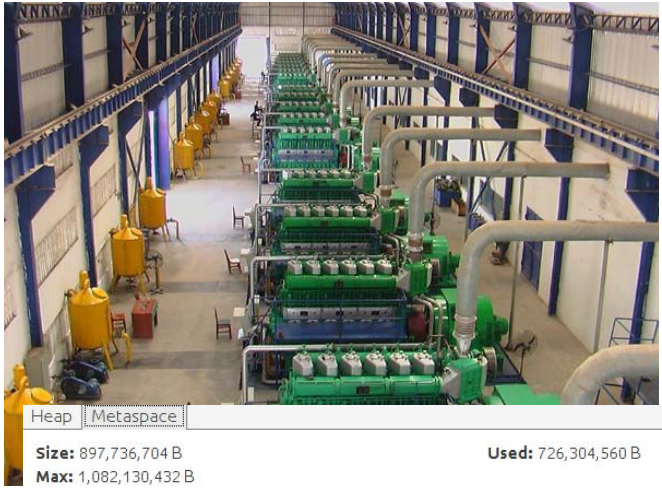
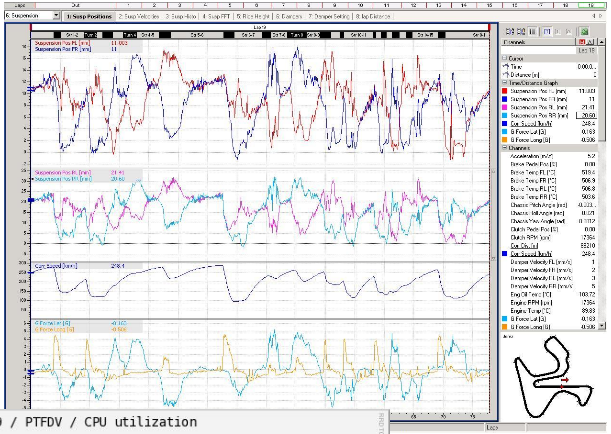
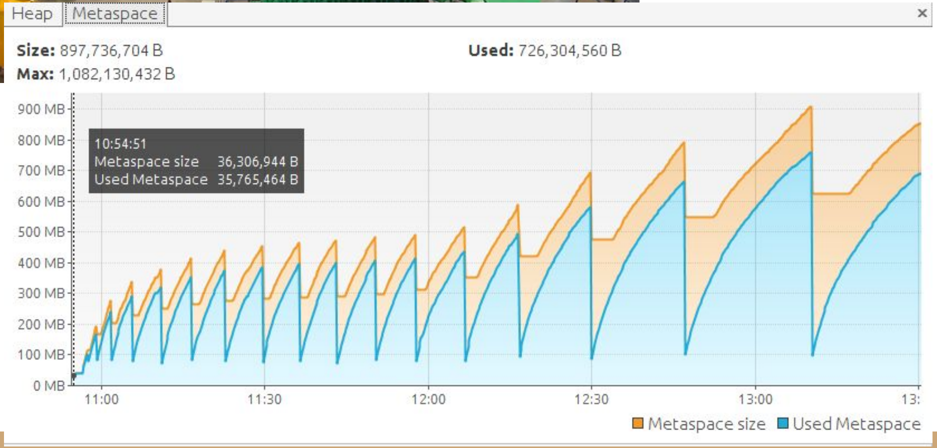
BTrDB: Optimizing Storage System Design for Timeseries Processing

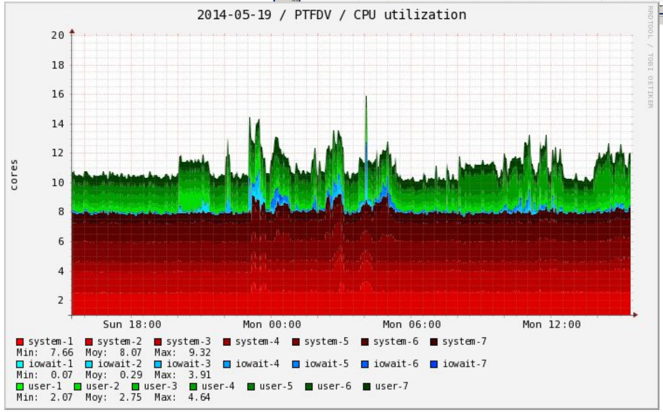
Michael P Andersen, David E. Culler
University of California, Berkeley

Fast scalar stream telemetry





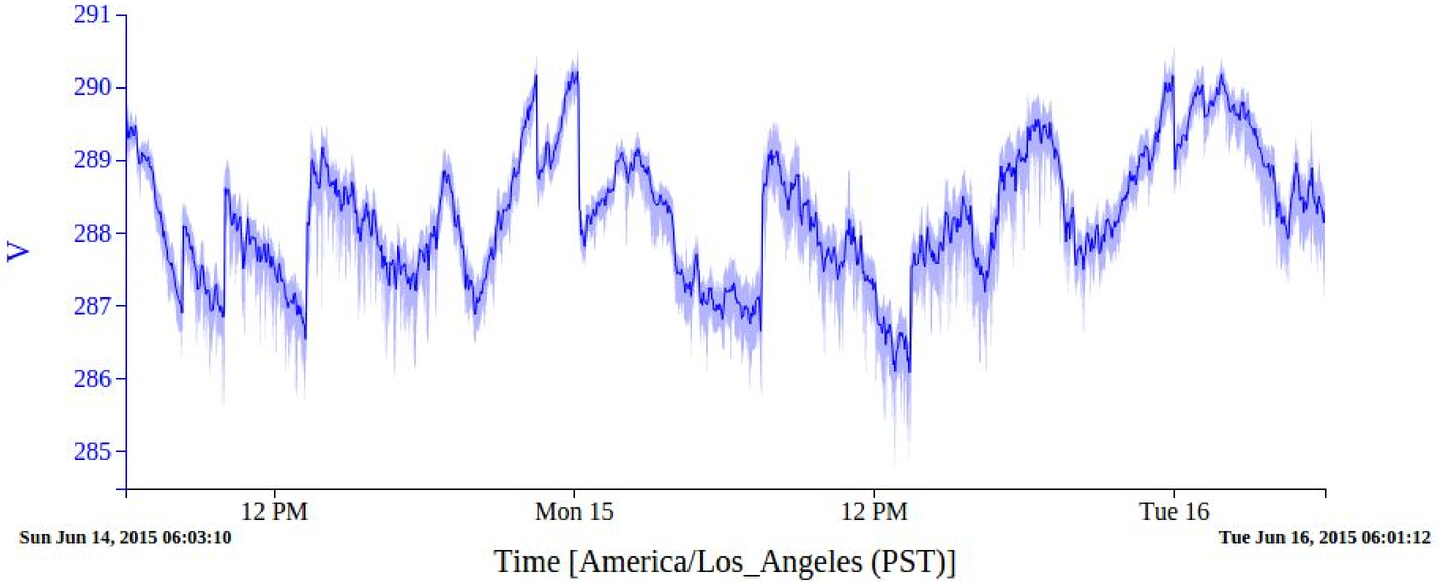




Overview

- Challenges with how this data is used and processed
- Solving this with the abstraction and operations that BTrDB provides
- BTrDB data structures
- Performance evaluation of a BTrDB Go implementation
- Idempotent distillation operations leveraging fast changeset calculation

Fast scalar stream telemetry



A stream is a list of (timestamp<int64> , value<float64>) tuples

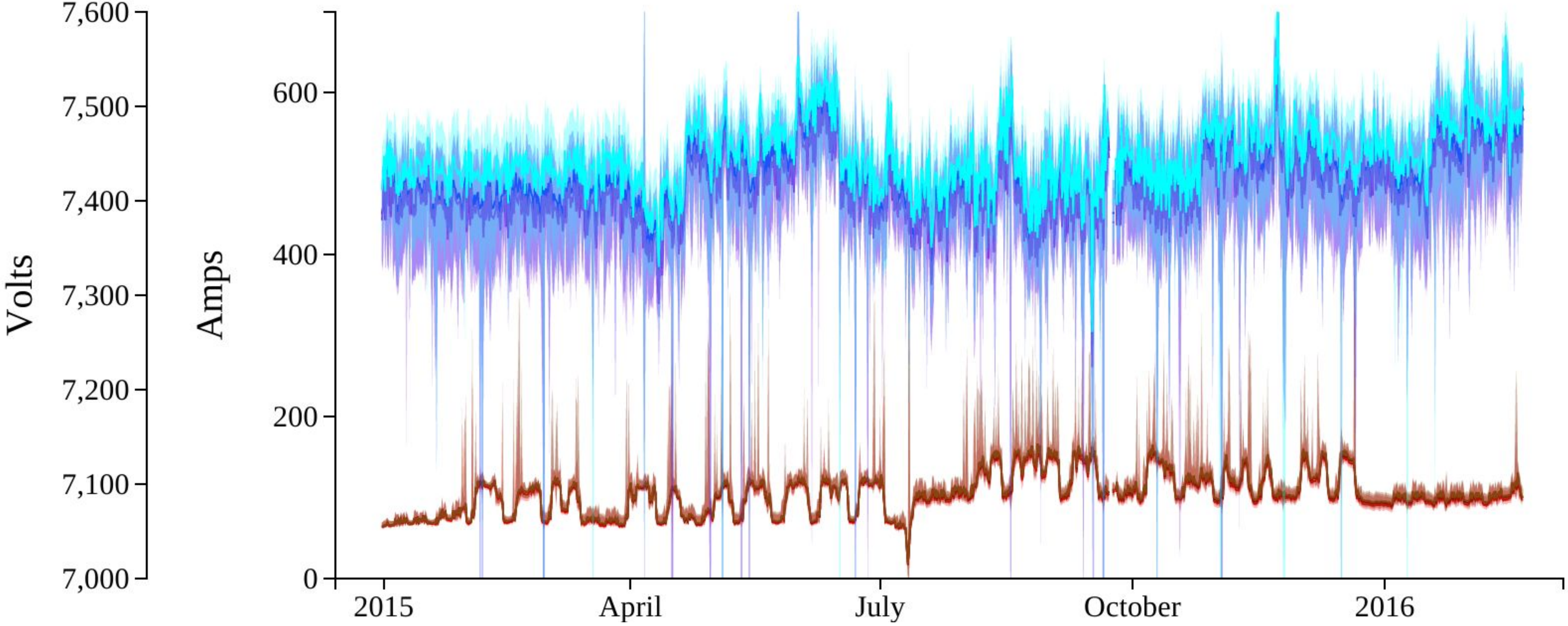
Challenges with this fast scalar telemetry

Data characteristics:

- High density: e.g. uPMUs are 120 Hz per stream, 1.4 kHz per device
- Varying lag and out of order delivery: e.g. delivered over intermittent LTE
- High precision timestamps: nanoseconds

Analysis characteristics:

- Aggregated queries more common than full-resolution queries



14 month overview from **just one** uPMU: 6 streams:
24 billion datapoints, 400GB of data

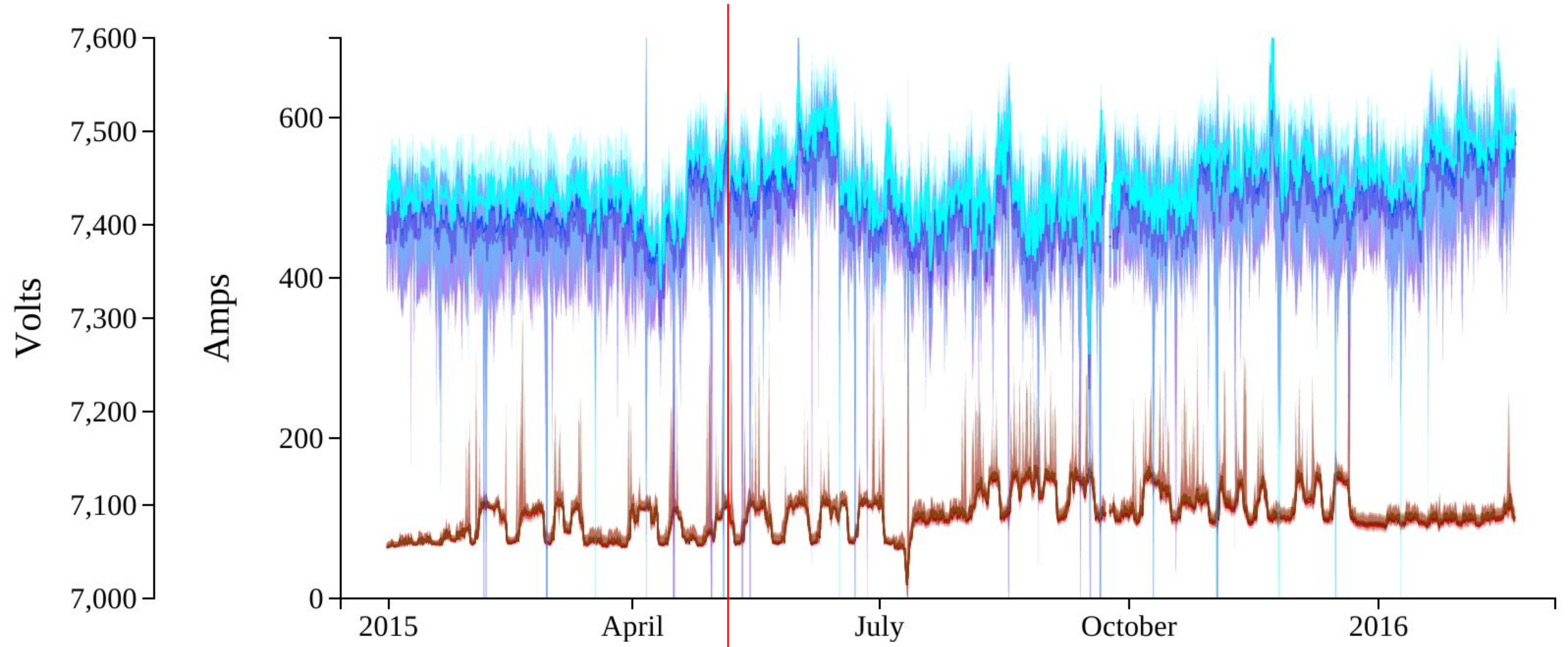
Challenges with this fast scalar telemetry

Data characteristics:

- High density: e.g. uPMUs are 120 Hz per stream, 1.4 kHz per device
- Varying lag and out of order delivery: e.g. delivered over intermittent LTE
- High precision timestamps: nanoseconds

Analysis characteristics:

- Aggregated queries more common than full-resolution queries
- Aggregation windows are much larger than the sample interval



One pixel column ^ is 4.2 million data points
Aggregation window is 6 orders of magnitude bigger than sample interval

Challenges with this fast scalar telemetry

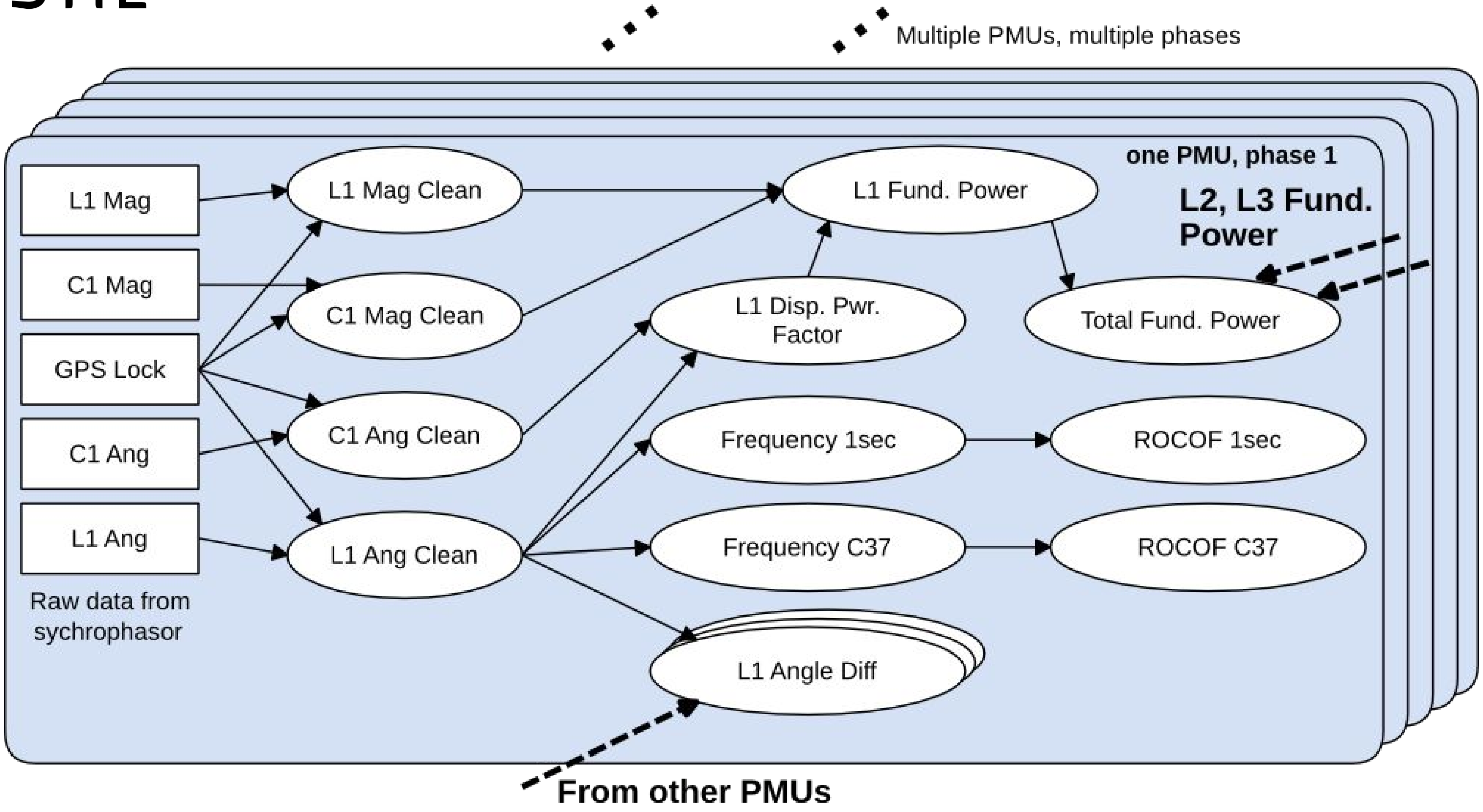
Data characteristics:

- High density: e.g. uPMUs are 120 Hz per stream, 1.4 kHz per device
- Varying lag and out of order delivery: e.g. delivered over intermittent LTE
- High precision timestamps: nanoseconds

Analysis characteristics:

- Aggregated queries more common than full-resolution queries
- Aggregation windows are much larger than the sample interval
- Data transformed in a DAG, creating multiple dependent streams

DISTIL



Multiple PMUs, multiple phases
one PMU, phase 1
L1 Mag
C1 Mag
GPS Lock
C1 Ang
L1 Ang
Raw data from sychrophasor
L1 Mag Clean
C1 Mag Clean
C1 Ang Clean
L1 Ang Clean
L1 Fund. Power
L1 Disp. Pwr. Factor
Frequency 1sec
Frequency C37
L1 Angle Diff
L2, L3 Fund. Power
Total Fund. Power
ROCOF 1sec
ROCOF C37
From other PMUs

Why can we not solve this with existing DBs

- Density:
        - 1.4 Million values/s/node raw data
        - >10 Million values/s /nodederived streams
    - Existing throughput is too low (<< 1 Million values/s/node)
- Aggregation capability mismatch:
    - Either done Just In Time (query time aggregation) - too expensive for 100’s of GB
    - Or done at insert time - doesn’t handle out of order / changed data
    - Don’t guarantee consistency of aggregate
- Hard to support analysis DAG:
    - Require per-consumer state
    - Don’t provide snapshot features - needed for idempotent analysis
    - Don’t guarantee consistency of result streams

Why do these shortcomings exist?

- Often, because they do too much:
  - Designed for data that is complex, multidimensional
  - Support queries based on multiple indexes, or on values
    - Find me **log messages** where the type is **error**
    - Find the sum of **session times** where the **advert** is from **vendor X**

Simple Abstraction for Timeseries Database

- QueryRange(uuid, start_time, end_time)
  - > <[<time, value>]>
- InsertValues(uuid, [<time, value>])
  - >
- DeleteRange(uuid, start_time, end_time)
  - >

Would this work?

- Analyse recently changed data **- HARD**
  - Not always at the end of the stream
- Perform computation idempotently **- HARD**
  - Snapshot the stream
- Compute dependent streams: B = f(A) **- HARD**
  - Run a function over everything in A that has changed since last computation
- Locate interesting events in large quantities of data **- HARD**
  - In the synchrophasors project, an event is ~100ms, and a year's worth of data from a single device is 670 GB
  - 'Interesting' is hard to define, but it often means:
    - above or below a threshold
    - more than X from the mean
    - having a different density than the rest (holes, timebase overlapping etc)

Improved Abstraction for Timeseries Database

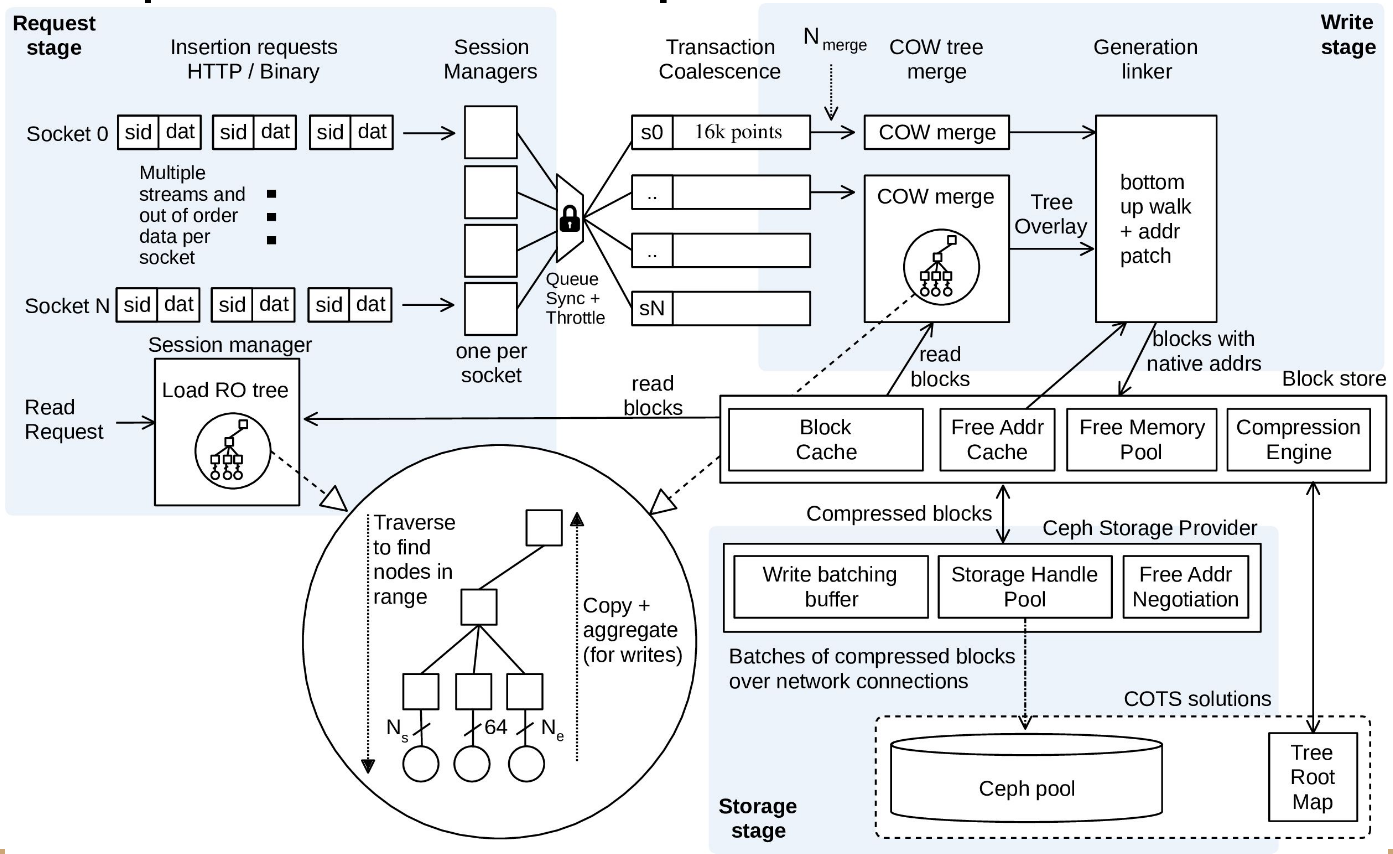
- QueryRange(uuid, start_time, end_time, **version**)
  -> (**version**, List of (time, value))
- InsertValues(uuid, [<time, value>])
  -> **version**
- DeleteRange(uuid, start_time, end_time)
  -> **version**

- **StatisticalWindow(uuid, start_time, end_time, version, windowsize)**
  -> (version, List of (time, min, mean, max, count))
- **ComputeDiff(uuid, fromversion, toversion, version, resolution)**
  -> List of (starttime, endttime)

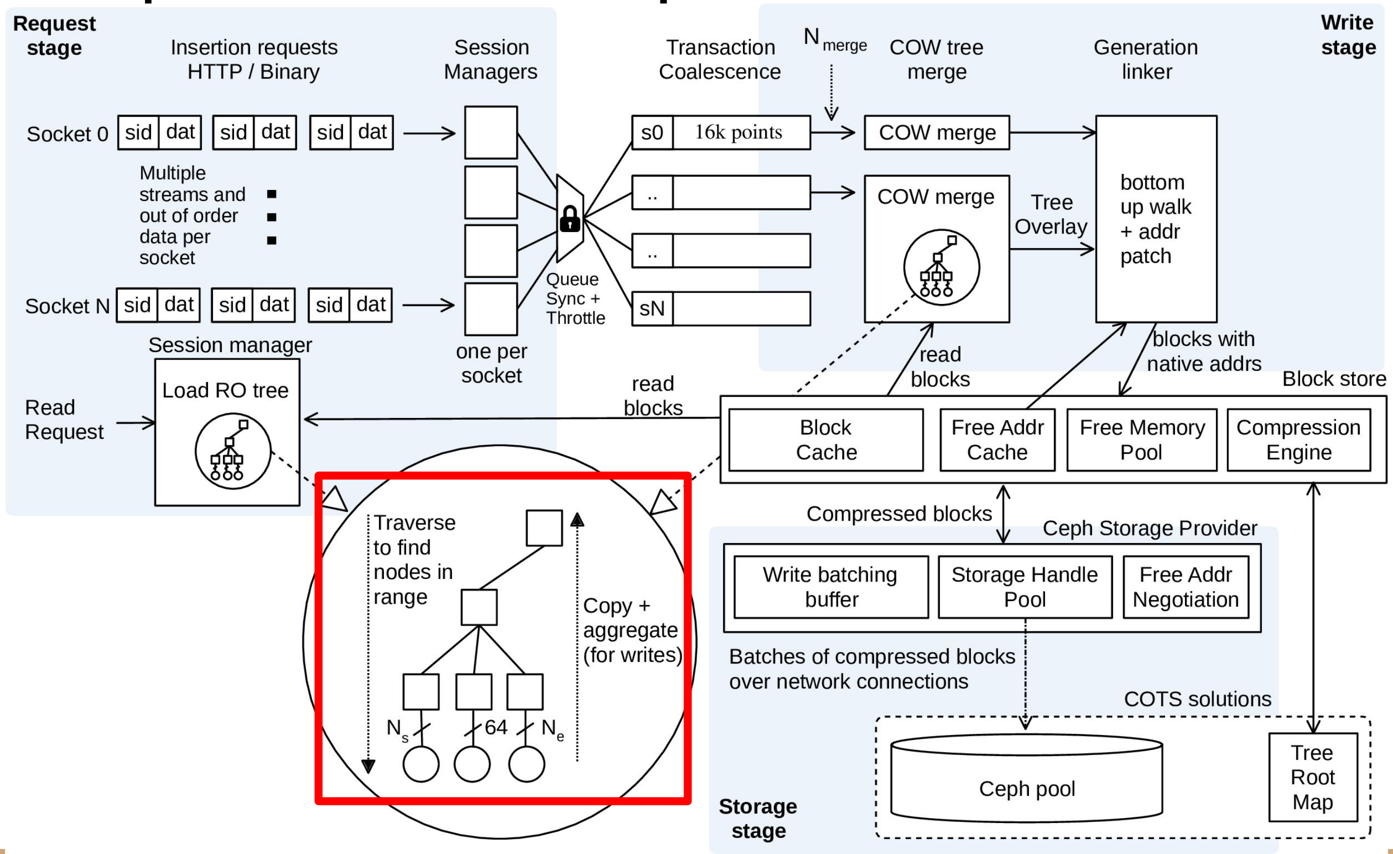
Would this work?

- Analyse recently changed data **- ComputeDiff**
  - Not always at the end of the stream
- Perform computation idempotently **- Version**
  - Snapshot the stream
- Compute dependent streams: B = f(A) **- ComputeDiff + Version**
  - Run a function over everything in A that has changed since last computation
- Locate interesting events in large quantities of data **- StatisticalWindow**
  - In the synchrophasors project, an event is ~100ms, and a year's worth of data from a single device is 670 GB
  - 'Interesting' is hard to define, but it often means:
    - above or below a threshold **- Mean/Min/Max**
    - more than X from the mean
    - having a different density than the rest (holes, timebase overlapping etc) **- Count**

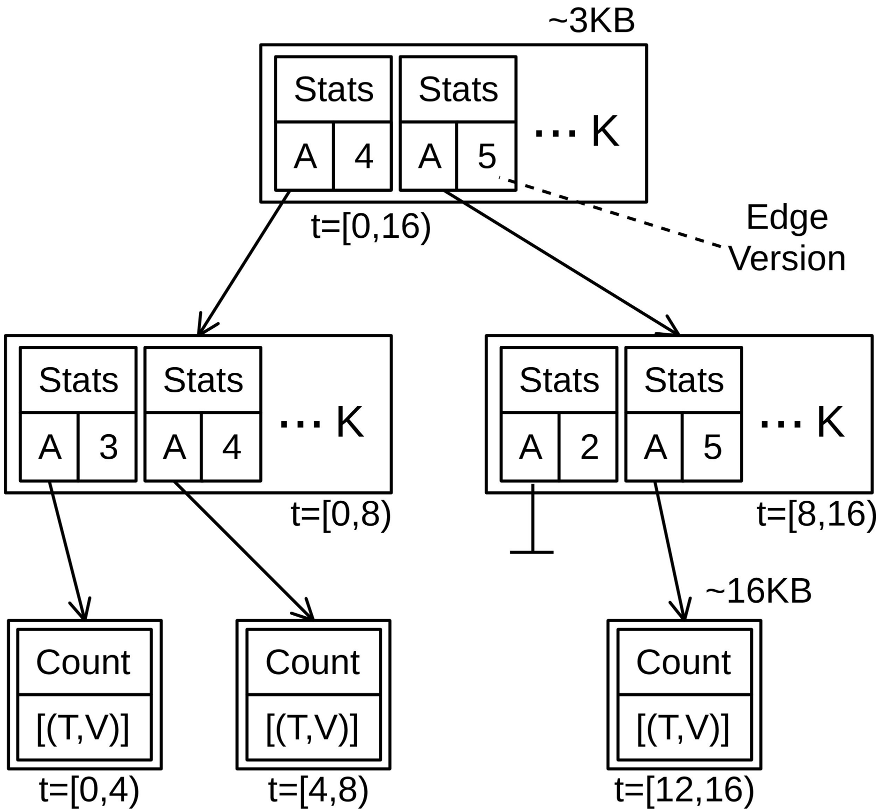
A Go implementation of BTrDB



A Go implementation of BTrDB



BTrDB Tree - a datastructure for this abstraction



Copy on write K-ary Tree

Partitioning static time (1933 to 2079)

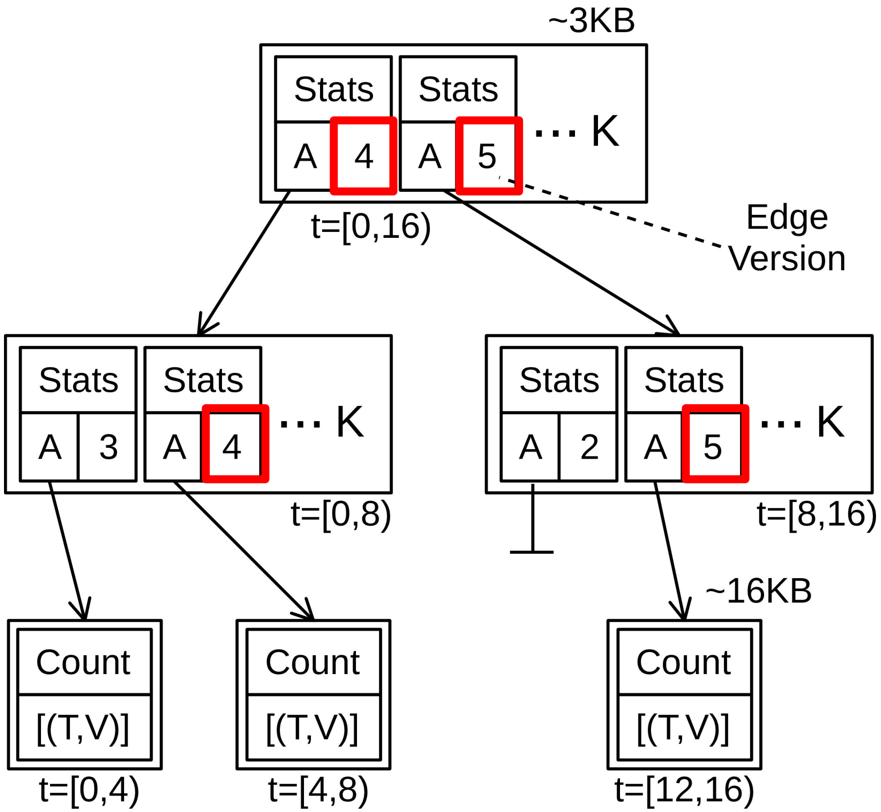
Leaf nodes

- Time, value pairs + length

Internal nodes

- Pointers to children
- **Version annotations** for children
- **Aggregates** for children
    - Min, Mean, Max, Count
    - Any associative operator

BTrDB Tree - a datastructure for this abstraction



Copy on write K-ary Tree

Partitioning static time (1933 to 2079)

<u>Leaf nodes</u>

- Time, value pairs + length

<u>Internal nodes</u>

- Pointers to children
- **Version annotations** for children
- **Aggregates** for children
    - Min, Mean, Max, Count
    - Any associative operator

More on the tree

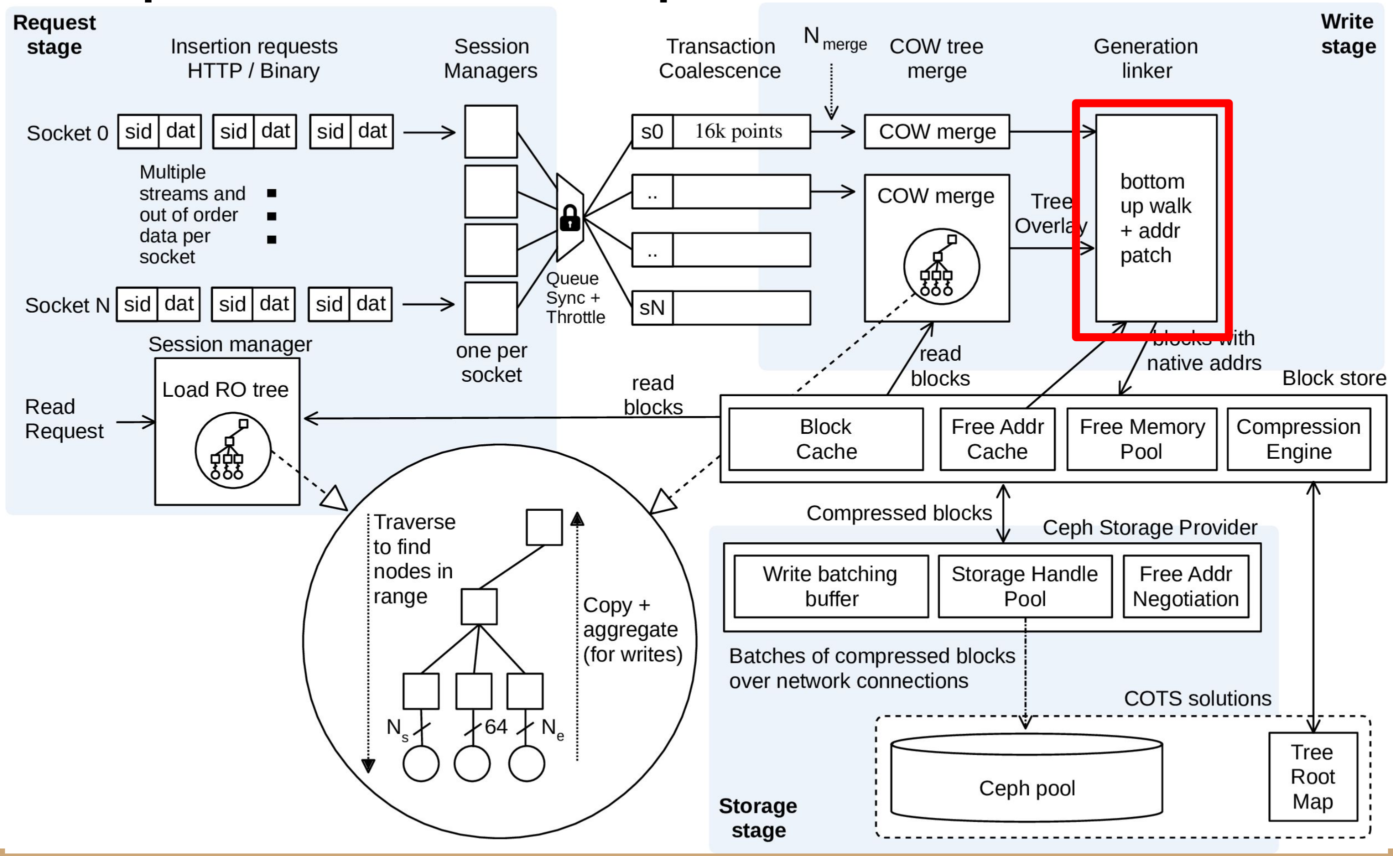
There are good reasons for doing aggregates at commit time, in the tree:

- Data already in memory, nodes already need COW: no additional IO
- If version is visible, root was written therefore aggregates are consistent
- They don't use much space: 0.3% of a K=64 tree

How to reduce RTTs in traversing tree?

- Edges use native addresses, directly resolvable by underlying storage

A Go implementation of BTrDB



More on the tree

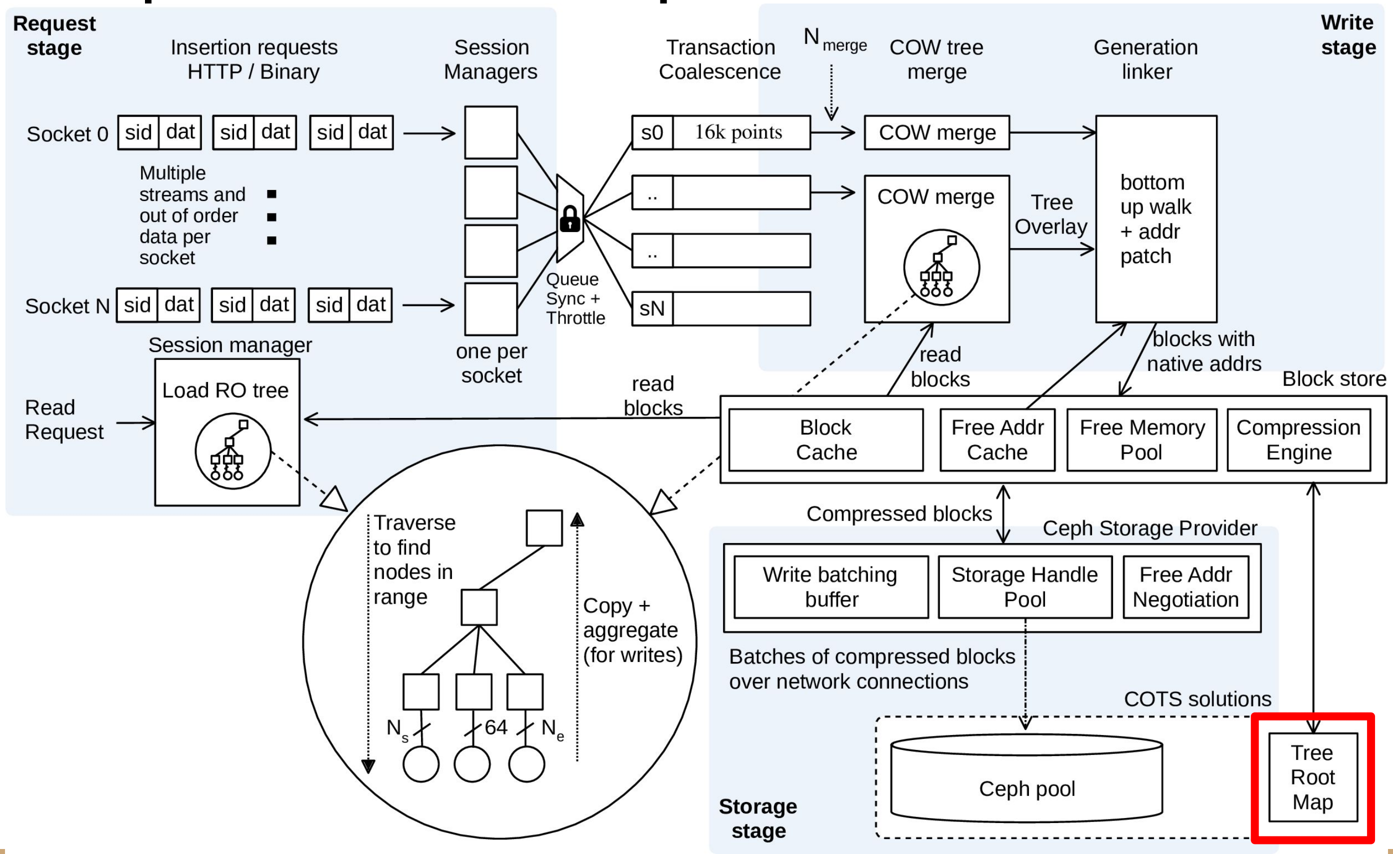
Why do aggregates in the tree?

- Data already in memory: no additional IO
- If version is visible, root was written therefore aggregates are consistent
- They don’t use much space: 0.3% of a K=64 tree

How to reduce RTTs in traversing tree?

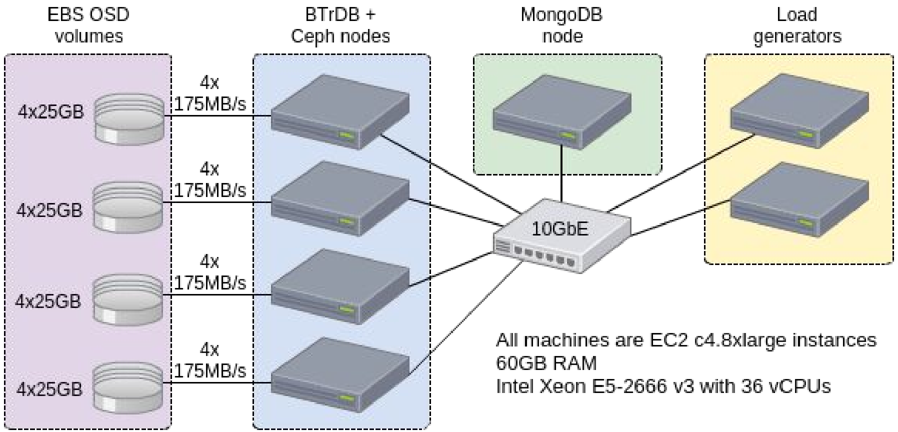
- Edges use native addresses, directly resolvable by underlying storage
- Only the root of the tree requires translation
    - uuid -> native address of the root

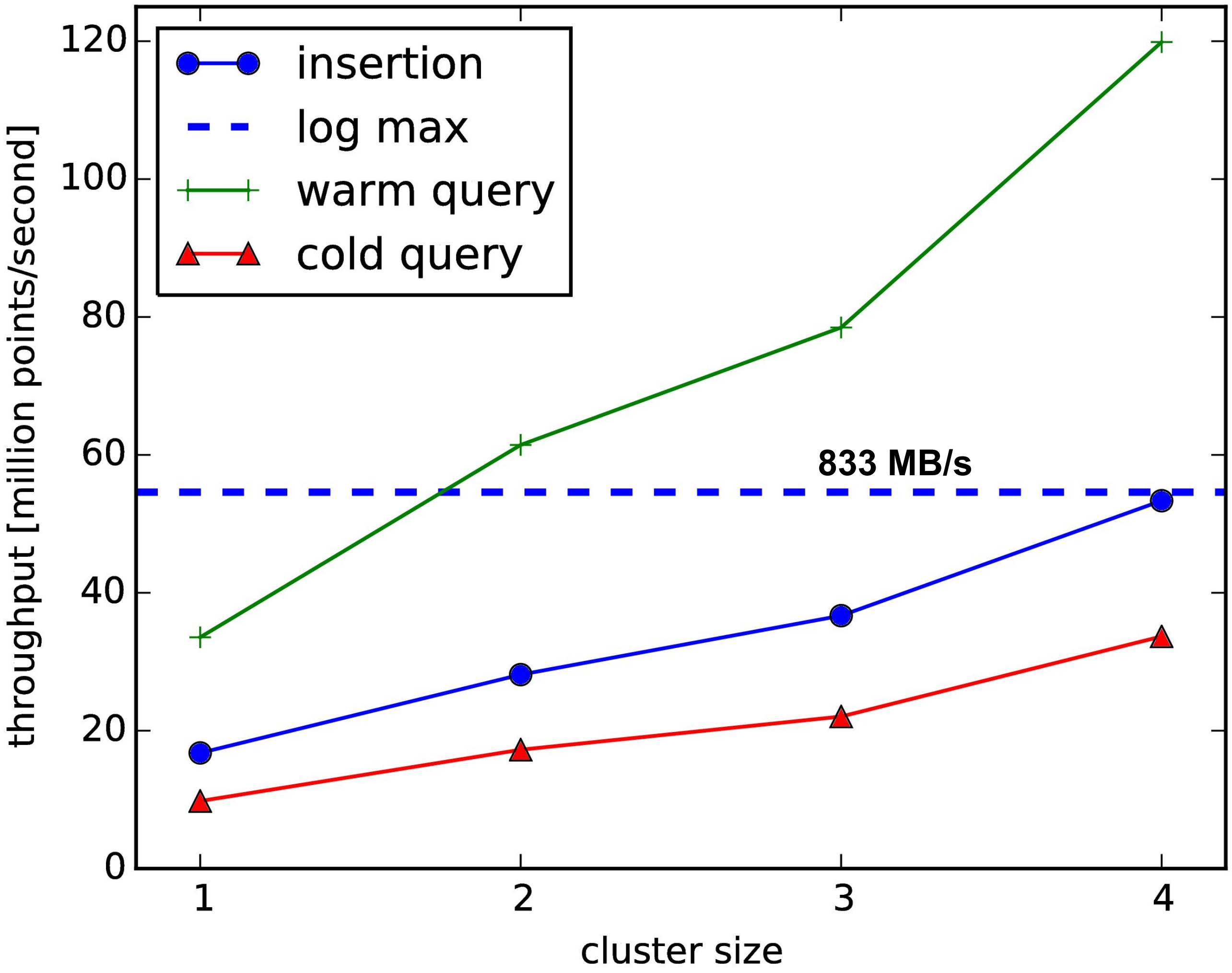
A Go implementation of BTrDB



Evaluation

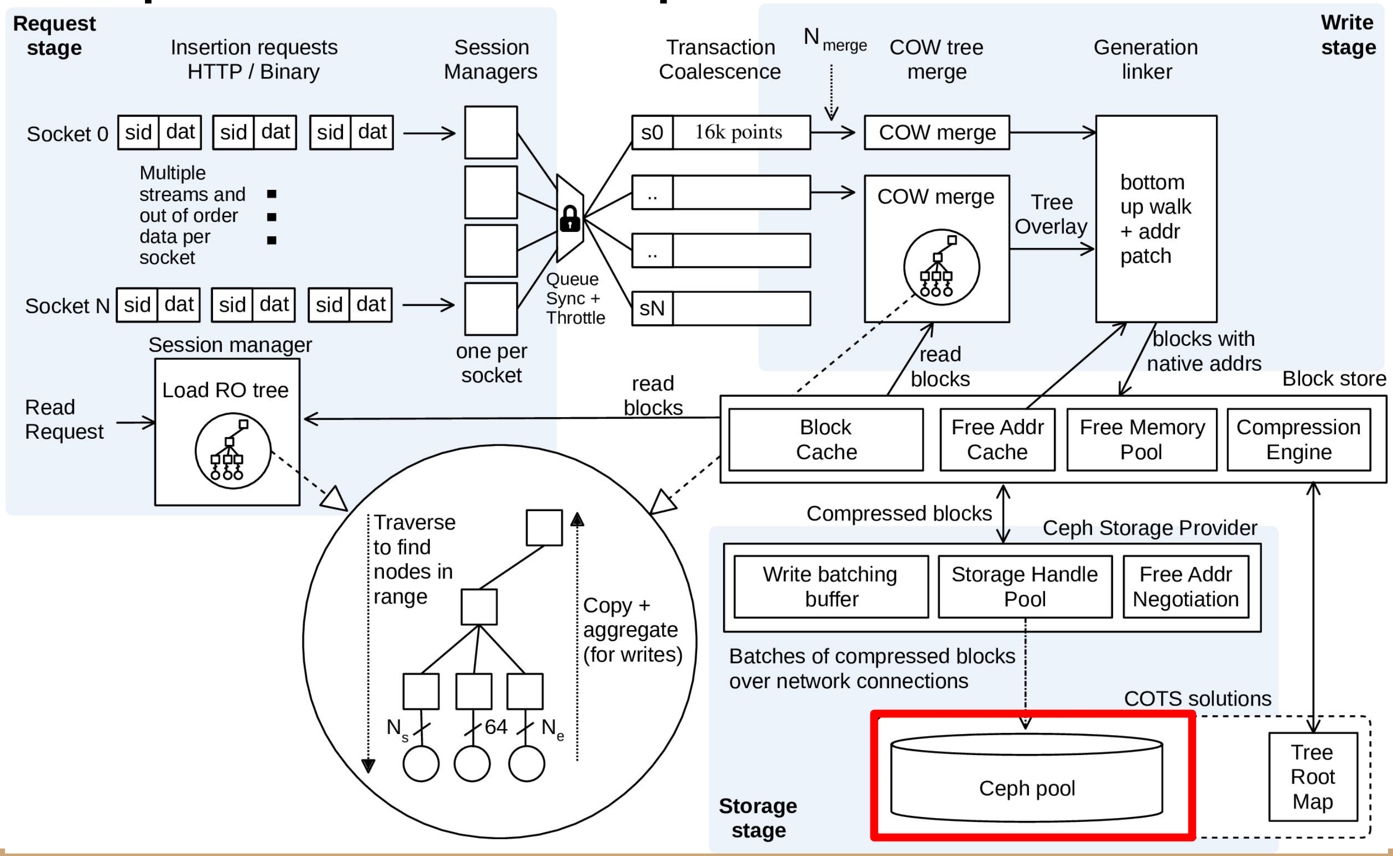
Raw throughput with chronological random inserts and queries on EC2

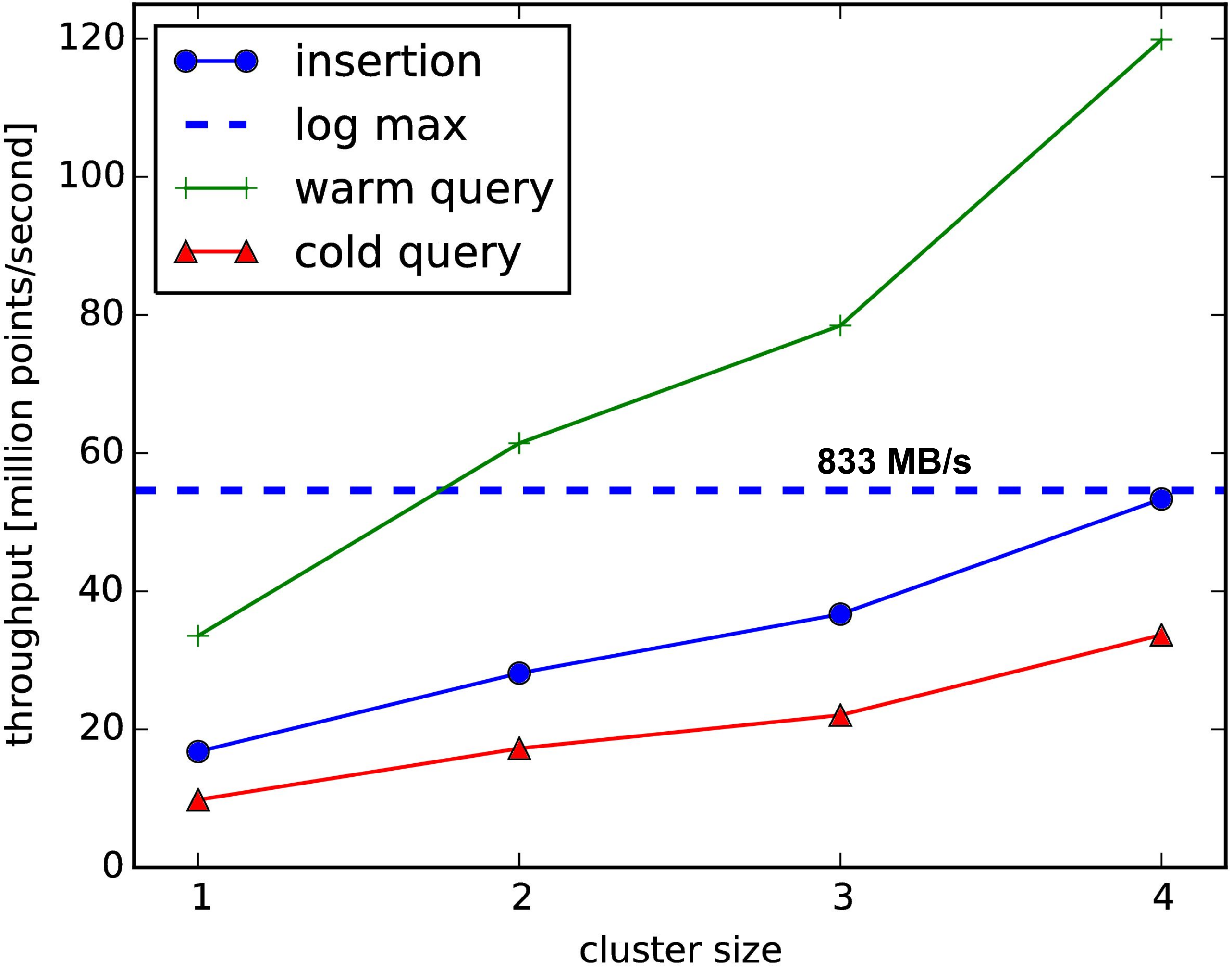




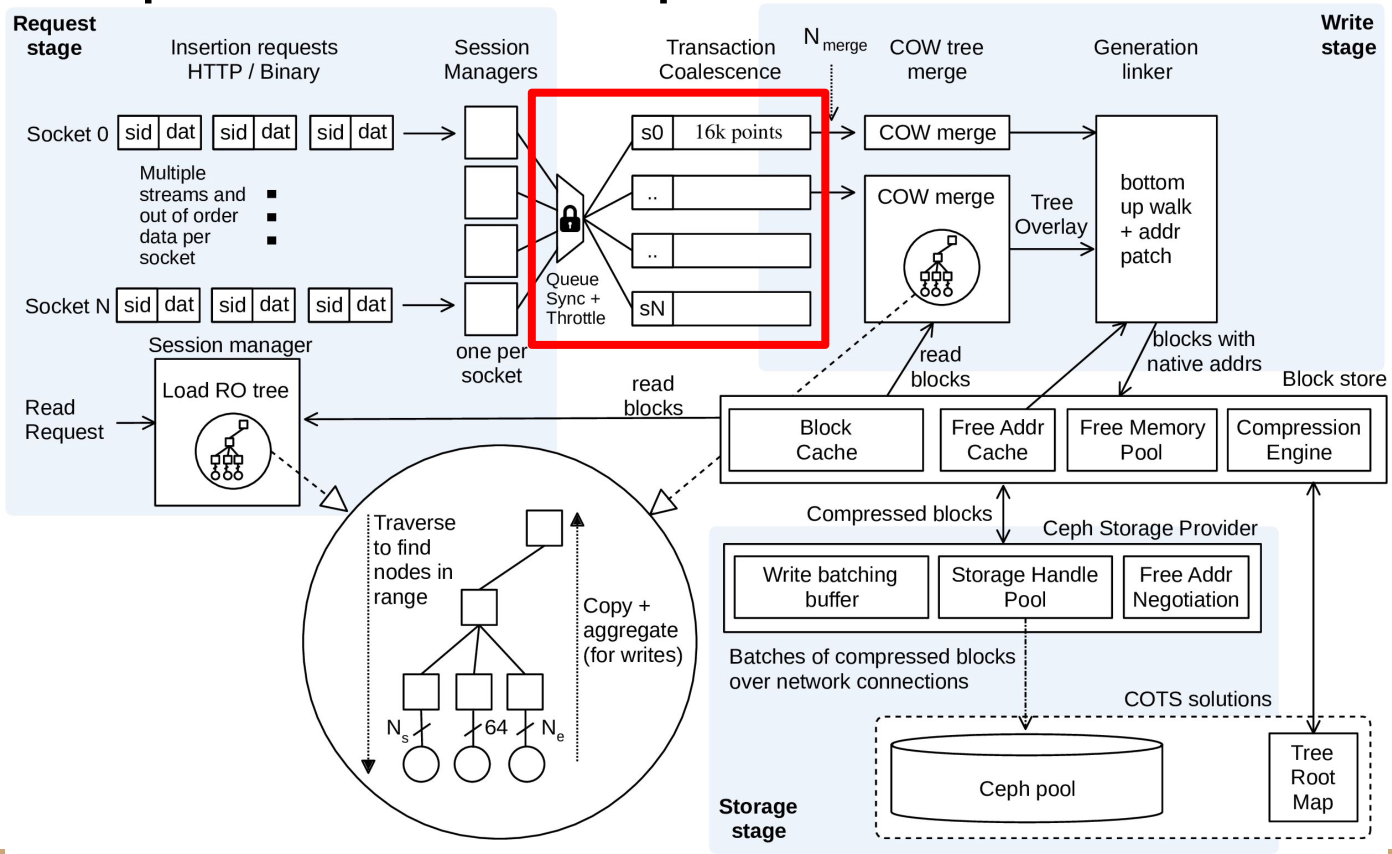
insertion
log max
warm query
cold query
833 MB/s
throughput [million points/second]
0
20
40
60
80
100
120
1
2
3
4
cluster size

A Go implementation of BTrDB




insertion
log max
warm query
cold query
833 MB/s
throughput [million points/second]
cluster size
0
20
40
60
80
100
120
1
2
3
4

A Go implementation of BTrDB

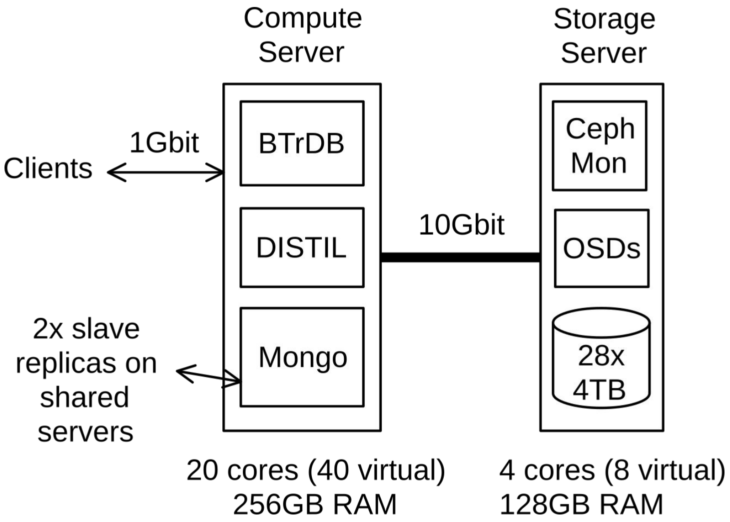


Out of order performance characteristics

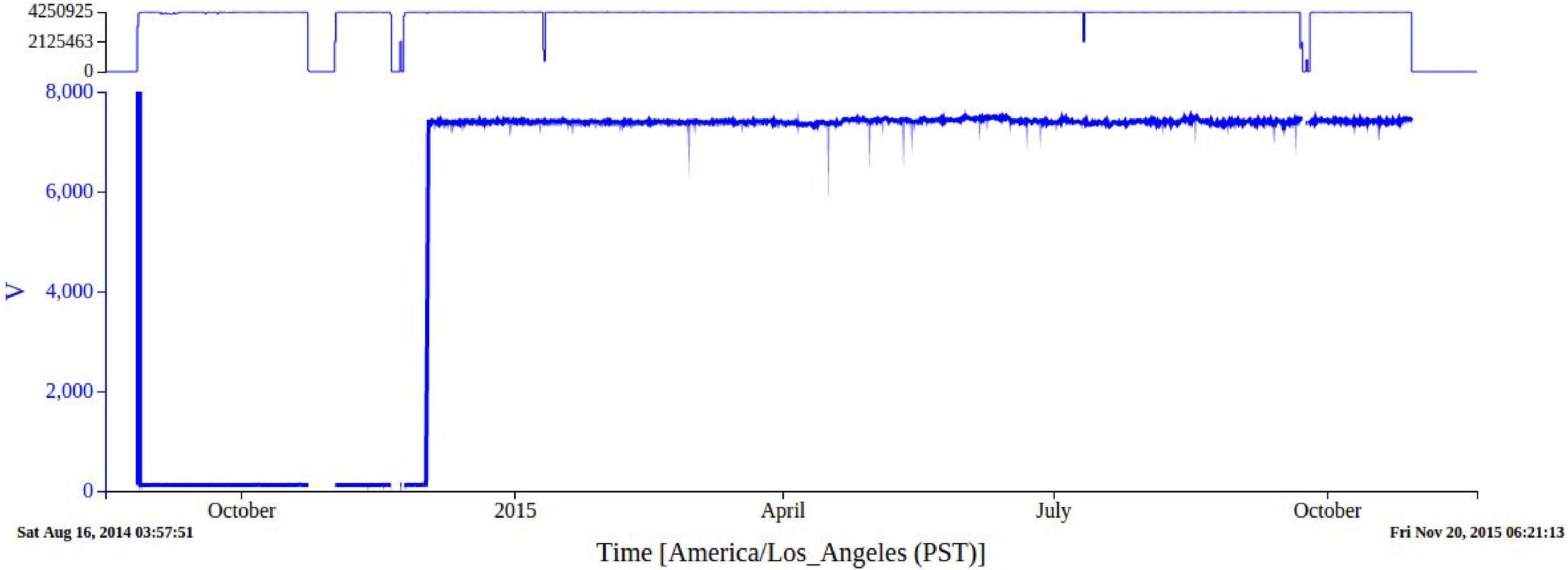
	When insertion was	
Throughput [million pt/s] for	Chrono.	Random
Insert	28.12	27.73
Cold query in chrono. order	31.41	31.67
Cold query in same order	-	32.61
Cold query in random order	29.67	28.26
Warm query in chrono. order	114.1	116.2
Warm query in same order	-	119.0
Warm query in random order	113.7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

Evaluation

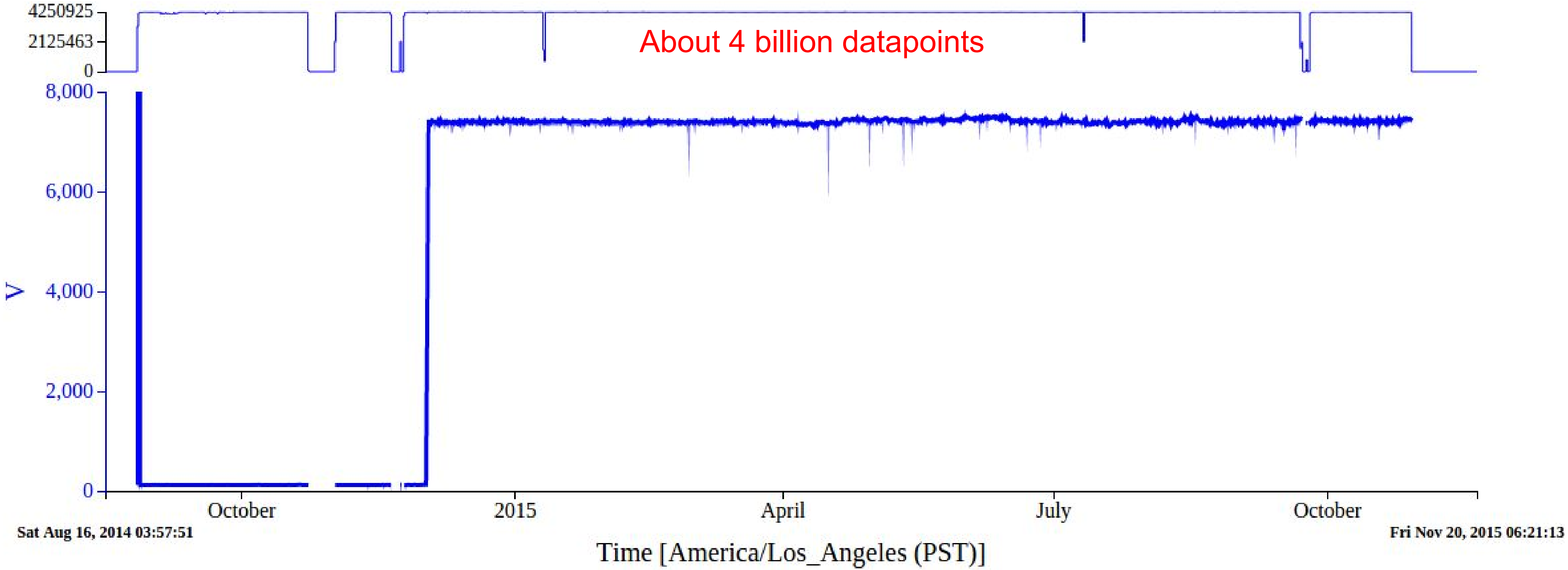
Statistical queries on a production server



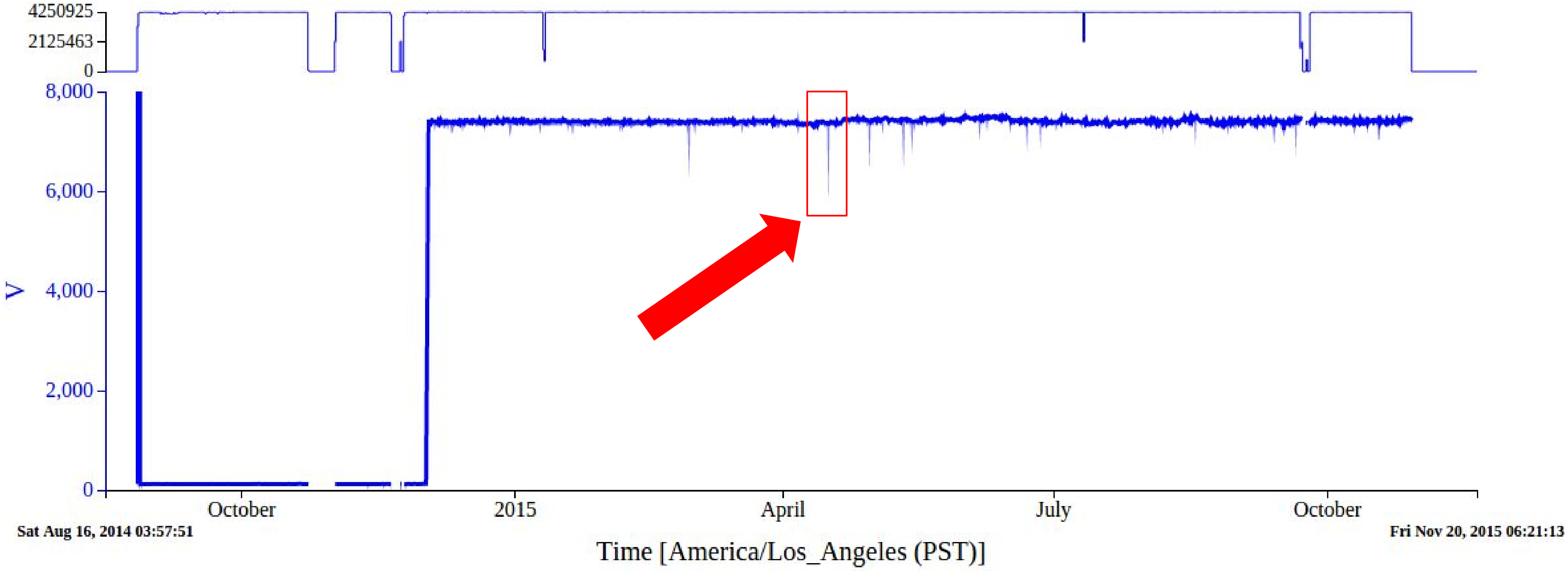
Statistic queries : visualisation



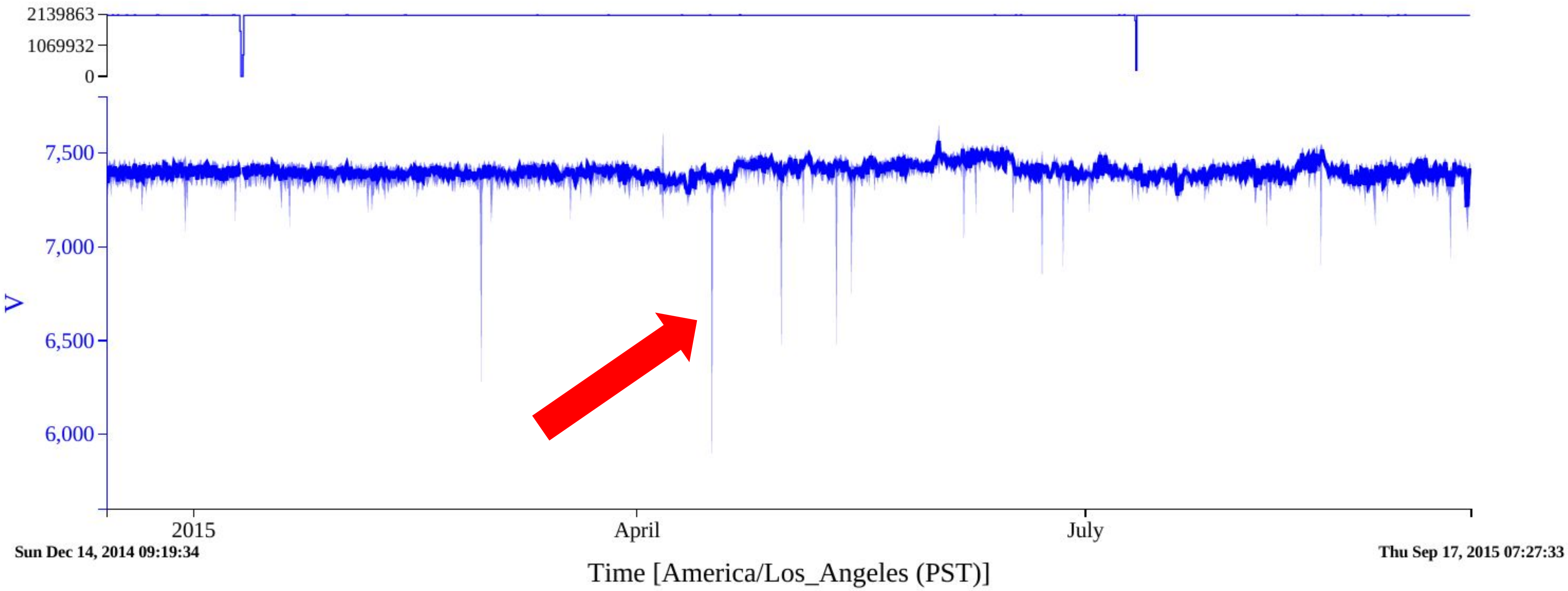
Statistic queries : visualisation



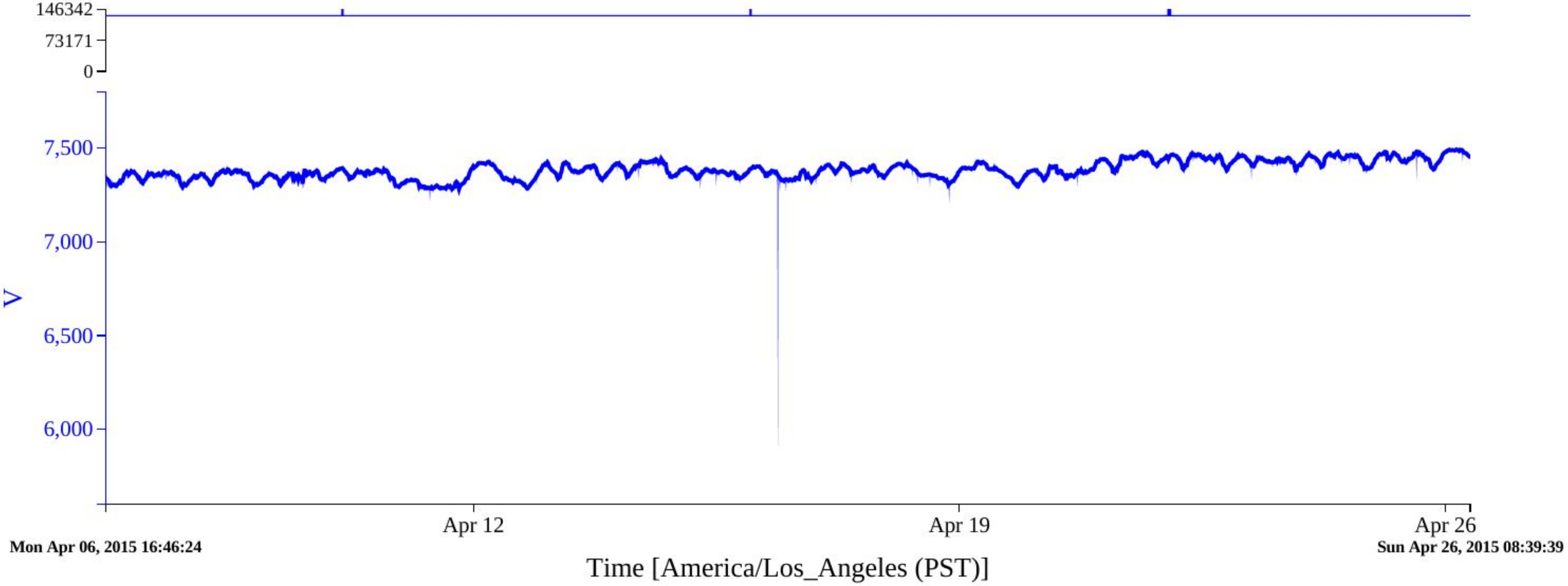
Statistic queries : visualisation



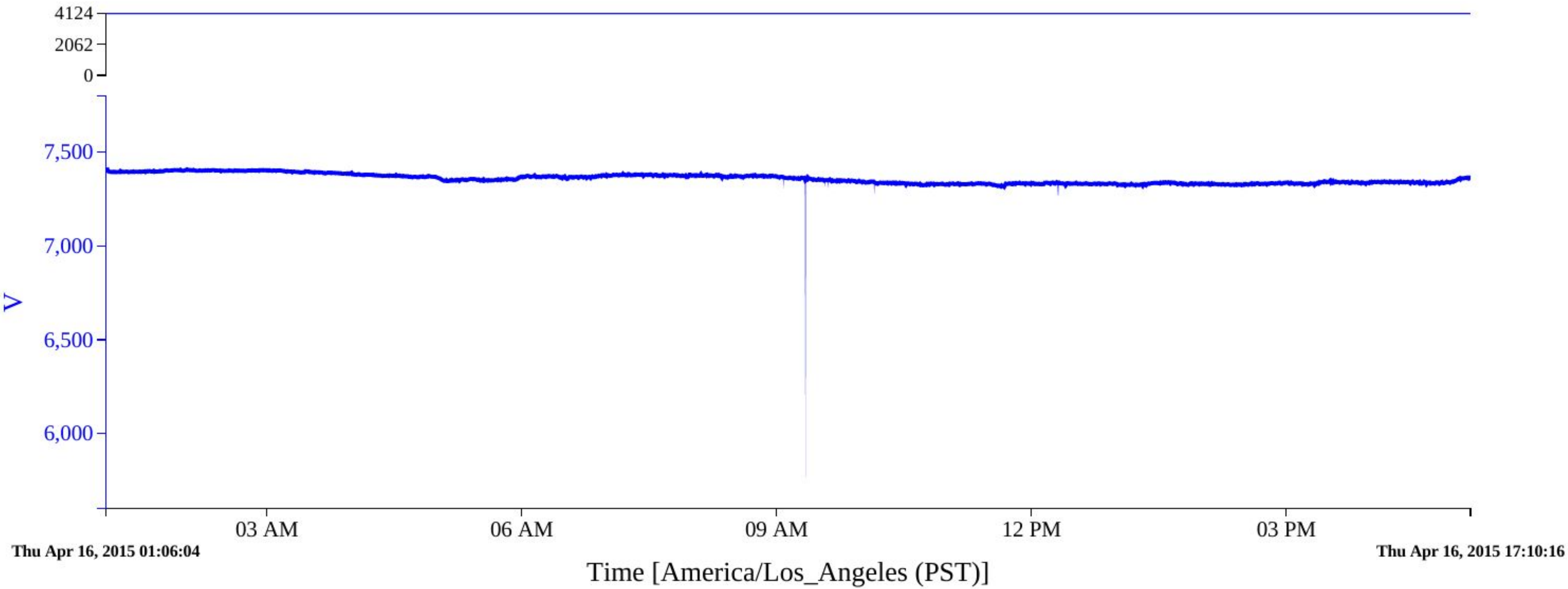
Statistic queries : visualisation



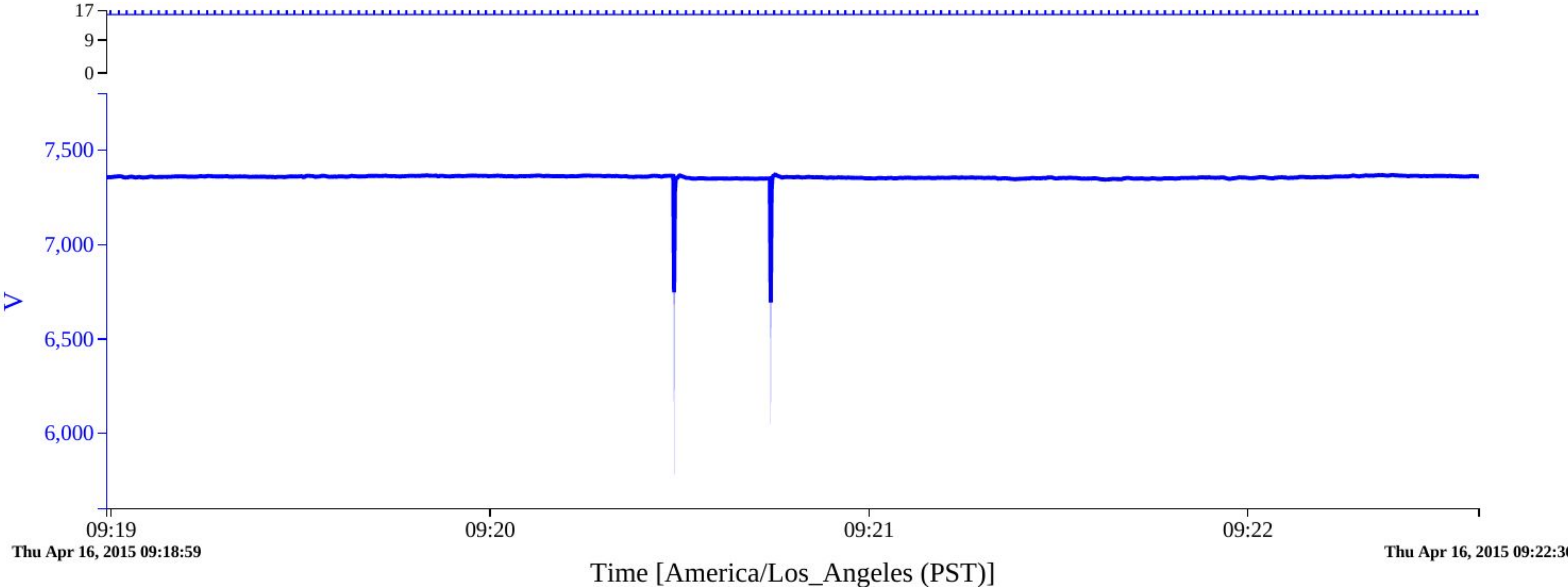
Statistic queries : visualisation



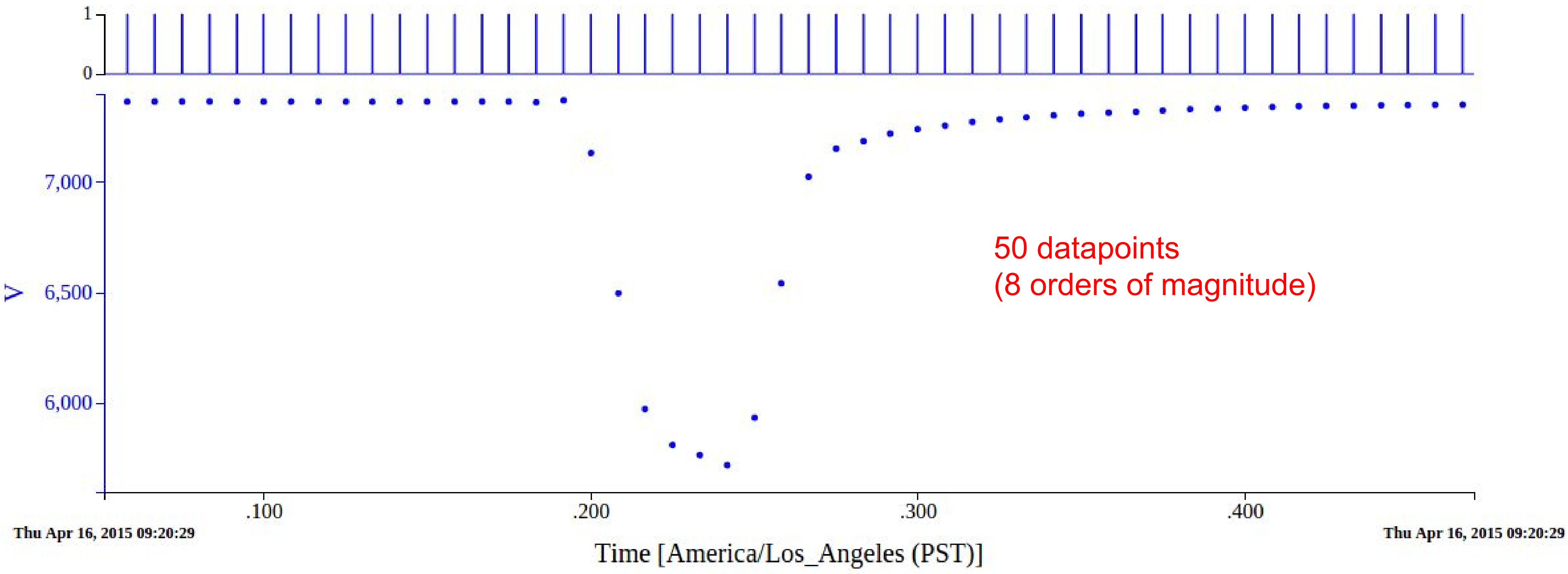
Statistic queries : visualisation

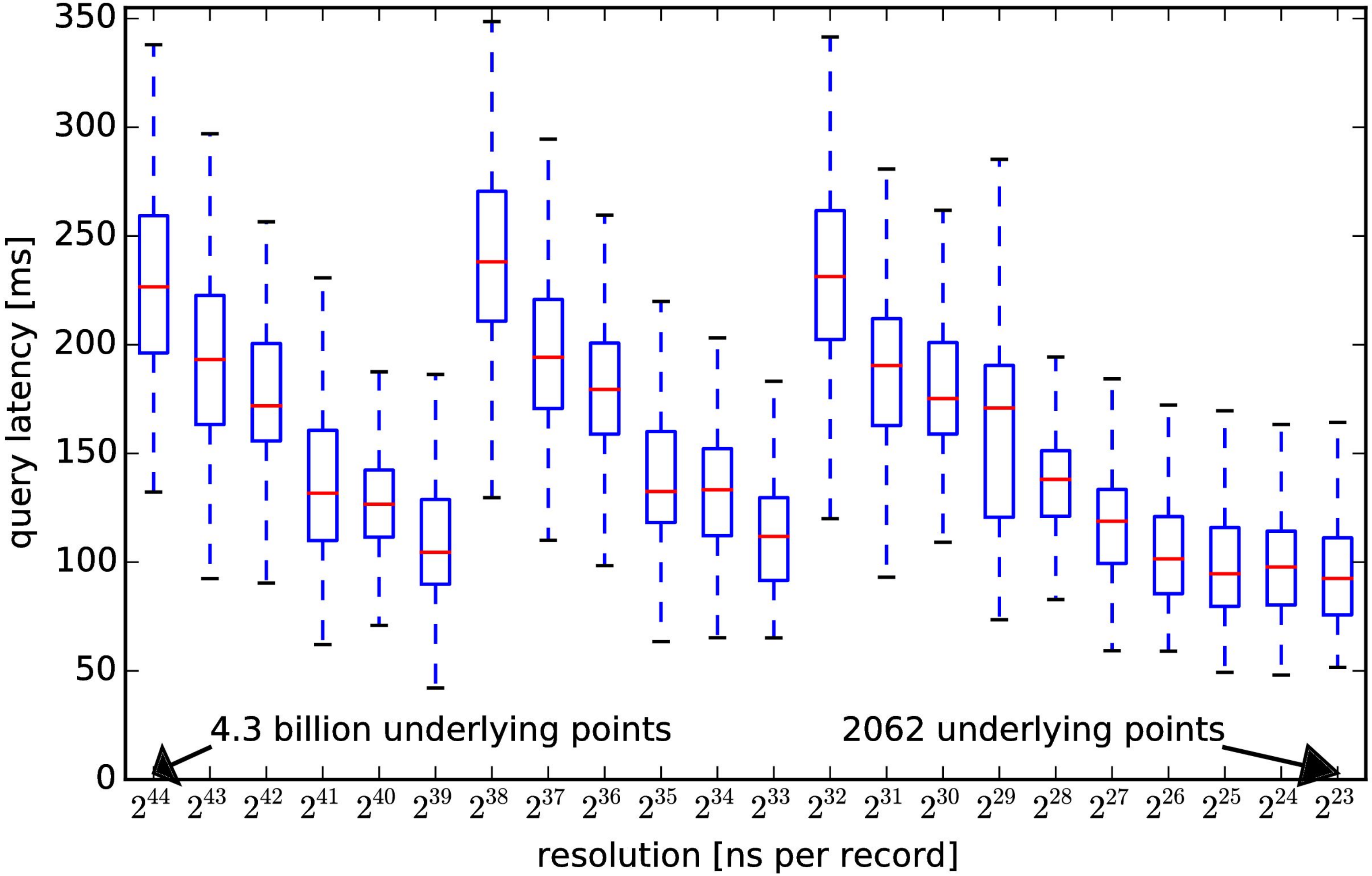


Statistic queries : visualisation

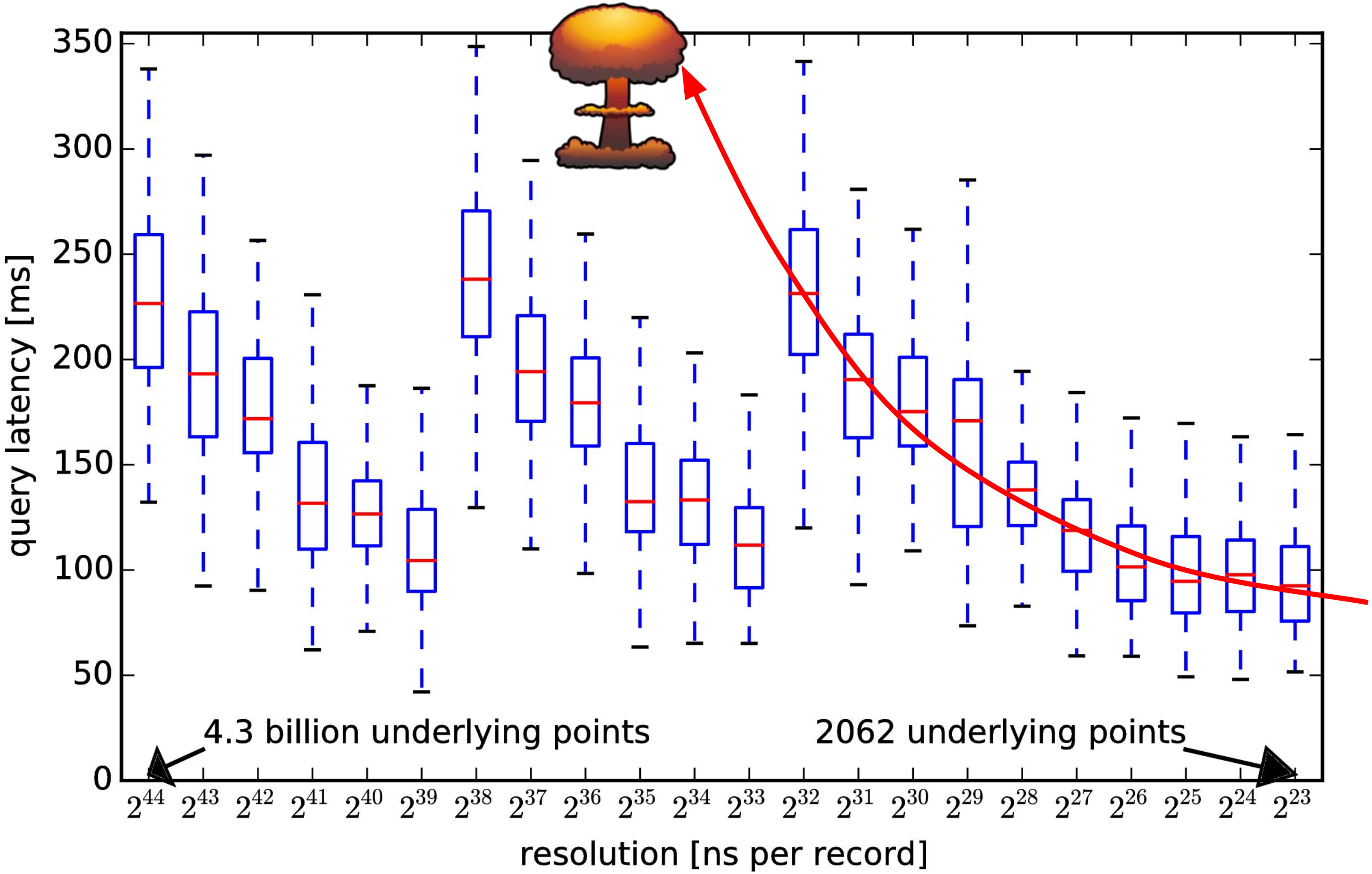


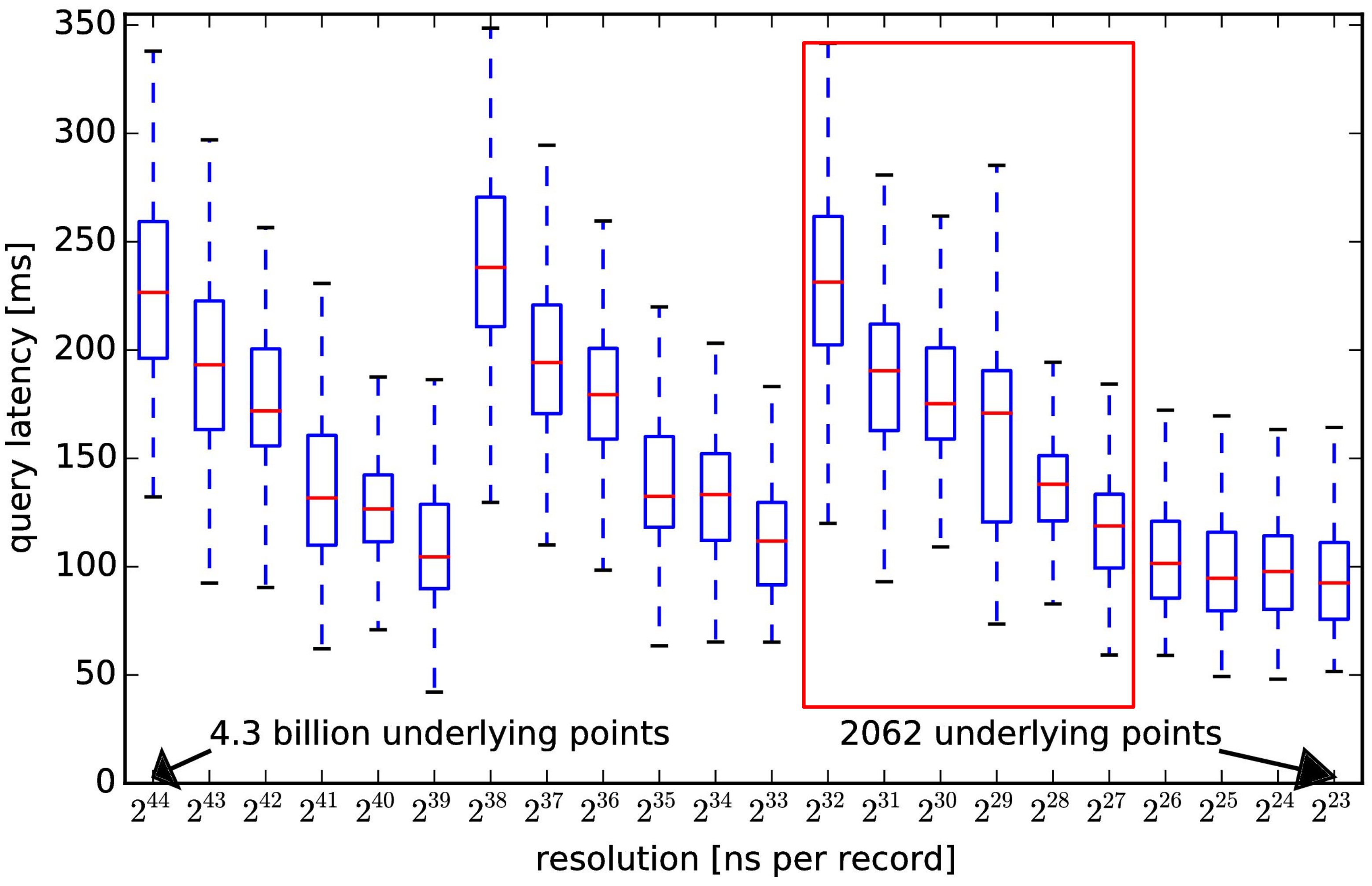
Statistic queries : visualisation

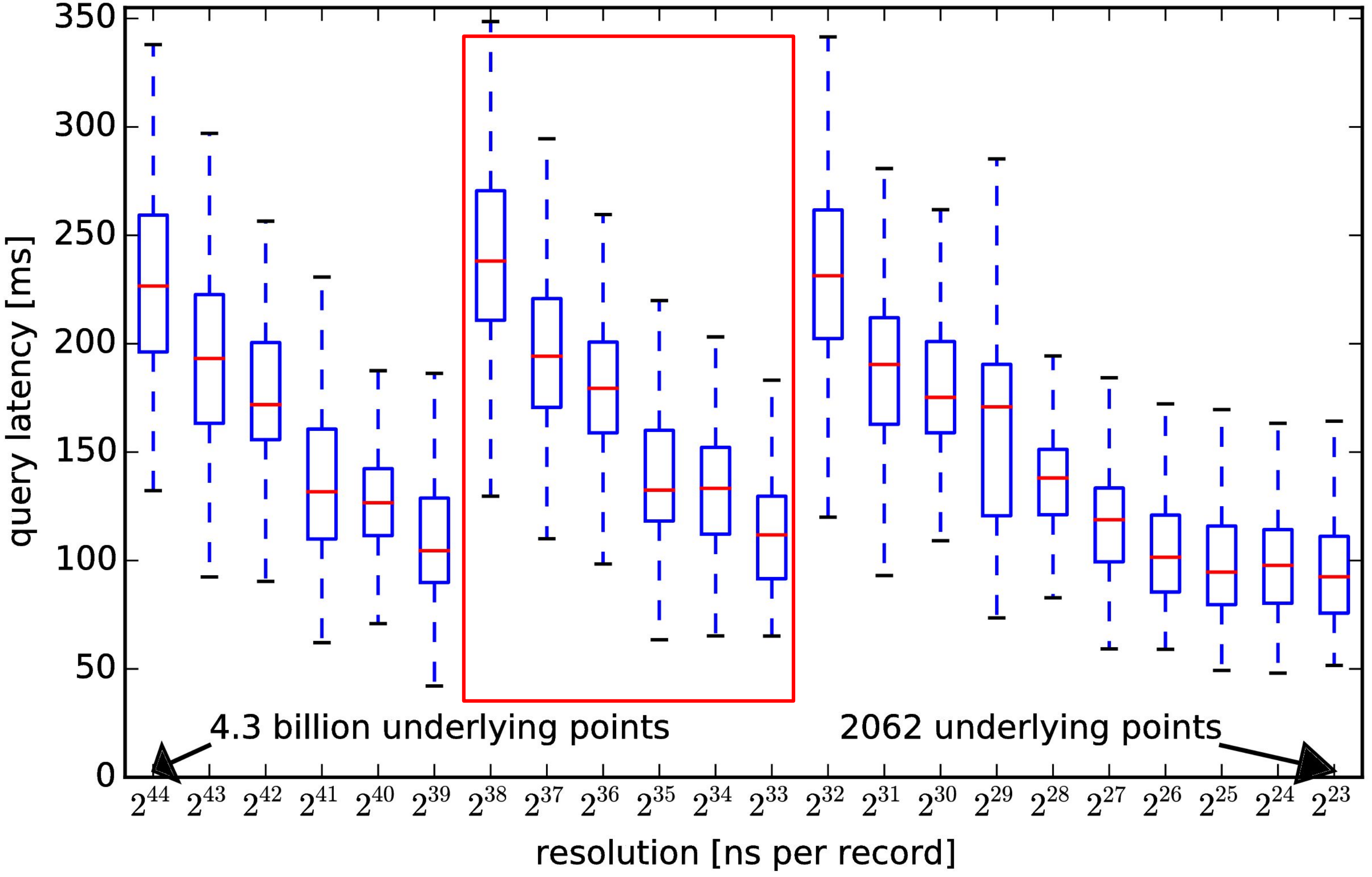




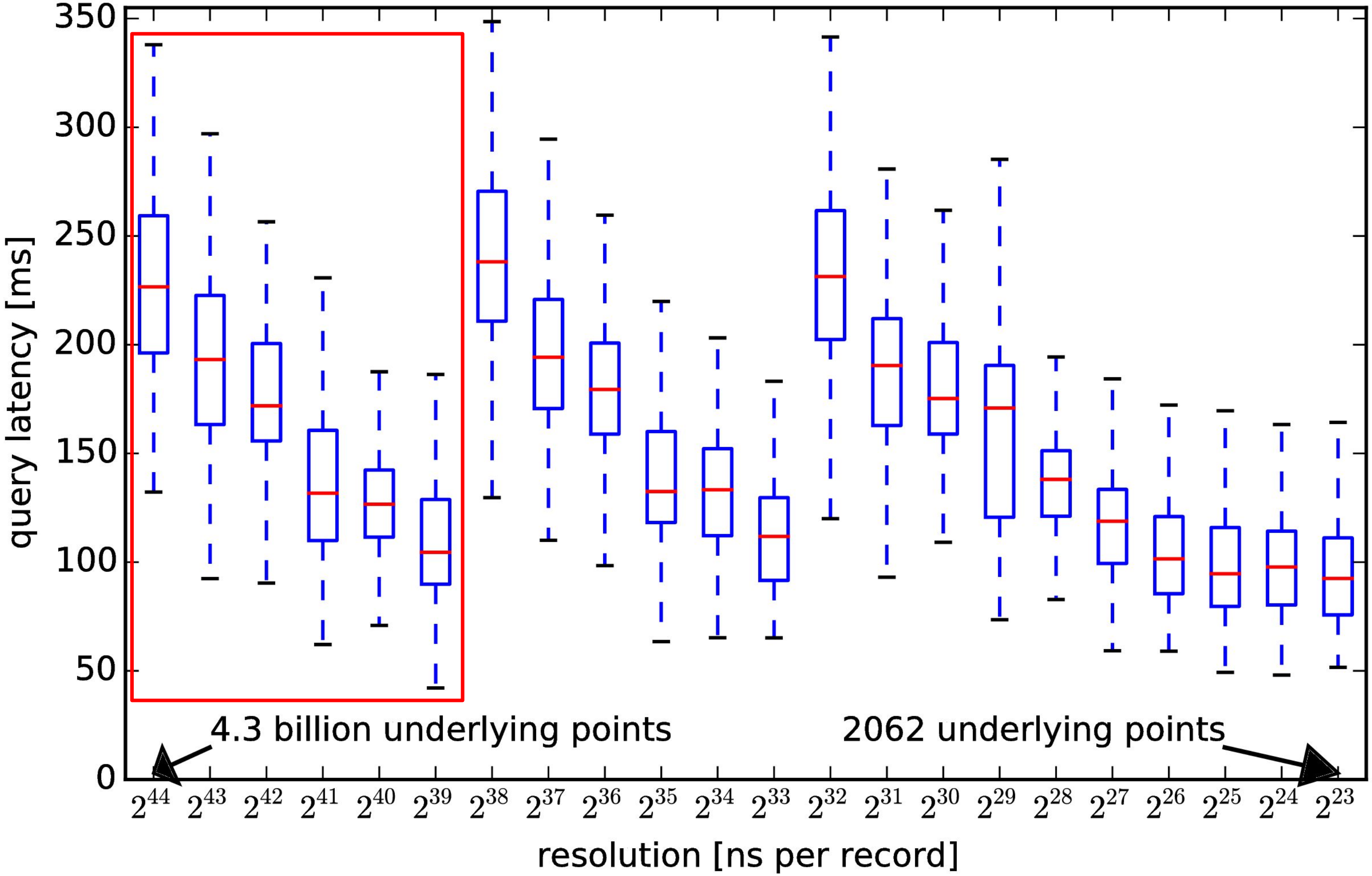
query latency [ms]
350
300
250
200
150
100
50
0
4.3 billion underlying points
2062 underlying points
2^{44} 2^{43} 2^{42} 2^{41} 2^{40} 2^{39} 2^{38} 2^{37} 2^{36} 2^{35} 2^{34} 2^{33} 2^{32} 2^{31} 2^{30} 2^{29} 2^{28} 2^{27} 2^{26} 2^{25} 2^{24} 2^{23}
resolution [ns per record]


query latency [ms]
350
300
250
200
150
100
50
0
4.3 billion underlying points
2062 underlying points
2^{44} 2^{43} 2^{42} 2^{41} 2^{40} 2^{39} 2^{38} 2^{37} 2^{36} 2^{35} 2^{34} 2^{33} 2^{32} 2^{31} 2^{30} 2^{29} 2^{28} 2^{27} 2^{26} 2^{25} 2^{24} 2^{23}
resolution [ns per record]


query latency [ms]
350
300
250
200
150
100
50
0
4.3 billion underlying points
2062 underlying points
2^{44} 2^{43} 2^{42} 2^{41} 2^{40} 2^{39} 2^{38} 2^{37} 2^{36} 2^{35} 2^{34} 2^{33} 2^{32} 2^{31} 2^{30} 2^{29} 2^{28} 2^{27} 2^{26} 2^{25} 2^{24} 2^{23}
resolution [ns per record]

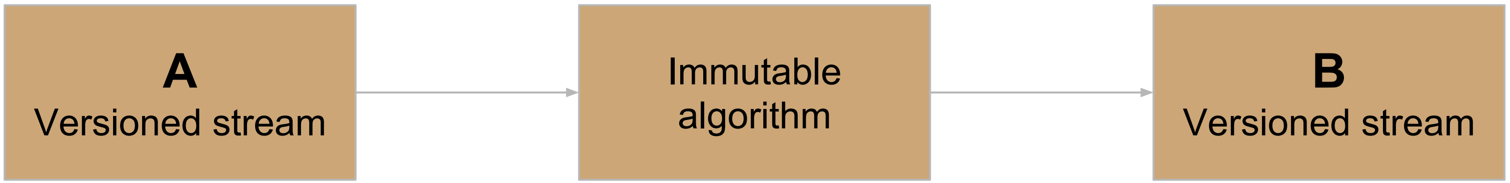


query latency [ms]
350
300
250
200
150
100
50
0
4.3 billion underlying points
2062 underlying points
2^{44} 2^{43} 2^{42} 2^{41} 2^{40} 2^{39} 2^{38} 2^{37} 2^{36} 2^{35} 2^{34} 2^{33} 2^{32} 2^{31} 2^{30} 2^{29} 2^{28} 2^{27} 2^{26} 2^{25} 2^{24} 2^{23}
resolution [ns per record]


query latency [ms]
350
300
250
200
150
100
50
0
4.3 billion underlying points
2062 underlying points
2^{44} 2^{43} 2^{42} 2^{41} 2^{40} 2^{39} 2^{38} 2^{37} 2^{36} 2^{35} 2^{34} 2^{33} 2^{32} 2^{31} 2^{30} 2^{29} 2^{28} 2^{27} 2^{26} 2^{25} 2^{24} 2^{23}
resolution [ns per record]

DISTIL - Eventually consistent derivative streams

- B = f(A)
  - Find differences in A since last update of B - ***CalculateDiff()***
  - Compute f(ΔA) and update B
  - If operation succeeds, update B's metadata with new version of A
- Sometimes keep A up to date all the time
- Sometimes materialize A just in time when needed



e.g Sliding window quartiles

algorithm, uuid(A), ver(A)

Summary

By leveraging qualities about the data

- Handle raw inserts/requests substantially faster than existing solutions (>16x faster than the new Cassandra C++ rewrite)
- Can analyse years of data in milliseconds, for a significant set of queries
- Aggregates are guaranteed to be consistent
- Can build elegant eventual consistency systems using multiversioning

Although simpler, relevant to a massive set of streams. Almost all physical quantity measurement quantities are scalar or vector-of-scalar streams.