How Our Time Series Database Works

April 2017
What is Time Series Data?

• Time series data is data you measure from something at points in time
  • EVENTS

• Normally labeled with a “metric” name, but not necessarily

• Time series data models vary; there isn’t one “true” time series datatype.

• Canonical example: stock ticker data
  • “The DJIA closed today at 19,238 on volume of 400 million shares”
Time Series Tradeoffs

• Time series data always originates from streams of events.
• If you’re like VividCortex and you’re measuring query events, it can be tens of thousands per server per second.
• What options are there?
  • Full event stream capture. Costly.
  • Aggregation. Metrics/aggregates discard some data.
  • Sampling. Also ignores/discards data.
  • Compression. Full data, but smaller; perhaps only approximate.
Full Event Streams

• The canonical example is “log everything and put into Splunk.”
• The storage and analytics over this tends to be extremely costly.
  • Far more costly than it’s worth.
• The usual tradeoff is sampling.
  • There are many ways to do this, all of them horrible
  • It’s 2017, why are we debating the best way to willfully ignore data?
• The “slow query log” is sampling. The criterion is e.g. “latency > 100ms”
  • This is not good at all. It ignores fast, frequent queries
Metrics/Aggregates

- The usual tradeoff in systems monitoring is to aggregate the events.
- The benefit is the results are small but represent the full original dataset
- If there were 24,617 queries in a second, {"qps", <timestamp>, 24,617}
- This is a type of compression, discarding redundancy
  - 24,617:1 compression ratio
- There are a couple of problems
  - You’ve aggregated, you can’t disaggregate anymore
  - At low throughput, you don’t gain a lot of compression
The VividCortex Solution

- We aggregate some stuff into metrics
- We capture samples of some other stuff
  - The sampling algorithm is non-trivial; see VividCortex’s ebook
- Some benefits of both worlds
VividCortex’s Time Series Data

• We measure machine data, primarily *query behavior*.
• For our purposes, time series data is a labeled, timestamped number
• The metrics can be e.g.
  • A value whose existence is known and expected, like CPU utilization
  • A value that’s not knowable in advance, like frequency of a new query

```
{"os.cpu.utilization", 2017-04-20 12:34:56, 34%}
{"queries.0xc0ffee.time_us", 2017-04-20 12:34:56, 181827}
```
Our Time Series Requirements

• We ingest and analyze A LOT of time series data.
  • Orders of magnitude more.
• We evaluated all the “standard” databases that are “good” for TS data.
• Our primary requirements are
  • Speed
  • Scalability
  • Efficiency
  • Fits the desired data model.
What’s Different About VividCortex?

- Many series. Tens of thousands per host/server. High cardinality.
- Sparse series. Series may exist for only 1 moment and never again.
- Categorized/related series that need to be analyzed together.
- High resolution. 1-second granularity, microsecond precision.
- Write-heavy
Typical TSDBs

- Most TSDBs make assumptions that aren’t true for VividCortex
- Little-to-no encryption, auditability, or access controls, not suitable for a SaaS service
- Optimized for “dense” metrics
  - Our workload has tons of sparse metrics
  - Non-trivial problem: “read all metrics that exist from point A to B”
- Assume low-cardinality, e.g. “each metric gets its own file in a directory”
- Optimized for “read metric A from timestamp B to C”, not much else
  - Our workload does a lot of top-K ranking across lots and lots of metrics
  - Many customers have ~10k-50k types of queries to rank
Our Architecture

- Multi-tenant SaaS architecture, with segregated tenancy
- Secure (in-flight and at-rest encryption)
- API-driven
- Sharded
- Robust and reliable: uses InnoDB transactional storage with crash recovery
Mapping Multi-Tenancy Onto MySQL

- Each tenant gets their own database
- Data is partitioned by time, but not with native MySQL partitioning
- Old data is purged with DROP TABLE, e.g. unlink()
- The schema design is crucial
  - InnoDB has clustered primary key indexes
The Central Time Series Data Problem

Written in columns

Read in rows

Time
How To Index?

• Index timestamp-first
  • Writes are efficient because you’re appending
  • “Read all data between A and B” is efficient
  • Single-metric reads are inefficient, a metric’s data is spread everywhere
  • Reading a large subset’s data, ditto

• Index metric-first
  • Reads are efficient because a metric’s data is densely packed together
  • Writes are penalized

• If you don’t index, reads are full-table-scan
  • Please study the Universal Scalability Law for how this ends :-(

Please study the Universal Scalability Law for how this ends :-(
#WhyNotBoth

MO’ INDEXES

MO’ PROBLEMS
Another Common TSDB Problem: Repetition

<table>
<thead>
<tr>
<th>Metric</th>
<th>Timestamp</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>os.cpu.utilization</td>
<td>1</td>
<td>34%</td>
</tr>
<tr>
<td>os.cpu.utilization</td>
<td>2</td>
<td>34%</td>
</tr>
<tr>
<td>os.cpu.utilization</td>
<td>3</td>
<td>34%</td>
</tr>
<tr>
<td>os.cpu.utilization</td>
<td>4</td>
<td>34%</td>
</tr>
<tr>
<td>os.cpu.utilization</td>
<td>5</td>
<td>34%</td>
</tr>
<tr>
<td>os.cpu.utilization</td>
<td>6</td>
<td>34%</td>
</tr>
<tr>
<td>os.cpu.utilization</td>
<td>7</td>
<td>34%</td>
</tr>
<tr>
<td>os.cpu.utilization</td>
<td>8</td>
<td>35%</td>
</tr>
<tr>
<td>os.cpu.utilization</td>
<td>9</td>
<td>34%</td>
</tr>
<tr>
<td>os.cpu.utilization</td>
<td>10</td>
<td>34%</td>
</tr>
<tr>
<td>os.cpu.utilization</td>
<td>11</td>
<td>34%</td>
</tr>
</tbody>
</table>
Vectorization

• Vectorizing chunks of metrics together reduces redundancy

{"os.cpu.utilization", 1, {34%, 34%, 34%, 34%, 34%, 34%, 35%, 34%}}

• This is somewhat annoying to query, but we accepted non-SQL API already
• It’s much more compact, without needing opaque compression
Metrics Categorization

- Our top-K ranking works across e.g. all “query.XXX.time_us” metrics
- Of which there could be 50k+
- Ideally we’d like these to be densely packed together in storage
- But our metric IDs are binary checksums, i.e. randomized, not sorted
- Thus, we label and prefix categories of metrics with a category ID
Time-Based Partitioning

• Time series data is immutable after a while, and stays static till purged
• Purging is ~continuous and needs to be really cheap
• We always access by time range, so partitioning by time is sensible
• We don’t use native MySQL partitioning because of locking
Schema Changes

• Schema changes are super painful at scale, so we don’t.
• We *version* our schema instead
• We maintain read compatibility for old versions
Time Series Schema, All Together Now

Table name: sprintf("observation_%d_%d_%d_%d",
version, granularity, beginTimestamp, endTimestamp)

Columns:

• host int unsigned
• metric bigint unsigned
• ts int unsigned
• points binary(60)
• Primary Key(host, metric, ts)
Sharding and Operations

- Sharding key is {customer, host, time range}
- A sharding directory service sits in front of a cluster of MySQLs
- Both the read path and write path consult this service
- The sharding directory service is also an Ansible inventory
  - To add a node, tell the directory service, then run Ansible
The Write Path

- Read-path and write-path APIs are separate for HA purposes
- Data comes into our APIs and goes into Kafka
  - Fallback #1: agents have a few retries
  - Fallback #2: encrypted write-only S3 buckets
    - Prioritize not losing data, even if it arrives late
- Kafka consumers write data into MySQL and Redis
  - The Redis cluster simply notes existence of a metric within a range of time
    - It is a coarse-grained “existence index” that’s cheap and fast
- We do not use MySQL replication
  - Kafka is our replication log
  - All MySQL shard servers are consumers of this log
Downsampling and Read Path

• We store 1-sec for 3 days, 1-min for 30 days, 10-min for 13 months
• This is configurable; premium accounts have 3-year retention
• Reads hit the lowest-resolution data they can
• If only high-resolution data exists for a time range they fill in from that
• The read path operates in parallel across shards
  • We handle tail latency with duplicate requests, ignoring late replies
Special Optimizations

- Our Top-K rankings are especially expensive, if done naively
- Remember, they run in parallel and data is sharded by time
- This means it’s a distributed Top-K problem, which is “hard”
- Many approaches are known to produce wrong results (see SCUBA paper)
- We use a patent-pending two-pass algorithm
  - Overly inclusive statistical pruning as a first pass approximation
  - Exact calculation over the pruned dataset as 2nd pass
Results

• Unit economics matter A LOT to us
  • Many monitoring companies go out of business because of low margins

• Extremely low-cost time series storage is a killer feature
  • It enables us to build high-resolution, high-cardinality monitoring
  • Other companies can’t do that because of their costs
  • Reducing cost dumbs down the solution till it’s non-functional

• We got raw efficiency, for sure!
  • We gave up some programmer efficiency and convenience
  • We had to treat MySQL like a dumb storage engine
  • We had to build our own time-series store on top of it
Alternatives

• We evaluated dozens of free and commercial offerings
• Cassandra, HBase, etc aren’t efficient enough
  • Many/most DBs aren’t efficient enough because “correctness”
• Most RRD-like TSDBs have assumptions that don’t work for us
• InfluxDB was too new and not yet able to handle our cardinality, etc
  • Looks very promising these days
• ElasticSearch might be a candidate
• Clickhouse, TimeScaleDB, CitusDB, etc didn’t exist yet
• Many of the above are possibly a fit now or in the future
@xaprb
baron@vividcortex.com