# Integration of a job completion accounting system in a HPC environment by extending a workload manager

Alejandro Sanchez Graells

July 3, 2015

Supervisor: Javier Bartolomé Rodríguez
Co-supervisor: Carles Fenoy Garcia
Tutor: Jordi Torres Viñals
Department of Computer Architecture

Màster en Enginyeria Informàtica
Facultat d'Informàtica de Barcelona (FIB)
Universitat Politècnica de Catalunya (UPC) - BarcelonaTech
Barcelona Supercomputing Center (BSC-CNS)

**Abstract**

During the last decades, the amount and heterogeneity of resources needed to satisfy the research community and company computing demands has increased considerably. Alongside this HPC[1] environment growth, new and rich-featured workload management systems have been appearing in order to solve the problem of configuring and managing in an efficient way the usage of a large variety of resources. Current batch systems provide a resource management framework that takes job[2] requirements, find the best resources to run the job, and monitor its progress. In addition to that, an important part of these managers is the job completion accounting sub-system. These sub-systems are used to gather and store job submission, execution and termination historical data so that relevant statistics can be generated for further analyzing, performance improvement and troubleshooting. The first part of this work has consisted in taking an existing open source workload manager, study its job completion sub-system and extend it by implementing a new plugin to store the information in a distributed, real-time search and analytics engine. The next goal has been to integrate the new featured manager in a real production cluster and add a new layer for user-friendly interaction. Finally, some basic experimentation through machine learning techniques has been carried out, pointing out a strategy for future work.

[1] High Performance Computing is the use of supercomputers and parallel processing techniques for solving complex computational problems.

[2] In the batch procsesing context, a job is a program or set of programs that are encapsulated as a single unit. They are prepared to be submitted and run to completion without further human interaction.

## Acknowledgements

First and foremost, I would like to express my sincere gratitude to my parents and my younger sister. After many years of unconditional support and encouragement, only beautiful words come to my mind when recalling how important and necessary they are in my life. I feel deeply indebted to their effort put on my personal growth, so this thesis is specially dedicated to them. *Us estimo.*

A very special thanks goes out to my co-supervisor and colleague Carles Fenoy, whose expertise, understanding and patience added considerably to this work, specially at the beginning. I've met very few people so technically skilled and with this level of profound knowledge in many areas of Computer Science, and I can assure that I've meet quite a few engineers, professors and researchers throughout my career and professional years. I wish you all the best, and hope you enjoy this great new stage of your life.

I must also acknowledge my supervisor and manager Javier Bartolomé, who gave me the opportunity to join this wonderful team at BSC and let me take out part of the work hours for this project. I honestly hope that it will be useful. Certainly appreciate the writing suggestions and advice too. I would also like to mention Jordi Torres, who didn't hesitate a second when I proposed him being my tutor for this thesis. He has continuously been contributing with evident experience during the meetings and emphasized the importance of achieving the key deadlines.

My gratitude also goes to Marta Arias from the CS department for her discussions and advice with the Machine Learning topics. I had the pleasure to be her student in Computing and Intelligent Systems, where I discovered a lot of interesting topics about Bayesian Networks and Machine Learning among other fields.

Another special thanks goes to Morris Jette, CTO at SchedMD. He thoroughly reviewed the code pointing out some minor problems and elegant enhancements. He also conducted a very professional and bold transition to the master branch and sent me a really nice compliment about the plugin code. The open source concept is just astonishing.

I can't forget my colleagues at BSC. Thanks to JR, Prode, Filip, Marcel, Benet, Sergi, Josep, Janko, Tripi, Anibal, Berni, Borja, Toni, Pedro, Riera,

Ferran, Pedro, Cristian, Armenta, Javi and David; and the mates that decided to take another direction but I had the pleasure to meet. They all create a really nice atmosphere at work which makes this team something worth being part of, despite I think that a little bit of investment in attracting female members to the team would be interesting.

I would like to say thanks to my friends too, you all know who you are (specially to David and Raul); to my cousin Dani and to special women that one way or another have been or still are part of my life: Judit, Anastasia, Miriam and Marta.

Finally, I send kisses to my grandmother, who peacefully passed away last year. She spent some time teaching me calligraphy when I was a child, and stirred up a few genuine values such as curiosity, humbleness and caring for the details that have been key catalysts in my personality.

# Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

## 1.1 Problem Statement. Motivation

Managing, monitoring and analyzing the performance of a medium to large scale HPC facility are quite complex tasks which require a considerable amount of resources and effort. Within this context, it is reasonable to believe that the more we know about how a HPC environment is being used, the better we'll be able to understand whether we are taking advantage of it in an efficient way.

The way workloads are submitted and executed in a HPC infrastructure is managed by workload managers. This kind of tools are highly parameterized, so that different policies, quality of services, partitions, limits and a rich variety of options can be configured. Today, modern workload managers are capable of achieving utilization rates up to 99 per cent because of the flexibility and efficiencies they introduce in HPC cluster environments.

Taking a real infrastructure solution as a case study (MareNostrumIII[1]), the aim of this essay consists in analyze, design and implement a similar system that gathers historical information about finished jobs for different infrastructures. In order to accomplish this goal, the functionality of the Slurm[1] workload manager has been extended.

A job completion logging plugin has been implemented and integrated in two production clusters. A web layer frontend with rich customizable features such as histograms and charts provides rich analysis options to the

[1]Please, refer to Appendix A.1 for a MareNostrumIII overview.

end user. Finally, some statistics and machine learning mechanisms have been explored, so they can be directly used in possible future works.

## 1.2 BSC-CNS Infrastructure Overview

As pointed out above, the starting point for this work has been taking a real specific production infrastructure as a case study and develop a general solution (bottom-up approach). This production infrastructure is the MareNostrumIII supercomputer under BSC-CNS[2]. A brief history and background about this organization is provided below.

Early in 2004 the Ministry of Education and Science (Spanish Government), Generalitat de Catalunya (local Catalan Government) and Technical University of Catalonia (UPC) took the initiative of creating a National Supercomputing Center in Barcelona. BSC-CNS (Barcelona Supercomputing Center - Centro Nacional de Supercomputación) is the National Supercomputing Facility in Spain and was officially constituted in April 2005. BSC-CNS manages MareNostrum, one of the most powerful supercomputers in Europe, located at the Torre Girona chapel. The mission of BSC-CNS is to research, develop and manage information technology in order to facilitate scientific progress. With this aim, special dedication has been taken to areas such as Computer Sciences, Life Sciences, Earth Sciences and Computational Applications in Science and Engineering.

The Barcelona Supercomputing Center has different HPC facilities and they are used for different purposes. Each cluster has its own characteristics, configuration and reason for being; and in the following table it's summarized an overview of the main ones.

| **Cluster** | **Nodes** | **Cores** | **Users** | **Details** |
|---|---|---|---|---|
| MareNostrumIII | 3,056 | 48,896 | ˜800 | Appendix A.1 |
| MinoTauro | 126 | 1512 | ˜690 | Appendix A.2 |
| CNAG | 120 | 1120 | ˜115 | Appendix A.3 |
| BSCCV | 22 | 352 | ˜100 | n/a |
| ICGC | 4 | 48 | ˜12 | n/a |
| Altix 2 UV100 | 1 | 96 | ˜112 | Appendix A.4 |
| Storage | 4 | 64 | ˜1870 | n/a |

Table 1.1: BSC-CNS main clusters overview

For the purpose of this work, a relevant characteristic is the workload manager used on each of the clusters. Basically, MareNostrumIII is currently managed by a privative solution, the IBM Platform LSF[3], and the rest of the clusters are managed with Slurm.

Prior to the start of this work, an information system was already implemented and deployed in MareNostrumIII in order to visualize and analyze finished jobs' historical data. In the next section, it's explained the design of this information system in its initial state.

Another characteristic to put the focus on is the volume of finished jobs per cluster. This volume is important because if the rest of clusters had a negligible or almost negligible amount of data to be analyzed, this work wouldn't have much sense. The following figure illustrates the number of finished jobs per month and per cluster during the year 2014.

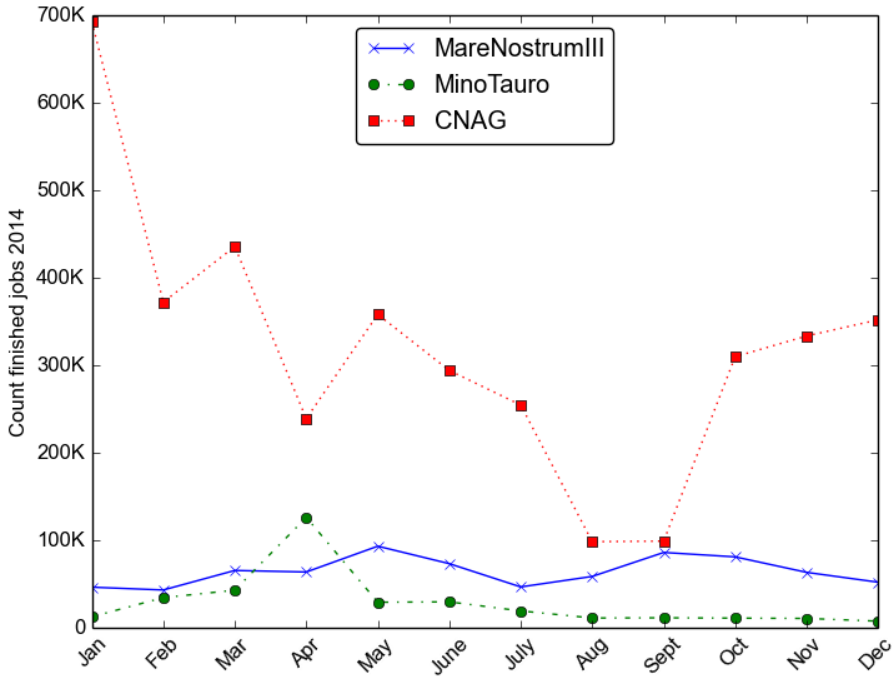



Figure 1.1: Number of finished jobs/month per cluster (2014)

There are two observations that can be extracted from the figure above:

1. The first one is that even the MinoTauro[2] cluster, which had the less amount of finished jobs out of the three main ones, had an average of 28455.16 finished jobs per month during the year 2014. This is a big amount of data from which valuable information can be analyzed.

2. The second observation, talking in relative terms, is that at first sight one may think that MareNostrumIII, with the highest amount of CPU cores (48,896), should be the cluster with more finished jobs. But it is not. The CNAG[3] cluster, with much fewer resources (compared to MareNostrumIII), is the one with the highest average of finished jobs per month. This is because the CNAG cluster is mainly used for genomics research purposes and, besides the jobs submitted by the researchers, there are many jobs automatically submitted with input from the DNA sequencers[4].

The fact that just having a look at this simple chart one can deduce some interesting information suggests that if more data could be gathered, more valuable information could be deduced through analysis. For instance, a CNAG system administrator might ask himself what percentage of the finished jobs come from the researchers and what comes from the sequencers. Another desirable information would be to know whether the different groups consuming resources on a cluster are really consuming or not the agreed SLA[5].

Without too much effort one can imagine a list of questions that would be nice to be answered if the right data is somehow stored and ready to be retrieved and analyzed. In some cases this information could not only be desirable but even critical to have. And that's why this work is useful and justifiable.

[2]Refer to Appendix A.2 for a MinoTauro overview.

[3]Refer to Appendix A.3 for a CNAG (Centre Nacional d'Anàlisi Genòmica) overview.

[4]A DNA sequencer is a scientific instrument used to automate the DNA sequencing process. Given a sample of DNA, a DNA sequencer is used to determine the order of the four bases: adenine, guanine, cytosine, and thymine. The order of the DNA bases is reported as a text string, called a read. Some DNA sequencers can be also considered optical instruments as they analyze light signals originating from fluorochromes attached to nucleotides.

[5]A service-level agreement (SLA) is part of a service contract where a service is formally defined.

# Chapter 2

# Underlying Technologies And Formats

The system design and development requires an understanding of two underlying technologies prior to its explanation. So this chapter is intended for clarifying purposes and the reader can skip it if being familiar with JSON and what's an Elasticsearch cluster, its components and how to interact with it. Most of the content in this chapter is transcribed from the official documentation.

## 2.1 JSON

JSON[8] stands for JavaScript Object Notation and it is a lightweight data-interchange format. An object in the JSON context is an unordered set of name/value pairs. An object begins with { (left brace) and ends with } (right brace). Each name is followed by : (colon) and the name/value pairs are separated by , (comma):

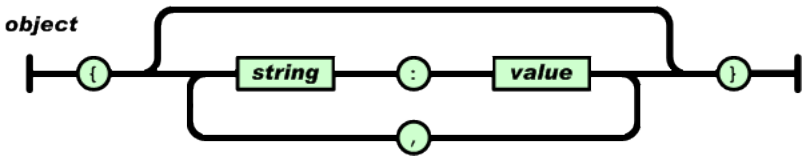



Figure 2.1: JSON object definition.

An array is an ordered collection of values. An array begins with [ (left bracket) and ends with ] (right bracket). Values are separated by , (comma):

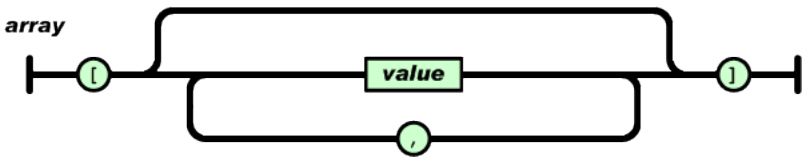



Figure 2.2: JSON array definition.

A value can be a string in double quotes, or a number, or true or false or null, or an object or an array. These structures can be nested:

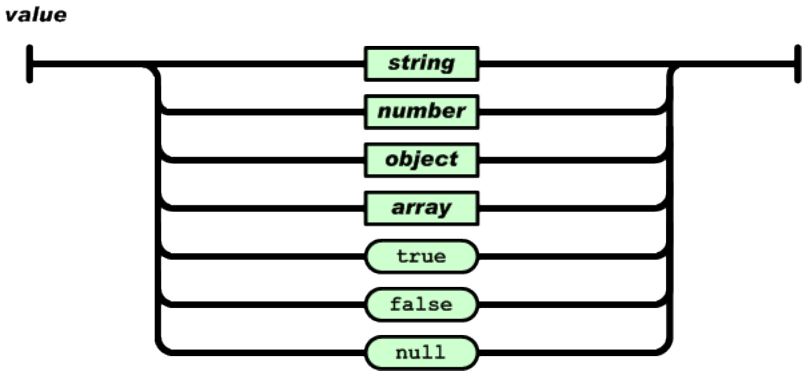



Figure 2.3: JSON value definition.

A number is very much like a C or Java number, except that the octal and hexadecimal formats are not used:

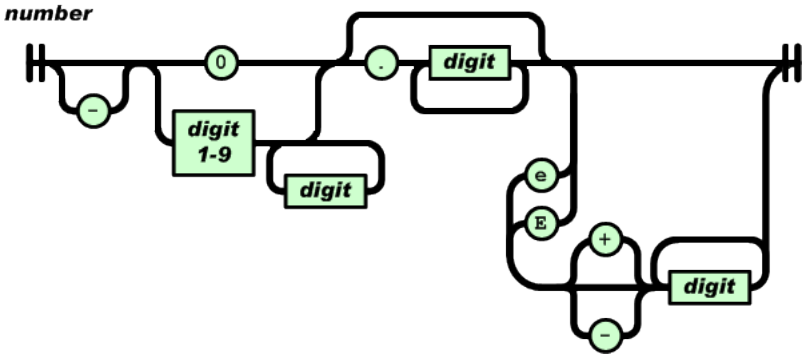



Figure 2.4: JSON number definition.

A string is a sequence of zero or more Unicode characters, wrapped in double quotes, using backslash escapes:

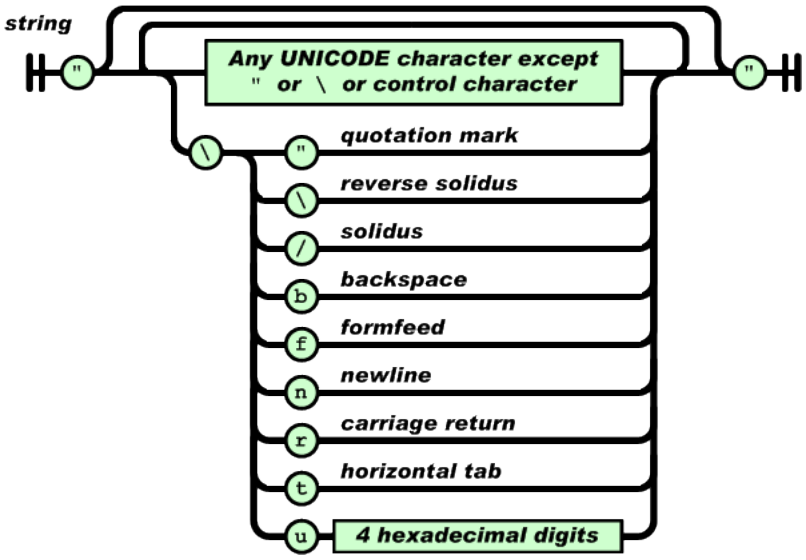



Figure 2.5: JSON string definition.

A character is represented as a single character string. Note that this escaping definition is specified in the `RFC7159`[9] and it's relevant for the system implementation.

## 2.2 Elasticsearch

### 2.2.1 Basic Concepts. Components

Elasticsearch is a highly scalable open-source full-text search and analytics engine. It allows to store, search, and analyze big volumes of data quickly and in near real time. It is generally used as the underlying engine/technology that powers applications that have complex search features and requirements.

Here are a few sample use-cases that Elasticsearch could be used for:

- To collect log or transaction data and to analyze and mine this data to look for trends, statistics, summarizations, or anomalies. Once the data is in Elasticsearch, it is possible to run searches and aggregations to mine any information that is of interest (this is exactly what is needed for this work).

- There are analytics/business-intelligence needs and there is a wish to quickly investigate, analyze, visualize, and ask ad-hoc questions on a lot of data (think millions or billions of records). In this case, it is possible to use Elasticsearch to store the data and then use Kibana (part of the Elasticsearch/Logstash/Kibana stack) to build custom dashboards that can visualize aspects of the data that are important. Additionally, you can use the Elasticsearch aggregations functionality to perform complex business intelligence queries against the data.

There exist more use-cases, but for the purpose of this work these two examples are more than enough to get a general description.

There are a few concepts that are core to Elasticsearch. Understanding these concepts from the outset will tremendously help to get a better idea of the plugin development.

- **Near Realtime (NRT)**. Elasticsearch is a near real time search platform. What this means is there is a slight latency (normally one second) from the time someone index a document until the time it becomes searchable.

- **Cluster**. A cluster is a collection of one or more nodes (servers) that together holds the entire data and provides federated indexing and search capabilities across all nodes. A cluster is identified by a unique name which by default is "elasticsearch". This name is important because a node can only be part of a cluster if the node is set up to join the cluster by its name. It is good practice to explicitly set the cluster name in production, but it is fine to use the default for testing/development purposes.

- **Node**. A node is a single server that is part of the cluster, stores the data, and participates in the cluster's indexing and search capabilities. Just like a cluster, a node is identified by a name which by default is a random Marvel character name that is assigned to the node at startup. Any node name can be defined, if not defined the default one will be used. This name is important for administration purposes where someone wants to identify which servers in the network correspond to which nodes in the Elasticsearch cluster.

  A node can be configured to join a specific cluster by the cluster name. By default, each node is set up to join a cluster named elasticsearch which means that if someone starts up a number of nodes on the

network and—assuming they can discover each other—they will all automatically form and join a single cluster named elasticsearch.

In a single cluster, it's possible to have as many nodes as desired. Furthermore, if there are no other Elasticsearch nodes currently running on the network, starting a single node will by default form a new single-node cluster named elasticsearch.

- **Index**. An index is a collection of documents that have somewhat similar characteristics. For example, someone can have an index for customer data, another index for a product catalog, and yet another index for order data. An index is identified by a name (that must be all lowercase) and this name is used to refer to the index when performing indexing, search, update, and delete operations against the documents in it.

  In a single cluster, it's possible to define as many indexes as desired.

- **Type**. Within an index, someone can define one or more types. A type is a logical category/partition of the index whose semantics is completely up to the administrator. In general, a type is defined for documents that have a set of common fields. For example, let's assume someone runs a blogging platform and store all his/her data in a single index. In this index, the administrator may define a type for user data, another type for blog data, and yet another type for comments data.

- **Document**. A document is a basic unit of information that can be indexed. For example, someone can have a document for a single customer, another document for a single product, and yet another for a single order. This document is expressed in JSON (JavaScript Object Notation) which is an ubiquitous internet data interchange format.

  Within an index/type, it's possible to store as many documents as desired. Note that although a document physically resides in an index, a document actually must be indexed/assigned to a type inside an index.

- **Shards and Replicas**. An index can potentially store a large amount of data that can exceed the hardware limits of a single node. For example, a single index of a billion documents taking up 1TB of disk space may not fit on the disk of a single node or may be too slow to serve search requests from a single node alone.

To solve this problem, Elasticsearch provides the ability to subdivide an index into multiple pieces called shards. When an index is created, one can simply define the number of shards. Each shard is in itself a fully-functional and independent "index" that can be hosted on any node in the cluster.

Sharding is important for two primary reasons:

- It allows administrators to horizontally split/scale your content volume.
- It allows administrators distribute and parallelize operations across shards (potentially on multiple nodes) thus increasing performance/throughput.

The mechanics of how a shard is distributed and also how its documents are aggregated back into search requests are completely managed by Elasticsearch and is transparent to the administrator as the user.

In a network/cloud environment where failures can be expected anytime, it is very useful and highly recommended to have a failover mechanism in case a shard/node somehow goes offline or disappears for whatever reason. To this end, Elasticsearch allows administrators to make one or more copies of their index's shards into what are called replica shards, or replicas for short.

Replication is important for two primary reasons:

- It provides high availability in case a shard/node fails. For this reason, it is important to note that a replica shard is never allocated on the same node as the original/primary shard that it was copied from.
- It allows to scale out the search volume/throughput since searches can be executed on all replicas in parallel.

To summarize, each index can be split into multiple shards. An index can also be replicated zero (meaning no replicas) or more times. Once replicated, each index will have primary shards (the original shards that were replicated from) and replica shards (the copies of the primary shards). The number of shards and replicas can be defined per index at the time the index is created. After the index is created, it's possible to change the number of replicas dynamically anytime but it's not possible to change the number shards after-the-fact.

Here's an example of an Elasticsearch configuration schema:

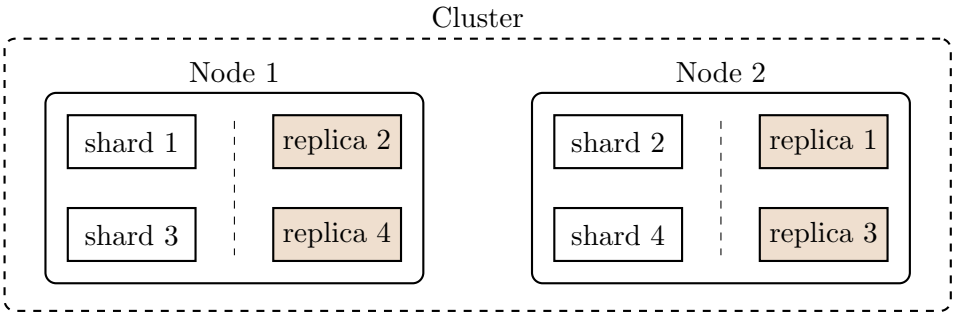



Figure 2.6: Example of Elasticsearch configuration.

### 2.2.2 Elasticsearch REST API

Once there is an Elasticsearch up and running, the next step is to understand how to communicate with it. Elasticsearch provides a very comprehensive and powerful REST API that can be used to interact with the cluster. Among the few things that can be done with the API are as follows:

- Check the cluster, node, and index health, status, and statistics.
- Administer the cluster, node, and index data and metadata.
- Perform CRUD (Create, Read, Update, and Delete) and search operations against the indexes.
- Execute advanced search operations such as paging, sorting, filtering, scripting, faceting, aggregations, and many others.

For the purpose of this project, it's just needed to understand how to perform index operations. These operations are performed through HTTP/REST calls. There are many tools to make this kind of calls and for the following examples curl is going to be used.

Let's assume there is an Elasticsearch cluster up and running. In order to index a document, PUT or POST methods can be used. For this example, POST is used because this method handles the automatic creation of the index if it's not previously created. Also, and id will be generated automatically.

```
$ curl -XPOST 'http://serv:9200/itest/ttest?pretty' -d '{
  "username" : "Alejandro Sanchez",
  "somefield" : "Example",
  "otherfield" : 1234
}'
{
  "_index" : "itest",
  "_type" : "ttest",
  "_id" : "xMOVtLX2RFCtB9DeXOJPBg",
  "_version" : 1,
  "created" : true
}
$
```

This is an example of an HTTP/REST call and the response from the Elasticsearch server. Being familiar with this operation is relevant for further chapters understanding.

# Chapter 3

# System Design

At the time of starting this work, a system for managing historical data about finished jobs in the MareNostrumIII cluster was already set up in production. The design and implementation of this system was lead by operators at the BSC-CNS facilities.

Along this chapter it's explained this baseline solution architecture, the different approaches toward the new design together with their discussion and the final selected design.

## 3.1 Baseline Architecture

As been mentioned, MareNostrumIII is managed by the IBM Platform LSF[3] workload manager. So the starting point of this sytem is `mbatchd`[4].

This process is the LSF's Master Batch Daemon and it runs on the master scheduling host. It is responsible for the overall state of jobs in the system. It also receives job submission and information query requests, manages jobs held in queues and dispatches jobs to hosts as determined by `mbschd` (Master Batch Scheduling Daemon).

It is shown below the schema with the different components and processes which compose the MareNostrumIII baseline architecture and the explanation of how the information flows through it.

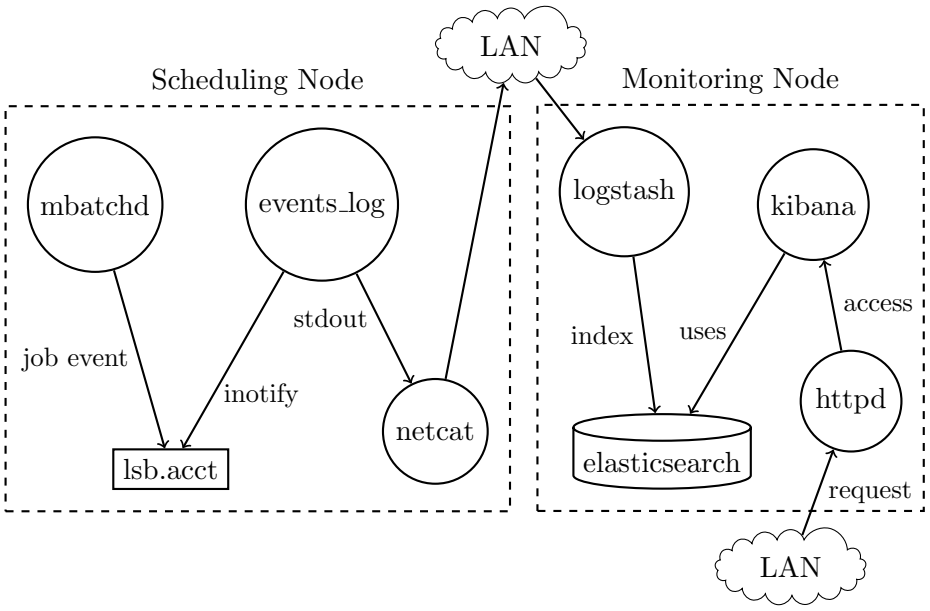



Figure 3.1: MN3 finished jobs system schema

The `lsb.acct`[5] file is the LSF's batch job log file[1]. The master batch daemon generates a record for each job completion or failure, for each job resize and for each expiration of an advance reservation[2]. The record is appended to the job log file `lsb.acct`. The job log file is an ASCII file with one record per line. The fields of a record are separated by blanks. If the value of some field is unavailable, a pair of double quotation marks ("") is logged for character string, 0 for time and number, and -1 for resource usage.

The second process that come into play is `events_log`. This one is a C compiled binary which is monitoring events on the `lsb.acct` file. In order to do so, this process makes use of `inotify`[6] (a Linux kernel subsytem). The `inotify` API provides a mechanism for monitoring filesystem events, individual files, or directories. When a directory is monitored, `inotify` will return events for the directory itself, and for files inside the directory. So every time a record is appended to `lsb.acct`, the `events_log` detects and parses it. Then the process discards lines which represent job resizes or expirations of advanced reservations events and just processes the job completion or failure events.

[1]Please, refer to Appendix B.1 for LSF's directory structure.
[2]Advance reservations ensure access to specific hosts during specified times.

After that, it prints the job information fields separated by blanks to stdout and pipes it to the `netcat` program, which is a feature-packed networking utility which reads and writes data across networks from the command line. In this context, netcat takes the job information from `events_log` process and redirects it to a concrete TCP port on the monitoring node.

In the monitoring node, there is an ELK Stack[7] and an Apache Web Server running. The ELK acronym stands for Elasticsearch, Logstash and Kibana. Logstash is a data pipeline that helps processing logs and other event data from a variety of systems. It can connect to a variety of sources and stream data at scale to a central analytics system.

So in this schema, Logstash is configured to listen to the TCP port where `netcat` is writing to, takes the job information and changes its format so that it can be indexed into Elasticsearch, which is a real-time distributed search and analytics engine. It is used for full-text search, structured search, analytics, and all three in combination. Detailed explanation can be found in the section 2.2.

Once data is gathered, Kibana comes into play. It's open source analytics and visualization platform designed to work with Elasticsearch. It is used to search, view, and interact with data stored in Elasticsearch indices. It allows users to perform advanced data analysis and visualize their data in a variety of charts, tables, and maps. Kibana makes it easy to understand large volumes of data and its simple, browser-based interface enables users to quickly create and share dynamic dashboards that display changes to Elasticsearch queries in real time.

Finally, a `httpd` web server is running to attend the requests to the Kibana webpage.

All in all, this was the BSC-CNS solution to analyze historical information about finished jobs in the MareNostrumIII cluster. One of the goals of this thesis was to have a similar solution for the rest of the clusters under BSC facilities. In the next sections, it's explained some alternative features that Slurm offers to accomplish the same task, which decisions have been taken and a detailed description of the different design approaches.

## 3.2 Approaching The System Design

The main target is to have a solution like the one in MareNostrumIII for the rest of the BSC clusters. Having in mind that the baseline architecture worked properly, a tempting solution could be applying the same design in the rest of the clusters. Let's call this approach *full replication* from now on. This approach is not viable due to the fact that MareNostrumIII works with IBM Platform LSF and the rest of clusters with Slurm, and this has an impact over the initial schema. The following sections detail the pitfalls encountered with different approaches.

## 3.3 Full Replication Approach

So this first approach would consist in replicating the MareNostrumIII baseline schema by just replacing the LSF `mbatchd` process by its analogous in Slurm, the `slurmctld` (Slurm controller daemon). The following figure illustrates this possible scenario. The green node means newly added and the yellow ones mean modified, as compared to the initial baseline architecture.

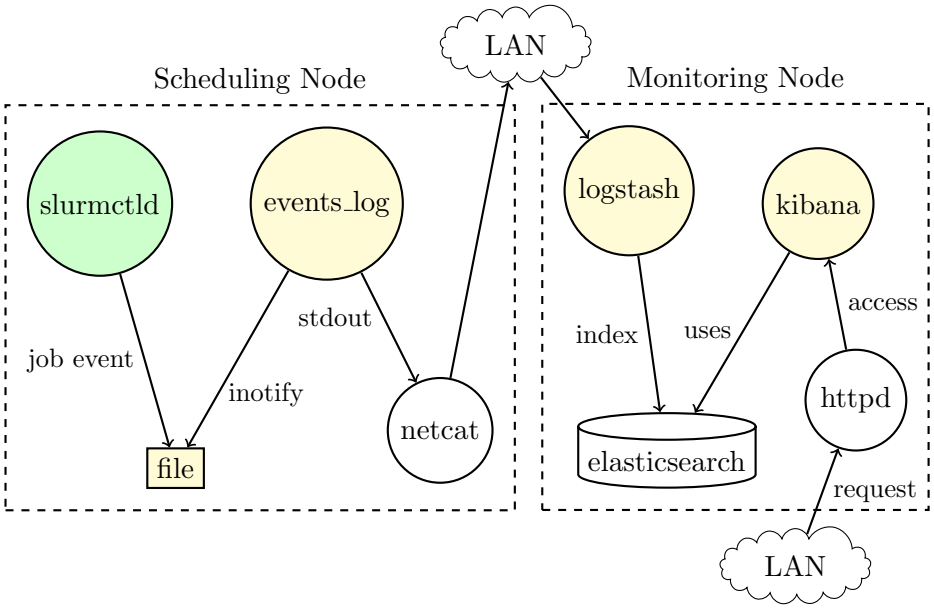



Figure 3.2: Possible alternative design. Full replication.

Let's explain why this approach is not viable. The `slurmctld` does not write job completion logging information by default to a file, as compared to LSF's `mbatchd`. However, it is possible to do that by enabling the `filetxt` job completion plugin. Anyhow, the schema would continue having some problems.

The first one is that this file would not have the same format as the original LSF's `lsb.acct`, so the `events_log` process source code should be adapted to interpret this new format and so with the Logstash source patterns. The Kibana dashboards configuration should also be adapted but it would be a minor change.

That would still be a feasible option, but there is a second problem. The `filetext` plugin does not record the information of all the requried job fields, it just records a small subset of them.

## 3.4 Using Alternative Plugins Approach

Another possible alternative would be looking to other job completion logging plugins which Slurm offers. The current official documentation[10] specifies that these plugins implement the Slurm API for logging job information upon their completion. This may be used to log job information to a text file, database, etc. The plugins must conform to the Slurm Plugin API. At the time of starting this work, there were four plugins of this type:

| **Plugin** | **Description** |
|---|---|
| **none** | No job logging |
| **filetxt** | Log job information to a text file |
| **mysql** | Job completion is written to a mysql database |
| **script** | Execute a script passing in job info in environment variables |

Table 3.1: Slurm jobcomp type available logging plugins.

The `filetxt` option is already discarded, so the other two options are analyzed. See below how would the schema change with these other two alternatives, `mysql` or `script`:

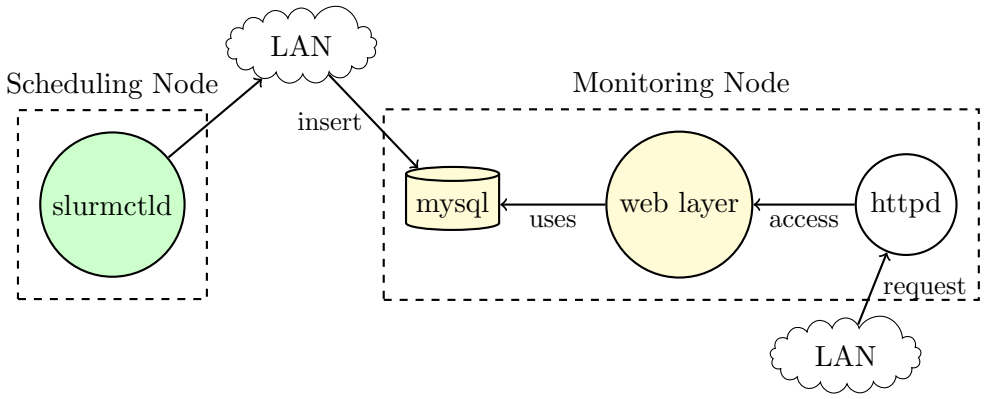



Figure 3.3: Possible alternative design. Mysql plugin.

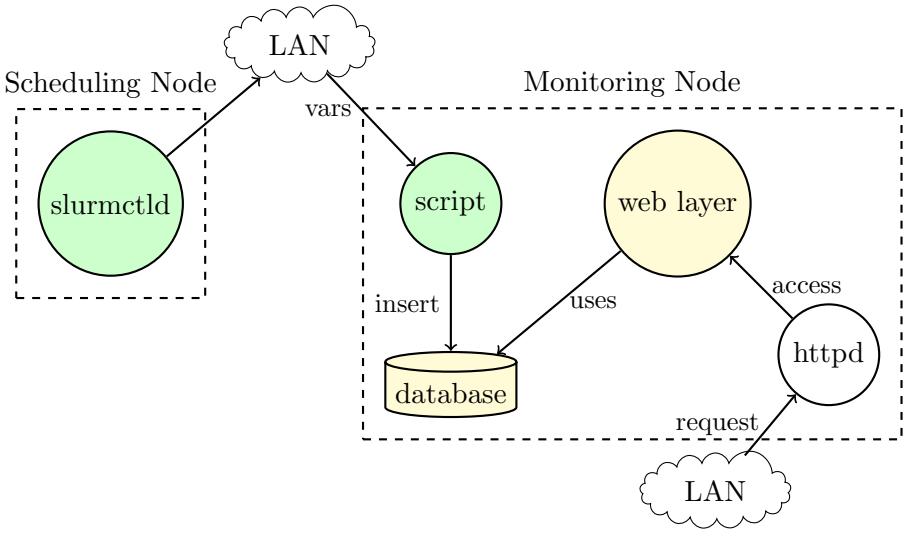



Figure 3.4: Possible alternative design. Script plugin.

These two alternatives have some pitfalls too. The first one is that the web layer should be re-designed so that it could gather the information not from Elasticsearch, but from a `mysql` database or maybe another kind of database in the case of the `script`. This has another consequence, and it is that the end users wouldn't have the same interface they are used to with the original schema with Kibana. The second one is that neither the `msyql` plugin nor the `script` one record, again, enough job related fields as it is requried.

## 3.5 Final Architecture

So the final selected design is illustrated in the following schema and discussed immediately.

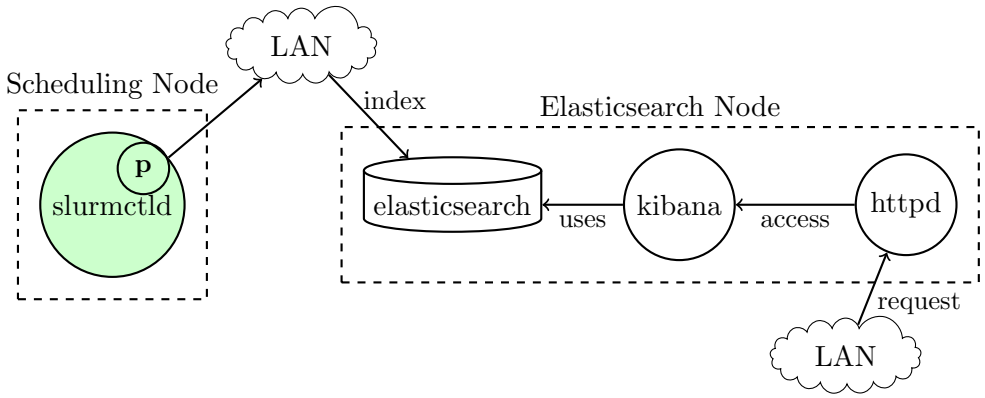



Figure 3.5: Final design. Jobcomp/elasticsearch plugin.

This solution implies the development of a new Slurm job completion plugin (denoted with the *p* circle in the figure) to index the job information into an Elasticsearch server. The solution is simple but removes the overhead and possible errors produced by the intermediate components in the initial schema. Note that the `lsb.acct` file, the `events_log` process, `netcat` and Logstash are not needed anymore, because the plugin already takes care of the proper information handling, processing and indexation into Elasticsearch. This solution extends the functionality offered by Slurm taking advantage of its open source nature.

In the Chapter 4, the development of this plugin, the problems encountered and how they have been solved are explained with detail.

# Chapter 4

# Slurm Plugin Implementation

As mentioned before, a Slurm job completion logging plugin is a Slurm plugin that implements the Slurm API for logging job information upon its completion. The plugin must conform to the Slurm Plugin API with a concrete specification[10] and it has been implemented from scratch as part of this thesis.

## 4.1 Jobcomp Logging Plugins API

We will highlight three main aspects of the API specification: the **required variables** by the plugin interface, the **error handling** and the plugin **API methods**. So let's start with the required variables.

### 4.1.1 Requried Variables By Plugin Interface

There are three variables required by the generic plugin interface. If they are not found in the plugin, the plugin loader will ignore it:

- `plugin_name` - a string giving a human-readable description of the plugin. There is no maximum length, but the symbol must refer to a valid string.
- `plugin_type` - a string suggesting the type of the plugin or its applicability to a particular form of data or method of data handling. If the

low-level plugin API is used, the contents of this string are unimportant and may be anything. Slurm uses the higher-level plugin interface which requires this string to be of the form

`<application>/<method>`

where `<application>` is a description of the intended application of the plugin (e.g., "jobcomp" for Slurm job completion logging) and `<method>` is a description of how this plugin satisfies that application. Slurm will only load job completion logging plugins if the `plugin_type` string has a prefix of "`jobcomp/`".

- `plugin_version` - an unsigned 32-bit integer containing the Slurm version (major.minor.micro combined into a single number).

```
const char plugin_name[] = "Job completion elasticsearch logging
                           plugin";
const char plugin_type[] = "jobcomp/elasticsearch";
const uint32_t plugin_version = SLURM_VERSION_NUMBER;
```

### 4.1.2 Error Handling

The implementation must maintain (though not necessarily directly export) an enumerated `errno` to allow Slurm to discover as practically as possible the reason for any failed API call. Plugin-specific enumerated integer values should be used when appropriate. The error number should be returned by the function `slurm_jobcomp_get_errno()` and this error number can be converted to an appropriate string description using the `slurm_jobcomp_strerror()` function described below.

These values must not be used as return values in integer-valued functions in the API. The proper error return value from integer-valued functions is `SLURM_ERROR`. The implementation should endeavor to provide useful and pertinent information by whatever means is practical. Successful API calls are not required to reset any errno to a known value. However, the initial value of any `errno`, prior to any error condition arising, should be `SLURM_SUCCESS`.

### 4.1.3 Plugin API Methods Specification

A complete list with the plugin functions is shown below. The subset of functions whose name is not prefixed with an underscore are the API functions, and they must appear. For these subset of functions, the ones which are not implemented should be at least stubbed. The remaining functions are auxiliary and that's why they are defined with the static keyword, because they are only needed to be visible in their own translation unit (thus limiting their scope).

```
static void _get_user_name (uint32_t user_id, char *user_name,
                            int buf_size)
static void _get_group_name (uint32_t group_id, char *group_name,
                             int buf_size)
static char * _lookup_slurm_api_errtab (int errnum)
static uint32_t _read_file (const char *file, char **data)
static int _load_pending_jobs (void)
static size_t _write_callback (void *contents, size_t size,
                               size_t nmemb, void *userp)
static int  _index_job (const char *jobcomp)
static char * _json_escape (const char *str)
static int _save_state (void)
static void _push_pending_job (char *j)
static void _update_pending_jobs (int *m)
static int _index_retry (void)
static void _make_time_str (time_t *time, char *string, int size)
int init (void)
int fini (void)
int slurm_jobcomp_set_location (char *location)
int slurm_jobcomp_log_record (struct job_record *job_ptr)
int slurm_jobcomp_get_errno (void)
char * slurm_jobcomp_strerror (int errnum)
List slurm_jobcomp_get_jobs (slurmdb_job_cond_t *job_cond)
int slurm_jobcomp_archive (slurmdb_archive_cond_t *arch_cond)
```

## 4.2 Plugin Methods Relationship

For a better understanding of the code[1], a call graph is shown below. A call graph (also known as a call multigraph) is a directed graph (and more specif-

[1]Please, refer to this bibliography link[11] to one of the public repositories where the plugin code is available. Viewing the code will help understanding this chapter.

ically a flow graph) that represents calling relationships between subroutines in a computer program. Specifically, each node represents a procedure and each edge $(f, g)$ indicates that procedure $f$ calls procedure $g$. Thus, a cycle in the graph indicates recursive procedure calls. For the sake of simplicity, just the relationships between the functions listed in the previous list are shown (so the calls to other functions are not explicitly stated).

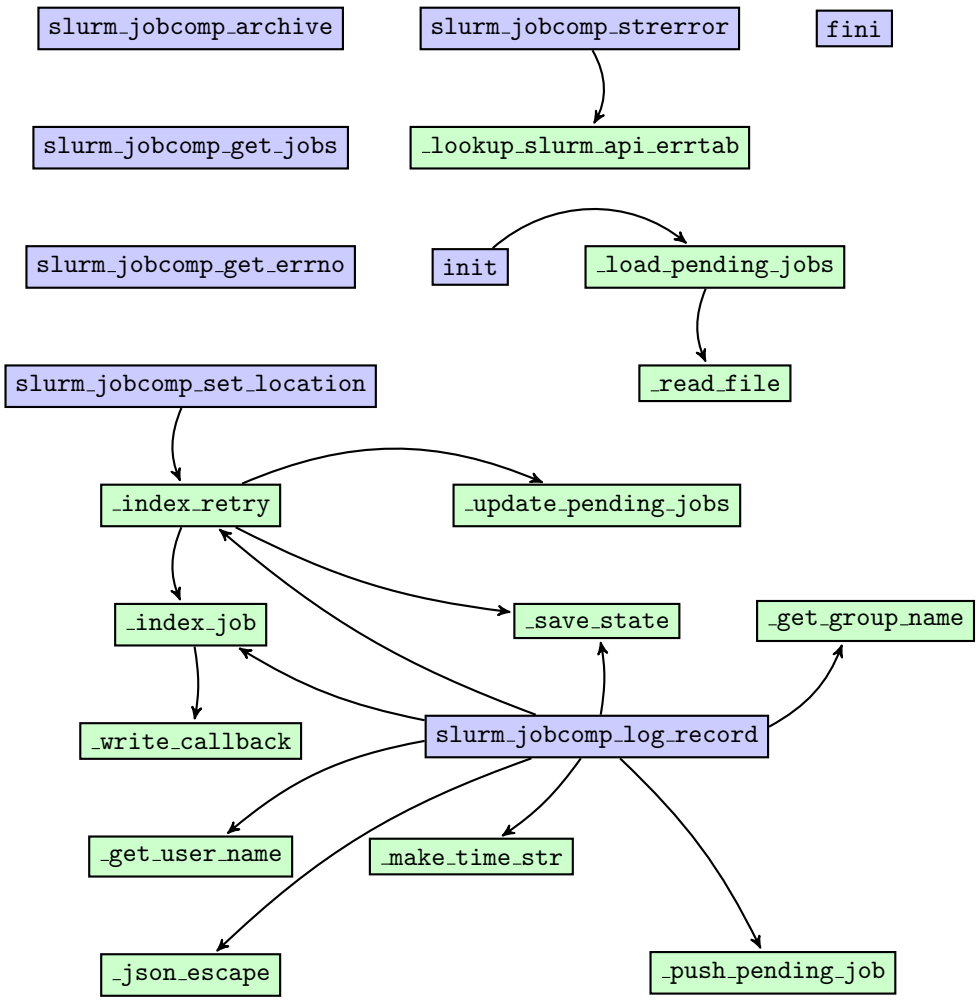



Figure 4.1: Call graph for jobcomp/elasticsearch plugin.

## 4.3 Slurm Plugin Design Management

The first plugin function executed is `init`. This function takes no arguments and is called when the plugin is loaded, before any other functions are called. It is intended to be used for global initializations and it returns `SLURM_SUCCESS` on sucess or `SLURM_ERROR` on failure.

How the `init` routine for a jobcomp plugin is called is part of the Slurm code and its plugin design. See below a call graph through the Slurm source internals until `init` is called.

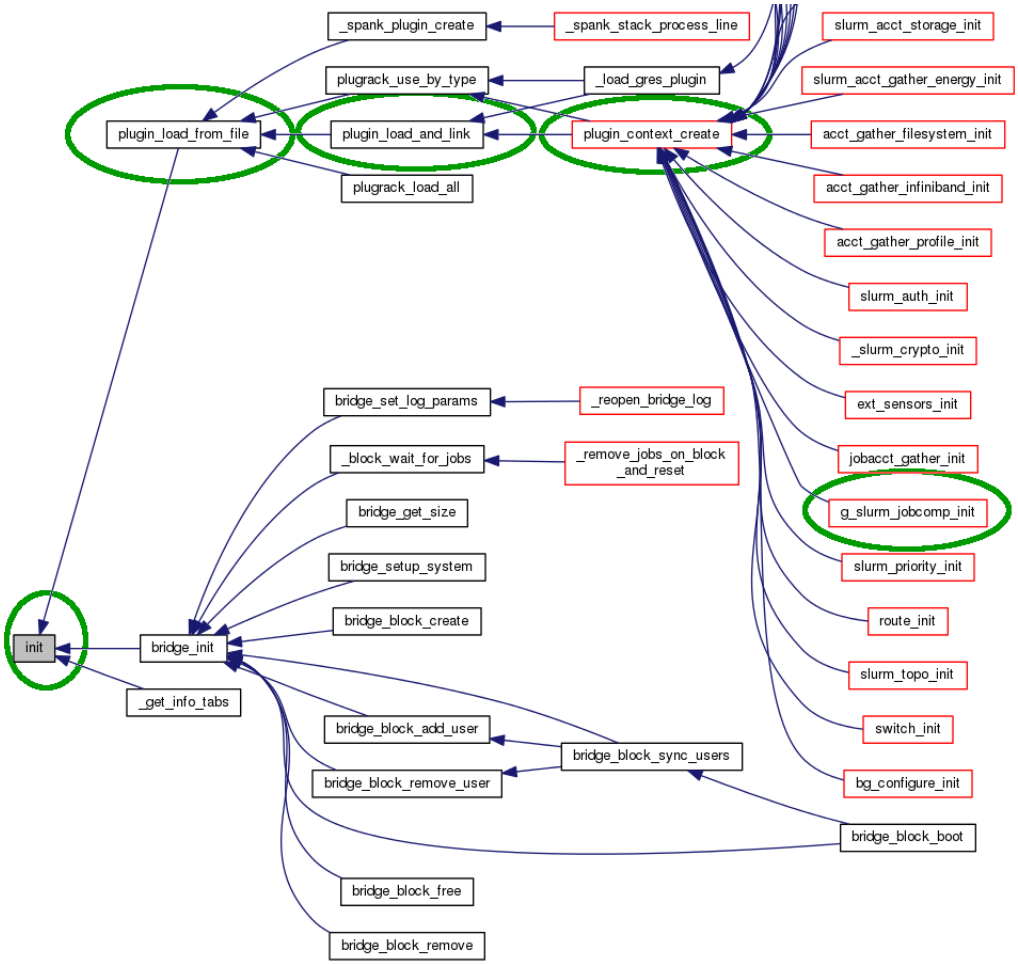



Figure 4.2: Call graph path from `g_slurm_jobcomp_init` to `init`.

So this figure shows that the g_slurm_jobcomp_init function calls to the plugin_context_create one. This, in turn, calls to plugin_load_and_link and this one to plugin_load_from_file respectively. Finally, the later calls to init.

All these functions are located across different source files. The following table shows which files contains each of these functions:

| Function | Relative path to Slurm build root directory |
|---|---|
| init | src/plugins/jobcomp/jobcomp_elasticsearch.c |
| plugin_load_from_file | src/common/plugin.c |
| plugin_load_and_link | src/common/plugin.c |
| plugin_context_create | src/common/plugin.c |
| g_slurm_jobcomp_init | src/common/slurm_jobcomp.c |

Table 4.1: Call graph functions mapped to their file location.

Digging deeper, g_slurm_jobcomp_init is called by a few more functions until the main function is reached. A call graph for this part is not shown anymore because it is not relevant for the scope of this work. It's just needed to understand that our init function is loaded once by the slurmctld daemon, which has this last main function located at:

src/slurmctld/controller.c.

If the init function returns SLURM_SUCCESS and a context can be created for the plugin, the slurm_jobcomp_set_location is called. This function is explained later and it's just called once during the plugin life cycle, as with init and fini. The slurm_jobcomp_strerror is called everytime an error occurs and finally the slurm_jobcomp_log_record is called everytime that a few type of events arise, such as a job's been completed, failed, it has received a signal because it has reached its time limit and so on.

So there is a coherence and a design behind the Slurm plugin structure. If the functions are implemented following the Slurm plugin API specification, this design will take care of them and it will call them when necessary. In the same way that there is a call graph back to the main function with init, the rest of the functions also have their call graph but it is not shown because it's just necessary to understand that they are called when appropriately and this management is done by the Slurm design. So we're just

going to focus at the level of our new developed plugin onwards:

`src/plugins/jobcomp/jobcomp_elasticsearch.c`

## 4.4 Jobcomp Elasticsearch Plugin Methods

The `init` function basically initializes some global variables and loads the information about pending jobs to be indexed from a state file into a global variable in memory. Detailed information about this global variable and how to access this state file is explained in other functions.

`slurm_jobcomp_set_location` specifies the location to be used for job logging. It takes a `char * location` as an argument. This location specifies where logging should be done. The interpretation of this string is at the discretion of the plugin implementation. In the case of this plugin, the value is read from the `JobCompLoc` parameter in the `slurm.conf` configuration file, and it should contain the URL and the port to reach the Elasticsearch server:

`JobCompLoc=http://ELASTICSEARCH_SERVER:PORT`

Example:

`JobCompLoc=http://localhost:9200`

In other jobcomp plugins, this value could be a MySQL server or the path to a text file, for example. The function tries to access the Elasticsearch server just to test if it is reachable and, in that case, it tries to reindex all the pending jobs located in the mentioned global memory variable. Pending jobs are jobs that, for whatever reason, couldn't be properly indexed in the past. For instance, the server could be temporarily unavailable or the index in read only mode. Later on it's explained how this `_index_retry` function is implemented.

A highly relevant function in the plugin is `slurm_jobcomp_log_record`. The goal of this API function, which is called every time a job is finished, is to log job related information somewhere. In the case of this plugin it tries to index the job information inside the Elasticsearch server configured in the `slurm.conf` parameter explained before. In order to do so, it first has to access the information, transform it and finally try to index it.

The function receives a pointer to a structure as a parameter. Here is the header:

```
int slurm_jobcomp_log_record (struct job_record *job_ptr);
```

The `job_ptr` is a pointer to a job record as defined in:

`src/slurmctld/slurmctld.h`

The implementation gathers a subset of the total desired information relative to a finished job from this structure. Later it is explained that not all the fields relative to a job can be retrieved from this variable, so more information sources need to be accessed. A collaboration diagram for a `job_record` struct is shown below:

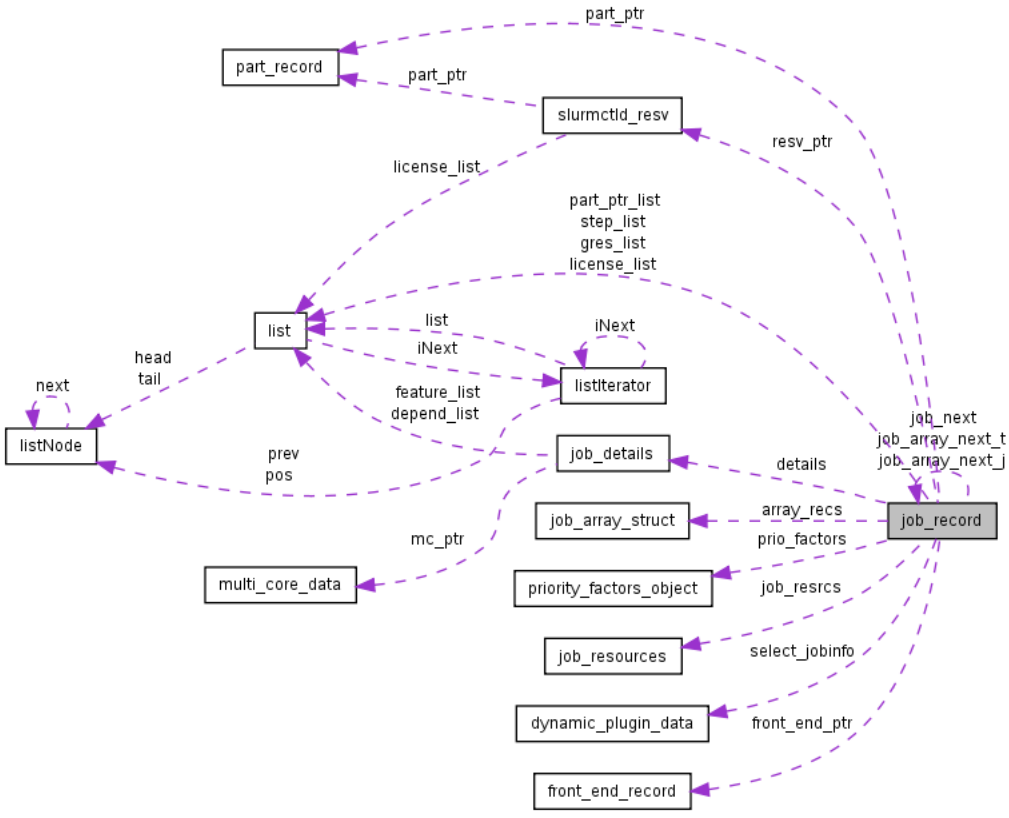



Figure 4.3: Collaboration graph for the `job_record` struct.

The job_record structure may contain information up to 115 fields, some of them being pointers to other structures that may respectively contain even more information as shown in the collaboration graph. For instance, job_ptr->user_id holds the information related to the user id bound to the current finished job. Then the function calls other non API functions to get even more information:

```
_get_user_name(job_ptr->user_id, usr_str, sizeof(usr_str));
_get_group_name(job_ptr->group_id, grp_str, sizeof(grp_str));
```

They obviously get the username and the groupname respectively. The slurm_jobcomp_log_record also calls a few times another function with different parameters:

```
_make_time_str(&job_ptr->start_time, start_str, sizeof(start_str));
_make_time_str(&job_ptr->end_time, end_str, sizeof(end_str));
```

The function _make_time_str transforms the start and end times of the job to the format YYYY-MM-DDTHH:MM:SS, which is the ISO8601 standard.

One by one, fields are being processed and saved in a local buffer. This buffer attempts to dynamically resize its allocated memory as needed depending on the size of the fields being added. Furthermore, as fields are being added to the buffer, they are previously formatted so that the buffer finally holds a correct JSON object.

One of the desired job fields is the job script. This job script is accessed through the get_job_script function and it returns a string representing the script. This string is very sensitive to errors because of its size variety and because it has to conform to the JSON string definition. And this is why prior to adding the job script to the local buffer, it is processed through the _json_escape non-API function.

As mentioned before, not all the desired fields are directly accessed through the job_record structure. For instance, some of the fields are gathered from the job_details one. There is a pointer inside the job_record pointing to job_details. There are also fields computed from the information of another fields, for example cpu_hours is calculated from elapsed_time and total_cpus. Summarizing, here's a list with all the 37 fields gathered for every job. Depending on the job, some of them may be empty.

| Field | Short description |
|---|---|
| jobid | job ID |
| user_id | user the job runs as |
| username | username associated to user_id |
| group_id | group submitted under |
| group_name | group associated to group_id |
| @start | time execution begins |
| @end | time execution ended |
| @submit | time of submission |
| elapsed | elapsed execution time |
| partition | name of the partition(s) |
| alloc_node | local node making resource alloc |
| nodes | list of nodes allocated to job |
| total_cpus | number of allocated cpus for accounting |
| total_nodes | number of allocated nodes for accounting |
| derived_exitcode | highest exit code of all job steps |
| exitcode | exit code for job (status from wait call) |
| state | state[2] of the job |
| cpu_hours | execution hours consumed by the job |
| eligible_time | time waited for resource allocation |
| work_dir | pathname of working directory |
| std_err | pathname of job's stderr file |
| std_in | pathname of job's stdin file |
| std_out | pathname of job's stdout file |
| cluster | name of cluster |
| qos | quality of service used for this job |
| ntasks | number of tasks to start |
| ntasks_per_node | number of tasks on each node |
| cpus_per_task | number of processors required for each task |
| orig_dependency | original value (for archiving) |
| excluded_nodes | excluded nodes |
| time_limit | time limit seconds[3] |
| reservation_name | reservation name |
| gres_req | Requested GRES[12] added over all nodes |
| gres_alloc | Actual GRES use added over all nodes |
| account | account number to charge |
| script | executed batch script |
| parent_accounts | accounts hierarchy up to root |

Table 4.2: Buffered fields to be indexed and their description.

Furthermore, another desired information is a slash (/) separated string of accounts representing the account bound to the job and its hierarchy of parent accounts up to the root. This field is the `parent_accounts` and in order to gather this information an iterative algorithm navigating the `slurmdb_assoc_rec_t` structure and the use of `assoc_mgr_fill_in_assoc` function is needed.

Once the job information has been collected and pushed to the buffer, the `slurm_jobcomp_log_record` continues with its execution. If the fields have been properly buffered, the function attempts to index the job completion information into Elasticsearch. This is done through the non-API function:

```
static int _index_job(const char *jobcomp)
```

As explained in the section 2.2, the index operations to Elasticsearch are performed through its HTTP/REST API. Instead of implementing the protocol operations, the plugin uses the `libcurl`[13] library, which is a free client-side URL transfer library. Specifically, the function invokes the following methods:

```
curl_global_init(CURL_GLOBAL_ALL)
```

This function is used for global initialisation and sets up the program environment that `libcurl` needs. Then the function calls to:

```
curl_easy_init()
```

which starts a `libcurl` easy session. This function must be the first function to call, and it returns a CURL easy handle that must be used as input to other functions in the easy interface.

This call must have a corresponding call to `curl_easy_cleanup` when the operation is complete. Then, the next step is to use the CURL handle to set some options:

[2] At this stage of the job execution, job state can be one of `COMPLETED`, `CANCELLED`, `FAILED`, `TIMEOUT` or `NODE_FAIL`

[3] `time_limit` can also be `INFINITE`. `NO_VAL` implies partition `max_time`.

```
curl_easy_setopt(handle, CURLOPT_URL, url);
curl_easy_setopt(handle, CURLOPT_POST, 1);
curl_easy_setopt(handle, CURLOPT_POSTFIELDS, jobcomp);
curl_easy_setopt(handle, CURLOPT_POSTFIELDSIZE, strlen(jobcomp));
curl_easy_setopt(handle, CURLOPT_HEADER, 1);
curl_easy_setopt(handle, CURLOPT_WRITEFUNCTION, _write_callback);
curl_easy_setopt(handle, CURLOPT_WRITEDATA, (void *) &chunk);
```

Note that the `_write_callback` is set up as a callback function. It is used to handle the HTTP responses from the Elasticsearch server once the HTTP request is done through:

```
curl_easy_perform(handle)
```

Once the plugin has the access to the response, it's parsed. The purpose of parsing is to search for HTTP status codes indicating whether the job information could be indexed or not. If the 200 or 201 status codes[14] are found, then the request has succeed.

At first sight, the status codes different to 200 or 201 were interpreted as a request failure by the plugin. However, on testing time, one of the bugs found and properly patched was a corner case scenario where the plugin informed that the job information could not be indexed, but directly querying to the Elasticsearch server the information was actually properly indexed. The problem was due to a status code 100 received. To cover this case, the plugin was patched to continue seeking for the next status code in the response data. See below two examples of HTTP responses, one with a success request and another with a failure one respectively.

```
HTTP/1.1 201 Created
Content-Type: application/json; charset=UTF-8
Content-Length: 92

{
  "_index" : "slurm",
  "_type" : "jobcomp",
  "_id" : "okHxrcz3Qx-Owg1WeaCeLw",
  "_version" : 1,
  "created":true
}
```

```
HTTP/1.1 403 Forbidden
Content-Type: application/json; charset=UTF-8
Content-Length: 96

{
  "error" : "ClusterBlockException[
    blocked by: [FORBIDDEN/5/index read-only (api)];]",
  "status":403
}
```

The `_index_job` function also uses a variable as an error counter so that every hundred failed requests, an error message is logged. This way, administrators with less verbosed debug modes[15] can be aware of the problem.

Getting back to the `slurm_jobcomp_log_record`, once the attempt to index the job has been performed and depending on the result, one of two paths of execution is followed. If the job completion information couldn't be properly indexed, this information has to be saved somewhere so that it can be indexed in future retries. So the `_push_pending_job` function is called and wrapped by mutex locks:

```
if (_index_job(buffer) == SLURM_ERROR) {
  slurm_mutex_lock(&pend_jobs_lock);
  _push_pending_job(buffer);
  slurm_mutex_unlock(&pend_jobs_lock);
  rc = _save_state();
} else {
  rc = _index_retry();
}
```

The `_push_pending_job` function adds the information related to the current job that could not be indexed to the global variable `pending_jobs_t pend_jobs`. This global variable is defined as follows:

```
/* Type for jobcomp data pending to be indexed */
typedef struct {
  uint32_t nelems;
  char **jobs;
} pending_jobs_t;
```

This structure has two fields, an unsigned integer used to count the number of jobs pending to be indexed and an array of pointers with the memory address for each of the pending jobs JSON information. So the `_push_pending_job` tries to reallocate memory for the new pending job and if success, adds its information and increases nelems by one.

Moreover, there is another place where the information related to all pending jobs to be indexed is saved. This place is a file that ensures information persistence as opposed to the `pend_jobs` structure which is a memory structure and therefore volatile. The function that deals with the file is `_save_state`. This function reads the information in the `pend_jobs` and saves it to a state file. This access is also done wrapped by mutex locks, but the mutex are inside the function itself. In fact, every time the `pend_jobs` structure is accessed and every time the state file is accessed, it's wrapped by mutex locks for the sake of thread safety.

So once the information is saved in a local buffer, `_save_state` function gets the value of the `slurm.conf` parameter `StateSaveLocation`[15]. This parameter specifies the fully qualified pathname of a directory into which the Slurm controller, `slurmctld`, saves its state (e.g. "/usr/local/slurm/checkpoint"). Slurm state is saved here to recover from system failures. The state file for this plugin is named `elasticsearch_state` following the convention used by other Slurm state files. In fact, the writing algorithm uses three files:

```
StateSaveLocation/elasticsearch_state.new
StateSaveLocation/elasticsearch_state
StateSaveLocation/elasticsearch_state.old
```

The function tries to write the information to the new file. If all goes smoothly, it rotates the final state file to the old file and the new to the final state through the `link` and `unlink` functions. This is a more secure algorithm to make the writing process than just writing to the final `elasticsearch_state`, because it prevents messing up the information in case of writing failure.

All in all, when there's a modification in the `pend_jobs` structure, a corresponding modification is done in the state file so that a coherence between memory and file state is preserved. In this case, it's somewhat similar to the *Write-through* cache-memory writing policy. In the `_index_retry` function it's a little bit different, and it's explained later.

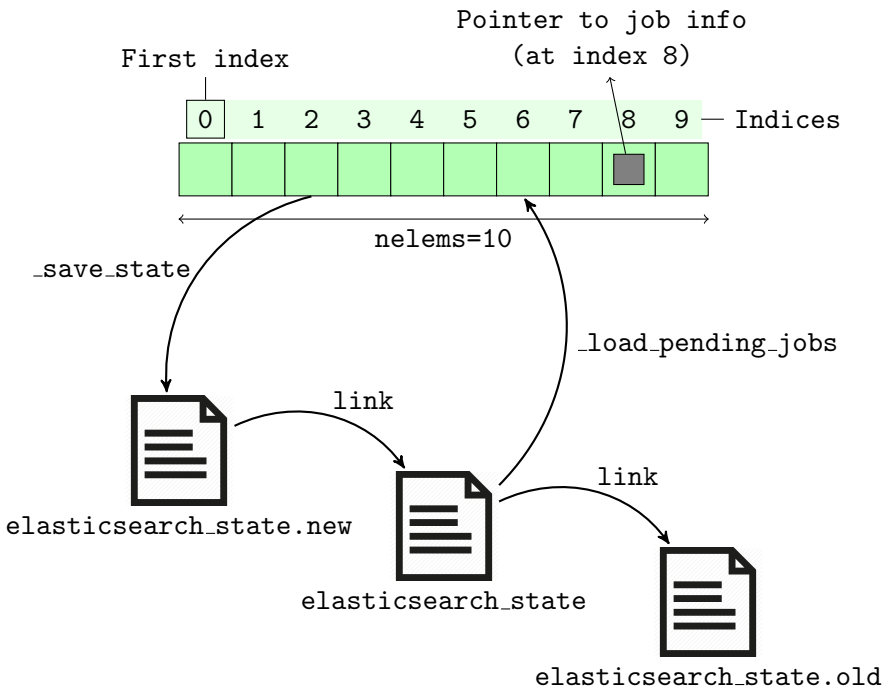



Figure 4.4: Management of `pend_jobs_t` coherence with the state file.

Another aspect which is worth mentioning is the way the buffer information is treated before written/read to/from the state file. The plugin, following again the style of other state files, uses these two routines[16]:

```
packstr_array
unpackstr_array
```

The `packstr_array` converts the job size information to network byte order. This is important because in some cases endiannes[17] could suppose a problem. Let's suppose int values are stored in a file, then the file is read from a machine with the opposite endianness. This situation would lead to problems. This is even worse over the network, because the other machine might not be able to determine the endianness of the machine that sent the data. The solution is to store the information in *network byte order* (big endian). Then the other machine just reads the file information with the `unpackstr_array` which knows that it was stored in network byte order.

## 4.5 Reindexing Pending Jobs

There are two points where the plugin attempts to reindex the pending jobs.

- The first one is each time the `slurm_jobcomp_log_record` is able to successfully index a job. If a job has just been able to be indexed, there is a highg probability that the Elasticsearch server is up, running and accepting requests. So it makes sense to try to reindex the rest of the pending jobs at this point.
- The other point where the plugin tries to reindex is when the plugin is loaded, specifically inside the `slurm_jobcomp_set_location`. Ideally, the SLURM administrator should start SLURM after verifying that the services which SLURM makes use of are up and running. So it's likely that Elasticsearch is accepting requests before `slurmctld` starts.

The reindexing process is done by calling to `_index_retry`. This function creates a local array of ints with the same number of elements as pending jobs to be reindexed.

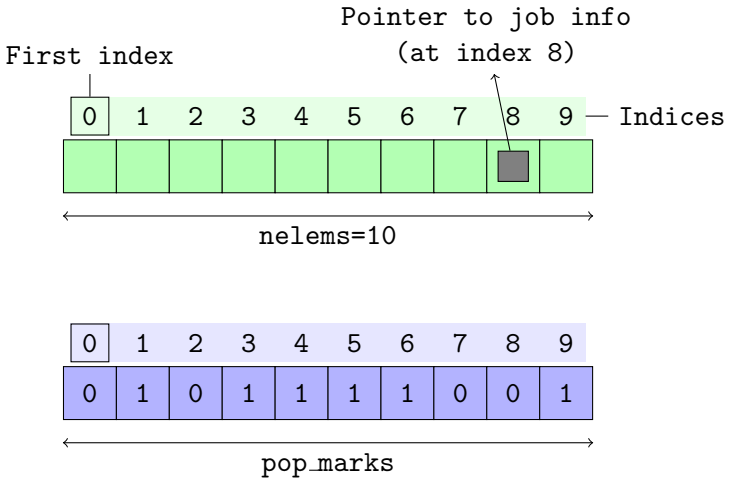



Figure 4.5: Pending jobs retry by using a `pop_marks` array.

See below the code snippet which allocates memory for this array.

```
...
int *pop_marks;

slurm_mutex_lock(&pend_jobs_lock);
pop_marks = xmalloc(sizeof(int) * pend_jobs.nelems);
...
```

The `pend_jobs_t` is sequentially traversed. If the job in the $i$-position is properly reindexed, `pop_marks[`$i$`]` is assigned with 1 (initially all `pop_marks` positions are assigned with 0) and the job information at `pend_jobs.jobs[`$i$`]` is freed. If there is an error while reindexing, nothing remarkable is done so that position remains with 0.

Once both arrays have been traversed, the `_index_retry` function calls to the `_update_pending_jobs` function passing the `pop_marks` array as an argument. This function restructures `pend_jobs_t` by reallocating the positions for the jobs that could not be indexed and freeing the ones that could be properly indexed as specified by the `pop_marks` vector. Then it returns the control back to `_index_retry`, which then calls to `_save_state` to maintain the coherence with the state file to reflect the changes.

Finally, the `fini` function is called when the plugin is removed. This function is just used to clear any allocated storage.

## 4.6 Development Environment

A development machine was used while implementing and testing the plugin, and it was needed that this machine was able to emulate a cluster with multiple nodes.

As explained in the Slurm Programmer's Guide[18], it is possible to make a single node appear to Slurm as a Linux cluster by running `configure` with the `--enable-front-end` option. This defines `HAVE_FRONT_END` with a nonzero value in the file `config.h`. All (fake) nodes should be defined in the `slurm.conf` file. These nodes should be configured with a single `NodeAddr` value indicating the node on which single `slurmd` daemon executes.

So Slurm was configured with this option and installed in the development machine, which was used for implementation and testing purposes.

## 4.7 Features Summary

The most relevant features related to the plugin are summarized in the following list:

- **Fault-tolerance**. The system is able to continue operating properly in the event of failure in the Elasticsearch server, saving the job information for future retries.
- **Thread safety**. The `slurmctld` is multi-threaded. The plugin implementation is guaranteed to be free of race conditions when accessed by multiple threads concurrently.
- **API specification.** The plugin has been implemented satisfying the Slurm Job Completion Logging Plugin API as specified in the official Slurm Developers guidelines.
- **Data pre-processing**. Data has been pre-processed and prepared to follow the JSON format prior to be indexed into the Elasticsearch server.
- **Curl integration**. The HTTP/REST requests and responses are executed using the `libcurl` library methods.

In the Chapter 5, the plugin integration and deployment singularities are thoroughly explained.

# Chapter 5

# Plugin Integration And Deployment

Once the plugin was developed, the next step is to understand which files are needed to compile and integrate the plugin with Slurm and to progressively deploy it in the rest of the BSC clusters. So this is what's this chapter about.

## 5.1 Plugin Integration Needed Files

The files needed for the plugin are divided in two categories: the ones that are newly created from scratch, and the ones that already existed in the Slurm build but were needed to be modified and adapted for the plugin to work. The following table illustrates the mapping between each of these files and their category.

| **c/m** | **Relative path to the Slurm build root directory** |
|---|---|
| c | src/plugins/jobcomp/elasticsearch/ |
| c | src/plugins/jobcomp/elasticsearch/jobcomp_elasticsearch.c |
| c | src/plugins/jobcomp/elasticsearch/Makefile.am |
| m | src/plugins/jobcomp/Makefile.am |
| c | auxdir/x_ac_curl.m4 |
| m | auxdir/Makefile.am |
| m | configure.ac |
| m | slurm.spec |

Table 5.1: Created and modified files/dirs for the plugin.

A small explanation of each file purpose is shown below:

- The newly created directory **src/plugins/jobcomp/elasticsearch/** contains the plugin files. Every jobcomp plugin has its own subdirectory.
- **src/plugins/jobcomp/elasticsearch/jobcomp_elasticsearch.c** is the plugin source code written in C which implements the the Slurm job completion logging plugin API (explained in the previous chapter).
- The new **src/plugins/jobcomp/elasticsearch/Makefile.am** file is used by `automake`[19] to generate the `Makefile.in` file. It's part of the GNU build system or Autotools. Among other things, the `Makefile.am` instructs `automake` so that it recognizes which are the plugin source files, libraries and dependencies. For instance, the library:

  ```
  jobcomp_elasticsearch.la
  ```

  will only be installed in the libdir directory if the `configure` script detects that the `libcurl` library is installed and usable at build time.

  ```
  ...
  if WITH_CURL
  ELASTICSEARCH = jobcomp_elasticsearch.la
  endif
  pkglib_LTLIBRARIES = $(ELASTICSEARCH)
  jobcomp_elasticsearch_la_SOURCES = jobcomp_elasticsearch.c
  ...
  ```

- The file **src/plugins/jobcomp/Makefile.am** was also modified, and the modification was as simple as adding elasticsearch to the `SUBDIRS`[20] variable.

  ```
  SUBDIRS = elasticsearch filetxt none script mysql
  ```

In packages using make recursion, the top level `Makefile.am` must tell Automake which subdirectories are to be built. This is done via the `SUBDIRS` variable.

- There was another newly **auxdir/x_ac_curl.m4** file. This file was just copied from the `libcurl` source repository[21] with a small modification. This is the macro file used by `autoconf` and specifies how to check that `libcurl` is installed and usable in the system. The `DEFAULT-ACTION` string can be `yes` or `no` to specify whether to default to `--with-libcurl` or `--without-libcurl`. If not supplied, `DEFAULT-ACTION` is `yes`.

- **auxdir/Makefile.am** was modified to include the previous macro in the list of distributed macros not covered in the automatic rules.

```
EXTRA_DIST = x_apthread.m4 slurm.m4 ...  x_ac_curl.m4 ...
```

- The **configure.ac** file was also modified. This is the input for the Autoconf tool used to generated the `configure` script.

```
dnl
dnl Check for compilation of SLURM with CURL support:
dnl
LIBCURL_CHECK_CONFIG
...
AC_CONFIG_FILES([Makefile
 src/plugins/jobcomp/filetxt/Makefile
 src/plugins/jobcomp/elasticsearch/Makefile
 src/plugins/jobcomp/filetxt/Makefile
 src/plugins/jobcomp/none/Makefile
 src/plugins/jobcomp/script/Makefile
 src/plugins/jobcomp/mysql/Makefile
...
 ]
)
```

- Finally, some modifications to the **slurm.spec** were made too. The `spec`[22] is at the heart of RPM's packaging building process. Similar in concept to a `makefile`, it contains information required by RPM to build the package, as well as instructions telling RPM how to build it. The `spec` file also dictates exactly what files are a part of the package, and where they should be installed. In this case, the plugin library is added if it's found under the libdir directory.

Some messages will indicate at configure time that libcurl is present and usable on the system:

```
...
checking for curl-config...  /usr/bin/curl-config
checking for the version of libcurl...  7.19.7
checking whether libcurl is usable...  yes
checking for curl_free...  yes
...
```

The rest of the Slurm compilation and installation process remains the same as illustrated in the official documentation.

## 5.2 Plugin Deployment

After submitting different job test cases in the development machine, patching a few errors and once the plugin seemed to be stable, the decision was to deploy the solution.

The Elasticsearch, Kibana, `libcurl` library and the web server are dependencies for the plugin to work. However, the installation process for each of these technologies is out of scope of this work as it is already documented in their respective official repository. So each of these technologies were installed in the target deployment clusters prior to the plugin integration.

Once they were installed, the plugin was integrated into Slurm (and enabled in `slurm.conf`) on two of the main clusters at BSC: MinoTauro and CNAG.

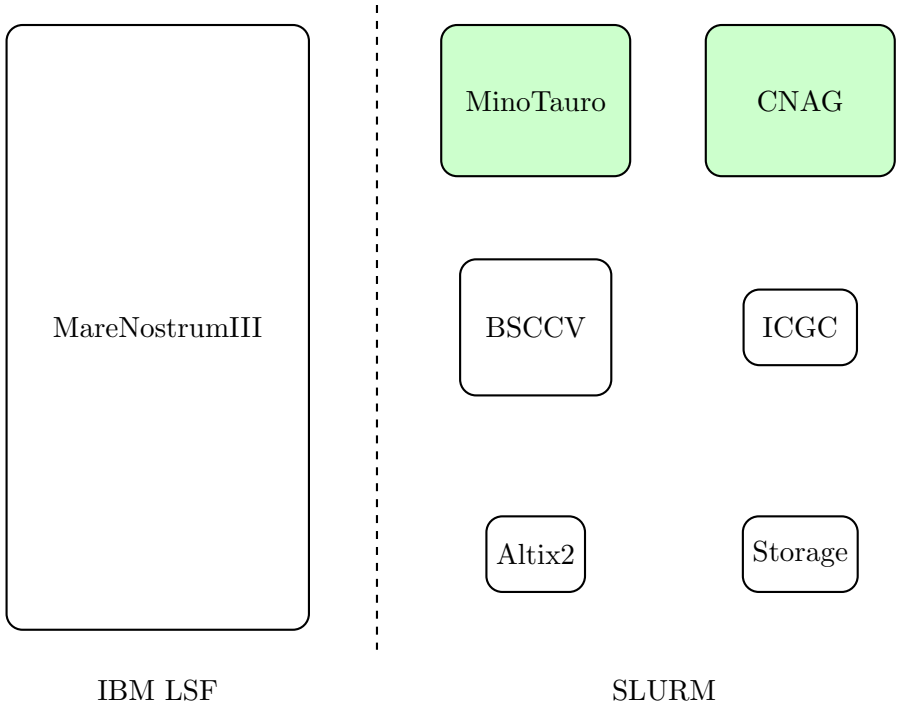



Figure 5.1: Solution integrated in two clusters.

Two tables with the Elasticsearch configuration for each of the target deployment clusters are showed below:

| **Parameter** | **Quantity** | **Additional information** |
|---|---|---|
| cluster | 1 | `minotauro` |
| nodes | 1 | `mino1` |
| indices | 2 | `kibana-int`, `slurm` |
| shards/index[1] | 5 | |
| shards `kibana-int` | 5 | |
| shards `slurm` | 5 | |
| replicas/shard[2] | 1 | |
| documents `kibana-int` | 1 | |
| documents `slurm` | 1417863 | 1/job. Across the 5 shards. |

Table 5.2: Elasticsearch configuration in MinoTauro cluster.

| Parameter | Quantity | Additional information |
|---|---|---|
| cluster | 1 | `cnag` |
| nodes | 2 | `cnag1`, `cnag2` |
| indices | 2 | `kibana-int`, `slurm` |
| shards/index | 2 | |
| shards `kibana-int` | 2 | |
| shards `slurm` | 2 | |
| replicas/shard[3] | 1 | |
| documents `kibana-int` | 1 | |
| documents `slurm` | 10529119 | 1/job. Across the 2 shards. |

Table 5.3: Elasticsearch configuration in CNAG cluster.

Once Kibana is installed, no more coding is needed and the administrator just has to configure[23] a dashboard to make it point to the Elasticsearch `slurm` index. Please, refer to Appendix C.1 to check a bit of the analyzing potential offered by the Kibana features on the plugin indexed data.

During the first days in production, the plugin was just enabled during office hours so it could remain under supervision to possible unexpected corner cases. Some of them were not covered on testing time and they appeared, sometimes even producing segmentation faults affecting `slurmctld` that needed immediate disabling of the plugin, `slurmctld` restarting and error debugging and patching.

One of the places used as a plugin repository contains a `NEWS`[24] file with the changes made during the initial beta versions. Some bugs had to do with forgotten releases of allocated memory, strings not properly resized or fields not properly formatted. Some tools like `gdb`[25] or `valgrind`[26] have been very useful during implementation time as well as during testing time.

[1]The plugin just requires being able to index to the `slurm` index an so Kibana with `kibana-int`. The default configuration is 5 shards per index and 1 replica per shard. However, adjusting the number of nodes, shards, replicas and so on for a better performance has more to do with sysadmin tasks and it's independent for the plugin to properly work.

[2]In this case, as there's just one node, the replicas are marked as UNASSIGNED by Elasticsearch because they need to exist in different nodes. So they are not useful for this cluster.

[3]In this case, there are 2 nodes available, so replicas are marked as STARTED by Elasticsearch and thus they are useful.

The work has not been done following a classical waterfall model[27] methodology, with rigid differentiated stages. Instead, the followed methodology can be said to be closer to a more flexible iterative[28] model with Agile[29] practices. This means that, for instance, if more fields were needed after production, they have been added, so another iteration through coding, testing and integration has been done. Dynamic changes to requirements are welcomed and can be adapted to the code if needed.

# Chapter 6

# Future Work

It would be interesting to initiate some tasks using the work done in this project. A good approach would be applying the output of existing machine learning[1] techniques to predict the jobs' elapsed time to feed back Slurm at job submission time. Before explaining in detail this schema, let's detail what's the elapsed time and why its value is important to be predicted.

The elapsed time is defined as the difference between the end time and the start time, measured in seconds. In order to understand why it is important to have a good estimation of the jobs elapsed time, one first needs to understand the *Backfill Scheduling*[30].

In the Slurm context, without backfill scheduling each job is scheduled strictly in priority order, which typically results in significantly lower system utilization and responsiveness than otherwise possible. Backfill scheduling will start lower priority jobs if doing so does not delay the expected start time of any higher priority jobs. Since the expected start time of pending jobs depends upon the expected completion time of running jobs, reasonably accurate time limits are important for backfill scheduling to work well. And here's where a machine learning component could help users by suggesting time litmis for their jobs.

[1]Machine learning is a subfield of computer science that evolved from the study of pattern recognition and computational learning theory in artificial intelligence. It explores the construction and study of algorithms that can learn from and make predictions on data. Such algorithms operate by building a model from example inputs in order to make data-driven predictions or decisions, rather than following strictly static program instructions.

The schema would consist in adding this new component that would make use of the gathered data from historical jobs and applies machine learning techniques to generate a formula. This formula could be a linear regression one but it would depend on the applied technique. Finally, the future work would be creating a submit plugin[31] that would make use of this formula to generate the predictions for the users.

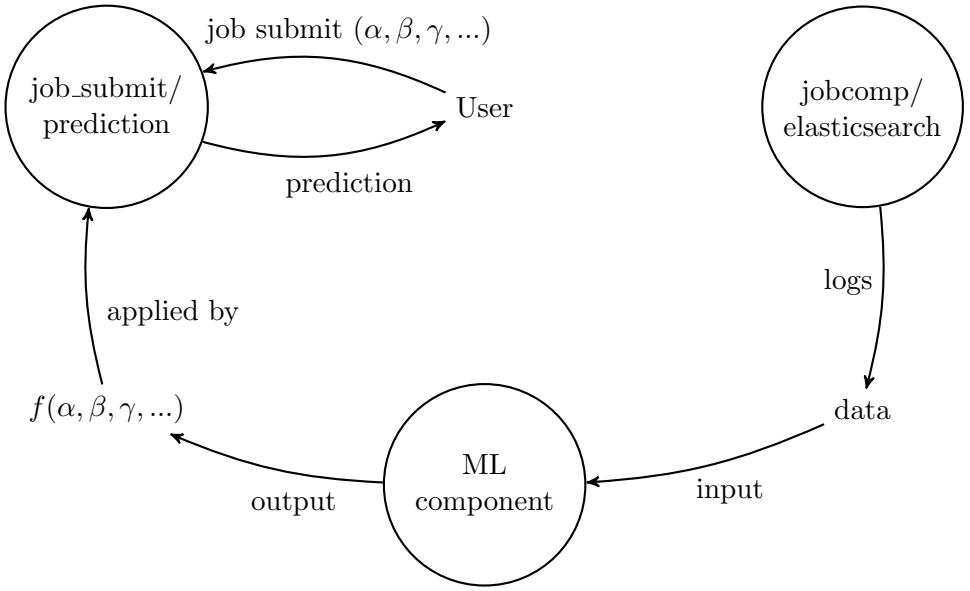



Figure 6.1: Possible future work prediction schema.

This schema somewhat makes up a cycle that feeds back their components.

The idea of predicting the elapsed time isn't something new and there's already much research literature[2] discussing predictions and data mining techniques in this context. But the future work would consist in applying one of these techniques to Slurm as shown in the previous schema.

Anyhow, despite it's not the purpose of this work, some basic exploration has been done towards this aim and it's explained in the Chapter 7.

[2]Please, refer to the bibliography[42, 43, 44, 45, 46, 47] to find some articles and books discussing machine learning techniques in this context.

# Chapter 7

# Machine Learning Exploration

Some basic machine learning exploration has been done, and to do so, KNIME[1] tool was used (`knime-2.11.1`).

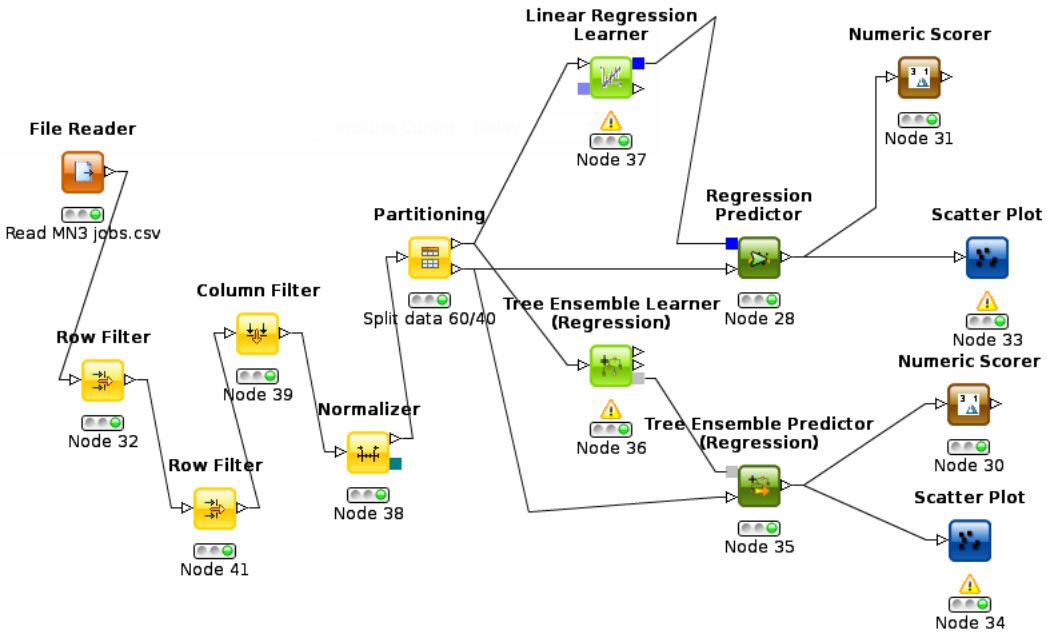



Figure 7.1: Basic Machine Learning exploration using KNIME software.

[1]KNIME is a modern data analytics platform that allows you to perform sophisticated statistics and data mining on your data to analyze trends and predict potential results

The tool provides a visual workbench where hundreds of processing nodes for data I/O, preprocessing and cleansing, modeling, analysis and data mining can be added and combined to obtain different results.

The first node reads a `.csv` file with information related to all the finished jobs from MareNostrumIII during the years 2013 and 2014. Specifically, there are 13149000 rows each one representing a subset of each job fields. Then some filters are applied, including the removal of missing values or jobs whose status was `exited` from the analysis. The column filter excludes some columns, like `jobid, waiting_time` or `status`, which shouldn't help for prediction purposes. After that, some numeric values like `cpus` are normalized. Then, the input table is split into two partitions (train and test data) using the stratified sampling option. The two partitions are available in the two output ports.

At this point, data is ready to be analyzed and two methods were experimented. The first one (top way seen in the workbench) was done through the regression model. The train partition was connected to the Linear Regression Learner. This node performs a linear regression selecting the `elapsed` column as the target to be predicted.

Given a data set $\{y_i, x_{i1}, ..., x_{ip}\}_{i=1}^{n}$ of $n$ statistical units, a linear regression model assumes that the relationship between the dependent variable $y_i$ and the $p$ vector of regressors (independent variables) $x_i$ is linear. This relationship is modeled through a *disturbance term* or *error variable* $\epsilon_i$. Thus the model takes the form

$$y_i = \beta_1 x_{i1} + ... + \beta_p x_{ip}$$

The $\beta_i$ elements are the regression coefficients. The higher the absolute value of the coefficient, the higher the weight of its parameter will be in the prediction.

Getting back to the tool, the test port from the partition is connected to the Regression Predictor, together with the output from the Learner. This node predicts the response using the regression model. Finally, the Numeric Scorer node computes certain statistics between the numeric columns's values and the predicted values.

| | |
|---|---|
| $R^2$ | 0.275 |
| mean absolute error | 0.003 |
| mean squared error | 0 |
| root mean squared error | 0.006 |
| mean signed difference | 0 |

Table 7.1: Numeric Scorer statistics from the linear regression.

The $R^2$ statistic is the coefficient of determination and indicates how well data fit a statistical model. An $R^2$ of 1 indicates that the regression line perfectly fits the data. So in this case, the prediction was not really good.

The other approach was to try with a different model and see whether the results got better. The Tree Ensemble Learner was used in this case (bottom way in the workbench figure). This model is way more sophisticated than the linear regression one. Each of the regression tree nodes is learned on a different set of rows. See below the results with this approach:

| | |
|---|---|
| $R^2$ | 0.531 |
| mean absolute error | 0.002 |
| mean squared error | 0 |
| root mean squared error | 0.005 |
| mean signed difference | 0 |

Table 7.2: Numeric Scorer statistics from the Tree Ensemble Predictor.

Despite the prediction is not significantly good in this case either, it's closer to 1 than the previous one.

The problem with this model is that it does not generate a low computationally expensive formula as in the case of the linear regression. It's important to take into account that the prediction formula should be used by the submit plugin, thus applied every time a job is submitted and generating some overhead as a consequence.

Anyhow, despite some basic exploration has been done, as it's been mentioned in the previous chapter there's already literature discussing predictions on the elapsed time and it would be nice if any of the technies were applied to the proposed schema.

# Chapter 8

# Conclusions

At the moment of writing this work, the plugin has been working (and still does) in both MinoTauro and CNAG clusters for more than two months 24x7, withstanding peaks of more than **15K finished jobs per day** with a big variety of field values. The solution in MinoTauro already contains 1417863 finished jobs and CNAG 10529119. The intention is to progressively deploy the plugin in more Slurm clusters.

Some important **decisions** are being taken related to clusters reprovision by analyzing the information generated by the solution, even having a positive impact in terms of costs saving.

From the point of view of BSC, the system is being useful and **actively used** by the supporters to search and filter job historical information which helps troubleshooting issues and better train the users on how to use the facilities. The systems team also finds it useful to analyze how the clusters are being used and try to improve the performance.

Moreover, the solution has also been presented at the **HPCKP Slurm Training'15**[32] with a very positive acceptance and feedback from other Slurm users in the audience.

Finally, apart from the plugin files being published in an own public repository[33], its existance was raised to the `slurm-dev`[36] mailing list. As a result, the plugin and its needed files are already merged[35] in the official Slurm master branch and **it has been announced[34] to be adopted in the next stable release** (`slurm-15.08`), planned in August 2015.

Taking into account that Slurm is the workload manager on about 60% of the TOP500[37] supercomputers, it would be a big satisfaction if any of the sites holding these supercomputers make use of this system. A solution that was initially required by BSC is now available for the rest of the community using HPC environments with a Slurm workload manager.

# Appendix A

# Clusters detailed overview

## A.1 MareNostrumIII

MareNostrum III[38] is a supercomputer based on Intel SandyBridge processors, iDataPlex Compute Racks, a Linux operating system and an Infiniband interconnection. The current Peak Performance is 1.1 Petaflops. The total number of processors is 48,896 Intel SandyBridge-EP E5–2670 cores at 2.6 GHz (3,056 compute nodes) with 103.5 TB of main memory. Listed below a summary of the system:

- 37 iDataPlex compute racks. Each one composed of:
  - 84 IBM dx360 M4 compute nodes
  - 4 Mellanox 36-port Managed FDR10 IB Switches
  - 2 BNT RackSwitch G8052F (Management Network)
  - 2 BNT RackSwitch G8052F (GPFS Network)
  - 4 Power Distribution Units
- All IBM dx360 M4 node contain:
  - 2x E5–2670 SandyBridge-EP 2.6GHz cache 20MB 8-core
  - 500GB 7200 rpm SATA II local HDD
- Nodes are differentiated as follows:
  - RAM:
    - 64 nodes contain 8x 16G DDR3–1600 DIMMS (8GB/core) Total: 128GB/node

        * 64 nodes contain 16x 4G DDR3–1600 DIMMs (4GB/core) Total: 64GB/node
        * 2880 nodes contain 8x 4G DDR3–1600 DIMMs (2GB/core) Total: 32GB/node
    - 42 heterogenenous nodes contain:
        * 8x 8G DDR3–1600 DIMMs (4GB/core) Total: 64GB/node
        * 2x Xeon Phi 5110P accelerators
- 1.9 PB of GPFS disk storage
- Interconnection Networks
    - Infiniband Mellanox FDR10: High bandwith network used by parallel applications communications (MPI)
    - Gigabit Ethernet: 10GbitEthernet network used by the GPFS filesystem
- Operating System: SLES 11 SP3
- 3 Login nodes and 2 nodes for interactive jobs

## A.2 MinoTauro

MinoTauro[39] is a NVIDIA GPU based cluster. A list of the main technical features is shown below:

- 126 compute nodes and 2 login nodes.
- Every node has:
    - 2x Intel Xeon E5649 6-Core at 2,53 GHz
    - 24 GB of RAM memory, 12MB of cache memory
    - 250 GB local disk storage (SSD)
    - 2x NVIDIA M2090, each one:
        * 512 CUDA Cores
        * 6GB of GDDR5 Memory
    - Access to BSC GPFS parallel filesystem disk storage (2 PB)
- The networks that interconnect the cluster using 14 switches are:

  - Infiniband Network: High bandwidth network used by parallel applications communications (MPI)
  - Gigabit Network: 10GbitEthernet network used by the GPFS Filesystem

- Red Hat Enterprise Linux operating system
- Some software available:
  - BullX Cluster Suite
  - Intel Cluster Studio
    * C/C++/Fortran Compilers
    * MKL
    * Intel MPI
    * Intel Trace Analyzer and Collector
  - PGI Accelerator Fortran Server
  - NVIDIA CUDA Toolkit
  - OpenMPI

## A.3 CNAG

The CNAG[40] cluster comprises 100 compute nodes, 2 login nodes and 20 HiMem nodes. The operative system is a Linux RedHat release 6.2.

Every compute node (and both logins) has two processors Xeon Quad Core at 2.93 GHz with 48 GB of RAM, that is 8 cores with 5,8GB per core, and 166 GB of local disk storage. Each HiMem node has two processors Intel Xeon E5–2670 at 2.60 GHz with 128 GB of RAM, 16 cores with 7,8 GB/core, and 394 GB of local disk storage.

Every node has access to a couple of Lustre parallel filesystems, */project* (1130T) and */scratch* (873T), providing a total of 2 PB of disk storage.

The networks that interconnect the CNAG cluster are:

- Infiniband Network: High bandwidth network used by parallel applications communications (MPI) available in all compute nodes except in the HiMem ones.
- Gigabit Network: Ethernet network used by the Lustre Filesystem and the communications between the HiMem nodes.

## A.4 Altix 2 UV100

Altix 2 UV100[41] is a shared memory machine, with a cc-NUMA architecture (Cache Coherent Non-Uniform Memory Access). The total amount is 96 CPUs with 1.5TB of RAM memory composed by:

- 3 IRUs, which are the basic SGI unit interconnected with Numalink 5 at 15GB/s
- Each IRU has 2 blades IP93
- Each IP93 blade has:
  - 2 Nehalem-EX Intel(R) Xeon(R) CPU E7-8837 8-core @ 2.67GHz
  - 16 DIMM of 16GB RAM DDR3 @ 1066MHz, with 256GB of total memery per blade

# Appendix B

# IBM LSF Build Structure

## B.1 UNIX/Linux LSF Directory Structure

The following figure shows typical directory structures for a new UNIX or Linux installation. Depending on which products are installed and platforms selected, the directory structure may vary. Note that the lsb.* files (like `lsb.acct`) are located under:

`LSF_TOP/conf/lsbatch/`*`cluster_name`*`/configdir/`

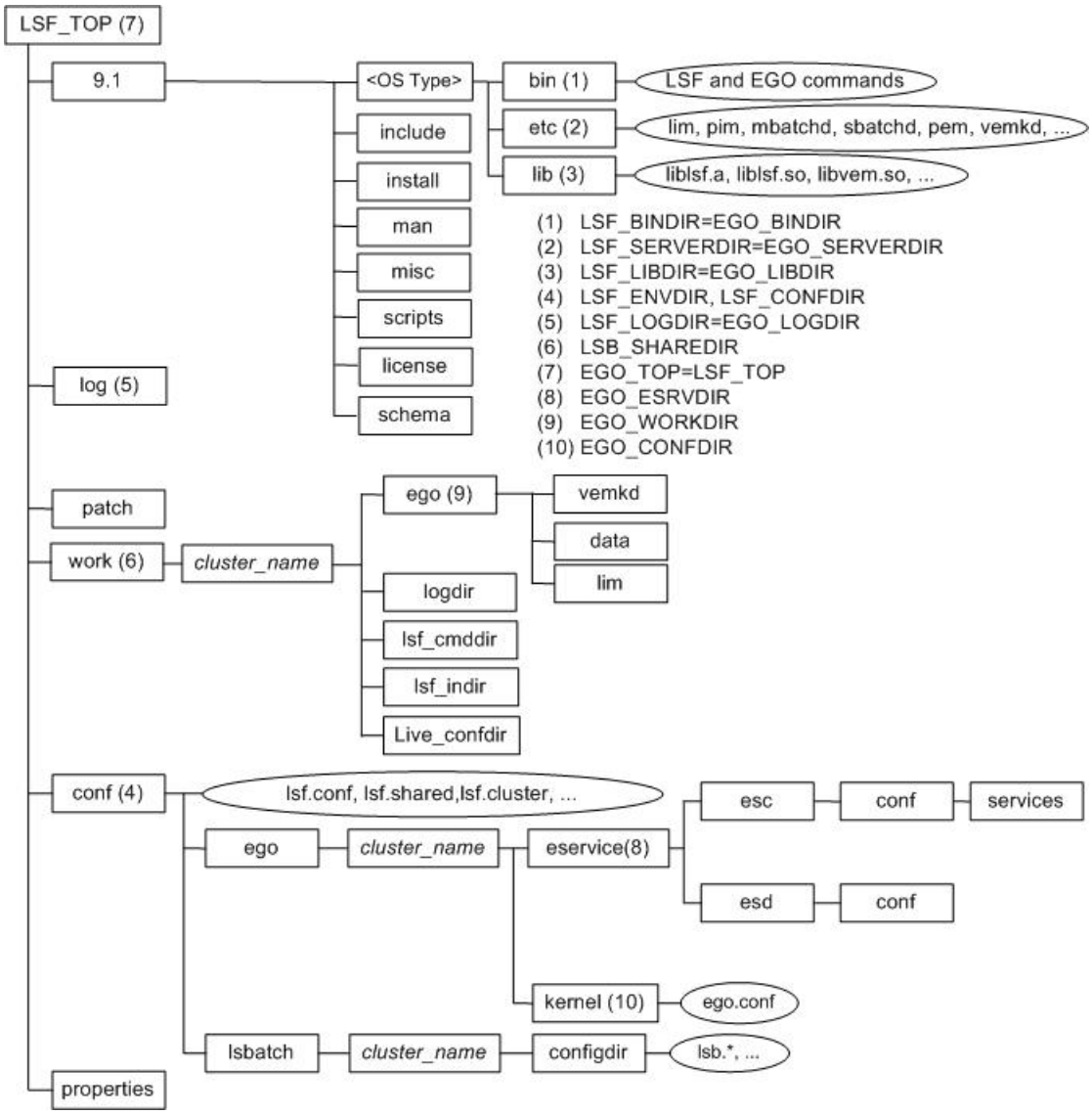



Figure B.1: LSF UNIX/Linux typical directory structure.

# Appendix C

# Web layer

## C.1 Kibana details

Some Kibana features, panels, stats and details[1] are showed in this section. The next picture shows how to indicate Kibana to gather the data[2] from the generated Elasticsearch `slurm` index. This is done under the dashboard general settings, which can be easily reached at the top of the webpage.

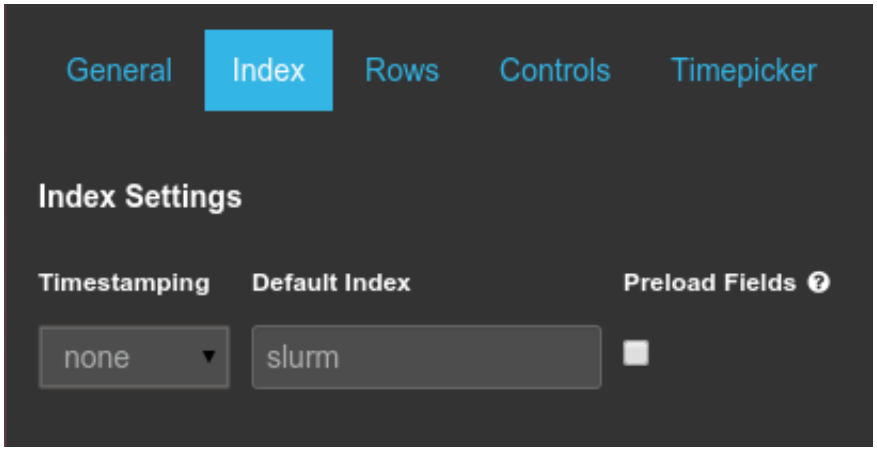



Figure C.1: Point Kibana to collect data from `slurm` index.

[1] A subset of the analysis potential is shown, but not the total.

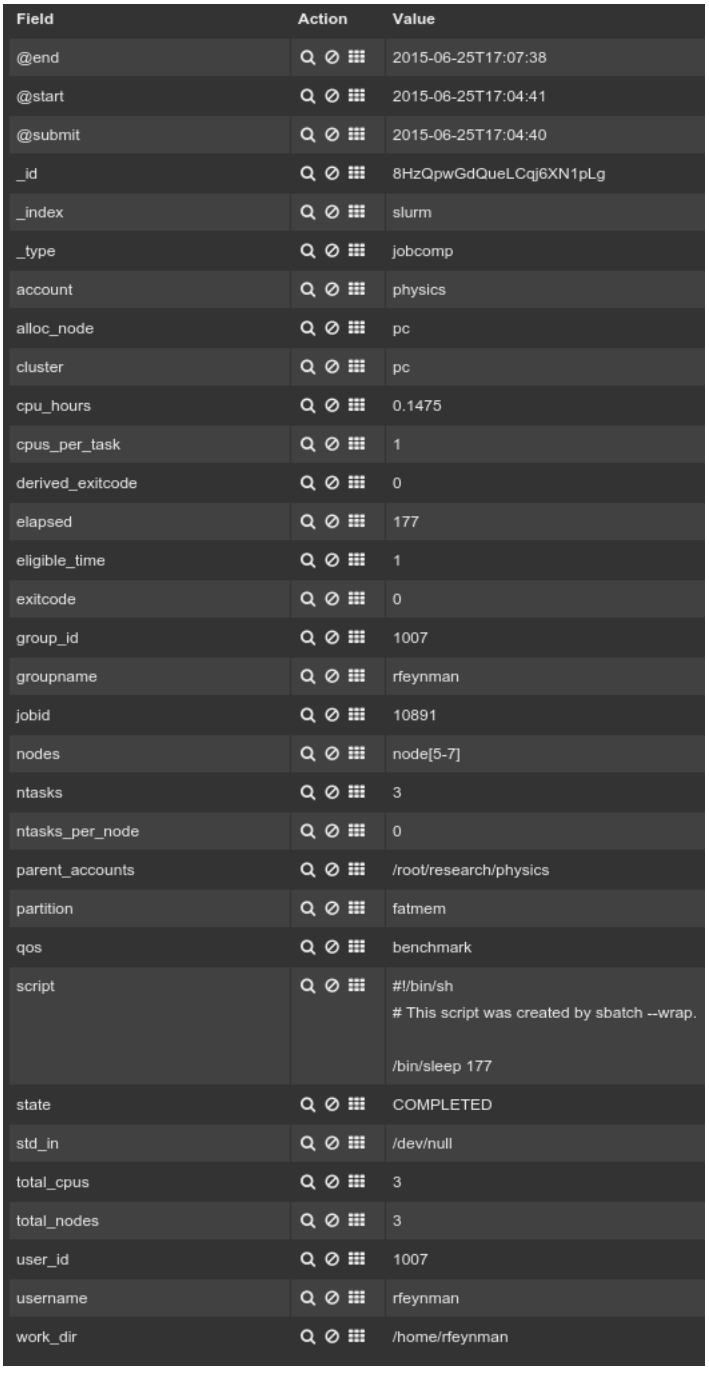
[2] Note that data is fake but most of the panel configurations showed here are applied in production with real data.

FINISHED JOBS

0 to 14 of 14 available for paging

| jobid | username | ntasks | total_nodes | partition | @submit | @start | @end | state | elapsed | parent_accounts |
|---|---|---|---|---|---|---|---|---|---|---|
| 10886 | tmckenna | 4 | 4 | townmem | 2015-06-25T16:39:09 | 2015-06-25T16:39:10 | 2015-06-25T16:39:10 | COMPLETED | 0 | /root/research/bio |
| 10885 | tmckenna | 2 | 2 | sharedmem | 2015-06-25T16:38:15 | 2015-06-25T16:38:16 | 2015-06-25T16:41:36 | COMPLETED | 200 | /root/research/bio |
| 10889 | tleary | 1 | 1 | debug | 2015-06-25T17:03:15 | 2015-06-25T17:03:16 | 2015-06-25T17:08:40 | COMPLETED | 324 | /root/research/geo |
| 10890 | tleary | 3 | 3 | debug | 2015-06-25T17:03:18 | 2015-06-25T17:03:19 | 2015-06-25T17:08:43 | COMPLETED | 324 | /root/research/geo |
| 10891 | rfeynman | 3 | 3 | fatmem | 2015-06-25T17:04:40 | 2015-06-25T17:04:41 | 2015-06-25T17:07:38 | COMPLETED | 177 | /root/research/physics |
| 10893 | mfaraday | 1 | 1 | townmem | 2015-06-25T17:07:49 | 2015-06-25T17:07:50 | 2015-06-25T17:12:17 | COMPLETED | 267 | /root/research/chemistry |
| 10894 | mfaraday | 1 | 1 | medmem | 2015-06-25T17:15:46 | 2015-06-25T17:15:46 | 2015-06-25T17:17:26 | COMPLETED | 100 | /root/research/chemistry |
| 10892 | leuler | 4 | 4 | sharedmem | 2015-06-25T17:07:09 | 2015-06-25T17:07:09 | 2015-06-25T17:14:45 | COMPLETED | 456 | /root/research/math |
| 10883 | jcampbell | 2 | 2 | medmem | 2015-06-25T16:35:43 | 2015-06-25T16:35:43 | 2015-06-25T16:35:58 | COMPLETED | 15 | /root/research/geo |
| 10884 | jcampbell | 3 | 3 | medmem | 2015-06-25T16:35:53 | 2015-06-25T16:35:53 | 2015-06-25T16:38:29 | COMPLETED | 156 | /root/research/geo |
| 10887 | ccastaneda | 4 | 4 | townmem | 2015-06-25T17:01:06 | 2015-06-25T17:01:07 | 2015-06-25T17:03:07 | COMPLETED | 120 | /root/research/bio |
| 10882 | ajodorowsky | 1 | 1 | fatmem | 2015-06-25T16:30:55 | 2015-06-25T16:30:56 | 2015-06-25T16:31:06 | COMPLETED | 10 | /root/research/geo |
| 10881 | ajodorowsky | 1 | 1 | debug | 2015-06-25T16:24:47 | 2015-06-25T16:24:48 | 2015-06-25T16:24:48 | COMPLETED | 0 | /root/research/geo |
| 10888 | ahuxley | 1 | 1 | fatmem | 2015-06-25T17:01:52 | 2015-06-25T17:01:52 | 2015-06-25T17:02:35 | COMPLETED | 43 | /root/research/physics |

Figure C.2: Table panel with one row per finished job.



| Field | Action | Value |
|---|---|---|
| @end | | 2015-06-25T17:07:38 |
| @start | | 2015-06-25T17:04:41 |
| @submit | | 2015-06-25T17:04:40 |
| _id | | 8HzQpwGdQueLCqj6XN1pLg |
| _index | | slurm |
| _type | | jobcomp |
| account | | physics |
| alloc_node | | pc |
| cluster | | pc |
| cpu_hours | | 0.1475 |
| cpus_per_task | | 1 |
| derived_exitcode | | 0 |
| elapsed | | 177 |
| eligible_time | | 1 |
| exitcode | | 0 |
| group_id | | 1007 |
| groupname | | rfeynman |
| jobid | | 10891 |
| nodes | | node[5-7] |
| ntasks | | 3 |
| ntasks_per_node | | 0 |
| parent_accounts | | /root/research/physics |
| partition | | fatmem |
| qos | | benchmark |
| script | | #!/bin/sh<br># This script was created by sbatch --wrap.<br><br>/bin/sleep 177 |
| state | | COMPLETED |
| std_in | | /dev/null |
| total_cpus | | 3 |
| total_nodes | | 3 |
| user_id | | 1007 |
| username | | rfeynman |
| work_dir | | /home/rfeynman |

Figure C.3: Table panel with finished job details expanded.

Figure C.2 shows a table panel with one row per finished job. A custom subset of the job fields is showed, despite the complete job information can be expanded as seen in detail in the Figure C.3. Moreover and as explained in prior sections, not all the jobs contain values for all the 37 fields and just the ones with non empty values are saved. In this panel, one of the features that can be configured is how many rows per page the user wants to be displayed.

Kibana also offers querying and filtering options so that jobs that match the criteria are displayed:

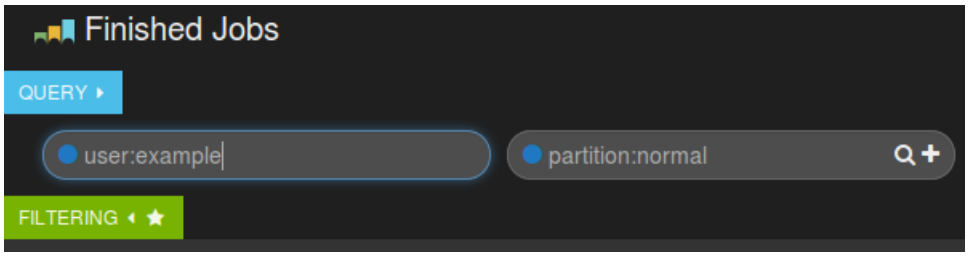



Figure C.4: Kibana querying and filtering options.

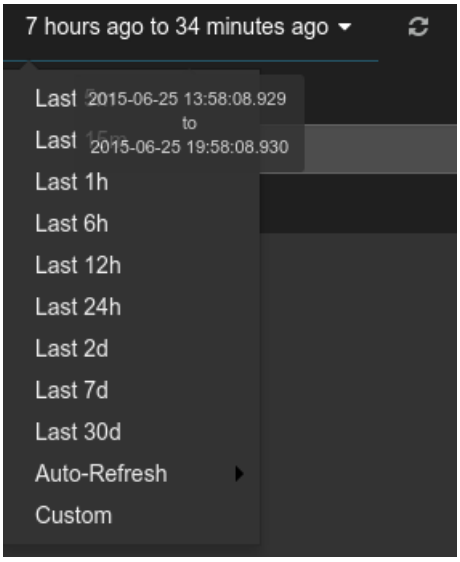



Figure C.5: Data time and Auto-Refresh options.

The figure above shows the dashboard time options. The timepicker reference field can also be configured, and in both of our production clusters the `@end` field is used.

The next figure shows two pie charts with the percentage of CPU hours per `partition` and per `qos` respectively. This kind of panels can be configured with different layouts, such as bar or numeric panels and some other features.

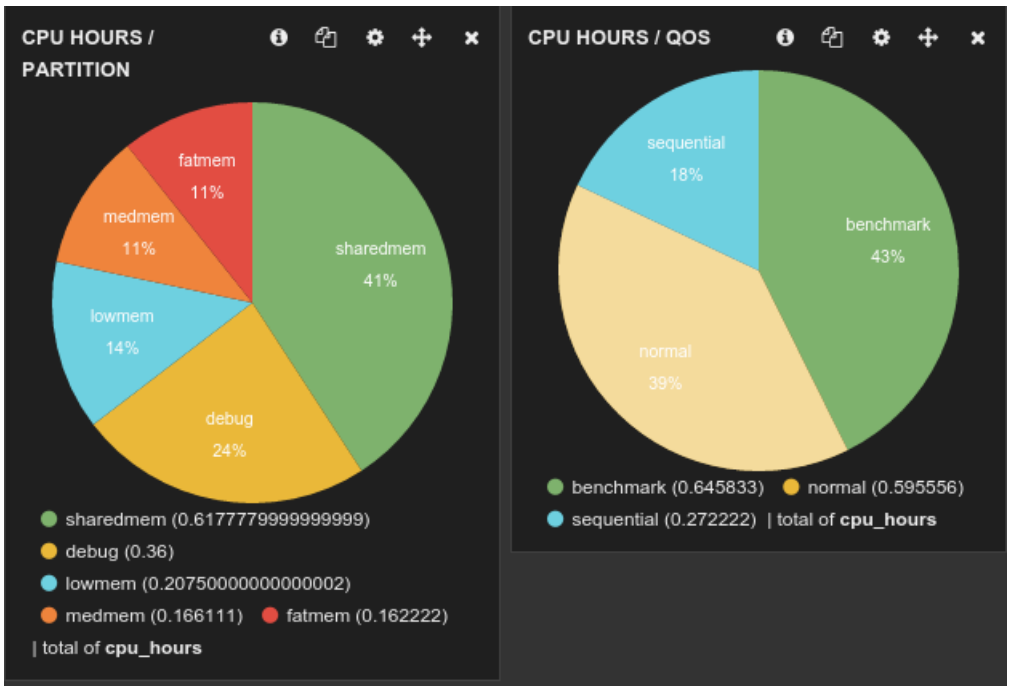



Figure C.6: CPU hours per partition and per queue pie charts.

The following figure shows an histogram panel with a line for every type of finished job state: COMPLETED, FAILED, TIMEOUT or NODE_FAIL (except CANCELLED state that was excluded).

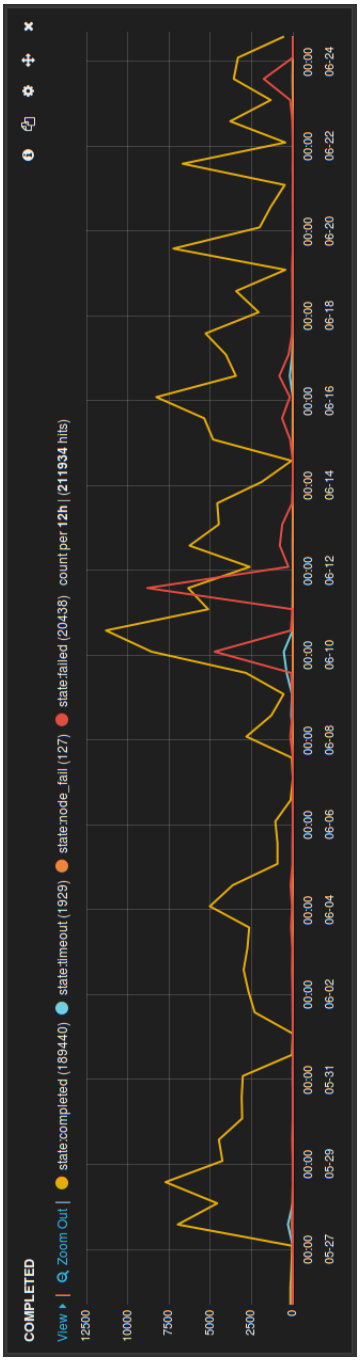



Figure C.7: Histogram of finished jobs break down in a time range.

Raw numerical data can be shown too. The next figure illustrates the top users with more accumulated elapsed time.

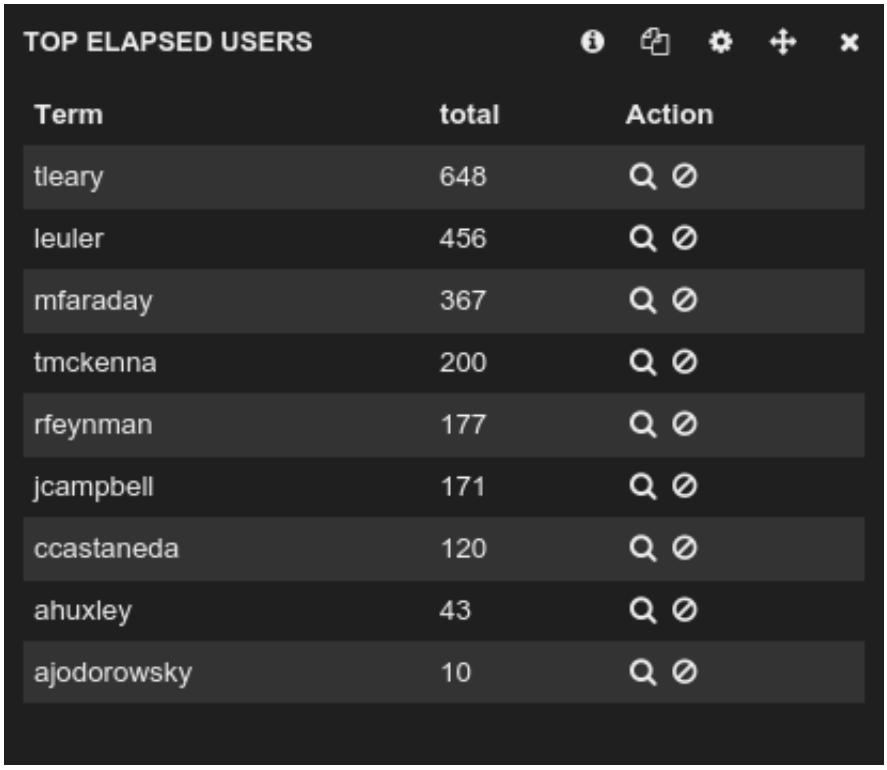



Figure C.8: Terms type panel with top elapsed time users.

More panel and feature options are described in the official Kibana webpage. The purpose of this section was to illustrate a little bit of the analysis potential offered by this tool with our work generated data. More features are being added in newer Kibana releases. At the time of doing this work, the version used was `kibana-3.1.2`.

# Bibliography


[1] SchedMD. Slurm Workload Manager. Site, accessed 2014.

[2] Barcelona Supercomputing Center. Site, accessed 2014.

[3] IBM Platform Computing. Site, accessed 2014.

[4] IBM Platform LSF Master Batch Daemon. Site, accessed 2015.

[5] IBM Platform LSF Batch Log File. Site, accessed 2014.

[6] Inotify Linux Kernel Subsystem. Site, accessed 2014.

[7] ELK Stack. Elasticsearch, Logstash and Kibana. Site, accessed 2015.

[8] The JSON Data-Interchange Format. Site, accessed 2014

[9] IETF. JSON Request for Comments, RFC7159. Site, accessed 2014.

[10] SchedMD. Job Completion Logging Plugins. Site, accessed 2014.

[11] Sanchez, A. Jobcomp Elasticsearch Code. Site, accessed 2015.

[12] SchedMD. Generic Resource (GRES) Scheduling. Site, accessed 2014.

[13] Libcurl. The Multiprotocol File Transfer Library. Site, accessed 2014.

[14] IETF. HTTP Request for Comments, RFC 2616. Site, accessed 2014.

[15] SchedMD. Slurm Configuration File. Site, accessed 2014.

[16] SchedMD. String Arrays (Un)Packaging. Site, accessed 2014

[17] Dept. of CS, University of Meryland. Endianness. Site, accessed, 2014.

[18] SchedMD. Slurm Programmer's Guide. Site, accessed 2014.

[19] GNU Project. Automake, GNU Coding Standards. Site, accessed 2015.

[20] GNU Project. Automake Recursing Subdirectories. Site, accessed 2015.

[21] CURL Project. Libcurl External Macro File. Site, accessed 2015.

[22] RPM. The Spec File. Site, accessed 2015.

[23] ELK Stack. Kibana Index Configuration. Site, accessed 2014

[24] Sanchez, A. Jobcomp Elasticsearch NEWS changes. Site, accessed 2015.

[25] GNU. GDB: The GNU Project Debugger. Site, accessed 2014.

[26] Valgrind. Debugging And Profiling Linux Programs. Site, accessed 2015.

[27] About The Waterfall Model. Site, accessed 2015.

[28] Wikipedia. Iterative And Incremental Development. Site, accessed 2015.

[29] Agile Software Development. Site, accessed 2015.

[30] SchedMD. Slurm Backfill Scheduling. Site, accessed 2015.

[31] SchedMD. Slurm Job Submit Plugin API. Site, accessed 2015.

[32] HPCKP. Slurm Training'15. Site, accessed 2015.

[33] Sanchez, A. Jobcomp Elasticsearch Github. Site, accessed 2015.

[34] SchedMD. Changes In Slurm 15.08.0pre5. Site, accessed 2015.

[35] SchedMD. Plugin Merged In Slurm Master Branch. Site, accessed 2015.

[36] SchedMD. Slurm-Dev Mailing List. Site, accessed 2015.

[37] Erich Strohmaier *et al.* The TOP500 Supercomputers List. Site, accessed 2015.

[38] BSC. MareNostrumIII Support Overview. Site, accessed 2015.

[39] BSC. MinoTauro Support Overview. Site, accessed 2015.

[40] BSC. CNAG Support Overview. Site, accessed 2015.

[41] BSC. Altix 2 UV100 Support Overview. Site, accessed 2015.

[42] F. Guim, J. Corbalan. A Job Self-Scheduling Policy for HPC Infrastructures. *13th Workshop on Job Scheduling Strategies for Parallel Processing (In Conjunction with ICS 2007), June 2007.*

[43] F. Guim, I. Rodero, J. Corbalan, A. Goyeneche. The Grid Backfilling: a Multi-Site Scheduling Architecture with Data Mining Prediction Techniques. *Coregrid Workshop In Grid Middleware 2007.*

[44] F. Guim, J. Corbalan, J. Labarta. Prediction f based Models for Evaluating Backfilling Scheduling Policies. *Accepted for publication in PDCAT 2007 (8th International Conference on Parallel and Distributed Computing, Applications and Technologies).*

[45] J. Han and M. Kamber. Book: Data mining: Concepts and techniques. *Book, 2001.*

[46] D. Heckerman. A tutorial on learning with bayesian networks. *1995.*

[47] A. Karalic. Employing linear regression in regression tree leaves. *In European Conference on Artificial Intelligence, pages 440–441, 1992..*