Time Series Databases and Streaming algorithms
Introduction and motivation for Time Series
Financial
Internet of things
Domotics
Predictive Maintenance
Environmental tracking

CarbonTracker free troposphere CO₂
2008-Jun-06

[CO₂] μmol mol⁻¹
375 380 385 390
A time series is a sequence of data points, typically consisting of successive measurements made over a time interval.
Why Time Series Databases?

- High Volume of Data
- Large Quantities of Immutable Data
- Is Primarily Sorted Temporally
- Needs to Be Rolled Up To Gain Majority of Insights
- Needs to Be Normalized Across Multiple Time Zones

Problems using Relational DBs

1. It’s Difficult to Change the Sample Rate

2. It’s Difficult To Use SQL Queries For Analysis

3. Time Zones Add Extra Complexity To Your Data Analysis

Advantages of NoSQL

1. Greater simplicity in the DB engine
2. Ability to handle semi-structured and denormalized data
3. Potentially much higher scalability
Disadvantages of NoSQL

1. Higher complexity in the application
2. Loss of abstraction provided by the query optimizer
Basic Operations on Time Series Data
What do we need to do with TS

- Acquire
  - Measurement, transmission, reception
- Store
- Retrieve
- Analyze and visualize
Rescaling

• Transform the range of variation to a given scale
• Useful for algorithms sensitive to the magnitude of the signal
Resampling

• Differences in sampling resolution
• Bring both series to the same sample frequency
• Requires a function for collapsing points together
Shifting

- Align series we know are misaligned
- Bad reference time, drifting clock, ...
Slicing

• Retrieve a time series based on a given time range
Dynamic Time Warping

- Used for measuring similarity between series that vary in time or speed
- Dynamic time warping is a sequence alignment technique used in speech recognition
- It is an algorithm that has $O(n^2)$ complexity
Subsequence Matching

- A sequence query is matched against a longer TS
- Also related with Chunking where we look for repeating patterns
Statistical measures

- Mean
- Median
- Standard Deviation
- Variance
- Quantiles
Statistical fitting

- Interpolation
- Linear models
- Non linear models
Data Storage for Time Series Data
Log Files

• Simplest solution

• Right solution when low number of time series or data fits in memory

1950    1  0.92000E+00
1950    2  0.40000E+00
1950    3 -0.36000E+00
1950    4  0.73000E+00
1950    5 -0.59000E+00
1950    6 -0.60000E-01
1950    7 -0.12600E+01
1950    8 -0.50000E-01
1950    9  0.25000E+00
1950   10  0.85000E+00
1950   11 -0.12600E+01
1950   12 -0.10200E+01
1951    1  0.80000E-01
1951    2  0.70000E+00
1951    3 -0.10200E+01
1951    4 -0.22000E+00
1951    5 -0.59000E+00
1951    6 -0.16400E+01
1951    7  0.13700E+01
1951    8 -0.22000E+00
1951    9 -0.13600E+01
1951   10  0.18700E+01
Advanced Log Files

• Same concept about storing TS in files
• Use a smart binary encoding format
• Allows less processing, aka no parsing
• Stores data more efficiently for scan readings

message AddressBook {
  required string owner;
  repeated string ownerPhoneNumbers;
  repeated group contacts {
    required string name;
    optional string phoneNumber;
  }
}
Advanced Log Files

- Lots of binary formats lately
  - Thrift
  - Avro
  - Parquet

*We created Parquet to make the advantages of compressed, efficient columnar data representation available to any project in the Hadoop ecosystem.*
Relational Databases

- True and tested technology validated in multitude of scenarios
- Allows indexing out of the box
- Allows data replication and sharding (to some extent)
Relational Databases

- Use the Star Schema
- The fact table contains the measurements
- The dimension tables contain info about the series
Relational Databases

- The Star Schema can work reasonably to the hundreds of millions
- We can even implement the Star Schema in a NoSQL database
- When data grows this size several problems arise mostly related to the Star Schema itself.
Limitations of the Star Schema

- It uses one row per measurement
- Limitants of retrieval speed:
  - number of rows scanned,
  - total number of values retrieved
  - total volume of data retrieved
NoSQL databases

- Most of TS DBs use a NoSQL engine
  - OpenTSB → Hbase
  - InfluxDB → BoltDB
  - Prometheus → LevelDB
  - Newts → Cassandra
NoSQL databases

- Tall and narrow vs Short and wide table designs
- Short and wide denormalizes data
- Short and wide provides several advantages over the columnar data model
NoSQL databases

- Indexed by TS and timestamp the most common access pattern
- Retrieving data is an almost sequential reading from disk
Improvements over the Wide Table Design

- Collapse all the data into a blob
- Compress the blob so less data has to be read
- Allow coexistence of wide table columns and the blob
Improvements over the Wide Table Design

- Avoid the reads in order to overcome insert bottlenecks
- Create a fallback system in order to prevent failures
- Allow access to the in-memory data
Why not with RDBMs?

- Why use a RDBMs when you're not using any of its strong points?
- Also some features, ie. transactions, get in your way for scaling
Time Series Databases

- TempoIQ
- OpenTSDB
- Prometheus
- InfluxDB
- Druid
InfluxDB
Features

- Written in Go
- Using BoltDB as its internal storage engine
- SQL-like language
- HTTP(S) API for querying data
- Stores metrics and event data
- Horizontally scalable
A **series** is a collection of data **points** along a **timeline** that share a common **key**, expressed as a **measurement** and **tag set** pairing, grouped under a **retention policy**.
Key concepts

• **Measurement**
  • It is the value being recorded
  • Can be shared amongst many series
  • All series under a given measurement have the same field keys and differ only in their tag set

https://influxdb.com/docs/v0.9/concepts/key_concepts.html
Key concepts

- Tag
  - It is a key-value pair.
  - A measurement could have several tags
  - Tags are indexed
  - Both the key and value are strings

https://influxdb.com/docs/v0.9/concepts/key_concepts.html
Key concepts

• Point
  • A point is a single collection of fields in a series.
  • It is uniquely identified by its series and timestamp

https://influxdb.com/docs/v0.9/concepts/key_concepts.html
Key concepts

• Field
  • A field is a key-value pair
  • It records an actual metric for a given point
  • They are not indexed
  • They are required at least 1 on each point

https://influxdb.com/docs/v0.9/concepts/key_concepts.html
Key concepts

- **Database**
  - similar in concept to RDBS groups series

- **Retention policy**
  - defines what to do with data that is older than the prescribed retention policy

https://influxdb.com/docs/v0.9/concepts/key_concepts.html
Logging points into InfluxDB

```
{
    "database": "mydb",
    "points": [
    {
        "measurement": "cpu_load",
        "tags": {
            "host": "server01",
            "core": "0"
        },
        "time": "2009-11-10T23:00:00Z",
        "fields": {
            "value": 0.45
        }
    },
    {
        "measurement": "cpu_load",
        "tags": {
            "host": "server01",
            "core": "1"
        },
        "time": "2009-11-10T23:00:00Z",
        "fields": {
            "value": 1.56
        }
    }
    ]
}
```
HTTP endpoint

/query GET
/write OPTIONS
/write POST
/ping GET
/ping HEAD
/data/process_continuous_queries POST
Query exploration

Queries like in RDBMs

```
SELECT * FROM cpu
WHERE cpu_1 = '1'
```

Querying by time

```
SELECT mean(value) FROM cpu
WHERE time > 12345678s
GROUP BY time(10m);
SELECT mean from "hour_summaries".cpu
WHERE time > now() - 7d
```
Dealing with Time

- Querying using time strings

```sql
SELECT value FROM response_times
WHERE time > '2013-08-12 23:32:01.232' and time < '2013-08-13';
```

- Relative time

```sql
SELECT value FROM response_times
WHERE time > now() - 1h limit 1000;
```

- Absolute time

```sql
SELECT value FROM response_times
WHERE time > now() - 1h limit 1000;
```
Dealing with missing values

- Use null, previous, none for missing values

```sql
SELECT COUNT(type) FROM events
WHERE time > now() - 3h
GROUP BY time(1h) fill(null)
```

```sql
SELECT COUNT(type) FROM events
WHERE time > now() - 3h
GROUP BY time(1h) fill(previous)
```

```sql
SELECT COUNT(type) FROM events
WHERE time > now() - 3h
GROUP BY time(1h) fill.none
```
Write data

- Ingest data into InfluxDB using the HTTP API

- Create the Database
  
  ```bash
curl -G http://localhost:8086/query --data-urlencode "q=CREATE DATABASE mydb"
  ```

- Write data into the database
  
  ```bash
curl -i -XPOST 'http://localhost:8086/write?db=mydb' --data-binary '
  cpu_load_short,host=server01,region=us-west value=0.64
  1434055562000000000'
  ```
Hands on

- Import data from Standard&Poor
- Explore the performance of different encodings:
  - Several fields for a single point
  - Each column as a separate TS
- Create the following queries:
  - Select maximum opening price on a given period for each quote
  - Select the monthly average
Hands on (Advanced)

- Import extra dataset
- Compare loading and querying data between MySQL and InfluxDB
Streaming data
Algorithms for processing data streams in which the input is presented as a sequence of items and can be examined in only a few passes.
Examples
Examples: Anomaly Detection
Real Time Telemetry
Trends in Social Networks
Streaming algorithms
Characteristics of streaming algorithms

- Operates on a continuous stream of data
- Unknown or infinite size
- Only one pass, that allows following options:
  - Store it
  - Lose it
  - Store an approximation of it
- Limited processing time per item
- Limited total memory
These algorithms produce an approximate answer based on a summary or "sketch" of the data stream in memory.
They have limited memory available to them (much less than the input size) and also limited processing time per item.
Questions to answer

- Frequency moments
- Counting distinct elements
- Heavy Hitters
- Anomaly detection / Membership query
- Online learning
10^7 elements
10^6 distinct values
domain of 32-bit integers

40 MB
Raw Data

0.6 MB
Membership Query
with 4% error – Bloom Filter

4 MB
Exact Membership Query,
Cardinality Estimation – Sorted IDs or Hash Table

48 KB
Frequency of top-100 most frequent
elements with 4% error – Count-Min Sketch

14 KB
Top-100 most frequent
elements with 4% error – Stream-Summary

7 MB
10^6 pairs
{32-bit value, 24-bit counter}

2 KB
Cardinality Estimation
with 4% error – Loglog Counter

125 KB
Cardinality Estimation
with 4% error – Linear Counter

Exact Frequency
Estimation, Range Query – Sorted Table or Hash Map

https://highlyscalable.wordpress.com/2012/05/01/probabilistic-structures-web-analytics-data-mining/
Cardinality estimation
Linear Counting

```java
class LinearCounter {
    BitSet mask = new BitSet(m) // m is a design parameter
    
    void add(value) {
        int position = hash(value) // map the value to the range 0..m
        mask.set(position) // sets a bit in the mask to 1
    }
}
```

Load Factor is the ratio of distinct elements over the size m
## Cardinality estimation
### Linear Counting

<table>
<thead>
<tr>
<th>Formula</th>
<th>Description</th>
</tr>
</thead>
</table>
| \( \hat{n} = -m \ln \frac{m - w}{m} \) | \( \hat{n} \) - cardinality estimation  
  \( w \) - mask weight (a number of 1's)  
  \( m \) - mask size |
| \( \text{bias}(\hat{n}) = E(\hat{n}) - 1 = \frac{\varepsilon^t - t - 1}{2n} \) | This equation expresses a bias of the estimation (the ratio between estimation and true cardinality) as a function of the load factor and expected cardinality (or upper bound).  
  \( t \) - load factor, \( n/m \)  
  \( \varepsilon(.) \) - mathematical expectation  
  \( n \) - maximum cardinality (or upper bound, or capacity) |
| \( m > \max(5, 1/(\varepsilon^t)^2) \cdot (\varepsilon^t - t - 1) \) | A practical formula that allow one to choose \( m \) by the standard error of the estimation.  
  \( m \) - mask size  
  \( \varepsilon \) - standard error of the estimation  
  \( t \) - load factor, \( n/m \) |
Cardinality estimation

Loglog Counting
Cardinality estimation

Loglog Counting

class LogLogCounter {
    int H = H // H is a design parameter
    int m = 2^k // k is a design parameter
    etype[] estimators = new etype[m] // etype is a design parameter

    void add(value) {
        hashedValue = hash(value)
        bucket = getBits(hashedValue, 0, k)
        estimators[bucket] = max(
            estimators[bucket],
            rank(getBits(hashedValue, k, H))
        )
    }

    getBits(value, int start, int end)
    rank(value)
Frequency Estimation: Count-Min Sketch
Frequency Estimation: Count-Min Sketch

class CountMinSketch {
    long estimators[][] = new long[d][w] // d and w are design parameters
    long a[] = new long[d]
    long b[] = new long[d]
    long p       // hashing parameter, a prime number. For example 2^31-1

    void initializeHashes() {
        for(i = 0; i < d; i++) {
            a[i] = random(p)     // random in range 1..p
            b[i] = random(p)
        }
    }

    void add(value) {
        for(i = 0; i < d; i++)
            estimators[i][hash(value, i)]++
    }

    long estimateFrequency(value) {
        long minimum = MAX_VALUE
        for(i = 0; i < d; i++)
            minimum = min(minimum,
                           estimators[i][hash(value, i)])
        return minimum
    }

    hash(value, i) {
        return ((a[i] * value + b[i]) mod p) mod w
    }
}
Frequency Estimation: Count-Mean-Min Sketch

class CountMeanMinSketch {
   // initialization and addition procedures as in CountMinSketch
   // n is total number of added elements

   long estimateFrequency(value) {
      long e[] = new long[d]
      for (i = 0; i < d; i++) {
         sketchCounter = estimators[i][hash(value, i)]
         noiseEstimation = (n - sketchCounter) / (w - 1)
         e[i] = sketchCounter - noiseEstimator
      }
      return median(e)
   }
}

class CountMeanMinSketch {
    // initialization and addition procedures as in CountMinSketch
    // n is total number of added elements

    long estimateFrequency(value) {
        long e[] = new long[d]
        for(i = 0; i < d; i++) {
            sketchCounter = estimators[i][hash(value, i)]
            noiseEstimation = (n - sketchCounter) / (w - 1)
            e[i] = sketchCounter - noiseEstimator
        }
        return median(e)
    }
}
Heavy Hitters
Stream-Summary
Membership Query
Bloom Filter
Feature Hashing

- John likes to watch movies.
- Mary likes movies too.
- John also likes football.

\[
\begin{pmatrix}
  \text{John likes to watch movies} & \text{Mary also likes football}
  \\
  1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0
  \\
  0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0
  \\
  1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1
\end{pmatrix}
\]
Feature Hashing

```plaintext
function hashing_vectorizer(features : array of string, N : integer):
    x := new vector[N]
    for f in features:
        h := hash(f)
        x[h mod N] += 1
    return x
```

Can be extended to use signed hashing functions

Pros:
• Extremely fast
• No memory footprint

Cons
• There is no way to reverse features
Stochastic Gradient Descents
Apache Spark
Storage System

Program Model + Storage System
Hello cruel world

Say hello! Hello!
Problem with Iterative Algos

Disk I/O is very expensive
Opportunity for a new approach

- Keep data in memory
- Use a new distribution model

Spark
Lightning-Fast Cluster Computing
Spark Streaming
Resilient Distributed Dataset (RDDs)

- A distributed and immutable collection of objects
- Each RDD can be split into multiple partitions
- RDDs allow two types of operations:
  - Transformations (lazy)
  - Actions (non-lazy)
A sequence of RDDs representing a stream of data
DStreams
Windows
Windowing computations

- **hashTags**: A sequence of tags at time points t-1 to t+2.
- **sliding window**: Represents the moving time window over which computations are performed.
- **countByValue**: A function that counts occurrences of values within the sliding window.
- **tagCounts**: The output of the countByValue function, indicating the frequency of tag occurrences.
Stateful computations
DStream API

http://spark.apache.org/docs/1.3.1/api/scala/index.html#org.apache.spark.streaming.dstream.DStream
Hands on Streaming

• Start the Spark server
• Create a Job
• Run netcat
• Send
Hands on Streaming (Advanced)

• Implement Count-Log on basic Spark
Setup environment

• Prerequisite:
  
  – Install latest version of Vagrant
    
    https://www.vagrantup.com/
  
  – Install latests version of Virtualbox
    
    https://www.virtualbox.org/

• Create the Virtual Machine:
  
  vagrant init codezomb/trusty64-docker
  vagrant up

http://blog.scottlowe.org/2015/02/10/using-docker-with-vagrant/
Setup environment

- Log in into the VM machine
  `vagrant ssh`

- Install some Ubuntu packages
  `sudo apt-get update`
  `sudo apt-get -y install docker`
  `openjdk-7-jdk`

- Pull docker images
  `docker pull tutum/influxdb`
  `docker pull sequenceiq/spark:1.3.0`

http://old.blog.phusion.nl/2013/11/08/docker-friendly-vagrant-boxes/