University of Victoria
Faculty of Engineering
Spring 2019 Work Term Report

Cloud Management: Time Series Data Visualization and Development

University of Victoria
Department of Physics and Astronomy
Victoria, BC

Shaelyn Tolkamp
V00875259
Work Term 2
Software Engineering
tolkamps1@uvic.ca

April 25, 2019

In partial fulfillment of the requirements of the
Bachelor of Engineering Degree

Supervisor's Approval: To be completed by Co-op Employer

This report will be handled by UVic Co-op staff and will be read by one assigned report marker who may be a co-op staff member within the Engineering and Computer Science Co-operative Education Program, or a UVic faculty member or teaching assistant. The report will be either returned to the student or, subject to the student's right to appeal a grade, held for one year after which it will be destroyed.

I approve the release of this report to the University of Victoria for evaluation purposes only.

Signature: ____________________________ Position: _____________ Date: ______________

Name (print):_________________________ E-Mail: ____

For (Company Name)____

Contents

List of Figures iii

Abstract iv

1 Glossary v

2 Introduction 1

3 Visualization for Cloudscheduler v2 1

3.1 Database Selection 2

3.1.1 MongoDB 2

3.1.2 Elasticsearch 2

3.1.3 Graphite 2

3.1.4 Prometheus 2

3.1.5 InfluxDB 3

3.2 Data Collection 3

3.3 Data Storage 4

3.4 Visualization 5

3.5 Retrieving Data 6

3.5.1 Websocket and Django Channels 6

3.5.2 AJAX, Django and the Fetch API 7

4 Amazon EC2 Support 8

4.1 Collecting Instance Types 8

4.2 Filtering and Segregation 10

4.3 Amazon Machine Images 10

5 Conclusions 10

6 Recommendations 11

7 Acknowledgements 11

References 12

List of Figures

1 CSV2's web front-end cloud status and job status tables for ATLAS 3
2 Sample cloudscheduler command 4
3 Excerpt from timeseriesPoller.py 4
4 Calculation for InfluxDB storage size 5
5 CSV2's web front-end with plot using Plotly.js 6
6 Django view excerpt from cloud_views.py 7
7 Sample product entry from Amazon JSON 9
8 ec2_instance_type_filters table from MariaDB 10

Cloud Management: Time Series Data Visualization and Development

Shaelyn Tolkamp
tolkamps1@uvic.ca

April 25, 2019

Abstract

The High Energy Physics Research Computing group (HEPRC) at the University of Victoria is dedicated to developing and maintaining cloud management software for the data analysis required for the ATLAS experiment at CERN Laboratories and for the Belle II experiment at KEK Laboratories. These particle physics experiments collect vast amounts of data that are best analyzed through cloud computing, due to their scale. HEPRC's Cloudscheduler software manages VMs on multiple computing clouds. HEPRC is permanently transitioning from Cloud Monitor (Cloudscheduler v1), soon to be deprecated, to Cloudscheduler v2 (CSV2) and requires a replacement visualization overview of the cloud scheduling statistics produced by their monitoring systems that is optimized for time series data. A time series database, InfluxDB, was selected to store the collected data, and a poller script developed to collect current data and send it to InfluxDB. A JavaScript graphing library, Plotly.js, was used to plot selected traces in the web browser to display historical data over a selected time range. This assisted with a quick visual overview for finding potential causes of errors. Additionally, CSV2 currently only supports Openstack clouds, but support is being added for Amazon Elastic Compute clouds. An instance type poller was written to gather instance types and their properties from an Amazon pricing JSON file. This information was added to MariaDB for use when a user is booting an instance to run a job.

1 Glossary

AJAX Asynchronous JavaScript and XML. Allows browsers to send and retrieve data from a server asynchronously.

API Application Programming Interface.

CLI Command Line Interface.

Cron Unix job scheduler for running tasks at specified times.

Daemon Computer process that runs in the background, usually continuously.

HTTP HyperText Transfer Protocol. Dictates the communication/data transfer of the World Wide Web.

Image A template of an operating system and often applications, or other software.

Instance type Specifies the hardware of a host computer for an instance or VM.

jQuery JavaScript library for simplified HTML document manipulation.

Line Protocol InfluxDB's format for writing data points to the database.

PhpMyAdmin Web application administration tool for MariaDB or MySQL.

RAM Random Access Memory. Storage for current data by a computing device.

SQLAlchemy SQL toolkit for the programming language Python.

Time Series Database Database specifically for storing time series data, ie. data with values and timestamps.

UI User Interface

VM Virtual Machine. An OS that can be run within a host computers environment and operate its own applications or programs.

2 Introduction

The High Energy Physics Research Computing group (HEPRC) at the University of Victoria is dedicated to developing and maintaining computing, networking, and storage systems. In particular, HEPRC, manages clouds for the data analysis required in large particle physics experiments, such as the ATLAS experiment at CERN Laboratories and the Belle II experiment at KEK Laboratories [1]. These research experiments collect large amounts of data that, because of their scale, are best analyzed through the resources available in cloud computing.

HEPRC's cloud scheduling software manages VMs on multiple computing clouds. HEPRC is permanently transitioning from Cloud Monitor and Cloudscheduler v1 (CSV1) to Cloudscheduler v2 (CSV2) and requires a replacement visualization overview of the cloud scheduling statistics that is optimized for time series data. Version 1 had multiple CSV1 servers that were running the cloudscheduler software for each project. They had a collection daemon that sent data to a separate server, Cloud Monitor, for visualization and to centralize monitoring data. Cloud Monitor had it's own web front-end and displayed this data in user-friendly, readable tables. CSV2 combines the cloud scheduling software and the visualization aspect into one CSV2 server. This integration streamlines the monitoring and scheduling processes by allowing direct communication and control.

A graphical visualization for CSV2 can efficiently assist in finding when an error or anomalous behaviour occurs. This visualization requires collecting and storing the time series data available in a separate database and displaying it in a time series plot at a user's request. The current database, MariaDB, stores only *current* state information and configurations, not any of the historical data that will be needed to create a time series plot.

CSV2 did not yet have support for Amazon computing clouds. The current implementation of the cloudscheduler software was structured around Openstack clouds for ATLAS. While Belle II clouds were currently being supported by CSV1, to successfully phase out CSV1 and integrate Belle II, additional support for Amazon computing clouds is required. This involves gathering the appropriate information from Amazon Elastic Compute clouds, processing it, and storing it in a database for further use.

3 Visualization for Cloudscheduler v2

CSV1 software handles time series data and visualization through a collector script on CSV1 servers that sends the data to MongoDB on a separate Cloud Monitor server. In CSV1, Elasticsearch is used for bench-marking, for storing log files in a centralized location, and for additional monitoring. The time series data from MongoDB was visualized on the Cloud Monitor web front-end in tabular displays with the PyMongo package, and graphed using Plotly.js, a python graphing library.

A new interactive visualization for CSV2 required custom software and several different technologies to be integrated. As the cloudscheduler software and web front-end are on the same server, unlike CSV1, it is especially important for this visualization and all the components it requires to be lightweight. A database was required to store the time series data, and a poller script was needed to regularly collect the current data from MariaDB and store it in a separate database. Prompted by the user, the data would then be retrieved and plotted in the web browser for a quick overview of the cloud, job, and service statuses throughout a specified time range, e.g. the last hour, day, week, etc.

3.1 Database Selection

The optimal database would be high performance for time series data; have little or no requirements and little overhead; and be open source, supported, and stable. In general, Time Series Databases (TSDBs) out-perform distributed databases, as they are optimized for storing and querying time series data [2]. For example, in a study done by InfluxData, their TSDB, InfluxDB, can handle almost 1,500,000 values/second while MongoDB, a distributed database, can only handle 10,000 values/second [3]. Most non-TSDBs do not take advantage of the highly-structured nature of time series data and must be specifically configured to work with time series data. After this preliminary research, it was decided that, for best performance, a database specifically for time series data would be used.

3.1.1 MongoDB

MongoDB, a distributed database, stores data in JSON-like documents. This means fields can vary and the overall structure is schema-free [4]. However, it does not take advantage of the structuring of time series data. It was utilized in version 1 of cloudscheduler on the Cloud Monitor server to store summary and detailed information sent by the collection daemons.

3.1.2 Elasticsearch

Elasticsearch is a distributed search-engine-based system that stores data in JSON files [5]. It is not primarily designed to store time series data, but it can be configured to do so [2]. It is currently being used by HEPRC as a monitoring/log centralization system, and Kabana, another component of the Elastic Stack [6], is used for detailed visualization of the logging, error, and other monitoring information stored in Elasticsearch.

3.1.3 Graphite

Graphite is a basic and traditionally used TSDB that "does two things (1) Stores numeric time-series data, [and] (2) renders graphs of this data on demand [7]". While this sounds ideal, users often rely on Grafana as a UI, as the built-in UI is generally considered insufficient [8]. Additional software or technology needed negates the benefits of being lightweight and simple. Data collection to Graphite is passive (i.e. data is collected from collection daemons such as *fluentd*, *statd*, or *collectd* [9], rather than by Graphite itself). Ganglia & Sensu, two technologies previously used by HEPRC in Cloud Monitor for collecting statistics, can also push data to Graphite [9]. While Graphite has been considered an industry standard (one of the reasons it was selected for Cloud Monitor), with advances in technology that produce Terabytes of time-series data, there are now many newer and more sophisticated TSDBs.

3.1.4 Prometheus

Prometheus is a comprehensive monitoring system that specializes in metrics and has built-in and active scraping, storing, querying, graphing, and alerting based on time series data [10]. It is usually paired with Grafana as a UI, as the built-in UI is generally insufficient. It has a powerful query language and alerting and notification functionality [10]. However, CSV2 already has most of this functionality, and these features of Prometheus are all a part of the default package.

3.1.5 InfluxDB

InfluxDB is the newest industry standard. While still fairly new, with its first stable release in September 2016, InfluxDB has become one of the most popular TSDBs on the market. The TICK stack by InfluxData includes four open source projects: a collection agent; a time series database (InfluxDB); a monitoring, alerting, and processing engine; and a customizable interface with graphical dashboards [3, 11]. InfluxDB is designed to handle high write and query loads and uses an SQL-like query language called InfluxQL. It can continuously calculate functions based on the input data and can store the results (e.g. calculation of an average per hour). Chronograf, the interface and data visualization part of InfluxData's TICK stack [12], allows for creating dashboards and graphs of stored data, although many users also integrate with Grafana. InfluxDB has options for full or incremental back-ups that can be done remotely and without taking the system offline [14]. The open source version runs on a single node and only commercial options offer clustering and high availability.

InfluxDB was selected as the most appropriate TSDB based off its continually rising popularity, long-term support, clear documentation, and efficiency for time series data.

3.2 Data Collection

CSV2's web front end status page displays the *current* values of the status data (Figure 1) that are queried from MariaDB directly. When a user selects a table value, the historical data of this measurement will need to be requested from the TSDB. To populate the TSDB, a poller script will need to query MariaDB every 30 seconds for the most current status values and aggregate the data in series and write it to the TSDB.. Two ways of achieving this were considered and implemented: directly interacting with the MariaDB, and using the Cloudscheduler CLI.

Group	Jobs	Idle	Running	Completed	Held	Other
atlas	192	58	126	6	2	0

Group	Clouds	VMs	Starting	Unreg.	Idle	Running	Retiring	Manual	Error	Native Cores Used	Native Cores Limit	RAM	Foreign VMs	Foreign Cores	RAM	Global Cores Used	Global Cores Quota	RAM	Slots	Slot Cores Busy	Slot Cores Idle
atlas	arbutus-a	40	1	0	0	39	0	0	0	320	320		282	564		884	5000		39	312	0
atlas	cc-east-a	16	0	0	0	11	0	0	5	128	128		1	2		130	768		11	88	0
atlas	cc-west-a	72	2	0	0	70	0	0	0	576	576		39	39		615	5600		70	560	8
atlas	chameleon-a	5	0	0	0	5	0	0	0	40	40		1	2		42	512		5	40	0

Figure 1: CSV2's web front-end cloud status and job status tables for ATLAS

First, utilizing the CLI was attempted. One consideration of choosing this option was the potential overhead. This was relatively straightforward and easy to implement as most of the coding was already a part of the CLI. A python script would run a subprocess `cloudscheduler` command and process the HTTP response JSON into Line Protocol for InfluxDB. An example `cloudscheduler` command can be seen in Figure 2, where the server is specified to be csv2-dev3 and the group (collection of clouds) is specified to be csv2-group. The results are requested back as comma separated values.

```
cloudscheduler cloud status -s csv2-dev3 -g csv2-group -CSV ‘’ -NV
```

Figure 2: Sample cloudscheduler command

The main disadvantage of this approach was that the CLI only queried one group at a time, and as CSV2 currently has two groups (ATLAS and BELLE) with the potential to add more, two commands are required. The drawback of this is that each command took approximately five to ten seconds to receive an HTTP response and sometimes ran over 15 seconds. This took much too long. Thus, the second approach was attempted and proved superior.

The second approach involved writing a python script that used SQLAlchemy to interact directly with MariaDB. The necessary views were already set up in phpMyAdmin, so the poller script queried each view and processed the returned data. After querying the database, an HTTP post request was sent to InfluxDB with the new data points. An excerpt from timeseriesPoller.py can be seen in Figure 3, where the cloud status measurements are parsed into Line Protocol. This approach usually took less than three seconds, depending on the system load.

```
# Parse cloud data into line protocol for influxdb
for line in cloud_status:
    column = 2
    group = line[0]
    if group not in groups and group != ‘’ or None:
        groups.append(group)
    cloud = line[1]
    for data in line[2:]:
        if data == -1 or data is None:
            column += 1
            continue
        if group == ‘’ or None:
            new_point = "{0},cloud={1} value={2}i {3}".format(
                column_list[column], cloud, data, ts)
        else:
            new_point = "{0},cloud={1},group={2} value={3}i {4}".format(
                column_list[column], cloud, group, data, ts)
        data_points.append(new_point)
        column += 1
```

Figure 3: Excerpt from timeseriesPoller.py

3.3 Data Storage

The data was stored as points in series in InfluxDB. The poller that queries data from MariaDB also needed to format that data appropriately and write it to the new TSDB with a timestamp. InfluxDB was set up to hold 30 second data for five years, with appropriate down-sampling to be added later as further testing was done to see how much space was used. The size and growth of the database depends on the number of unique series, tags, and fields. In InfluxDB, “[d]atabase names, measurements, tag keys, field keys, and tag values are stored only once and always as strings.

Only field values and timestamps are stored per-point. Non-string values require approximately three bytes [15]." One series is required for every unique measurement:cloud pair and one for every group:total-job-measurements pair. There are five job status measurements: jobs-total, -completed, -held, -idle, and -other. Additionally, there are 25 cloud status measurements that include the status of VMs-total, -starting, etc. and cores used, RAM used, etc. Based off InfluxDBs storage guide, this is a growth of almost 1.2 GB/year (see Figure 4 for calculation). This is a reasonable amount to keep for five years, considering the low granularity of this data; however, this was an estimated calculation, and the database size should be monitored to calculate an accurate measurement of growth.

```
Number of series = 5*<numberofgroups> + 25*<numberofclouds> + 11
                 = 5*2 + 25*7 + 11 = 196

Number of series * (<numberofvalues> + 1 timestamp) = 391

Number of bytes collected every 30s = 391 * 3 B
                                    = 1173 B

Number of 30s periods in 1 week = 20160
20160 * 1173 B = 23.6 MB/week or 1.23 GB/year.
```

Figure 4: Calculation for InfluxDB storage size

3.4 Visualization

The HEPRC group currently uses Elasticsearch for logging and monitoring and Kabana for detailed graphs and dashboards. Using Elasticsearch is time consuming for users due to Kabana's requirement to log in using a separate browser window. Additionally, the back-end is slow as the search engine is handling all the log and monitoring files.

Cloud Monitor implemented Plotly.js, a JavaScript graphing library for visualizing time series data [17]. This library allowed for a graph to be plotted within the current HTML page in a specified `<div>` element. This was the desired outcome, but other ways for visualizing the time series data were also examined.

Grafana first appeared to be the best choice. It provided customizable dashboards and built-in querying and basic analysis [18]. However, Grafana requires a search for the desired pre-prepared graph. While these graphs were arbitrarily editable, this was not ideal for the desired use. Furthermore, a user must be logged into Grafana in a separate browser window to view live graphs. Alternatively, a specific dashboard could be implemented in an `<iframe>` element within the current browser window, but this would not allow users to easily switch between time series, as the dashboard and graphs would need to be created prior to use.

While Chronograf, a dashboarding system, was part of the InfluxData TICK stack [12], and thus could almost be completely integrated with a simple download, it had many of the same limitations as Grafana in requiring a separate login/browsing window.

Cloud Monitor had success with Plotly.js. The in-web browser capabilities and easy integration into CSV2's current status page made it the most desirable option. There also existed some brief

documentation and tutorials with integrating InfluxDB and Plotly. Plotly.js also had proficient documentation and a large, active community forum [13].

Previously, CSV2's web front-end did not require JavaScript and was very mobile friendly as a result. Implementing Plotly involved writing JavaScript to interact with the library. When a user selects a measurement from one of the status tables, that measurement name and the current desired time range (with a default of the last one hour) is sent to InfluxDB. The returned data points are then added to a Plotly plot. However, to keep this implementation as lightweight and efficient as possible, no additional libraries (including jQuery) were used. This was very different from the implementation of Plotly in CSV1 and required more JavaScript to accomplish what may have been one or two lines in jQuery; however, as a result, CSV2's web front-end has no unnecessary code or dependability. A plot in CSV2's web front-end implemented with JavaScript and Plotyly.js can be seen in Figure 5. Additionally, unlike Cloud Monitor, all time series plots are updated every 30s with real-time status data, as discussed in section 3.5.2.

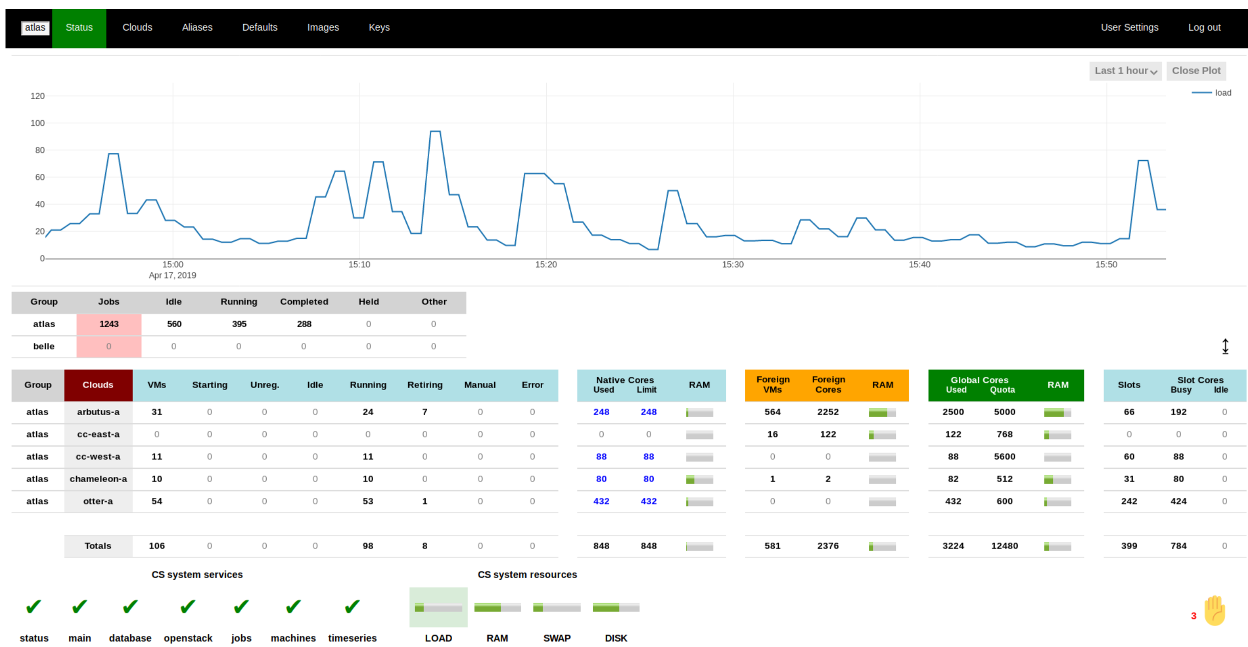

Figure 5: CSV2's web front-end with plot using Plotly.js

3.5 Retrieving Data

The data points for a specific trace needed to be queried upon user prompt. Using JavaScript, a connection to the TSDB was set up to request the full traces of status measurements as they were selected by a user. Two technologies were researched to determine the best approach: WebSocket and AJAX.

3.5.1 Websocket and Django Channels

The WebSocket API implements WebSocket, a communication protocol, which allows a direct, two-way connection between a browser and a server [19]. This type of connection only needs to be set up upon the initial interaction, and then it will remain open, ensuring a faster request/response

rate. It is ideal for real-time data transfer, particularly where there are multiple clients (e.g. in a chat-room). Django Channels allows Django to make use of the server-browser connection of WebSocket [20].

While Django Channels would enable the use of WebSocket and maintain a connection from server-side to client-side, this set up was unnecessary. "WebSocket's protocol allows bi-directional communication, meaning that the server can push data to the client without being prompted by the user [19]." This is entirely superfluous for HEPRC's use-case because the time series data should only be sent on client-side request.

3.5.2 AJAX, Django and the Fetch API

AJAX, an asynchronous process for sending HTTP requests in JavaScript, would set up an HTTP connection with the server for the requested URL. Each connection requires sending and receiving HTTP headers, increasing the overall time of one request. JavaScript has a Fetch API that returns promise objects (representing an asynchronous operations success or failure and result) and simplifies the use of AJAX [21, 22]. CSV2 is built upon Django web framework that allows views to be set up to handle specific URL requests. This allows a view in Django to directly query InfluxDB based upon the requested traces and return the response in JSON format. An example view that uses *Requests*, a python library for HTTP requests, can be seen in Figure 6.

```
def request_ts_data(request):
"""
This function should receive a post request with a payload of an influxdb
query to update the timeseries plot.
"""

params = {'db': 'csv2_timeseries','epoch': 'ms', 'q':request.body}
url_string = 'http://localhost:8086/query'
r = requests.get(url_string, params=params)
# Check response status code
r.raise_for_status()

return JsonResponse(r.json())
```

Figure 6: Django view excerpt from cloud_views.py

Based on a brief analysis of CSV2's use-case and research of the benefits of WebSocket vs AJAX, AJAX was determined as sufficient. After the web front-end of CSV2 loads, the initial client-server communication is limited to one HTTP request for each trace a user selects. Typically, users view between one and five traces each session. Cloud Monitor updates its status tables every 60 seconds; however, its graphs are not updated dynamically but instead require the user to manually refresh the page. CSV2's front-end automatically refreshes the status tables every 30 seconds, much like Cloud Monitor, but support for viewing live graphs that updated with the rest of the status tables was ideal. In MariaDB, the cloud, job, and system status information is updated every 30 seconds. The time series poller collects this data every 30 seconds as it is updated and stores it with a timestamp in InfluxDB. For viewing real-time graphs of this data in CSV2's web front-end, the

JavaScript was altered to send requests for the newest data point for each of the currently plotted series every 30 seconds. This provides live data and allows users to plot a trace with a range of the last five minutes and watch as the plot is continuously updated every 30 seconds to include the most recent data points. CSV2's 30 second refresh requires two HTTP requests: one for the status tables and one for live data of each series.

AJAX is sufficient for this number of requests. The one to two seconds needed to update the time series traces is inconsequential, as this result is not prompted by user request, and therefore a delay is not noticeable for a general user.

4 Amazon EC2 Support

Currently, HEPRC is adding support for Amazon Elastic Compute clouds (EC2) in CSV2. CSV2 was initially developed around Openstack clouds for ATLAS, with a plan to implement Amazon support for Belle II as CSV1 was phased out. Many polling features needed to be added to query Amazon EC2 service for information required to boot VMs and run jobs that included instance types and available Amazon Machine Images (AMIs). These scripts need to gather this data and insert it into MariaDB. The data would then be pulled from MariaDB and a user would select the appropriate options for booting VMs to run jobs either through the command line or web front-end.

4.1 Collecting Instance Types

Amazon provides public information on their available products and corresponding pricing in a global JSON offer file available at https://pricing.us-east-1.amazonaws.com/offers/v1.0/aws/ AmazonEC2/current/index.json [23]. The JSON is structured as a list of products with pricing available at the bottom of the file. Each product contains attributes that define an instance type: RAM, disk, and number of CPU cores. In Figure 7, instance type, m1.small, has 1.7 GB RAM, 160 GB disk, and one core. This is the information a user would need when selecting the appropriate instance type for their job(s).

```
"sku" : "CKRWNJGU4J7F9WE7",
"productFamily" : "Compute Instance",
"attributes" : {
  "servicecode" : "AmazonEC2",
  "location" : "US West (Oregon)",
  "locationType" : "AWS Region",
  "instanceType" : "m1.small",
  "currentGeneration" : "No",
  "instanceFamily" : "General purpose",
  "vcpu" : "1",
  "physicalProcessor" : "Intel Xeon Family",
  "memory" : "1.7 GiB",
  "storage" : "1 x 160",
  "networkPerformance" : "Low",
  "processorArchitecture" : "32-bit or 64-bit",
  "tenancy" : "Shared",
  "operatingSystem" : "Windows",
  "licenseModel" : "No License required",
  "usagetype" : "USW2-BoxUsage",
  "operation" : "RunInstances:0006",
  "capacitystatus" : "Used",
  "ecu" : "1",
  "normalizationSizeFactor" : "1",
  "preInstalledSw" : "SQL Std",
  "servicename" : "Amazon Elastic Compute Cloud"
}
```


Figure 7: Sample product entry from Amazon JSON

The poller script for collecting instance types queries a MariaDB table for a list of all EC2 clouds. A list of EC2 regions is then compiled from these entries. Using the python library *Requests*, a GET request for the Amazon offer's URL is sent and the returned JSON is parsed. Originally, the global offer file was requested as it would be easiest to only deal with one request and one response; however, this file turned out to be over 600 MB and was raising memory errors when it was loaded during the execution of the python script. A different solution was needed. After researching more into Amazons offer files, it was discovered that this information was available in region-specific files (i.e. one file for each region, such as us-west-2). The largest of these region-specific files is 38 MB. The instance type poller was re-written to query each region separately for the JSON file.

Testing the instance type poller was done using the region us-west-2. This region has approximately 190 unique instance types, which includes many that are superfluous to CSV2s users. Ideally, this number would be reduced to remove the unnecessary options and not overwhelm a user when selecting their preferred instance type. Filtering was required to manage the number of instance types.

4.2 Filtering and Segregation

A scheme was defined for filtering out the obsolete instance types. In MariaDB, a table was created called ec2_instance_type_filters. This table had columns for instance family, number of cores, processor types, and min and max memory (in GB) per core. A set of default filters were tested (Figure 8). The instance type poller was then written to query these filters from MariaDB and apply them to the JSON as each instance type was processed. This removed extraneous instance types from being added to MariaDB, and, in turn, would assist users with selecting the appropriate instance type. With the default filters, the number of instance types decreased to a more reasonable 19.

group_name	families	processor_type	cores	min_memory_gigabytes_per_core	max_memory_gigabytes_per_core
testing	Compute optimized,general purpose	Intel	1,2,4,8	1.5	3.0

Figure 8: ec2_instance_type_filters table from MariaDB

An issue that arose early on was lack of the consistency of the response from the Amazon pricing server. Occasionally, requesting the JSON for one file would take upwards of 4 minutes. Multiplied by 19 different regions could result in a delay over an hour. As the instance type poller would ideally be triggered every time a user updated the filter table so that it would populate MariaDB with the new instance types, this was not acceptable. It was decided that retrieving the Amazon JSON files would occur separately in a cron process and the files would be stored locally. This allowed the instance type poller to quickly grab the local region files and filter based on a users input. The separate cron task, ec2_retrieve_instance_type_files.py, was set up to query MariaDB for a list of all EC2 regions and then request the JSON file and store it locally in a configurable location. A crontab was written for this task to run every 24 hours. The instance type poller could then immediately open the local file when needed and parse the JSON.

4.3 Amazon Machine Images

Specifying an Amazon Machine Image (AMI) is required by users when setting up an instance. An image poller is required to compile a list of available AMIs much like instance types. However, Amazon has public images, Amazon-owned images, and images available on AWS Marketplace. This results in approximately 90 thousand different options, some of which are submitted by the general public and may lack security. Adding filters for which images a user may be interested in is required to manage this number. It was determined that some default filters would be applied, including only returning machine images that were currently available for use. While this decreased the number of options significantly, more user-specified filtering is required based on operating system, architecture, etc. and a scheme for doing so will be implemented in the future.

5 Conclusions

Because of the unique nature of time-series data, an optimal database would take advantage of its highly-structured character. Many databases were analyzed and InfluxDB, a database created

specifically for time series, was selected as the most appropriate because of its efficiency and popularity. A custom poller script, written in Python, was developed to transfer and parse data from the existing database, MariaDB, to InfluxDB. Using too many different technologies for clean and clear visualization of data can become cumbersome. Plotly.js was chosen as the most suitable for a quick, easy overview that was familiar to CSV2's current users and allowed for data to be plotted without requiring a user to open another browsing window or log in with their credentials. Plotly.js was implemented with additional JavaScript to allow for the plots to be customized to suit HEPRC's desired use-case, including being updated every 30 seconds with the most recent data available.

An instance type poller script was written to start the process of fully integrating support for Amazon EC2. These instance types were stored in MariaDB for the use of cloud scheduling software when the front-end and CLI are implemented to allow for a user to boot an appropriate VM. Filtering was added to the returned instance types to allow a user more control in narrowing a search or instance types.

6 Recommendations

While InfluxDB was originally set up to expire data after five years, the database offers down-sampling. This allows for older data to be stored at a lower resolution and kept for much longer in the database. Additionally, although the estimated growth of the database is 1.2 GB/year, watching the database size over a few months, up to a year, will provide a much more accurate measurement of growth. This growth rate will allow for an ideal down-sampling and data retention schema to be implemented.

Furthering the Amazon EC2 support requires a clear scheme for filtering out unnecessary AMIs, while still allowing a user control, along with UI implementation in the web front-end and CLI for both selecting instance types and images.

7 Acknowledgements

I would like to thank my manager, Dr. Randall Sobie, for providing me with this opportunity; my supervisor, Dr. Kevin Casteels, who guided me in this project while encouraging me to be independent, and who assisted with providing the access and resources this project required; and the rest of the HEPRC group, who were always willing to offer assistance.

References

[1] HEPRC, "HEP Research Computing," *High Energy Physics Research Computing.* [Online]. Available: http://heprc.phys.uvic.ca. [Accessed: Feb. 21, 2019].

[2] Bader, Andreas & Kopp, Oliver & Falkenthal, Michael. (2017). Survey and Comparison of Open Source Time Series Databases.

[3] Influx Data, "Time Series Database (TSDB) Explained," *Influx Data, Inc*, 2019. [Online]. Available: https://www.influxdata.com/time-series-database/. [Accessed: Feb. 21, 2019].

[4] MongoDB, Inc, "What is MongoDB?," *MongoDB, Inc*, 2019. [Online]. Available: https://www.mongodb.com/what-is-mongodb. [Accessed: Apr. 23, 2019].

[5] Elastic, "Elasticsearch," *Elasticsearch B.V.*, 2019. [Online]. Available: https://www.elastic.co/products/elasticsearch. [Accessed: Apr. 23, 2019].

[6] Elastic, "Kibana," *Elasticsearch B.V.*, 2019. [Online]. Available: https://www.elastic.co/products/kibana. [Accessed: Apr. 23, 2019].

[7] Graphite Project, "Overview," *The Graphite Project Revision*, 2017 [Online]. Available: https://graphite.readthedocs.io/en/latest/overview.html [Accessed: Feb 21, 2019].

[8] Graphite Project, "Graphite Documentation," *The Graphite Project Revision*, 2017. [Online]. Available: https://graphite.readthedocs.io/en/latest/faq.html. [Accessed: Feb. 21, 2019].

[9] Graphite Project, "Tools that work with Graphite," *The Graphite Project Revision*, 2017. [Online]. Available: https://graphite.readthedocs.io/en/latest/tools.html. [Accessed: Feb. 21, 2019].

[10] Prometheus, "Comparison to Alternatives," *Prometheus*, 2019. [Online]. Available: https://prometheus.io/docs/introduction/comparison/ [Accessed: Feb. 21, 2019].

[11] Influx Data, "Open Source Time Series Platform," *Influx Data, Inc*, 2019. [Online]. Available: https://www.influxdata.com/time-series-platform/. [Accessed: Feb. 21, 2019].

[12] Influx Data, "Chronograf," *Influx Data, Inc*, 2019. [Online]. Available: https://www.influxdata.com/time-series-platform/chronograf/. [Accessed: Feb. 21, 2019].

[13] Plotly, "Plotly.js," *Plot.ly*, 2019. [Online]. Available: https://community.plot.ly/c/plotly-js. [Accessed: Mar. 25, 2019].

[14] Influx Data, "Backing up and restoring in InfluxDB OSS," *Influx Data, Inc*, 2019. [Online]. Available: https://docs.influxdata.com/influxdb/v1.7/administration/backup_and_restore/. [Accessed: Mar. 7, 2019].

[15] Influx Data, "Hardware sizing guidelines," *Influx Data, Inc*, 2019. [Online]. Available: https://docs.influxdata.com/influxdb/v1.7/guides/hardware_sizing. [Accessed: Feb. 21, 2019].

[16] Influx Data, "Compare" *Influx Data, Inc*, 2019. [Online]. Available: https://www.influxdata.com/products/compare/. [Accessed: Feb. 21, 2019].

[17] Plotly, (2019, Feb) Plotly.js GitHub repository [Online]. Available: https://github.com/plotly/plotly.js. [Accessed: Feb. 21, 2019].

[18] Grafana Labs, "Grafana" *Grafana Labs*, 2019. [Online]. Available: https://grafana.com/grafana [Accessed: Feb. 21, 2019].

[19] Mozilla, "The WebSocket API (WebSockets)," *Mozilla*, 2019. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API [Accessed: Feb. 21, 2019].

[20] Channels, "Introduction," *Django Software Foundation Revision*, 2018. [Online]. Available: https://channels.readthedocs.io/en/stable/introduction.html. [Accessed: Feb. 21, 2019].

[21] Mozilla, "Fetch API," *Mozilla*, 2019. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API. [Accessed: Feb. 21, 2019].

[22] Mozilla, "Promises," *Mozilla*, 2019. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise [Accessed: Apr. 25, 2019].

[23] Amazon Web Services, "Using the Bulk API," *Amazon Web Services, Inc*, 2019. [Online]. Available: https://docs.aws.amazon.com/awsaccountbilling/latest/aboutv2/using-ppslong.html. [Accessed: Apr. 23, 2019].