1724–1735.

[25] REDHAT. Gluster home page. `http://www.gluster.org/`, 9 2015.

[26] REDIS LABS. Redis home page. `http://redis.io/`, 9 2015.

[27] SCOTT, JIM. MapR-DB OpenTSDB bulk inserter code. https://github.com/mapr-demos/opentsdb/commit/c732f817498db317b8078fa5b53441a9ec0766ce, 9 2014.

[28] SCOTT, JIM. MapR: Loading a time series database at 100 million points per second. https://www.mapr.com/blog/loading-time-series-database-100-million-points-second, 9 2014.

[29] STONEBRAKER, M., AND WEISBERG, A. The voltdb main memory dbms. *IEEE Data Eng. Bull. 36*, 2 (2013), 21–27.

[30] TOP, P., AND BRENEMAN, J. Compressing Phasor Measurement data. In *North American Power Symposium (NAPS), 2013* (Sept 2013), pp. 1–4.

[31] VOLTDB INC. VoltDB home page. https://voltdb.com/, 9 2015.

[32] WEIL, S. A., BRANDT, S. A., MILLER, E. L., LONG, D. D., AND MALTZAHN, C. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th symposium on Operating systems design and implementation* (2006), USENIX Association, pp. 307–320.

[33] WEIL, S. A., LEUNG, A. W., BRANDT, S. A., AND MALTZAHN, C. Rados: a scalable, reliable storage service for petabyte-scale storage clusters. In *Proceedings of the 2nd international workshop on Petascale data storage: held in conjunction with Supercomputing'07* (2007), ACM, pp. 35–44.

[34] WELSH, M., CULLER, D., AND BREWER, E. Seda: an architecture for well-conditioned, scalable internet services. *ACM SIGOPS Operating Systems Review 35*, 5 (2001), 230–243.