BTrDB: Optimizing Storage System Design for Timeseries Processing

Authors: Michael P Anderson, David E Culler
Presented By: Radhir Kothuri

Key Techniques & Ideas

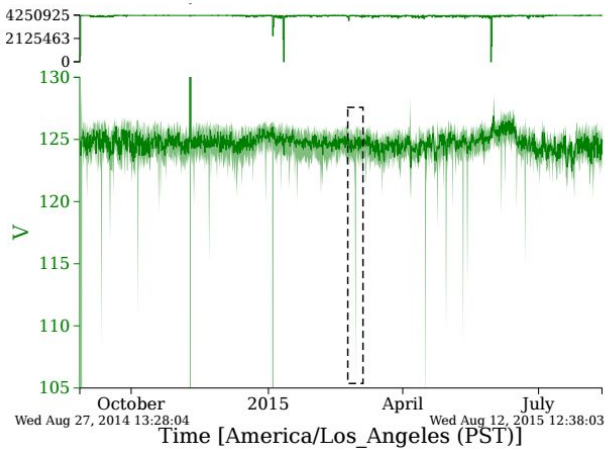
- Motivation & Background
- BTrDB Design & Data Structure
- System Design
- System Implementation
- Evaluation
- Conclusion

Motivation & Background

- Increase of IOT (Internet of Things) devices calls for more high-precision, high-sample rate telemetry timeseries DBs
- Existing timeseries DBs cannot cope with the demanding throughput required by these telemetry systems nor provide the analysis
  - Mainly useful for analyzing multi-dimensional data with much lower throughput and far less data points.
- BTrDB solves this niche case of providing a viable timeseries DB for IOT

Real-Time Telemetry Data

- Primary application of BTrDB is recording, distilling, and analyzing telemetry data.
- Paper focuses on uPMUs, precision power meters that are deployed in distribution tier of electrical grid
  - Produces 12 streams of 120 Hz high-precision values with up to 100 nanosecond accuracy
  - 1.4 million inserted points per second
  - Points can come out of order -> consistency must be guaranteed of both the raw data and the analytics



Existing Competition

- Only real competitors to BTrDB are OpenTSB and KairosDB
  - Throughput is only 403,500 inserts per second and 133k respectively
- Facebook's in memory Gorilla database has similar model but cannot handle sub-second timestamps and does not permit out-of-order insertion
- Even if some competitors can handle throughput, cannot satisfy statistical queries.

Intuitive Timeseries Abstraction

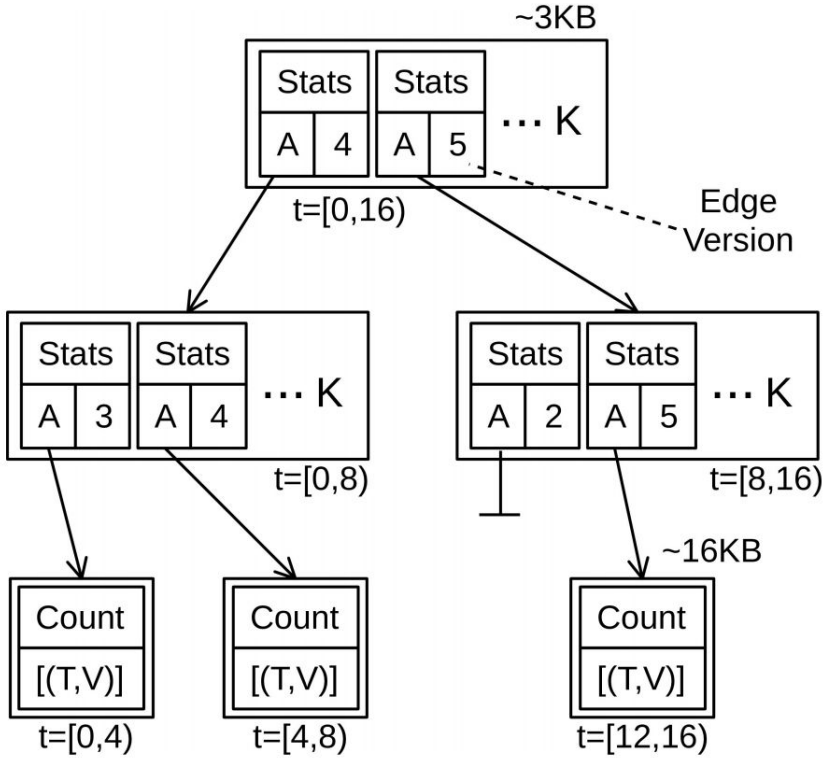
- Each stream is identified by a unique UUID
- QueryRange(uuid, start_time, end_time) -> Returns [(time,value)]
- InsertValues(uuid, [(time,value)])
- DeleteRange(uuid, start_time, end_time)
- However, hard to accomplish key characteristics of timeseries database
    - Analyse recently changed data
    - Perform computation idempotently
    - Computer dependent streams
    - Locate interesting events in large quantities of data

BTrDB Final Abstraction

- 6 Functions
  - InsertValues(UUID, [time,value])
  - GetRange(UUID, StartTime, EndTime, Version) -> (Version, [Time,Value])
    - Returns all values from latest version if version not specified
  - GetStatisticalRange(UUID, StartTime, EndTime, Version, Resolution) -> (Version, [(Min,Mean,Max,Count)]
  - GetNearestValue(UUID,Time,Version,Direction) -> (Version, (Time,Value))
  - ComputeDiff(UUID,FromVersion,ToVersion,Resolution) ->[(StartTime,EndTime)]
  - DeleteRange(UUID,StartTime,EndTime)
- Satisfies the key characteristics of a timeseries database

Time Partitioned Tree Data Structure

- Copy on write K-ary tree
- Copy on write (COW)
  - Each insert forms an overlay on previous tree.
  - Each query no matter how historic will take same time
- Leaf Nodes
  - Time, Value pairs
- Depth of tree defined by interval between data points
- Internal Nodes
  - Stats on the subtrees below it and links to subtrees
  - Any associative operation not requiring iteration over raw data can be used

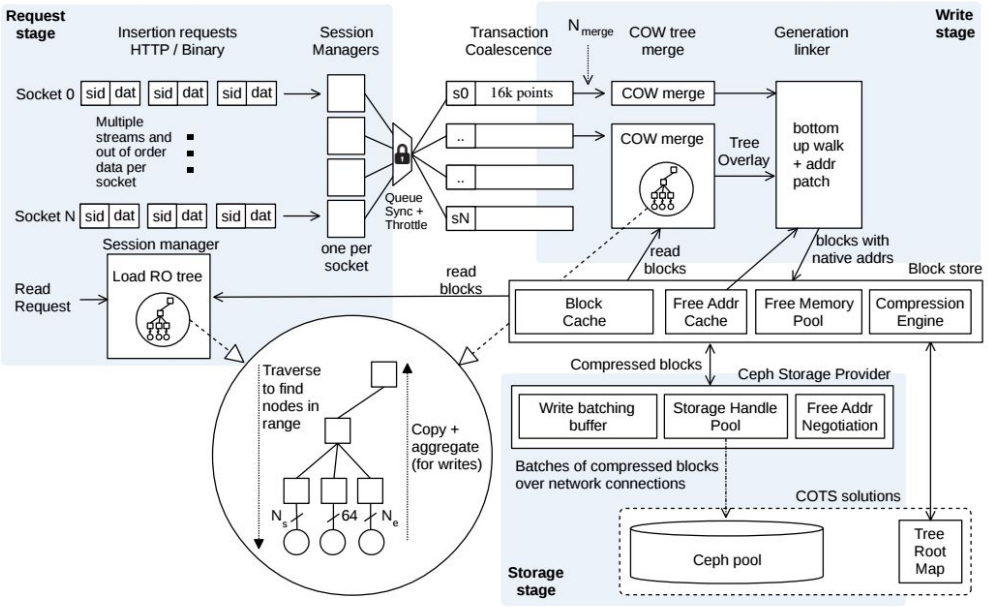


Time Partitioned Data Structure

- Use “native” addresses so an extra step for translation is not necessary
  - Directly resolvable by the storage layer
- Internal blocks have size 2x8xk for child addresses and child pointer versions. Statistics require an additional 4x8xk for min, max, mean, and count operations making them 3KB in size for k=64.
- By storing statistics, future reads save on IO operations since the stats are already in memory.

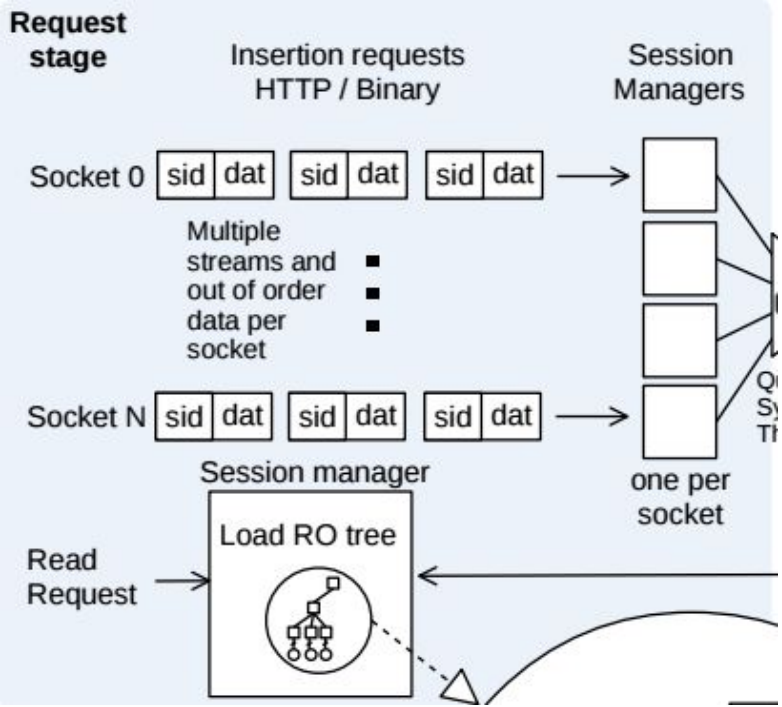
System Design

- Consists of several modules
  - Request handling
  - Transaction coalescence
  - COW Tree construction and merge
  - Generation Link
  - Block Processing and storage



Request Handling, COW Merge

- Handles binary and HTTP input
- For read inputs
  - Load the tree and read a partial view of the tree
- For write inputs
  - Demultiplexed into per stream coalescence buffers
  - Session managers grab a lock on the buffer
  - After a certain number of points that have arrived, a write is made to the tree
  - Average commit is around 14400 points
- COW Merge
  - Traverse tree and pick up free blocks from block store. Have temporary addresses which will be resolved by linker



System Design Continued

- Block Store
  - Allocates empty blocks, stores new blocks and fetches stored blocks
- Compression Engine
  - Uses delta delta encoding
- Generation Linker
  - Receives new tree from COW merge and replaces all temporary addresses with native ones
- Root Map
  - Used for tree construction. Resolves a UUID to a version and a storage “address.”
  - Maps UUID to storage block address
  - Single Point of Failure -> Needs to be fault tolerant

System Implementation

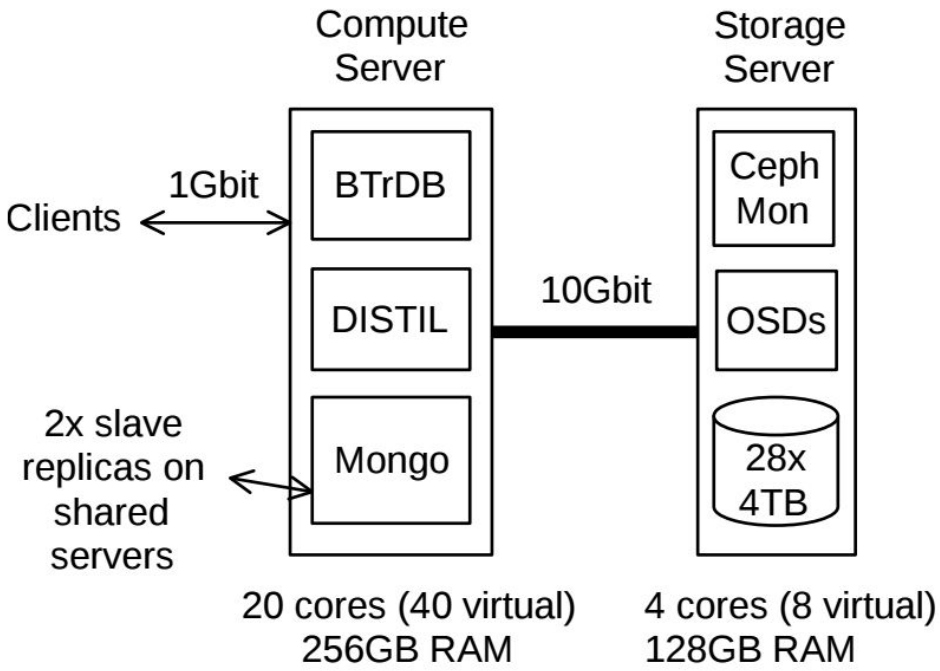


Figure 5: The architecture of our production system

4 nodes deployed in production

Scalability Evaluation

#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

Insertions, Cold Queries, and Warm queries are linear with respect to Streams

Throughput Evaluation

Cold queries in chrono. Order does worse in chrono. Shows that out of order performance does well

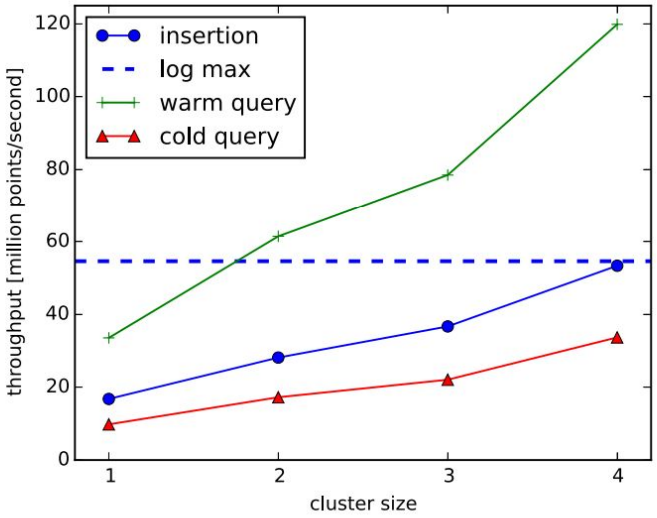
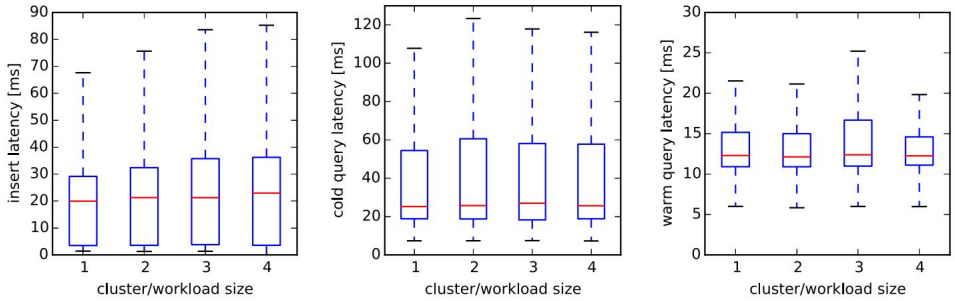


Figure 7: Throughput as the number of BTrDB nodes increases. The horizontal dashed line indicates the independently benchmarked write bandwidth of the underlying storage system.

	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

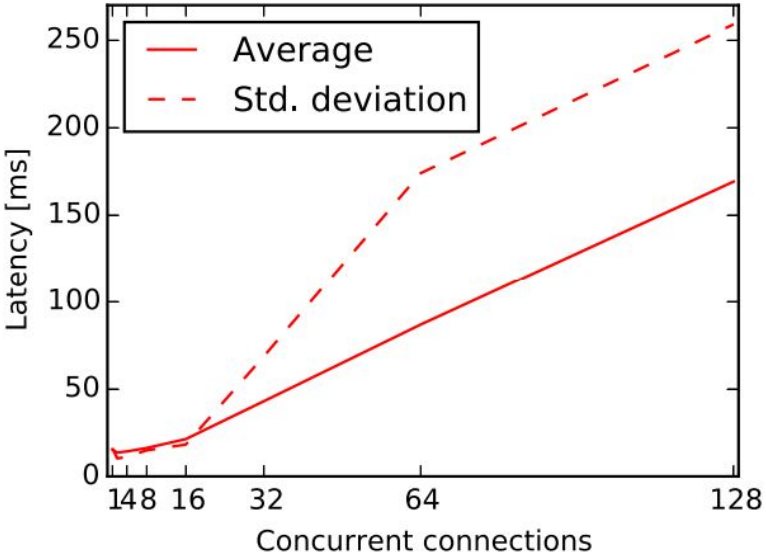
Table 3: The influence of query/insert order on throughput

Latency Evaluation



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

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



(a) Latency

Conclusion

- Provides a novel set of primitives
- Enable statistical queries on subsecond timeseries data
- Still provide high throughput and low latency in order to deliver the results given the high workloads

Discussion Points

- Cache is not optimized
  - Cache rate hit is 95%. This can still be optimized with tailored cache eviction policies
  - Policies based on versions. If querying for most recent stream, then evict all the ones from COW merge phase.
- Scalability of system
  - 4 nodes were used. Still handled upwards of millions of queries.
  - Competitors used 36 node cluster and still only achieved around ~450k max
  - One limitation will be the physical drives
- Working with other kinds of data
  - Would still work well. The system is scaled and optimized in order to handle millions of insertions and queries. Other data sources like stock market data could still be applicable.