COMP5338 – Advanced Data Models

Week 12: Time Series Database

Dr. Ying Zhou
School of Computer Science

Outline

- **Time Series Data**
- **Facebook's Gorilla In-Memory TSDB**
- **InfluxDB Data Model and Storage**

Time Series Data

- "A **time series** is a series of *data points* indexed (or listed or graphed) in time order. Most commonly, a time series is a sequence taken at successive equally spaced points in time."
- Sample time series data
  - Financial data
  - Scientific data
  - Health and personal data
  - Engineering data
    - Computer Hardware, software monitoring data
    - Other devices monitoring data
  - Traffic data
  - Business data

Time Series Analysis

- Time series analysis has a long history in various disciplines
- There are established ways of running such analysis developed in those disciplines
- The purpose is to use past behaviors (observations) to predict future
- Two main approaches
  - Statistical time series analysis
  - Machine learning time series analysis

Time Series Database: Motivations

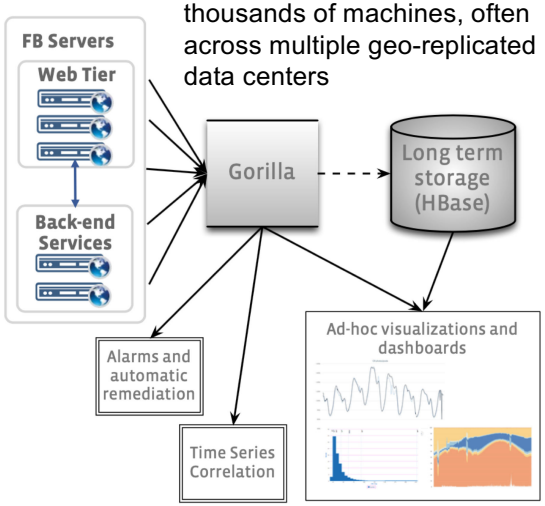
- Scalability
  - Data are collected and accumulated from a large variety of devices and in high frequency sometimes
  - Traditional database are not designed to handle data at such scale
- Relative standards set of functions and operations
  - Data retention policies
  - Continuous queries
  - Flexible Time aggregation

Outline

- Time Series Data
- **Facebook's Gorilla In-Memory TSDB**
  - **Motivation**
  - **Data Model**
  - **Compression Algorithms**
- InfluxDB Data Model and Storage

Facebook's time series data and management

thousands of individual systems running on many thousands of machines, often across multiple geo-replicated data centers



As of Spring 2015, Facebook's monitoring systems generate more than **2 billion unique time series** of counters, with about **12 million data points added per second.** This represents over **1 trillion points per day**.

Requirements of TSDB in Facebook

- Writes dominate
  - It should always be available to take writes. The write rate might easily exceed tens of millions of data points each second.
  - In contrast, the read rate is usually a couple orders of magnitude lower
- State transitions
  - Quickly identify the effect of new software, configuration, etc through fine-grained aggregation over short-time windows
- Highly available
  - Fail over to local storage when network partition happens
- Fault Tolerance/durability
  - Data Replication at multiple regions

What can be given up

- Users of monitoring systems do not place much emphasis on individual data points but rather on aggregate analysis.
- Traditional ACID guarantees are not a core requirement for TSDB
- Recent data points are of higher value than older points
  - Intuition: knowing if a particular system or service is broken *right now* is more valuable to an operations engineer than knowing if it was broken *an hour ago*.

Issue with HBase based solution

- HBase is an open source implementation of Bigtable
  - Optimize write performance at the cost of read performance
- With the increase of data size, the old monitoring system cannot scale to meet the read performance
- Issues observed
  - While the average read latency was acceptable for interactive charts, the 90th percentile query time had increased to multiple seconds
  - Queries of a few thousand time series took tens of seconds to execute
  - Larger queries executing over sparse datasets would time out
- Solution:
  - Design a memory based TSDB (Gorilla) to handle recent data
  - Older data are still persisted in HBase

Gorilla's Requirements

- 2 billion unique time series identified by a string key.
- 700 million data points (time stamp and value) added per minute.
- Store data for 26 hours.
- More than 40,000 queries per second at peak
- Reads succeed in under one millisecond.
- Support time series with 15 second granularity (4 points per minute per time series).
- Two in-memory, not co-located replicas (for disaster recovery capacity).
- Always serve reads even when a single server crashes.
- Ability to quickly scan over all in memory data.
- Support at least 2x growth per year.

Gorilla Data Model

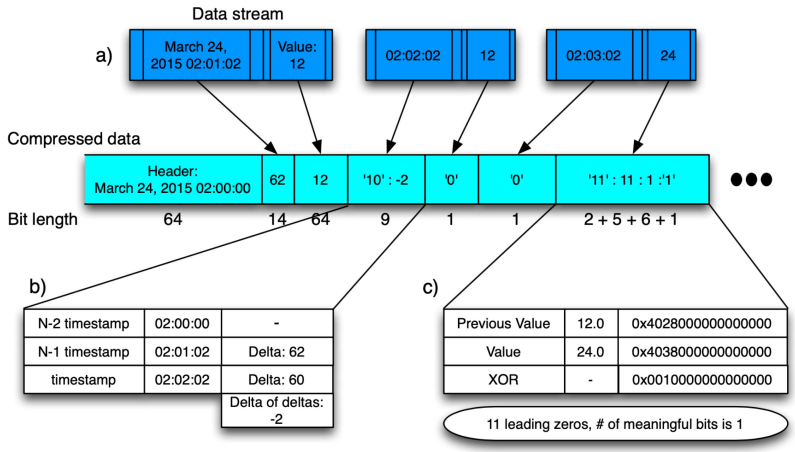
- The monitoring data is a 3- tuple
  - (**String** key, **64 bit** timestamp, **double precision** value)
- The **key** uniquely identifies a time series, e.g. the cpu usage of a web server *w1* in data center *c1*
- The key is used to shard the monitoring data,
  - Each time series dataset can be mapped to a single Gorilla host.
  - Very easy to scale
- At storage level each data point consists of only the tuple (**timestamp**, **value**)
  - Raw size is 16 bytes

Main Contribution of Gorilla

- Very efficient compression algorithm
  - Compress each data point down from 16 bytes to an average of 1.37 bytes, a 12x reduction in size.
  - Used in many other systems
- Efficient memory data structures to allow fast query
  - Fast and efficient scan of all data
  - Constant time lookup of individual time series

Compression Algorithm

- Compress integer timestamp and double precision value separately
- Compress time stream into blocks partitioned by time, e.g. 2 hours data is compressed in a block



Timestamp Compression

- Vast majority of monitoring data arrived at a fixed interval, with occasional slightly early or late data points
  - E.g. a time series to log a single point every 60 seconds. Occasionally, the point may have a time stamp that is 1 second early or late, but the window is usually constrained.
- Solution
  - Do not store timestamps in their entirety (64 bit)
  - Store must smaller ***delta of delta***
    - Difference of difference between adjacent timestamps
  - The block header stores the starting time stamp, t-1,which is aligned to a two hour window; the first timestamp, t0, in 14 bit, the block is stored as a delta from t-1, the rest are stored as delta of delta
- Result: 96% of all timestamps can be compressed to a single bit.

Value Compression

- Using compression scheme similar to existing floating point compression algorithms
- Simplify the implementation based on data feature observed
  - Neighbouring data points have similar values
  - Many data point are of integer type
- Simple property of binary representation of double precision value
  - If values are close together, the sign, exponent, and first few bits of the mantissa will be identical.
- Solution
  - Compute XOR of current and previous value
  - Only stores information about the non-zero bits in the XOR value

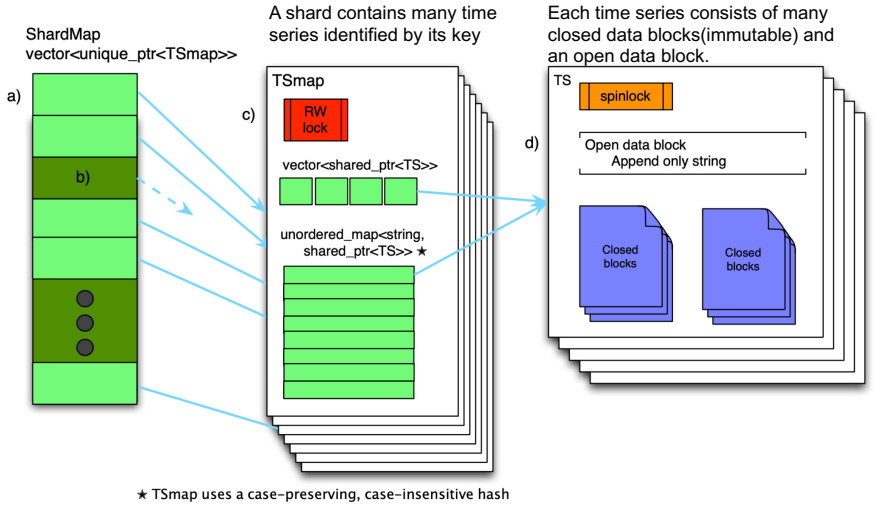
Property of XOR

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

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

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

In memory data structure



At top level, there is mapping from time series key to shard (TSmap)

Then there is mapping between key to the actual time series

Outline

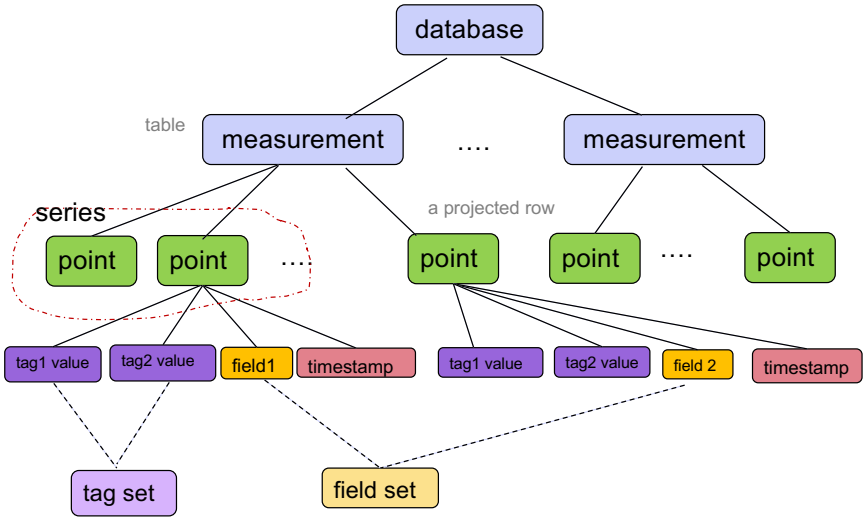
- Time Series Data
- Facebook's Gorilla In-Memory TSDB
- **InfluxDB Data Model and Storage**

InfluxDB data model

- A single time series is just a set of (*timestamp*, *value*) pairs
- A TSDB manages many time series, it needs a key to identify unique time series
- Gorilla uses a flat string key for that purpose
  - (String **key**, 64 bit **timestamp**, double precision **value**)
- At the highest level, InfluxDB data point is also a 3-tuple
  - (**Series Key**, **timestamp**, **value**)
- **Series Key** has *multiple* dimensions defined by
  - **Field** (required) to represent the actual measurement
  - **tag** (optional) to represent meta data
- Data points with the same series key form a **time series**
- Related **series** can be grouped as a **measurement**

https://docs.influxdata.com/influxdb/v2.0/reference/key-concepts/data-elements/#point

InfluxDB Key Concepts (1.x)



https://docs.influxdata.com/influxdb/v1.8/concepts/key_concepts/

Example Data

- System load (cpu and memory) monitoring data of two virtual machines
- Four series represented by four keys in **Gorilla** model
  - "`vm1.cpu`","`vm1.mem`", "`vm2.cpu`", "`vm2.mem`"
  - A particular data point look like ("`vm1.cpu`", **`1605181049, 0.65`**)
- One **measurement** with one **tag** and two **field**s in InfluxDB model
  - **Measurement**: **`load`**
  - **Tag**: vm with two values: "vm1" and "vm2"
  - **Field**: cpu and mem
  - A particular data point would look like

```
2020-11-12T22:00:00Z, load, vm="vm1", cpu=0.65
```

- The rich data model allows flexible query and aggregation

Gorilla vs. InfluxDB model

`vm1.cup:`

Timestamp	Value
1605181049	0.65
1605182049	0.63
1605183049	0.64
…	…

`vm1.mem:`

Timestamp	Value
1605181049	0.40
1605182049	0.49
1605183049	0.45
…	…

`load:`

time	VM	cpu	mem
1605181049	"vm1"	0.65	0.40
1605181049	"vm2"	0.71	0.58
1605182049	"vm1"	0.63	0.49
1605183049	"vm2"	0.64	0.45
…	…	…	…

Meta data is modelled as tag

Actual measurements are modelled as fields, multiple fields form a field set

A More Complex Example

- Assume we want fine grained monitoring data at service level, instead of VM level
- In Gorilla model, we may need to come up with different key strings: "`vm1.hdfs.dn.cpu`", "`vm1.hbase.rs.cpu`", …
- If we are monitoring the **cpu** and **memory** usage of 2 services on 2 virtual machines, there will be in total 8 time series
- In **InfluxDB** model, we need another tag **`service`** to indicate the services

The InfluxDB example model

time	VM	Service	cpu	mem
1605181049	"vm1"	"hdfs.dn"	0.35	0.10
1605181049	"vm1"	"hbase.rs"	0.30	0.40
1605181049	"vm2"	"hdfs.dn"	0.33	0.12
1605181049	"vm2"	"hbase.rs"	0.29	0.10
1605182049	"vm1"	"hdfs.dn"	0.34	0.20
1605182049	"vm1"	"hbase.rs"	0.25	0.13
…	…		…	…

Meta data are modelled as tags and multiple tags form a tag set

Actual measurements are modelled as field set

Another example

"the number of **butterflies** and **honeybees** counted by **two scientists** (langstroth and perpetua) **in two locations** (location 1 and location 2) over the time period from August 18, 2015 at midnight through August 18, 2015 at 6:12 AM"

```
Name: census
```

time	butterflies	honeybees	location	scientist
2015-08-18T00:00:00Z	12	23	1	langstroth
2015-08-18T00:00:00Z	1	30	1	perpetua
2015-08-18T00:06:00Z	11	28	1	langstroth
2015-08-18T00:06:00Z	3	28	1	perpetua
2015-08-18T05:54:00Z	2	11	2	langstroth
2015-08-18T06:00:00Z	1	10	2	langstroth
2015-08-18T06:06:00Z	8	23	2	perpetua
2015-08-18T06:12:00Z	7	22	2	perpetua

timestamp field set tag set

https://docs.influxdata.com/influxdb/v1.8/concepts/key_concepts/

Series

- In InfluxDB, a ***series*** is a collection of points that share a *measurement*, *tag set*, and *field key*.
  - E.g. time based data points belonging to the same feature of the same "entity"
    - cpu usage of hdfs data node process on vm 1
    - Stock price of a particular stock
    - Location of a particular car
  - In the insect counting data set, a *particular insect*'s count at a *particular location* made by a *particular scientist* form a series
  - There are in total 8 series: 2 (insects) x 2 (locations) x 2 (scientists)
  - The following is a data series representing *butterfly* count made by *Langstroth* at *location 1*

time	butterflies	location	scientist
2015-08-18T00:00:00Z	12	1	langstroth
2015-08-18T00:06:00Z	11	1	langstroth

Fields and Tags

- Fields contain the actual data, they are required pieces of the **`InfluxDB`** data model
- Tags contain auxiliary data to differentiate series, they are optional
- Tags are indexed but fields are not
  - Find out the time instances when butterfly count is greater than 10 will need full table scan
  - But such query is not meaningful and considered very rare
  - More meaningful one that involves field value would be: find out the time instances when butterfly number in location 1 as observed by perpetua is greater than 10

InfluxDB Storage

- InfluxDB's storage follows LSM tree design
  - Early version used LSM tree based engine like LevelDB, which optimized for write
  - It also used BoltDB, an engine based on memory mapped B+ tree, which is optimized for read
  - Eventually build a special LSM tree engine, called Time Structured Merge Tree.
- The main issue with general LSM tree based engine as identified by InfluxDB is delete performance
  - Data retention policy means large amount of data may expire at any time point
  - The original LSM approach to delete with tombstone marker makes it more expensive than other writes

InfluxDB Time Structured Merge Tree

- TSM tree is very similar to LSM tree
  - It has **WAL** to log the write query for durability and recovery
  - It has **cache** for recent updates
  - Data are stored as read-only files similar to SSTable format, called TSM files
    - Each contains data blocks
    - Data belonging to the same series would be saved in the same block(s) in timestamp order, the *series key* will be indexed, instead of incorporating it in every data point.
  - Most write processing are similar to LSM based engine
    - Only WAL and cache are updated
- Handling delete query
  - Write a tombstone file for each TSM file that contains relevant data. These tombstone files are used at start up time to ignore blocks as well as during compactions to remove deleted entries.

References

- Pelkonen, T., et al. (2015). "Gorilla: A fast, scalable, in-memory time series database." Proceedings of the VLDB Endowment **8**(12): 1816-1827.
  - Opensource project
    - https://github.com/facebookarchive/beringei
- **What the heck is time-series data (and why do I need a time-series database)?**
  - https://blog.timescale.com/blog/what-the-heck-is-time-series-data-and-why-do-i-need-a-time-series-database-dcf3b1b18563/
- InfluxDB data elements
- https://docs.influxdata.com/influxdb/v2.0/reference/key-concepts/data-elements/