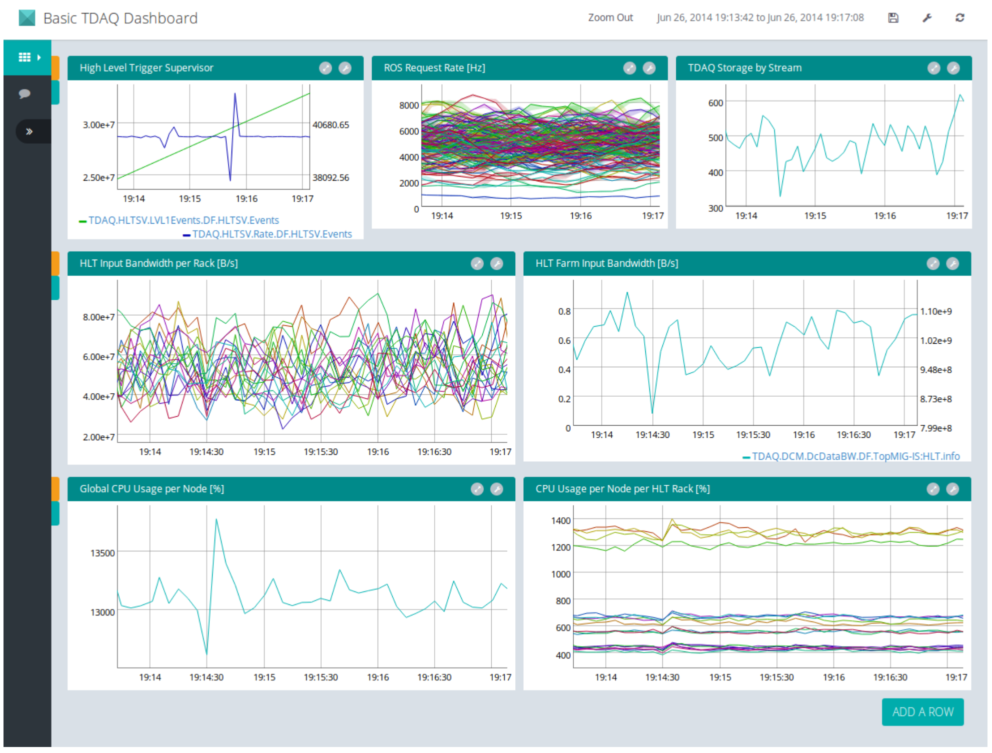
ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Autumn Semester 2014, IC-SIN

Start: 01.02.2013
Finish: 31.07.2014

Visualization of operational monitoring data for the ATLAS experiment

Florian Laurent


Basic TDAQ Dashboard
Zoom Out
Jun 26, 2014 19:13:42 to Jun 26, 2014 19:17:08
High Level Trigger Supervisor
ROS Request Rate [Hz]
TDAQ Storage by Stream
TDAQ.HLTSV.LVL1Events.DF.HLTSV.Events
TDAQ.HLTSV.Rate.DF.HLTSV.Events
HLT Input Bandwidth per Rack [B/s]
HLT Farm Input Bandwidth [B/s]
TDAQ.DCM.DcDataBW.DF.TopMIG-IS:HLT.info
Global CPU Usage per Node [%]
CPU Usage per Node per HLT Rack [%]
ADD A ROW

Professor: Anastasia Ailamaki (EPFL, IC IIF DIAS)
Supervisor: Giovanna Lehmann Miotto (CERN, PH-ADT-DQ)

Contents

Chapter 1 Introduction

1.1 ATLAS and the TDAQ System

ATLAS (A Toroidal LHC Apparatus) is the largest of several particle physics experiments built along the Large Hadron Collider (LHC) at CERN, Geneva. It is expected to address some of the unsolved questions of physics, advancing human understanding of physical laws. The Higgs particle was last year confirmed by data from the LHC [2].

The ATLAS Trigger and Data Acquisition (TDAQ) is in charge of selecting and transferring ATLAS experimental data from the detectors to mass storage. It uses over 3000 computers running more than 15000 concurrent processes to perform the recording. These processes produce monitoring data used for both inter-process communication and analysis of the systems. This data is exhaustively archived in order to perform offline analysis of data taking and to help investigating system behavior [3] [4].

The archiving system is called P-BEAST (Persistent Back-End for the Atlas Information System of TDAQ). It was developed and deployed in the second half of 2012.

1.2 Project Goals

The purpose of this project is to develop a visualization tool for the ATLAS monitoring data, stored in P-BEAST.

A wealth of monitoring tools are currently available and could potentially be used for this task. ElasticSearch, Graphite and OpenTSDB are the most common, however they do not precisely match our needs. These are typically two-tiered systems, with a backend storing the data and making it available for retrieval, and a frontend displaying it to the users. In our case, we already have P-BEAST acting as the backend. However P-BEAST is not suitable to be used directly from a web frontend: it is too slow, and contains more data than a browser could handle.

A custom system is therefore needed. The questions we will answer are:

- What are the requirements of the system we need to build?
- Could ElasticSearch, Graphite, OpenTSDB or other existing tools be adapted to our needs?

The solution chosen in the end is a custom system reusing parts of Grafana, the Graphite frontend, and storing data in memory in a custom high-performance backend acting as a P-BEAST cache. Multiple strategies were considered to store the data in memory, in the end the Redis database was found to be the most appropriate solution.

The LHC is currently in a ”Long Sleep” phase, undergoing maintenance for nearly 2 years. P-BEAST and the visualization system are parts of the upgrades planned for when the LHC comes back online. They will be used to monitor data both in the control room, with live data, and outside, to analyze archived data.

1.3 Report Structure

This report is divided in five parts, that follow the chronological development of the project.

The first part documents the requirements for the final software, as well as the study and comparison of the existing tools for the task at hand.

In the second part the implementation of the caching backend is described. The backend reads, pre-processes and stores in memory the data from P-BEAST requested by the users.

The third part describes the frontend architecture. The web-based frontend lets the user select, visualize and organize in dashboards data archived in P-BEAST.

The fourth part is a conclusion and mentions ideas for future work.

Finally the fifth part is a short user guide on how to deploy the system.

Part I

Context

Chapter 2 Requirements

2.1 Existing Infrastructure

2.1.1 Information System

The Information System (IS) is used to gather information about the TDAQ applications running in the experiment [41]. It runs on a multitude of dedicated machines. IS clients can publish data to the IS repository, and read value or subscribe to changes for specific objects. The IS objects are typed, using C++ basic types and arrays of those. The updates are published with microsecond precision timestamps. The communication between the IS repository and its client is implemented on top of CORBA [42]. P-BEAST stores exhaustively the information published to IS [32].

2.1.2 P-BEAST

P-BEAST was made specifically for the archival of IS data, and is still under active development. It evolved significantly during this project, in part because of this project's specific needs.

It uses a custom binary data format based on Google's Protocol Buffers [33] for performance and compactness reasons [9]. Initially the only way to access this data was to parse the output of the P-BEAST command lines tool. This obviously was neither efficient nor elegant. One of the major P-BEAST update was the implementation of a Java API built on top of CORBA. This made data retrieval much more efficient and reliable.

Another major change was the implementation of a P-BEAST *service* API. Initially it was necessary to have access to the archive files to read them. This was unpractical since the machine where the archive is recorded and stored is usually under a heavy load when live data is coming in. With the service API the data can be accessed remotely. It is now also possible to access the data that has been received by P-BEAST but hasn't been archived to disk yet. This happens transparently, and make it possible to access data less than 30 seconds after it was published.

Each IS object is an instantiation of a class within a data taking partitions that contains different attributes. The concatenation of the elements of this hierarchy makes up what we will call the ID of an object. For example this is one of the requests made on the test dashboard: *ATLAS.DCM.DcDataBW.DF.TopMIG-IS:HLT.info*

This request will read the attribute called "DcDataBW" of class "DCM" for the instance "DF.TopMIG-IS:HLT.info" in the "ATLAS" partition.

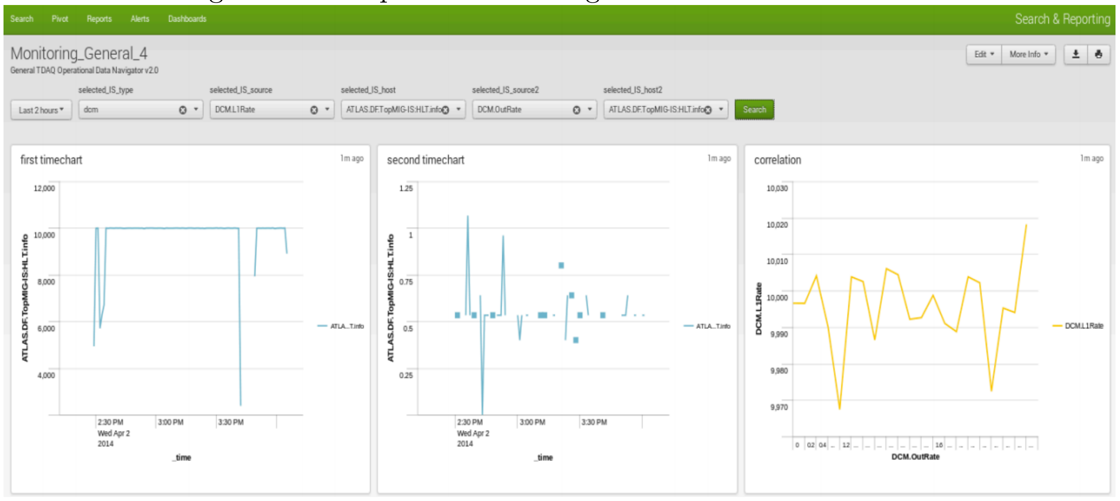
2.1.3 Splunk

Splunk is a professional tool to "search, monitor, analyze and visualize machine data" [7]. It generates dashboards, graphs, reports and alerts. It is developed by a multinational corporation of the same name, with over a thousand employees.

It is very complete, and is useful to both casual users and experts. Non-technical users can take advantage of the visual options and configurations. Experts can use its rich query syntax, with over a hundred search commands available.

Recent versions use a MapReduce mechanism to retrieve and analyze data sets [39]. As a result it scales very well with the number of nodes. On a single (modest) server it is able to index up to 100 GB per day [40].

Figure 2.1: A Splunk monitoring dashboard used at CERN



Splunk as it is used today has two major drawbacks.

First, the events need to be imported before they can be analyzed. This importing step is fast, but Splunk will consequently store a duplicate version of the data in its own custom format. This is problematic from a storage point of view.

The other problem is the licensing cost, which grows unreasonably large for the volume of data published to IS. For now tests are conducted using bandwidth-limited trial versions of the tool.

A way to mitigate both of these issues is to filter the data fed to Splunk to the subset that is the most interesting to analyze. This is impractical and limiting, which is why a new custom tool is needed.

It was also noted that Splunk is not always appropriate for scientific data. It is mostly targeted at business intelligence, and the way it handles data can sometimes be problematic. For example it makes debatable choices when downsampling time series.

The evaluation of Splunk for the TDAQ system is still ongoing.

2.2 System Requirements

Our visualization tool must fit in this infrastructure. The interface should be browser-based and allow dynamic exploration of the data.

The data should be displayed using a dashboard system, similar to what Splunk provides.

An instance should be accessible from the restricted network in which the ATLAS detector is running. As the number of servers running there is limited, the system should rely on a single powerful server rather than on a cluster of commodity machines.

Another requirement is that unlike Splunk, the system should not need to duplicate data that is already archived by P-BEAST. Ideally all the operations should be done in memory.

Of course not all the data can be loaded in memory, so at any time the backend only holds the data that was recently requested while older data is evicted. In that sense the backend acts as a P-BEAST cache. The complete archives takes terabytes of disk space, but only a small portion of it will be looked at.

Efficient support for sparse time series is critical. It is common to not receive data for days and this shouldn't take prohibitive space. The data is sparse because experiments are run at irregular intervals, and between those no data is recorded at all.

The system should be able to display numeric data. While the IS data also contains strings and arrays, it was established from the start those were not a priority. The users are usually interested in the evolution of values such as CPU load, number of events or bandwidth used that can be plotted as a line chart. (Eventually limited support was also implemented for strings and arrays as well, using pie charts and array flattening.)

A time precision of one second was initially supposed to be sufficient. This was a cause of problems as in practice data often changes at millisecond intervals. The system was later adapted to this higher time granularity.

In order to allow dynamic exploration of the data, the system needs to ingest and process the IS data efficiently. The initial constraints was to handle at least 300 thousand data points per second. The system also needs to handle up to a hundred concurrent users.

A test dashboard displaying data from 15 different objects was established to check how the system would react with a realistic workload. A ”stress” dashboard, simulation requests from a hundred users, was used for stress-testing.

The target machine used for development and testing is called *pc-atd-cc01*. The CPU is a 24 cores Intel Xeon X5690 running at 3.47GHz. The system has 24 GB of main memory.

Chapter 3 Initial Research

A number of time-series database solutions are available. Looking at our requirements, three solutions stand out: ElasticSearch, Graphite and OpenTSDB. These software provide end-to-end solutions: they store the data and provide visualization on demand.

We do not need persistent storage since the data we consider is already archived by P-BEAST. However P-BEAST can't be used directly from a web interface. There are two reasons for this:

The first problem is that retrieving large amount of data with P-BEAST is not fast enough. This is because P-BEAST doesn't use a time index for the stored series. It uses indexes to efficiently access a given *object* in a file, so this operation is fast. But when reading a specific range out of a large time series an overhead occurs. Additionally, P-BEAST loads the whole requested range in memory before writing it out. Retrieval is implemented this way because disk usage is more important for P-BEAST than speed.

The second issue is that the amount of data stored with P-BEAST is orders of magnitude too large to be handled by a web browser. JavaScript is used to load and display the data. Even with the impressive recent improvements in JavaScript performance, it is still not possible to load millions of data points while keeping the browser responsive. Therefore it is necessary to downsample the data on the server side.

Ideally, we would want a time-series database that handles downsampling without needing to persist the data to disk, ie keeping everything in memory. We are also looking for a complete and configurable dashboard system to visualize that data.

3.1 ElasticSearch and Kibana

ElasticSearch is a search server based on Lucene. It provides a distributed full-text search engine with a REST API. It is developed in Java and released as open source under the Apache License [14].

On paper it looks like the perfect tool for the job. It is often mentioned as a Splunk open-source alternative [15]. Its index can be kept fully in memory, and it comes bundled with Kibana, a very complete and elegant frontend [49]. It also provides a bulk import API that should have been useful to load data from P-BEAST.

As we will see ElasticSearch unfortunately gives disappointing results for our use case.

3.1.1 ElasticSearch

The issue with ElasticSearch is its data ingestion rate. A lot of time was spent optimizing ElasticSearch's configuration and the mappings of its documents to speed things up. Ultimately the results remained too slow.

Importing data

ElasticSearch comes bundled with LogStash, which is used to collect and format the events to be recorded. It quickly appeared LogStash incurs unnecessary overhead: it is faster to directly import the data in ElasticSearch using a custom script and the bulk API [34]. To test the insertion speed, a Python script was used to read data from P-BEAST and format it in the JSON structure appropriate for bulk import, then the *curl* tool was used to perform the insertion.

Shards and Replicas

Two of the parameters that affect ElasticSearch's performance are the number of *shards* and *replicas*. Shards are under the hood Lucene instances [35]. They are "worker" units that are managed automatically be ElasticSearch. Replicas are copies of shards. The rule of thumb is that having more shards increases the indexing performance while having more replicas increases search performance.

Mappings and Index Settings

A "mapping" is to ElasticSearch what a "schema definition" would be to relational databases. Similarly an "index" corresponds to a database, and a "type" to a table. A mapping defines the types of an index, and possibly index settings.

Multiple solutions were tried as far as mappings are concerned. The timestamp was stored either as a *double* or as a *date*. The value was stored either as a *string* or as a *float*. The second solution would be more flexible but the first is sufficient given our requirements.

Benchmarks

The tests were performed by inserting 10 million points. The results are the average over 3 runs. The tests were run on the pc-atd-cc-01 machine. No replication was used since we are only looking at insertion for now. The default mapping stores the timestamps as *double*s and the values as a *float*s. The optimized mapping stores the timestamps as *date*s and the values as a *string*s.

Number of shards	Mapping	Importing speed
2	default	11.7 K/s
4	default	14.5 K/s
8	default	32.0K/s
8	optimized	34.6K/s

Table 3.1: Test results

These results are very far from the minimum ingestion rate of 300K data points per second.

Translog and Segment Merging

Two other parameters that were studied but did not bring any significant improvement were the transaction log flush interval, and the segment merging policy.

Each shard has an associated write ahead log, also called transaction log (referred to as *translog* in the configuration) [37]. The translog is flushed ("commited") when some conditions happen, for example after a given time or number of logged operations. Modifications of the translog flush values didn't have any significant impacts on the importing speeds.

The shards, being Lucene indexes, are broken down into segments. These segments are sections of the index that are periodically merged into larger segments to limit the index size [36]. Tuning the index merging policy brought no significant improvements.

In-memory Index

It is possible to configure ES to keep the index in memory. This means the data is not stored on disk, as per our requirements. The file system is then only used for the transaction logs [16]. Unfortunately, this improved ingestion rate only marginally.

The problem is that ElasticSearch is fundamentally too complex for what we are trying to do. It can run advanced aggregation queries, can perform full-text search on large corpus or be used as a recommender system. It is also a fully distributed system meant to be run on multiple nodes and handling node failure. This is way beyond what we need! Some of these extra features would be useful but they come with too high of a performance price.

3.1.2 Kibana

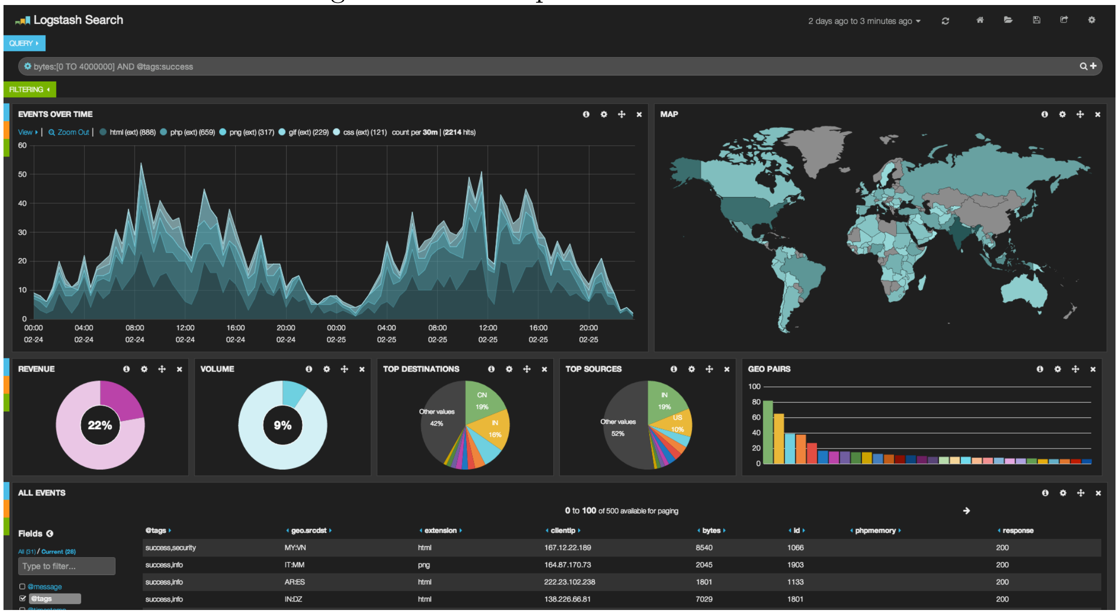
The Kibana dashboard is very impressive. It can display many types of panels, such a bar and line charts, gauges, and even maps.

It is not completely user-friendly, in the sense that it often works in unpredictable ways. For example, clicking on a country on a map filters the data displayed on the whole dashboard. How to remove this filter to see the complete data set again? An intuitive way would be to click on the same country again. This doesn't work, and doesn't seem to do anything. Another solution would be to click on another country. But then all the data disappears!

What actually happened is that clicking on a country added a filter. But clicking on the same country again added this filter again. Clicking on another country added a filter that excluded any data from the visualized data set.

This is just an example but is representative of the major problem with Kibana: it looks good, but it is confusing to use.

Figure 3.1: A sample Kibana dashboard



```
bytes:[0 TO 4000000] AND @tags:success
```

@tags	geo.srcdst	extension	clientip	bytes	id	phpmemory	response
success,security	MY:VN	html	167.12.22.189	8540	1066		200
success,info	IT:MM	png	164.87.170.73	2045	1903		200
success,info	AR:ES	html	222.23.102.238	1801	1133		200
success,info	IN:DZ	html	138.226.66.81	7029	1801		200

3.2 Graphite and Grafana

3.2.1 Graphite

Graphite is a popular real-time graphing system typically used to monitor application metrics, such as database performance [17]. It scales well and runs efficiently on modest hardware [18]. Initially an internal monitoring tool for the travel booking site Orbitz, Graphite is now used by heavyweights such as Etsy, Sears and Canonical.

Rollup Aggregation

A major difference with ElasticSearch is that the precision at which Graphite stores the data points decreases with time. Graphite is meant to store, for example, data with a resolution of 1 second up to an hour, 60 seconds up to 24 hours and 1 hour up to a month. The data points are regularly "rolled up" from one file to another of lower precision. The rollup aggregation function can be set to either *min*, *max*, *average* or *last*.

These time granularities need to be specified at startup and it is non-trivial to change them afterward. It is also impossible to select a granularity of less than a second.

Underlying Database

Graphite initially used the RRD file format. This is the format used by RRDTools, a high performance data logging and graphing system for time series data [19]. It ran into limitations of this format and subsequently switched to *Whisper*, a custom made database [17]. Whisper improves data storage in two ways: multiple points can now be written to disk at once, and points saved irregularly are better handled (but are still limited by the time granularity!).

Whisper is meant to store data at a regular interval in a systematic way, and it will use up space whether a point is recorded or not. This means a time series with three data points spread across a year will take as much space as a full year of recorded data.

An alternative custom database called *Ceres* that supports sparse data is currently in development, but it is not yet ready to be used in production.

Issues

Graphite would be difficult to adapt to our needs. One problem is that as part of our requirements we need to handle sub-seconds timestamps. This would require in-depth modifications of Graphite's internal. The other problem is the lack of mature support for sparse data.

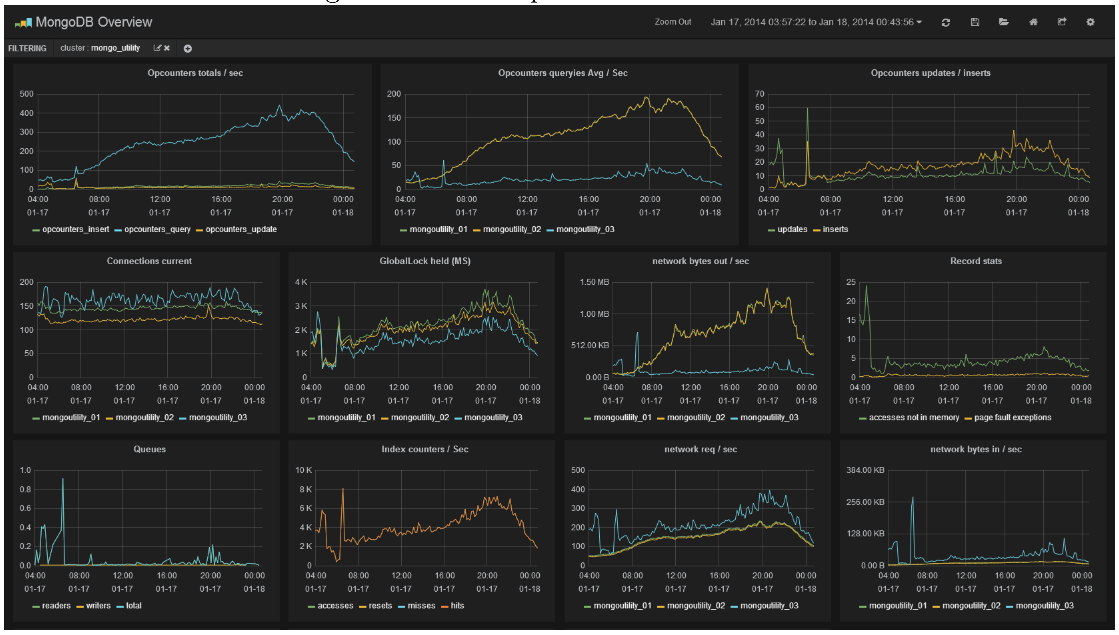
These issues rule out Graphite as a suitable backend.

3.2.2 Grafana

Grafana is a visually impressive dashboard system for Graphite [50]. It is a fork of Kibana, that kept its good look and added some common sense.

Grafana actually removed a lot of features from Kibana. The only panel available is the line chart. The panels can be moved around using drag and drop, the charts can be dynamically panned and zoomed. The whole dashboard is configurable from the web interface, and each chart can be styled independently.

Figure 3.2: A sample Grafana dashboard



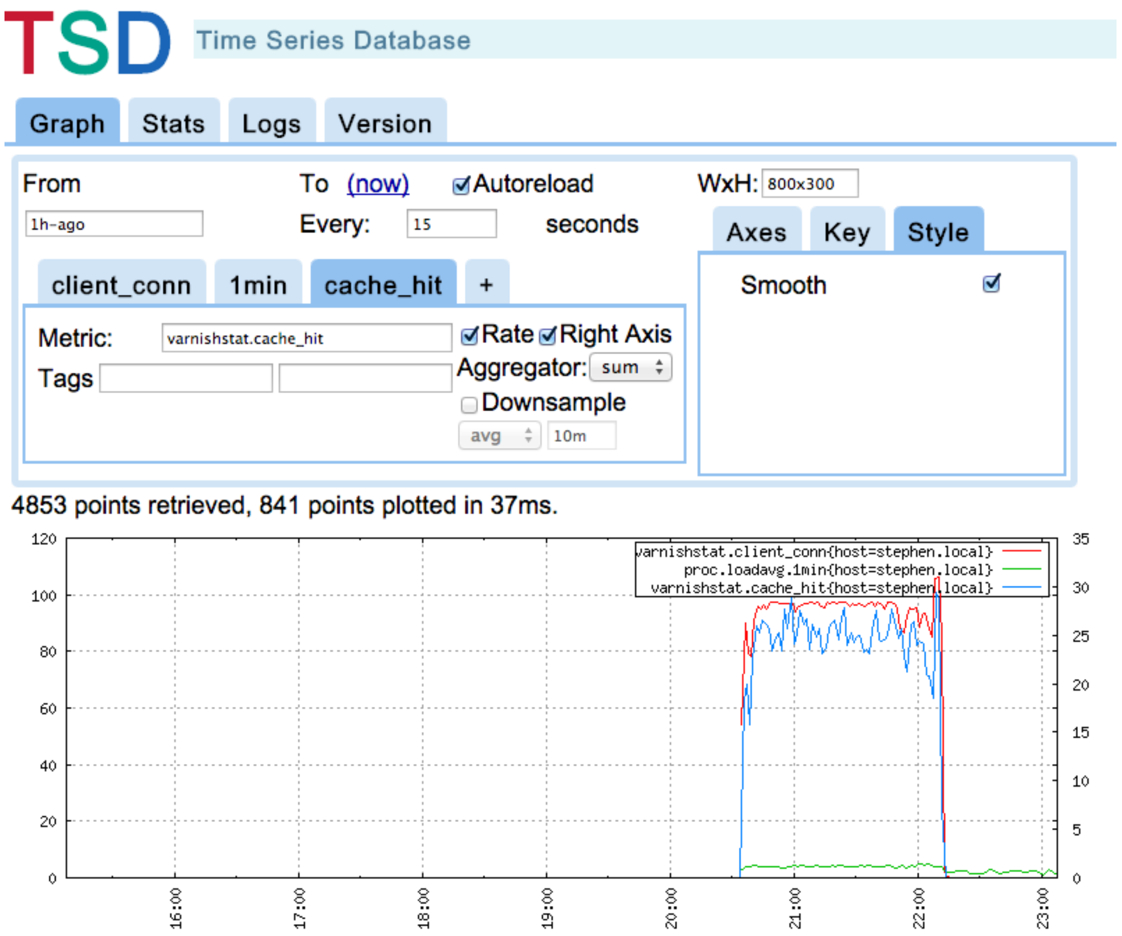
3.3 OpenTSDB

> "OpenTSDB is a distributed, scalable Time Series Database (TSDB) written on top of HBase. OpenTSDB was written to address a common need: store, index and serve metrics collected from computer systems (network gear, operating systems, applications) at a large scale, and make this data easily accessible and graphable." [20]

After ElasticSearch was considered, OpenTSDB was quickly discarded as it has the same issues. OpenTSDB runs on top of HBase, itself running on Hadoop. Just as ElasticSearch, it goes to great length to store the data to disk efficiently and reliably. Again this is useless for use and comes at a major performance cost.

OpenTSDB's frontend is clean and efficient, but it is too simple for what we want to do.

Figure 3.3: A sample OpenTSDB chart


TSD Time Series Database

Graph | Stats | Logs | Version

From: 1h-ago
To (now)
✓ Autoreload
Every: 15 seconds
WxH: 800x300

Axes | Key | Style
Smooth ✓

client_conn | 1min | cache_hit | +

Metric: varnishstat.cache_hit
Tags:
✓ Rate ✓ Right Axis
Aggregator: sum
☐ Downsample
avg 10m

4853 points retrieved, 841 points plotted in 37ms.

```
varnishstat.client_conn{host=stephen.local}
proc.loadavg.1min{host=stephen.local}
varnishstat.cache_hit{host=stephen.local}
```

Chapter 4 Chosen Solution

4.1 Overview

The conclusion of our initial research is that none of the existing solutions fulfills our requirements out of the box.

We need a backend that ingests points faster than anything we have considered, and that keeps everything in memory instead of creating a duplicate of the existing data on disk. For the frontend, Grafana is the best candidate and will make a good starting point.

A custom backend was implemented from scratch. It is a Java application that acts as a cache for P-BEAST. It loads IS data on demand and stores it in memory using either a Java skip list or the Redis in-memory database. Its purpose is to process the IS data so that the frontend will be able to retrieve and render it efficiently. For this purpose it downsamples the raw data, performs data aggregations to render pie charts and more.

A custom frontend was implemented based on Grafana. The graph rendering engine was changed to a more performant one. Support for pie charts was added. The user interface was completely redone to be more user-friendly and to make space for more features. Other deep modifications were performed to accommodate the Grafana code to the new backend. In the end it was so extensively modified it doesn't look or feel like Grafana anymore. It communicates with the backend using a JSON API.

The next two parts will describe in details the inner workings of the implemented solution, first its backend then its frontend.

4.2 Recent Developments

This initial research was conducted in the first months of 2014. Since then, the landscape dramatically changed.

Graphite doesn't rely on an RDD-style database anymore: the Ceres engine has been polished and is now used as the default. The data precision is still limited to one second. Grafana, the Graphite web frontend, gained significant traction. It is now being used as a frontend for OpenTSDB as well as for InfluxDB, a new time-series database. InfluxDB is written in Go, and was created from scratch by Y Combinator alumni for their own company. An interesting aspect is that their implementation of downsampling, querying and insertion is completely decoupled from the database engine they are using. LevelDB and LLDB are exemples of databases that can be used to handle the actual data persistence. OpenTSDB made no significant change and seems to be losing speed to InfluxDB. ElasticSearch is gaining traction as well, and Kibana seems to be more user-friendly now. Its parent company raised $70 millions in a series C funding round in June.

In definitive, despite the impressive evolution of these tools in the past 6 months, the choices made during the design phase of the projects are not put into questions. These tools evolved but they remain incompatible with our requirements.

What could be interesting would be to setup InfluxDB with Redis as a database engine. Indeed, InfluxDB already provides better querying features than our custom backend, and it has already been natively integrated with Grafana.

Part II

Backend

Chapter 5 Overview

Our research for a backend that could ingest data quickly and keep it in memory was unfruitful. ElasticSearch is too slow. OpenTSDB is too complex to setup, Graphite doesn't handle sparse data well, and both of those don't support in-memory storage anyway. A custom solution is needed.

Looking back at our requirements, which ones apply to the backend?

- High ingestion rates
- Fast retrieval
- Support for sparse time series with millisecond timestamps
- Support for numeric values, as well as arrays of numeric values

The custom backend is made of three parts.

The first is an in-memory data store. We will consider multiple solutions for the store, to make their comparison easier their implementations use a common interface and we can decide at start-up time which one we want to use.

A server part acts as the interface between the backend and the rest of the world. It provides a JSON REST API to be used by the frontend, and serves the static pages of the frontend itself. We use the Jetty HTTP server for this [25].

Most of the logic happens in the middle part, which reads the data from P-BEAST, pre-processes the time-series before caching them, and handles data aggregation as requested by the clients. It is also in charge of listing the various partitions, class, attributes and objects for which we have data in order to provide autocompletions to the frontend. The P-BEAST API is evolving, so its interface has been abstracted as well. At this point it is possible to read data from the command-line output of P-BEAST tools, from the Java API or from a static file.

As described in the Requirements (section 2.2) we rely on a single powerful server rather than on a cluster of commodity machines. To take advantage of the high CPU counts of the server, the two most CPU intensive tasks can be spread across an arbitrary number of threads.

The first task is the parsing and downsampling of imported data. Parsing is no longer a bottleneck now that a Java API is available for P-BEAST. Downsampling however is a CPU-intensive operation. We will see how it is handled in section 7.1. The second task is database insertion. The downsampler adds the points to be inserted in a central asynchronous queue. From there multiple threads add these points to the data store.

Finally it should be noted that since the loaded series are kept exclusively in memory, restarting the backend means losing them all. The only information persisted to disk is the list of series that have been loaded, along with some meta-data.

Chapter 6 Time Series Persistence

6.1 Data Stores

Multiple solutions were considered to keep the series in memory. In order to compare their performance the data store is abstracted from the rest of the system. This way it is possible to benchmark each solution individually to select the most appropriate one for each situation. The data store selection is done at start-up time.

As we will see, Redis is the best solution for the production setup. Redis let us store data in an efficient an scalable way. The second choice is to use a Java skip list. This works well and is actually faster when the amount of data to store is limited (up to 2 GB), but fails to scale beyond that point due to garbage collection performance limitations.

6.1.1 Java Skip List

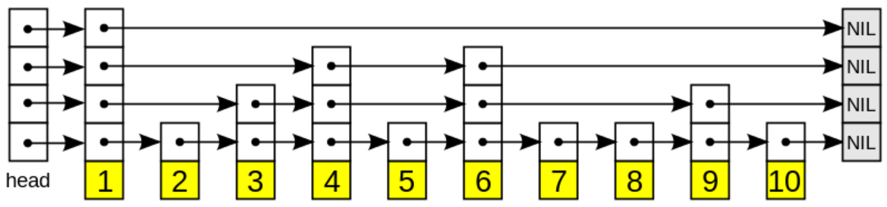
The obvious way to store the time series is in some Java list type. There are a number of constraints this list type should fulfill:

- Fast search in an ordered sequence, to handle time range queries.
- Fast ordered insertion, to support high data point ingestion rates.
- Thread safety. For performance reasons the list will be accessed and modified by many concurrent threads.

A skip list appears to be a good candidate. A skip list allows search within an ordered sequence with *O(log n)* performance (with n the number of elements retrieved). Ordered insertion is also performed in *O(log n)* of the number of inserted elements [21]. This is achieved by storing a hierarchy of ordered subsequences, each skipping over fewer elements, as seen on figure 6.1. A concurrent implementation is part of the Java *java.util.concurrent* package.

The Java *java.util.concurrent.ConcurrentSkipListMap<K, V>* class provides an efficient and concurrent implementation of skip lists. One *ConcurrentSkipListMap* is used per time series, and these the series are additionally indexed in a *ConcurrentHashMap*, mapping a series ID to its skip list.

Figure 6.1: Example skip list showing the hierarchy of subsequences [5]



A downside of this implementation is that the size of a list can't be computed in constant time. As the Java documentation warns:

> "Because of the asynchronous nature of these maps, determining the current number of elements requires traversing them all to count them."

As we will see when describing the downsampling mechanism (section 7.1), to retrieve data for a given time range we count the number of points at each downsampling granularity until we find a suitable one. The counting complexity incurs a delay in this process.

Support for skip lists is handled in the *beast.store.impl.skip* package. Multiple benchmarks were run to test its speed while inserting, retrieving and flushing data points.

The results are good as long as less than 2 GB of data is stored. Beyond this point, the Java garbage collector starts to incur significant slowdowns. This is even more problematic when large part of the heap is modified in a short amount of time, which is the case when long time series are imported. [6]

This makes a native Java skip list a good match in situations where a limited amount of system memory is available. Not only are the performances good, this also means the system can run with no dependencies besides a JVM.

In the final system it is possible to use the skip list based store by giving the *-store skip* argument at startup. See the running guide (section 15.3) for more details. This store is used by default if the system memory is less than 2 GB.

6.1.2 Redis

Redis is an in-memory key-value data store, implemented in ANSI C. The name comes from "REmote DIctionary Server". At this time it is the most popular key-value store [28].

As opposed to other structured storage systems Redis supports abstract data types such as sets, sorted sets and hashes. In that sense it is sometimes referred to as a "data structures server" [26] [27].

The sorted set data type is the most advanced one and is suitable for time series. The data is organized in sets, with each set element having an associated "score" used as an index to order it. For time series the timestamp is used the score.

The implementation is efficient:

> ”With sorted sets you can add, remove, or update elements in a very fast way (in a time proportional to the logarithm of the number of elements). Since elements are taken in order and not ordered afterwards, you can also get ranges by score or by rank (position) in a very fast way.”

Under the hood Redis uses either a ziplist or a skip list. The ziplist is a specially encoded dually linked list that is designed to be very memory efficient. By default a sorted set starts out as a ziplist, which is more efficient for shorter lists, and is converted into a skip list when it reaches a predefined size [29] [30]. The tipping point is defined in the configuration file by the *zset-max-ziplist-entries* and *zset-max-ziplist-value* values, with defaults set to 64 and 128 [31]. Since in practice we never store time series so short, and because the conversion from ziplist to skip list incurs an overhead, we set these values to 0 to use skip lists from the start.

A sorted *list* would be best for us. Indeed in a sorted set the values are expected to be unique. This means with a naive implementation only one data point would be kept for all the data points having the same value. As is usually done in this case, the timestamps is prepended to the values to insure uniqueness. [23] [24] This incurs some overhead in term of storage space and reading speed.

Support for Redis is implemented in the *beast.store.impl.redis* package. Redis is accessed through the *jedis* Java client [51]. This client implements pipelining, meaning we can send multiple commands to Redis in a row without having to wait for the successive answers. This speeds up importing considerably.

Redis is single-threaded. To take better advantage of the many cores of the server, multiple instances are started. The series are then partitioned between the instances based on their ID.

The following table shows the importing performance of the backend depending on the number of importer threads and Redis instances. The test is run on the pc-atd-cc-01 machine, 11 million points are imported. The speed is in data points per second.

Importer threads	Partitions	Importing speed
1	1	103.5 K/s
6	1	504.5 K/s
12	1	912.0 K/s
1	2	127.1 K/s
6	2	783.3 K/s
12	2	1325.0 K/s
1	3	97.9 K/s
6	3	629.0 K/s
12	3	1312.4 K/s

Table 6.1: Test results

The performances with Redis are really good. The most efficient configuration is with 12 importer threads and 2 Redis instances. It is not necessary to use more than 2 instances, and the importing speed stops increasing with more than 12 importer threads.

6.2 Out Of Memory Behavior

The backend acts as a cache, speeding up the retrieval of the IS data initially stored in P-BEAST. As with any caching system an eviction policy has to be defined to decide which item to discard when it fills up. In our case this happens when too many time series have been requested, and either the Java heap or the memory allocated to Redis runs short.

To deal with this case some meta-data is stored for each loaded time series: its creation time, the number of time it was requested, and the timestamp of the last access. For now a simple "least recently used" policy is used when a series needs to be deleted (this means only that last piece of information is used).

Potentially it could be improved to a smarter choice, keeping for example the series that have been "popular recently". With the current setup the very large amount of main memory available didn't make such an implementation necessary.

The series meta-data is stored in the server's SQLite database. As an additional benefit, since the SQLite database is persisted to disk this means we can reload all the series that were loaded in memory if the server needs to be restarted or crashes.

Chapter 7 Time Series Processing

The raw IS data is loaded in the data store, that acts as a cache. The main purpose of the backend is to make this data available to the client in a way it can handle. The time series are processed in two ways for this purpose: they are downsampled on the fly while they are being loaded, and they are aggregated on demand to generate histograms from the series.

7.1 Downsampling

The series from the IS typically contain hundreds of thousands of points each. On the other hand the graphs displayed on the dashboard are 500 to 2000 pixels wide. Loading and rendering all the points would therefore be wasteful and result in unreadable graphs.

Two strategies were considered to solve this issue.

7.1.1 Largest Triangle Three Buckets

The Largest Triangle Three Buckets (LTTB) algorithm was developed in 2013 for DataMarket, a data hosting company. The idea is to keep only the points which are the most visually significant. It is based on a technique from cartography which involves forming triangles between adjacent data points and using the area of the triangles to determine the perceptual importance of the individual points. [22]

Below is a comparison between an original series and a version downsampled by a factor 25 (ie 1 in 25 points is kept). Here the graph is 868 pixels wide. With 5000 points the line graph is over-crowded. With 200 points the visual characteristics are preserved and the result is much clearer.

The resulting chart is not "correct" in the sense that points are removed at places that do not make physical sense. However the goal is to keep the general appearance faithful to the original, since the dashboards are used to get an intuition about how the data behaves rather than to make precise measurements. Because the graphs are dynamic it is also possible to zoom in to see the raw data.

This methods works well for most series when the downsampling factor is less than 10. However it is difficult to go beyond this point without ruining the series appearance. The problem is that the LTTB algorithm splits the series in time windows of equal

Figure 7.1: Original series (5000 points)

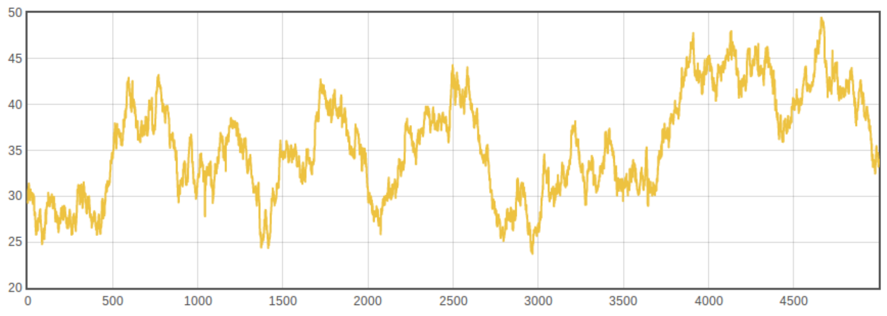
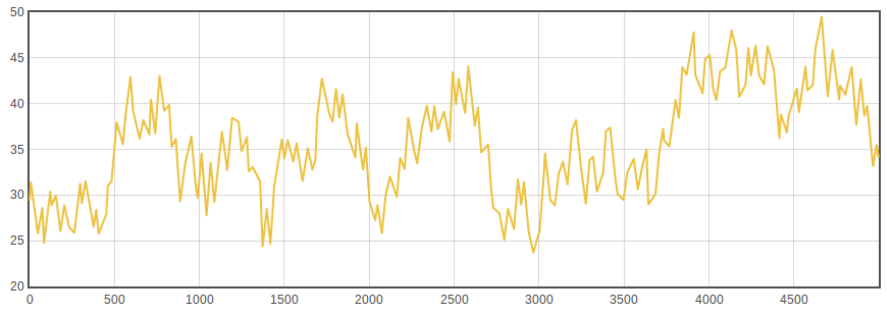


Figure 7.2: Downsampled version (200 points)



size, then picks one point per window. But the original series may contain much more information in some time ranges and than others. This is what makes LTTB unreliable and limited. A dynamic version that computes the time windows more intelligently exists, but its running time is prohibitive.

7.1.2 Min-max chart

Another solution is to cut the original series into slices, and to keep for each slice the maximum, minimum and last values. Then we can display a graph with 3 lines per series, one for each of these values.

This is very cheap from a processing point of view, as we read data from P-BEAST in chronological order. Therefore keeping the maximum and minimum value within a time range can be done trivially at import time.

Each series is sliced at 4 different time intervals to get a more flexible result. We use slices of 10, 100, 1000 and 10000 seconds. These downsampled series are saved in the data store along with the original one.

When the frontend requests a series, it specifies how many points should be returned. To pick the most appropriate granularity the server check first in the downsampled series that uses the largest time interval. If it finds enough points in the requested time interval there then it sends them back to the client. If not, it checks in the next series with largest time interval, if there are none it returns the raw data. An advantage of this solution is that the series are downsampled once and for all. This way a request for a time range which is already in the cache has a very low performance cost.

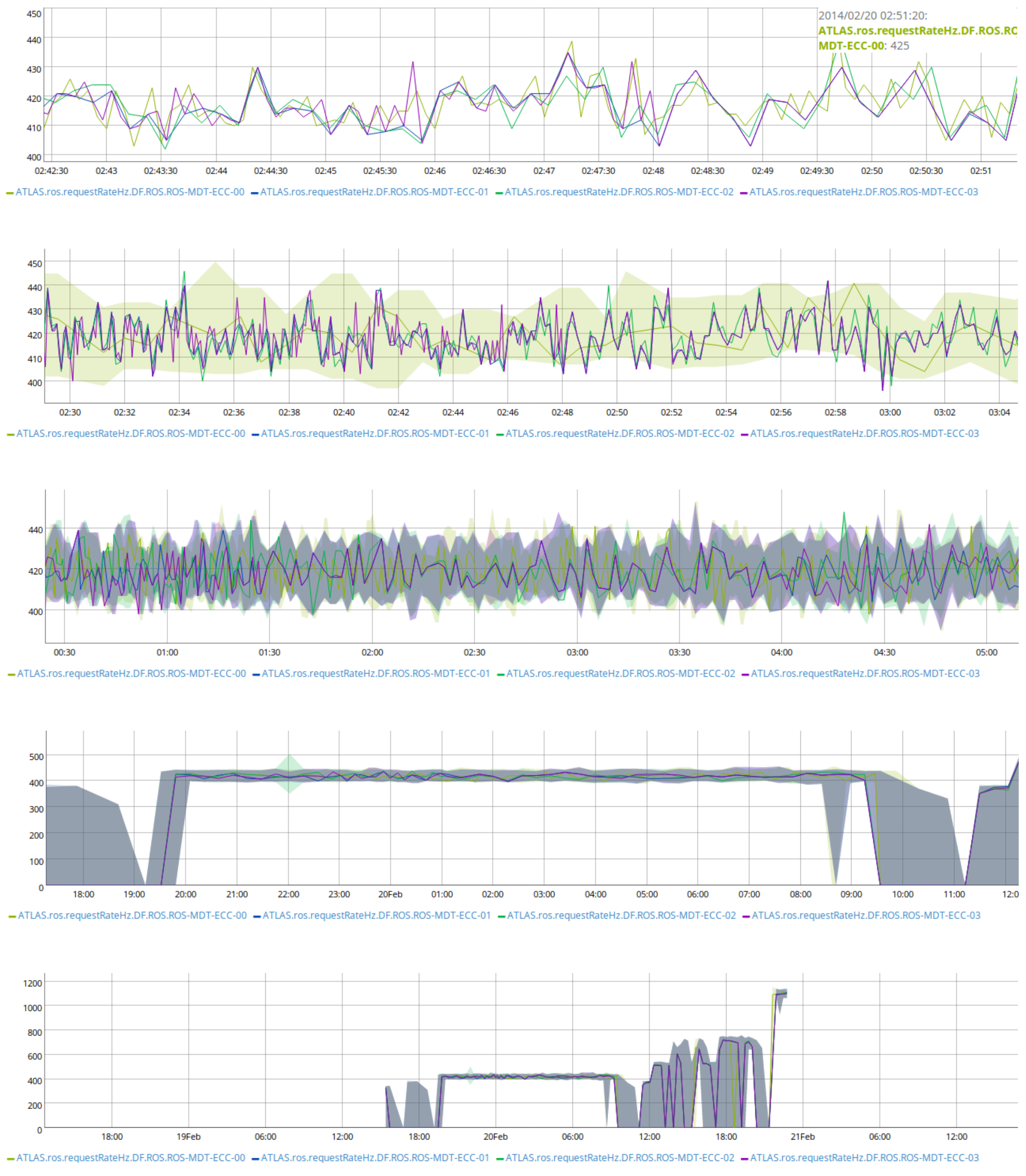


Figure 7.3: Progressively zooming out of a min-max line chart

This solution was kept as it scales to arbitrarily large series and display "correct" data, in the sense that the result can't get misleading. The LTTB algorithm produces more readable results but it is too limited and unreliable for our use case.

An interesting extension would be to use min-max charts that slice the series at the most visually significant points, as selected by the LTTB algorithm.

7.2 Aggregation

The other processing done on the time series is aggregation. This is used to display pie charts on the dashboards. Pie charts are useful to display non-numeric values, which typically represent states.

The aggregation is done on demand. The distinct values are binned, then the 15 bins with the highest counts are kept. If more than 15 distinct values exist in the series, an extra bin labeled ”others” is used. This threshold can be configured but the current value is a good trade-off between the precision of the pie chart and its readability.

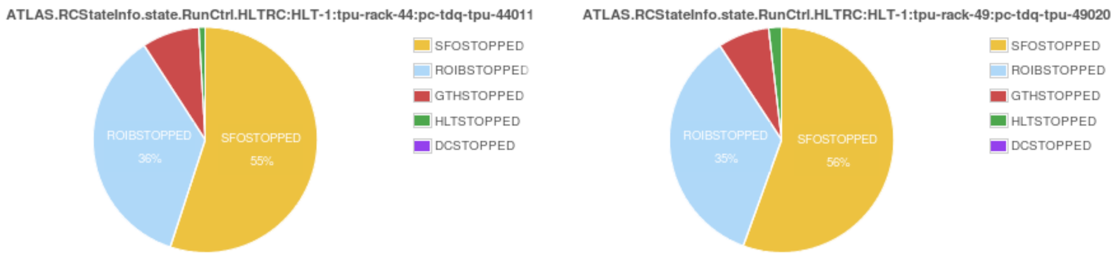


Figure 7.4: Pie charts as rendered on the dashboard

Both numeric and non-numeric series can be displayed as pie charts. Since the bins each contain a single value, pie charts don’t make sense for series with high cardinalities. To get an idea of whether a pie charts representation makes sense the total cardinality of each series can be computed at import time.

Hyperloglog counters are used to do this efficiently. Hyperloglog is a technique for efficiently computing the number of distinct entries in a set. Using the Redis implementation, it uses 12 KB per counter to count with a standard error of 0.81%up to 2^{64} items. Adding a new item and getting the value of a counters are both constant time operations. [10] [11] [12] [13]

Note that while the graphs are usually used to display multiple time series at once, a pie chart only works with one. In practice if a wildcard is used in the series ID, only the first matching series will be used (using lexicographic ordering).

Chapter 8 API

8.1 Implementation details

The backend API is optimized to minimize parsing and rendering time on the client.

The results are returned in the JSON format. The advantage of JSON is that the result string is a valid JavaScript object that does not need a custom parser like would be the case with XML. In general, most web applications would *still* use a custom parser for security reasons. As the result is valid Javascript, an attacker could potentially use it to inject malicious code that would get run by the client. In our case however as the dashboards are accessed locally and in a trusted environment, this overhead is skipped and the frontend directly ”eval’s” the result.

The structure of the result string is also optimized for speed. When the frontend requests a list of time series, they are read sequentially from the backend data store at the adequate downsampling rate. However before being written, the series are globally sorted, and data points at the same timestamps are grouped together. An example follows, the *null* values mean one of the series doesn’t have a point at that timestamp.

```
{"datapoints":[
        [1.392855418784E12,[7.76270762E8,776655085,7.76655085E8],[37625.0,38474.8,38474.8]],
        [1.392855433784E12,[7.76847938E8,777233185,7.77233185E8],[38497.8,38497.8,38641.4]],
        [1.392855448784E12,[7.77427548E8,777813336,7.77813336E8],[38550.4,38568.6,38911.2]],
        [1.392855463784E12,[7.77999966E8,778383168,7.78383168E8],[37285.2,38495.6,38495.6]],
        [1.392855478784E12,[7.78575243E8,778960872,7.78960872E8],null],
        [1.392855493784E12,[7.7915336E8,779536530,7.7953653E8],[38489.0,38489,38576.8]],
        [1.392855508784E12,null,[37722.6,37997,37997.0]],
        [1.392855523784E12,[7.80290968E8,780676277,7.80676277E8],[37521.0,38572.2,38572.2]],
        [1.392855538784E12,[7.80866001E8,781245676,7.81245676E8],[37888.8,38128.2,38387.8]],
        [1.392855553784E12,[7.81434217E8,781811987,7.81811987E8],[37736.8,37783.8,37783.8]],
        [1.392855568784E12,[7.82000688E8,782382139,7.82382139E8],[37739.0,38152.6,38152.6]],
        [1.392855583784E12,[7.82569432E8,782948072,7.82948072E8],null],
        [1.392860218784E12,[9.5908402E8,959462895,9.59462895E8],[37769.0,37910.2,38213.0]],
        ...
], "labels": ["ATLAS.HLTSV.LVL1Events.DF.HLTSV.Events","ATLAS.HLTSV.Rate.DF.HLTSV.Events"]}
```

This speed up rendering in two ways.

First, the final response size is smaller because the timestamps are not repeated. The gain is much larger than was anticipated because related time series are often started at the same time and recorded with the same time interval. In practice with test dashboards the points "line up" very well, and the same timestamps can frequently be used for dozen of series.

Second, by formatting the series this way we can directly feed them to the Dygraphs graph rendering engine (see (frontend part about dygraphs)). Dygraphs surprisingly expects the data points to be grouped in that way.

Additional optimizations were considered but ultimately rejected.

For example when a downsampled point is transferred it is formatted as a triplet: *min, max, last.* The triplet contains the minimum, maximum and last value on that range. The downsampling scheme is described in more details in section 7.1. Not all points that are transferred are downsampled, as sometimes we can render the recorded information exhaustively on the graph. It would make sense in that case to only transfer one value. However in practice it is more efficient to use the same encoding everywhere than to use a special case for non-downsampled points.

Similarly, the timestamps could be compressed by subtracting the start date of the range from each of them. Again in practice the decompression process on the client side is not worth it.

In general, it is not worth optimizing for response size over response complexity. The dashboards are accessed through a fast local network. The raw decoding speed is what matters.

8.2 API Documentation

Here is the list of the API calls available from the backend, along with the list of their GET parameters. The endpoints is located at:

http://servername:8080/rest/{API_CALL}

An example command to test a running backend server follows. It would list the partitions starting with "A" for which we have data.

```
curl http://servername:8080/rest/autocompleteSeries?stub=A
```

8.2.1 loadSeries

Loads a series using its ID on a given time range. Wildcards ("*") can be used for the used for the ID.

- **id** the series complete ID (a partial ID will be rejected)
- **start** beginning of the range as a Unix timestamps (in seconds)
- **end** end of the range as a Unix timespamp in seconds

8.2.2 readSeries

Reads a time range from a series loaded in the cache.

- **id** a list of semi-colon separated series ID, widlcards ("*") are supported.
- **start** beginning of the range as a Unix timespamp (in seconds)
- **end** end of the range as a Unix timespamp (in seconds)
- **maxDataPoints** the maximum number of points to be returned per series
- **histogram** whether to return histogram data (typically for pie charts)

8.2.3 autocompleteSeries

Provides auto-completion for a series path. Returns 10 results at most.

- **stub** the partial series ID to get autocomplete suggestions for.

8.2.4 saveDashboard

Saves a dashboard on the server. Since the dashboards are saved as large JSON files they are sent using a POST request.

8.2.5 Others

A few other parameter-less API calls are available for testing and maintenance purposes:

- **listLoadedSeries** returns the complete list of series loaded in the backend store (may get very large!)
- **deleteAllSeries** flushes all the loaded series
- **getUptime** returns the server uptime in seconds
- **getSystemSpecs** returns how much main memory and disk space is available on the server
- **getLogs** returns the complete server logs
- **getStoreStats** returns the data store current importing queue size and maximum size

Part III

Frontend

Chapter 9 Overview

The purpose of the frontend is to make the IS data accessible in an intuitive and dynamic way through a web browser. For this a dashboard system is used. Users can create, configure and save their own dashboards from their browser. Each dashboard contains panels, which can be either line charts or a pie charts. Line charts can be dynamically zoomed in and out and panned horizontally. Pie charts can represent numeric as well as non-numeric values. The data displayed by each panel, and the chart style, can be finely edited.

The development of the custom frontend was the most challenging part of this project. As we've seen the backend takes in charge a lot of the work by loading, downsampling and formatting the series into a format the frontend can easily handle. But still we can only simplify the data so much and in the end hundred of thousands of points need to be rendered as graphs by the browser. Making the interface readable, user-friendly and usable on multiple browsers was also harder than expected.

The Grafana frontend was used as a starting point. A number of elements needed to be adapted, updated or optimized to suit our needs.
Grafana uses the Bootstrap HTML frontend framework. To improve cross-browser performance and to take full advantage of it, Bootstrap was upgraded from version 2 to version 3. The Grafana theme was also completely modified to make space for more features, including advanced dashboard management.
The charts are rendered using JavaScript. With Grafana only line charts were supported, using the Flot rendering library. It appeared the Dygraph library from Google is much faster for that purpose, so it took Flot's place. Pie charts were also implemented, using Flot this time.
The dashboard editing features of Grafana were complete but sometimes confused users. Nearly all the interface elements, from the buttons to the menus to the dialog boxes, were redone taking into account user feedback.
New editing features were also needed to accommodate the custom backend. The most requested feature was autocompletion for the series ID. This was somehow challenging as the number of objects from the IS is huge.

We will review in more details how dashboards work, then how the various libraries and frameworks are used, and finally what were the major challenges.

Chapter 10 Dashboards

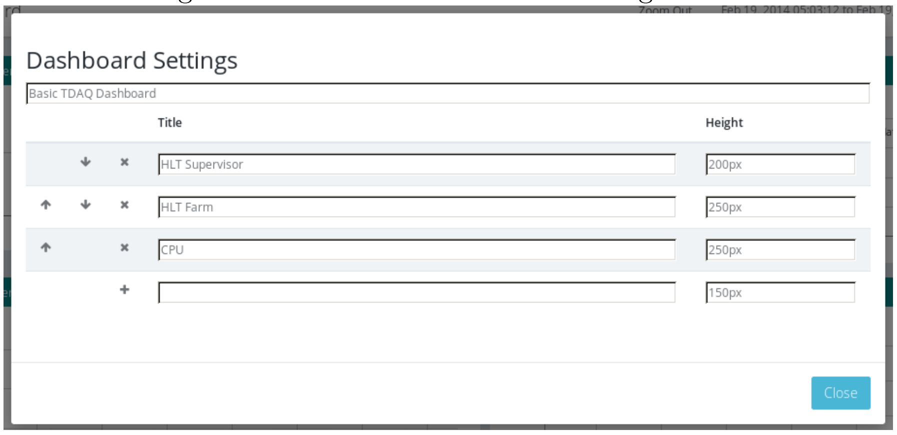
The data is displayed in dashboards. The height of each dashboard row can be individually configured in pixels. The width of each panel can be configured as a fraction of the page's total width. The fractions are multiple of 1/12, for example a panel taking half the horizontal space has a width of 6.

10.1 User Configuration

Three user dialogs are available to edit the arrangement of the dashboard.

The main dashboard dialog lets the user configure the row names, heights and orders, as well as the dashboard's name. It can be accessed from the top-right wrench button.

Figure 10.1: General Dashboard Configuration



Dashboard Settings

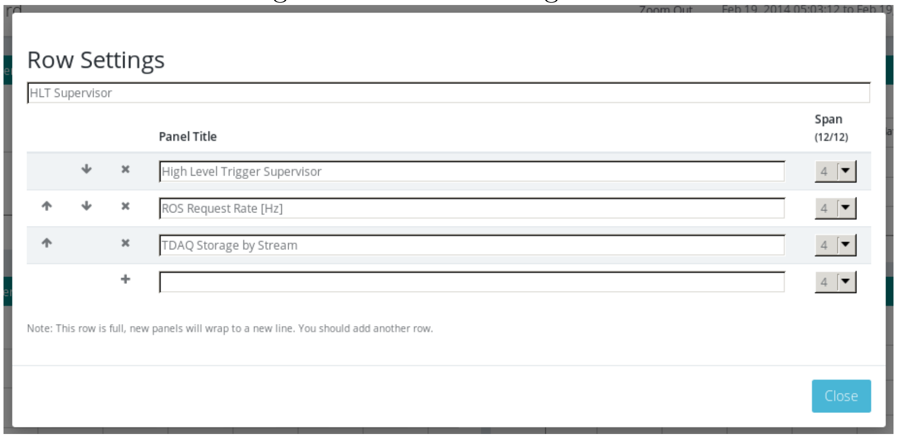
Basic TDAQ Dashboard

		Title	Height
	↓ ×	HLT Supervisor	200px
↑	↓ ×	HLT Farm	250px
↑	×	CPU	250px
	+		150px

For each row, the user can edit the width of each panel as well as its individual title. The panel can also be reordered. A name can be given to the full row, it is not visible by default but will be shown when the row is collapsed.

It is also possible to rearrange the panels by dragging and dropping them around the dashboard.

Figure 10.2: Row Configuration



Row Settings

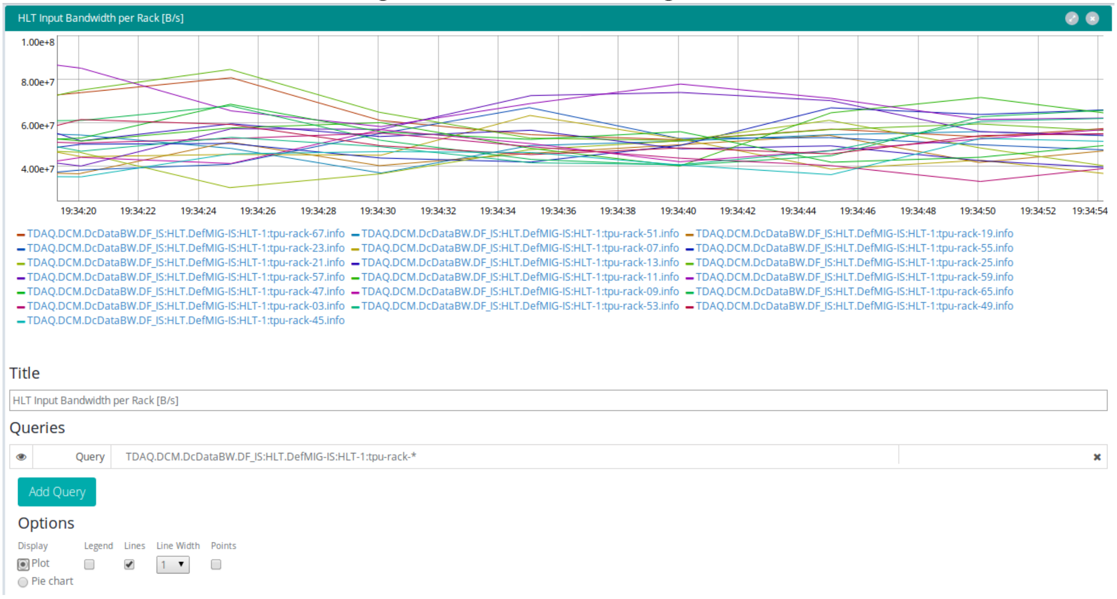
HLT Supervisor

		Panel Title	Span (12/12)
	↓ ✖	High Level Trigger Supervisor	4
↑	↓ ✖	ROS Request Rate [Hz]	4
↑	✖	TDAQ Storage by Stream	4
	✚		4

Note: This row is full, new panels will wrap to a new line. You should add another row.

Finally for each panel the queries, as well as the graph options, can be configured. The user can select between a line chart and a pie chart. The changes are applied at once. Multiple queries can be displayed on a single graph. As discussed earlier, only one time series can be used to display a pie chart; if a query returns multiple series only the first one will be used. For each line chart it is possible to use a second Y axis to display series with different dimensions. A click in front of a series' name in the legend will toggle its Y axis. It is also possible to toggle the visibility of any series by clicking on its name in the legend. Clicking a series while holding the *CTRL* key will keep it visible, but hide all the others. All of these settings are saved along with the dashboard.

Figure 10.3: Panel Configuration



- TDAQ.DCM.DcDataBW.DF_IS:HLT.DefMIG-IS:HLT-1:tpu-rack-67.info
- TDAQ.DCM.DcDataBW.DF_IS:HLT.DefMIG-IS:HLT-1:tpu-rack-51.info
- TDAQ.DCM.DcDataBW.DF_IS:HLT.DefMIG-IS:HLT-1:tpu-rack-19.info
- TDAQ.DCM.DcDataBW.DF_IS:HLT.DefMIG-IS:HLT-1:tpu-rack-23.info
- TDAQ.DCM.DcDataBW.DF_IS:HLT.DefMIG-IS:HLT-1:tpu-rack-07.info
- TDAQ.DCM.DcDataBW.DF_IS:HLT.DefMIG-IS:HLT-1:tpu-rack-55.info
- TDAQ.DCM.DcDataBW.DF_IS:HLT.DefMIG-IS:HLT-1:tpu-rack-21.info
- TDAQ.DCM.DcDataBW.DF_IS:HLT.DefMIG-IS:HLT-1:tpu-rack-13.info
- TDAQ.DCM.DcDataBW.DF_IS:HLT.DefMIG-IS:HLT-1:tpu-rack-25.info
- TDAQ.DCM.DcDataBW.DF_IS:HLT.DefMIG-IS:HLT-1:tpu-rack-57.info
- TDAQ.DCM.DcDataBW.DF_IS:HLT.DefMIG-IS:HLT-1:tpu-rack-11.info
- TDAQ.DCM.DcDataBW.DF_IS:HLT.DefMIG-IS:HLT-1:tpu-rack-59.info
- TDAQ.DCM.DcDataBW.DF_IS:HLT.DefMIG-IS:HLT-1:tpu-rack-47.info
- TDAQ.DCM.DcDataBW.DF_IS:HLT.DefMIG-IS:HLT-1:tpu-rack-09.info
- TDAQ.DCM.DcDataBW.DF_IS:HLT.DefMIG-IS:HLT-1:tpu-rack-65.info
- TDAQ.DCM.DcDataBW.DF_IS:HLT.DefMIG-IS:HLT-1:tpu-rack-03.info
- TDAQ.DCM.DcDataBW.DF_IS:HLT.DefMIG-IS:HLT-1:tpu-rack-53.info
- TDAQ.DCM.DcDataBW.DF_IS:HLT.DefMIG-IS:HLT-1:tpu-rack-49.info
- TDAQ.DCM.DcDataBW.DF_IS:HLT.DefMIG-IS:HLT-1:tpu-rack-45.info

Title

HLT Input Bandwidth per Rack [B/s]

Queries

Query TDAQ.DCM.DcDataBW.DF_IS:HLT.DefMIG-IS:HLT-1:tpu-rack-*

Options

Display	Legend	Lines	Line Width	Points
[✓ Plot]	☐	✓	1	☐
Pie chart				

10.2 Dashboard Management

The dashboards are saved on the server. A unique, shareable URL is created for each new one. For dashboards that needs to be made available to everyone, it is also possible to "pin" them. This means their name will appear on the dashboard list on the left side-bar of the interface, from where they can be opened.

For now any user can save, pin or override any dashboard. This is sufficient as only a restricted number of users have access to the interface, but in the future it will be necessary to add finer-grained controls. Separate user permissions should exist to allow a user to explore the data, create and save new dashboards, or to pin or override them.

It is important to note that simply exploring the data will load it into the cache. This means an overly general query (for example using just a wildcard as an object) could fill up the cache, and evict other user's data.

Chapter 11 Frameworks and Libraries

The frontend is a single-page application built with AngularJS and Bootstrap. The line charts are rendered using Dygraphs, and the pie charts using Flot. We will see in more details what is the role of each of these components.

11.1 AngularJS

AngularJS is the frontend'd spine. It's a web application framework that provides model-view-controller (MVC) capability [43]. It extends the usual HTML syntax with additional tag attributes to bind the page to the application model through directives.

Grafana as an AngularJS application, provided a solid foundation to build on. A custom notification has been added using AngularJS to show unobstrusive alerts to the user. Custom bindings were also developed to handle keyboard input, for example for autocompletion.

11.2 Bootstrap

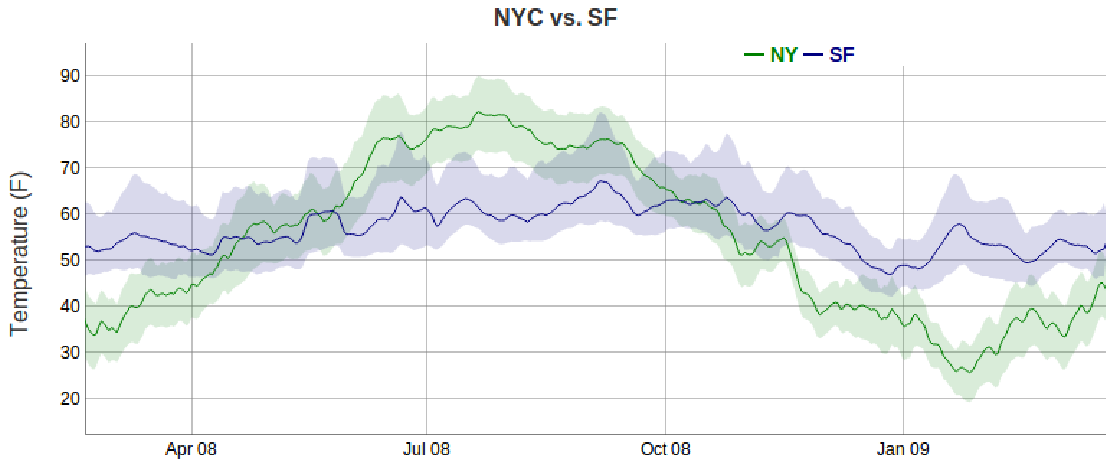
Bootstrap is a hugely popular frontend framework, that comes with a collection of tools for creating websites. It provides graphical elements for typography, buttons, navigation and more [44].

Grafana used an older version of Bootstrap. To take full advantage of this framework, it was upgraded to the latest version (3.2.0). The upgrade process is not trivial since the semantic changed in some major ways [45]. The angular bindings also needed to be updated to accommodate these changes.

11.3 Dygraphs

Dygraphs is a very fast charting library. It replaced Flot, which was used to render the plots in Grafana, in order to improve the overall dashboard performance. Another reason for using Dygraphs over Flot is the support for "error bars", ie the series' "envelope" visible on this graph 11.1. That feature is used to show the minimum and maximum of a series when using downsampling.

Figure 11.1: An example Dygraphs chart, with error bars



In some situations a huge number a series need to be displayed on the same graph. This is the case for example in the *ROS Request Rate* panel visible on this document's cover (middle panel, the top row). Beyond 500 series the graph would take multiple seconds to render and would start freezing the browser. This issue was due to the way the colors of the series are generated and handled by Dygraphs. Additional caching and optimization was done to the engine's internal to fix this issue.

Another issue that was harder to fix is that the charts have a tendency to "spill" outside of their designated rendering element. Typically a graph is rendered inside an HTML *div* element, and it should fit in that given space. Dygraphs sometimes takes more space than allocated. In our specific case this caused problems with the dashboard's layout, since an overly large panel would cause line breaks. A few modifications had to be done to Dygraph's internal to avoid this issue. The length of the axis labels was the problem.

11.4 Flot

Flot is another charting library [46], which is more expansible and customizable than Dygraphs, at the cost of lower performances. It also doesn't look as good by default. It is used in the frontend to generate pie charts, which are handled by an external plugin [47].

Chapter 12 Difficulties Encountered

12.1 Graph Rendering Speed

In the beginning the main difficulty was to render the charts fast enough. This problem was solved in three times: first Flot was replaced by Dygraphs (section 11.3), then min-max charts were implemented (section 7.1.2). Finally now the backend provides the data points in a way that speeds up the rendering still a bit more (section 8.1).

12.2 Sub-second Timestamps

After the Java API for P-BEAST was implemented, the charts appeared to be displaying very surprising data. It looked like around a third of the data points were ignored.

After investigation, it appeared the problem was due to the speed at which IS objects change. They were initially assumed to change at most every second. In practice changes happening milliseconds apart are common.

Why? This is actually due to the way P-BEAST stores events. As long as a value is constant, it doesn't record anything. But when it changes, it records the last known time at which the old value was still valid, and records the new value right afterward. The time interval between the last value and the new one is in the order of a few milliseconds.

The system had to be adapted to deal with such low intervals.

12.3 Memory Consumption

The frontend has some issues on machines with limited memory. The problem is that a large number of JavaScript objects needs to be kept in memory by the browser even after the charts are fully rendered. This is because the charts are dynamic: if the user hovers a data point, the value at this point will be shown. For this to be possible all the values of each series displayed on the dashboard must be kept. The only solution in a memory-limited environment would be to render the charts on the server side.

Additionally some memory leaks caused problems at some points. These were due to the non-obvious way Dygraphs charts needs to be disposed of [48].

Part IV

Conclusion and Outlook

Chapter 13 Conclusion

In the end the custom system works well. The performances are way beyond the requirements in terms of data ingestion. Some additional types of data, namely strings and simple arrays, can be handled as well. The user interface is responsive and user-friendly.

The dashboards were tested during a few CERN "Technical Runs" (test weeks) with simulated live data. They performed well during the last one, now that the frontend problems have been worked out. Using the latest version of P-BEAST it was possible to display data updated every second with less than 30 seconds of delay.

The results of the initial research were a bit frustrating since nothing seemed to fit our needs. But by studying and comparing the existing solutions, it was possible to borrow pieces from some and ideas from others.

Splunk is also being considered as a visualization system for the same data, so the two systems were in a way competing. The solution chosen ultimately is to keep both side by side: the custom system to make the entirety of the data available for quick display, and Splunk to perform more advanced analysis on specific portions of the data.

Chapter 14 Future Work

Advanced Series Analysis The data stored in the cache can be accessed very quickly. Using either Redis or a Java SkipList, both interval and random access reads are extremely fast. This makes the visualization backend a good place to perform more advanced analytics.

An interesting addition would be the ability to correlate pairs of series. Out of all the tools we have considered, only Splunk handles this. Users of Graphite for example have been requesting this feature for years [38].

Anomaly Detection Another exciting feature that could be added is anomaly detection. Work is being done to automatically detect "surprising" data coming from the various IS objects, by creating a model of the typical data and raising an alert when the actual values differ too much from it. This could be visualized as highlighted areas on the dashboard, as well as dashboard alerts using the existing notification system.

Graph and Dashboard Exporting A feature that was requested by some users, but which couldn't be implemented due to lack of time, is data exporting. Two exporting formats could be interesting: images, and text for raw data analysis. Images could be useful for example for reports. A textual export, in CSV, JSON or XML could be used for further data analysis in more specialized tools.

Sorted Lists for Redis Redis is open source and its architecture is extensible [1]. It would be possible to implement a custom data type for sorted lists. For now we concatenate the timestamps to the stored values to make them unique: a custom type would avoid this unnecessary overhead.

Dashboard Management Finally as described earlier (section 10.2) the dashboard editing rights should be linked to the CERN accounts permissions.

Part V

Instructions

Chapter 15 Setup Guide

15.1 Dependencies

Java The backend requires that the system has an installed version of at least Java SE 6.0.

Redis If the Redis store is used, a Redis server needs to be running on the default port (6379). To use partitioning the other Redis instances are expected to run on successive port numbers, ie the second instance on port 6380, the third on port 6381 etc. The redis_start.sh script at the root of the backend directory will start the specified number of instances assuming Redis is on the machine's path.

To use HyperLogLog counters for cardinality estimation, the installed version must be at least 2.8.9 (released April 22 2014). Otherwise any version from 2.8 (released November 22 2013) will work.

15.2 Logging

The first place to look in case of trouble is the server logs. They can be fetched remotely from the /rest/getLogs endpoint. It may be useful to turn on more verbose logging to gather more information, see the running configuration below for this.

The client logs debugging and error messages to the browser's console. Angular outputs meaningful messages there as well when something goes wrong.

15.3 Running Configuration

Here's the list of arguments that can be provided when starting up the server:

-log, -l Log level. Can be one of: *all*, *info*, *warn* or *error*. The default is *info*.

-nbIndexerThreads, -i Number of importer threads.

-nbReaderThreads, -r Number of reader threads.

-nbPartitions, -p Number of partitions of the data store; this is only applicable when using Redis. The default is 1.

-store, -s The data store to use. Can be either *skip* or *redis*, to use a Java skip list or the Redis database respectively. By default *skip* is used if the system has up to 2 GB of main memory, otherwise *redis* is chosen.

-port, -p Port on which the HTTP server will make the frontend available.

-downsampling, -d Downsampling factor adjustement. This will force the backend to return more or less data points than the frontend asked for. For example an adjustment of 3 will return 3 times more points than was requested. The frontend typically requests one data point per horizontal pixel of the graph. This is useful either for stress-testing the client (by overloading it with points) or to perform quick tests (to reduce how many points are loaded).

-index Path to the series index. This is an experimental feature. This parameter expects an OSIRIS SQLite file (which is a complete index of all series), and will use it to autocomplete the series IDs. This makes autocompletions faster.

-help, -h Displays the list of supported arguments.

Bibliography

[1] *Hacking Redis: Adding Interval Sets*, Kenny Hoxworth, http://www.starkiller.net/2013/05/03/hacking-redis-adding-interval-sets/

[2] ATLAS Collaboration 2008, *The ATLAS Experiment at the CERN Large Hadron Collider*, Journal of Instrumentation, S08003

[3] ATLAS Collaboration 2003, *ATLAS High-Level Trigger, Data Acquisition and Controls Technical Design*, http://cdsweb.cern.ch/record/616089/

[4] Ciobotaru M, Leahu L, Martin B, Meirosu C and Stancu S, *Networks for ATLAS trigger and data acquisition Conference on Computing in High Energy and Nuclear Physics*, S 2006

[5] Sample skip list, http://en.wikipedia.org/wiki/File:Skip_list.svg

[6] *How to tame java GC pauses?*, Alexey Ragozin, http://java.dzone.com/articles/how-tame-java-gc-pauses

[7] Splunk Website, http://www.splunk.com/

[8] *Persistent Back-End for the Atlas Information System of TDAQ*, Alexandru Sicoe, https://twiki.cern.ch/twiki/bin/view/Sandbox/PersistentBEAST

[9] *Persistent Back-End for the Atlas Information System of TDAQ*, Igor Soloviev, https://twiki.cern.ch/twiki/bin/viewauth/Atlas/DaqHltPBeast

[10] *HyperLogLog*, http://en.wikipedia.org/wiki/HyperLogLog

[11] *HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm*, Philippe Flajolet, Eric Fusy, Olivier Gandouet and Frederic Meunier, http://algo.inria.fr/flajolet/Publications/FlFuGaMe07.pdf

[12] *PFCOUNT*, Redis Documentation, http://redis.io/commands/pfcount

[13] *PFADD*, Redis Documentation, http://redis.io/commands/pfadd

[14] *Elasticsearch*, http://en.wikipedia.org/wiki/Elasticsearch

[15] *In A World Of Open Source Big Data, Splunk Should Not Exist*, Matt Asay, `http://readwrite.com/2014/01/24/open-source-big-data-splunk`

[16] *Pure in-memory option for running elasticsearch ?*, ElasticSearch Google Group, answer by ElasticSearch main developer "kimchy", `https://groups.google.com/forum/#!topic/elasticsearch/ltykdraIFqA`

[17] *Graphite FAQ*, `http://graphite.wikidot.com/faq`

[18] *Graphite 5-minute Overview*, Graphite Documentation, `http://graphite.readthedocs.org/en/1.0/overview.html`

[19] *About RRDtool*, RRDtool Documentation, `http://oss.oetiker.ch/rrdtool/index.en.html`

[20] *OpenTSDB*, OpenTSDB GitHub project page, `https://github.com/OpenTSDB/opentsdb`

[21] *Skip list*, `http://en.wikipedia.org/wiki/Skip_list`

[22] *Downsampling Time Series for Visual Representation*, Sveinn Steinarsson, `http://skemman.is/stream/get/1946/15343/37285/3/SS_MSthesis.pdf`

[23] *Time Series: What is the best way to store time series data in Redis?*, Reza Lotun's answer on Quora, `http://www.quora.com/Time-Series/What-is-the-best-way-to-store-time-series-data-in-Redis/answer/Reza-Lotun?srid=3XEK&share=1`

[24] *Time-series data in Redis*, James Harrison, `http://www.talkunafraid.co.uk/2010/12/time-series-data-in-redis/`

[25] *Jetty*, Jetty Website, `http://www.eclipse.org/jetty/`

[26] *FAQ*, Redis Documentation, `http://redis.io/topics/faq`

[27] *An introduction to Redis data types and abstractions*, Redis Documentation, `http://redis.io/topics/data-types-intro`

[28] *DB-Engines Ranking of Key-value Stores*, `http://db-engines.com/en/ranking/key-value+store`

[29] *What are the underlying data structures used for Redis?*, StackOverflow answer from "antirez", Redis' main developer, `http://stackoverflow.com/a/9626334/318557`

[30] ziplist.c source code, `http://download.redis.io/redis-stable/src/ziplist.c`

[31] redis.conf source code, `http://download.redis.io/redis-stable/redis.conf`

[32] *Persistent Back End for the ATLAS Information Service of Trigger and Data Acquisition (P BEAST)*, Alexandru Dan Sicoe, `http://cds.cern.ch/record/1509559/files/CERN-THESIS-2011-292.pdf.pdf`

[33] Protocol Buffers, `https://developers.google.com/protocol-buffers/`

[34] bulk api, ElasticSearch documentation, `http://www.elasticsearch.org/guide/en/elasticsearch/reference/current/docs-bulk.html`

[35] glossary of terms, ElasticSearch documentation, `http://www.elasticsearch.org/guide/en/elasticsearch/reference/current/glossary.html`

[36] merge, ElasticSearch documentation, `http://www.elasticsearch.org/guide/en/elasticsearch/reference/current/index-modules-merge.html`

[37] translog, ElasticSearch documentation, `http://www.elasticsearch.org/guide/en/elasticsearch/reference/current/index-modules-translog.html`

[38] Graphite needs a scatter graph mode, Graphite Bug Tracker, `https://bugs.launchpad.net/graphite/+bug/886411`

[39] *Large-Scale, Unstructured Data Retrieval and Analysis Using Splunk*, Splunk Technical Paper, `http://www.splunk.com/web_assets/pdfs/secure/Splunk_and_MapReduce.pdf`

[40] *Introduction to Splunk*, Alex Iribarren, `http://indico.cern.ch/event/199175/material/slides/0.pdf`

[41] *High-performance scalable information service for the ATLAS experiment*, Kolos, S., Boutsioukis, G., Hauser, R., `http://ieeexplore.ieee.org/xpl/login.jsp?tp=&arnumber=6418091&url=http%3A%2F%2Fieeexplore.ieee.org%2Fiel5%2F6410221%2F6418089%2F06418091.pdf%3Farnumber%3D6418091`

[42] CORBA, `http://corba-directory.omg.org/`

[43] Google's AngularJS, `https://angularjs.org/`

[44] Twitter's Bootstrap, `http://getbootstrap.com/`

[45] *Migrating to v3.x*, `http://getbootstrap.com/migration/`

[46] Flot Website, `http://www.flotcharts.org/`

[47] *Pie Charts*, `http://www.flotcharts.org/flot/examples/series-pie/index.html`

[48] *Preventing Dygraphs Memory Leaks*, Robert Konigsberg, `http://blog.dygraphs.com/2012/01/preventing-dygraphs-memory-leaks.html`

[49] Kibana Website, `http://www.elasticsearch.org/overview/kibana/`

[50] Grafana Website, `http://grafana.org/`

[51] Jedis GitHub Page, `https://github.com/xetorthio/jedis`