

Loginson: a transform and load system for very large scale log analysis in large IT infrastructures

Carlos Vega · Paula Roquero · Rafael
Leira · Ivan Gonzalez · Javier Aracil

This is a version of an unedited manuscript that
accepted for publication. Please, cite as:

Vega, C., Roquero, P., Leira, R. et al.,
Loginson: a transform and load system for
very large-scale log analysis in large IT
infrastructures, The Journal of Supercom-
puting (2017)

Received: 27 Jun 2016 / Accepted: 22 Feb 2017

The final publication is available at Springer via:
<http://dx.doi.org/10.1007/s11227-017-1990-1>

Abstract Nowadays most systems and applications produce log records that are useful for security and monitoring purposes such as debugging programming errors, checking system status and detecting configuration problems or even attacks. To this end, a log repository becomes necessary whereby logs can be accessed and visualized in a timely manner.

This paper presents *Loginson*, a high performance log centralization system for large-scale log collection and processing in large IT infrastructures. Besides log collection, *Loginson* provides high level analytics through a visual interface for the purpose of troubleshooting critical incidents.

We note that *Loginson* outperforms all of the other log centralization solutions by taking full advantage of the vertical scalability, and therefore decreasing Capital Expenditure (CAPEX) and Operating Expense (OPEX) costs for deployment scenarios with a huge volume of log data.

Keywords Log analysis · operational intelligence · data repositories for log collection · large datacenters

1 Introduction

In large datacenters and, in general, IT infrastructures, monitoring becomes an essential activity to ensure continuity and quality of service [1]. To this end, systems and applications produce log records, that provides valuable information about the system or application health. For example, system logs serve to detect high memory usage or swapping activity in a given node, which is useful to indicate a possible anomaly in the node's behavior.

In this paper we focus on log collection and processing for large IT infrastructures. The distinguishing features of such use case, compared to generic data col-

lection systems (Hadoop¹ [2], HBase² [4], Elasticsearch³ [5], etc. [6]), are twofold. First, logs are produced by a large number of servers and routers and the resulting amount of data is huge. They are also produced at a very high rate, which poses a severe challenge for log collection. Second, the task of log inspection encompasses two phases that come in sequence: high-level browsing and, then, detailed analysis of selected parts of the log. Actually, in the first place, a high-level data summary must be provided; for instance, a graph showing different HTTP response codes versus time. Whenever an anomaly occurs, such as a sudden increase in the number of internal server error codes (500), the manager should be able to drill down into the data to find the root cause. However, *this second phase only applies to the log data within the time interval when the anomaly occurred*. Such two-phase nature of log data examination calls for a distributed data processing and storage system specifically tailored to this use case, which is the focus of our research.

Actually, in this paper we propose a novel log processing and storage system that we call *Loginson* which is specifically targeted towards processing, storing and visualizing massive amounts of logs. As it turns out, detecting incidents within millions of log messages is a very challenging task, that requires a high performance system providing the above-mentioned two-phase browsing capabilities, namely high-level and detailed or drilled-down. At this point, before we provide further insights into the *Loginson architecture*, let us review the motivation behind this research: providing the network and system manager with a powerful, scalable and easy-to-use system which is good to tackle the daunting task of log analysis.

1.1 Motivation

The current Internet will likely grow [7][8][9] and evolve into the so-called Internet of Things [10][11][12], such that many small devices (meters, wearable devices, etc) will be producing data every second, not to mention large servers and routers. By data we refer to logs in the broad sense of the term, namely files with sequentially timestamped lines that have some interesting data for the operators. For example, the following box shows a log entry from the well-known Apache server:

```
64.123.18.10 - - [07/Mar/2015:16:10:02 -0800] "GET /format.css
HTTP/1.1" 200 6291
```

In the latter example, we can easily recognize the timestamp (date and time), IP address of the server, response code (200 OK) and response time (6291 microseconds). However, we note the log line *must be parsed in order to extract such values*.

As for the log capture and processing workload, both the volume and the rate at which logs are produced are important. We set our objective to the ambitious goal of handling *several millions of lines of log per second*. For instance, if log lines are 300 bytes long then a 10 Gbps link will be fully saturated at a rate of roughly 3 million logs per second. That is the offered rate of a population of 30 million

¹ <http://hadoop.apache.org/>

² <https://hbase.apache.org/>

³ <https://www.elastic.co/products/elasticsearch>

small devices that generate a log line every ten seconds or a thousand servers that generate three thousand lines of logs per second each. Such a vast workload, far beyond the state of the art as we will show, must be stored, filtered, aggregated and visualized for the purpose of detecting unexpected events or incidents.

In what follows we describe the main motivation behind this paper, i.e. the requirements that network and system managers have for log analysis. We have gained this experience through many different projects with large banking datacenters and operators, which have been carried out in our spin-off company *Naudit*⁴. Precisely, this work is motivated by many use cases in large IT infrastructures with a massive amount of logs being produced, which could hardly be managed with traditional techniques.

The first requirement is that network managers must centralise the logs generated by systems currently in production. We note that this is not only an operational requirement but a legal one in the banking sector⁵. In order to send the logs from the servers or routers, Syslog [13] constitutes a de facto standard, either through UDP, TCP or TLS. Therefore, we believe that Syslog is the best option due to its widespread implantation, which will continue to grow in the future. Consequently, in the central log collection repository, a *Loginson* component must capture the log stream at the targeted rate of millions of log lines per second. Such component provides load-balancing by splitting the data stream as fast as possible to the different processing and database nodes, in round-robin fashion.

The second requirement is to pre-process the raw data and provide a high-level summarized data overview whereby pre-processing implies filtering, aggregating and, in general, performing a pre-analysis of the raw log data. This third hurdle has the extra handicap of dealing with the specific characteristics of each kind of log message and its format. Hence, each log type demands a different filter or aggregator (for example number of HTTP Internal Server Error codes -500- every five minutes). Furthermore, flexibility is a key requirement in the *Loginson* pre-processing component. The output of this component is a summarized version of the raw data that is manageable for plotting in charts, time series and histograms and offers a high-level overview which is useful to detect incidents.

The third requirement is to provide a graphical representation of the summarized data in a user-friendly interface that allows to visualize incidents. We note that the data trimming and aggregation of the previous paragraph largely reduces the dataset to be plotted and paves the way for using open-source dashboard visualization solutions (such as *Kibana*)⁶ with modest resources (small number of virtual machines). Therefore we decided to incorporate *Kibana* (with the *Elastic Stack*⁷) as a visualization tool rather than providing our own dashboard implementation tool.

1.2 Novelty

In this paper, we are specifically considering the case of real-time log monitoring in large IT infrastructures. We note that monitoring is cornerstone for networks and

⁴ <http://www.naudit.es>

⁵ <https://www.pcisecuritystandards.org/>

⁶ <https://www.elastic.co/products/kibana>

⁷ <https://www.elastic.co/v5>

datacenters and, consequently, the proposed use case covers a widespread demand for monitoring at higher rates. As it turns out, monitoring demands log ingestion, processing and visualization systems whose underlying database sacrifices classical database functionalities for the sake of speed, centralization and adaptation to dashboard implementation tools.

First of all, syslog has become a de-facto standard for sending the logs at the server side. The introduction of software agents at the server, which could perform log pre-processing and distribution to many log collection nodes, is costly and in many cases not feasible for critical servers. Such requirement impedes the use of distributed database systems in which the data producers send data to the consumer nodes by means of smart load balancing algorithms [14], instead of using a central point for data collection. Actually, making all servers send the logs to a central point is much simpler, because all servers use the same destination IP address, namely that of the central collection point. Such scheme does not preclude high availability, as the central point can be implemented in several hosts. Remarkably, the central collection system is stateless, which simplifies matters for redundancy.

Second, the queries performed by system and network managers are much simpler than standard SQL statements. Whenever an incident happens, the queries that are typically performed consist of simple IP/port range searches within a given time interval (that of the incident), not complex SQL statements. While classical databases provide a better query language capabilities, our use case sacrifices query language capabilities for speed, analysis capabilities [15] and better data visualization features. We also note that the recent log messages are the most important and will be most queried, whilst the old ones are less and less relevant as time goes by.

Third, we stress the importance of data visualization for system and network monitoring. With many different servers and network segments to monitor, system and network managers strive for building a simplified, yet complete, graphical dashboard of the datacenter. Such a requirement calls for online data summarization capabilities, as all the data cannot be displayed in a dashboard.

Loginson's novelty lies in tackling all the above requirements at a very high log ingestion speeds. Thus, *Loginson* differs from more complex distributed database systems that are targeted to more generic data collection and search use cases, providing a novel ad-hoc system for log collection, processing, storage and visualization.

2 State of the Art

Most of the distributed processing and storage systems are useful for generic data analysis scenarios, not specifically for log collection, processing and visualization. Their main goal is to perform complex analysis of a large quantity of persistent data [16] that is inserted at a relatively low rate.

Actually, traditional big data systems can scale horizontally [17] to a large number of nodes, which are load-balanced through complex algorithms. Such systems try to hide low-level complexity from the programmers, allowing them to focus on the problem to solve. The tradeoff is that this abstraction adds a no-

ticeable overhead when using fewer nodes, as vertical scalability is sacrificed to achieve simplicity.

In the case of *Loginson*, the loss of vertical scalability is not acceptable. We have soft real-time constraints that cannot be met by any of the generic systems we have tested unless a very large number of nodes is used[20]. Furthermore, vertical scalability makes it feasible to use active-active configuration for data replication, because the number of nodes involved is not so high. Our conclusion is that a custom system is necessary for the real-time analysis and storage components of *Loginson*. After the amount of data is reduced by the former, traditional systems like *Elasticsearch* can be used to offer high level analytics.

In what follows we will review the state-of-art for our specific use case of log storage and visualization to conclude that no system to date fulfills the requirements set forth in the previous section. We divide the state of the art in four sections: message brokering and load balancing systems, database nodes, pre-processing (data transformation) components and graphical representation. As a summary, Table 1 presents the most salient performance aspects of the different tools included in the state of art.

Table 1: System Comparison Summary

Purpose	System	Performance
Log Message Brokers	Syslog-ng	650K log/s multithread [18]
	Apache Kafka	300K log/s Using one producer and one consumer
	Apache Flume	0.77M log/s/thread
	Logstash	29.5K log/s/thread
	FluentD	13K log/s/thread
Database Storage	Cassandra	251K TPS ¹ on a single node [19]
	ScyllaDB	1.8M TPS on a single node [19]
Pre-process (Split and join the log fields)	Apache Storm	900K logs/s Using 100% of all cores
	AWK	900K log/s/thread
	Python	500K log/s/thread
	Perl / Ruby	300K log/s/thread
Graphical Representation	Kibana	Official Elastic tool, like ES ² Proper dashboard composer tool User friendly and chart versatility
	Grafana	Unofficial, lacks support Proper dashboard composer tool Charts almost restricted to time series
	D3.js	Requires complex middleware Big catalog of charts JS ³ library, requires custom development

¹ Transactions per second

² Elasticsearch

³ JavaScript

2.1 Message brokering and load balancing for log centralization

The most straightforward choice for log centralization is to adopt one of the syslog daemons already existing in the open-source community. Out of these, **Syslog-ng** is the fastest, because it was designed for speed. It can be easily configured to receive logs from a client syslog daemon and to store them in a centralized location, which make it the ideal solution for our problem. Unfortunately, **Syslog-ng** cannot achieve a throughput of three million log lines per second, peaking at 650,000 [18].

Another popular system that comes at hand to centralize logs is called **Apache Kafka** [21], a broker that can receive data from multiple sources and distribute it to several destinations. Kafka uses a publish-subscribe pattern which allows to persist the data in disk as well as to distribute it to the subscribed consumers (logstash, fluentd, TCP clients, etc.). Kafka categorizes messages by topics, making it possible for consumers to subscribe to one or many topics. Such topics are divided into a number of partitions, which can be accessed in parallel by multiple consumers. In the case of producers, they can publish their log messages to a specific topic.

In benchmarks published by LinkedIn [22], a three node cluster is capable of ingesting 2 million events/s generated by three producers. The tests also show that producer throughput is not reduced by adding consumers, as one producer is capable of inserting 800,000 events/s with or without consumers.

Unlike Syslog-ng, Kafka is not focused on log processing and additional work becomes necessary. Kafka seemed to meet the needs of our use case, so we tested it using a server with two Intel Xeon E5-2630 v2 @ 2.60GHz CPUs, 32GB of RAM and a RAID 0 with 10 HDD. We found that Kafka did not satisfy our requirements, it used 100% of the machine CPU, showing an ingestion rate of 300,000 log entries (size 117 bytes) per second using one producer while reading from one consumer. Other log collection systems⁸ that use Kafka also get similar results. This throughput is an order of magnitude below our requirements, so we had to discard the use of Kafka in *Loginson*.

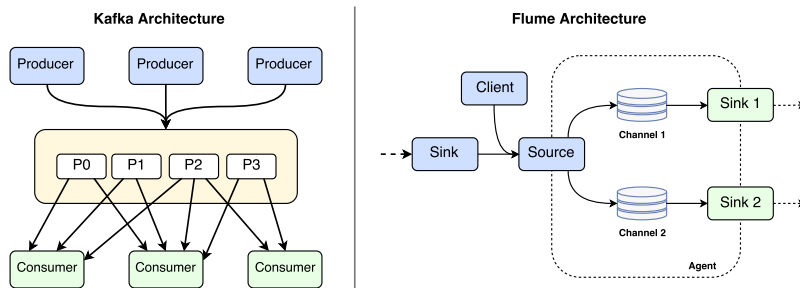


Fig. 1: Flume and Kafka architectures

An alternative to Apache Kafka is **Apache Flume**⁹, with the main difference being that Flume pushes the data to the destination, whilst Kafka's clients pull

⁸ <https://www.loggly.com/>

⁹ <https://flume.apache.org/>

the data from Kafka's queues at their own pace. While Flume's sinks can read only from one channel, multiple sinks can be read from the same channel. Furthermore, multiple sources can write on the same channel. Figure 1 compares both architectures. As Kafka's subscribers, Flume's sinks can be of different types, like HDFS[3], Elasticsearch, HBase, etc.

We tested Flume on the same machine with two different sink configurations: Null Sink, that discards all the events it receives from the channel[23]; and File Roll Sink that stores the events in files. To achieve maximum performance the output files were stored in a RAM disk. Flume achieved a performance of 1,500,000 log messages per second and thread with the former kind of sink and 770,000 logs per second and thread, with the latter using a log size of 291 bytes. More details about the test can be found on Figure 9 in Section 4. Again, this alternative did not match our performance needs.

There are other systems dedicated to log processing that are worth mentioning even if they do not provide the necessary throughput. **Logstash**¹⁰, developed by *Elastic*, collects, processes and forwards log messages from and to different systems (TCP, UDP, *Elasticsearch*, *Kafka*, *Omq*, etc.) promising 50,000 events per second¹¹ when using grok filters for log patterns.

Similarly, Treasure Data developed **FluentD**¹², an alternative to Logstash that presumes of a throughput of 800,000 events per second [24] in their Big Data cloud service, but without specifying any details [25]. On their website they describe the system as being able to process 13,000 events/second/core [26]. We tested both Logstash and FluentD on the same machine as the previous tests, without performing any process or transformation on the log entries but the default, which is to add a timestamp to the log, with a first test forwarding messages to stdout after being read from files on a RAM disk; a second test in which the log entries are received via TCP; and a last test also getting the log lines via TCP but then sent to null using their corresponding null output plugins. As seen in the Figure 2, both Logstash and FluentD do not surpass 60,000 log messages per second except for the tests in which log messages are sent to null. Hence, we had to dismiss such systems.

2.2 Database nodes

After receiving the log messages in a load-balancer they must be split among several database/storage nodes. That way the write workload per disk is reduced, which leaves room for the read operations not to affect the write performance.

There are several options to store this data, from big data systems to simple key-value storage. Our extensive tests have shown that, while **our solution needs only one server to process up to 10 million lines of log per second**, big data storage solutions like **Apache's HDFS** or **Cassandra**¹³ are only capable of meeting our targeted input rate by using at least 5 servers, so we had to discard them for cost-effectiveness [27]. Such systems are better tailored to a more generic

¹⁰ <https://www.elastic.co/products/logstash>

¹¹ <https://www.elastic.co/blog/logstash-1-5-0-ga-released>

¹² <http://www.fluentd.org/>

¹³ <http://cassandra.apache.org/>

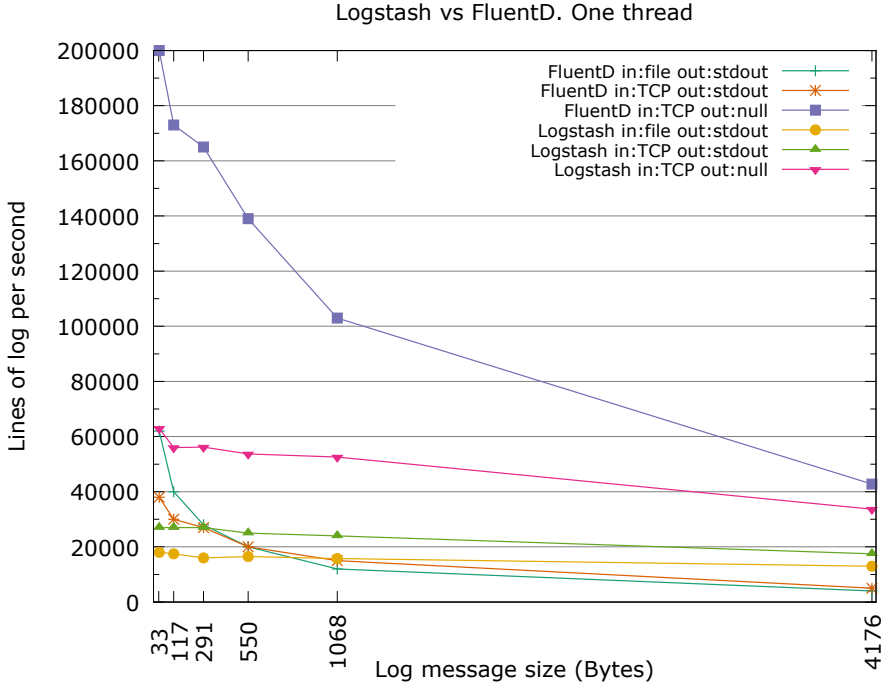


Fig. 2: Logstash and FluentD benchmarks

use case at the expense of performance, with features like fault-tolerance and high availability that, despite its value, can be achieved by other means such as system redundancy.

In the case of key-value storage, there are several benchmarks [28] that show that none of them can achieve a write speed fast enough for our purposes. Moreover, such systems are designed for random reads and writes, so they were discarded in favor of sequential flat file storage, as log entries are usually accessed in sequential chunks.

Some systems like **ScyllaDB**¹⁴ [29] (an open-source C++ rewrite of Cassandra) claim to process up to 1M transactions per second per server. One of their benchmarks [30] uses a cluster with 3 nodes with 128GB of RAM and 4x 960GB SSDs on RAID each. Using a replication factor of 3, the cluster provides an average performance of 1.9M transactions per second (TPS).

Another benchmark has been reported in [19] using a single node which gives a throughput of 1.8M TPS. While the performance of ScyllaDB surpasses Cassandra's (251K TPS for the latter test), the tests of these benchmarks were done using SSDs disks, which have a better performance on simultaneous write/read operations than the conventional HDDs at a higher cost. For example, a system

¹⁴ <http://www.scylladb.com/>

that collects 3 millions log lines per second produces an approximate 42 TB worth of log messages per day, so using SSDs would be extremely expensive.

Other solutions such as **MapRDB**¹⁵ have a commercial license of 4,000\$ per node (2011) [31]. MapR accomplish the ingest of 100,000,000 points per second with 4 nodes and a replication factor of 3 thanks to its own filesystem mapR-FS [32].

Splunk, is a search server similar to Elasticsearch, which is available with a commercial license at a price of 28,750\$ for 10GB/day, 172,500\$ for 100GB/day on their Enterprise Version and 15,870\$ for 10GB/day for their cloud service. It is a closed-source software which, according to their own tests [33] [34], gives a performance of 80,000 events per second on a Linux 64-bit HP DL380G7 machine with 2x6 Xeon @2.67 Ghz and 12GB of RAM. **Elasticsearch**, which, contrary to Splunk, is an open-source solution, achieves the same speed with a dual Xeon X2699 @2.3 Ghz and 256GB of RAM using just a 4GB heap, according to their benchmarks [35].

2.3 Pre-processing the logs

To pre-process the raw data, a real time streaming system is necessary. A popular solution is **Apache Storm**¹⁶, a distributed realtime computation system. We ran a benchmark in one node with two 6 core Intel Xeon E5-2630 v2 CPUs. Using a 100% of all cores, storm is capable of transferring 900,000 log lines per second, just splitting and joining the fields of the log line. Other systems like **Apache Spark**¹⁷ streaming and **Apache Flink**¹⁸ had worse performance.

Such performance figure of 900,000 log lines per second was obtained using the AWK programming language, and with a single core. Furthermore, such throughput decreases to 500,000 log entries per second when using Python and to 300,000 with Ruby or Perl.

Languages like AWK or Python can be linked using small wrappers or unix pipes to create a streaming pipeline. They offer the benefit of being easy to use and well-known, with no need of specialized programmers.

2.3.1 Serialization

The problem of serialization arises when parsing at high speed becomes necessary to convert log messages from its original format to *JavaScript Object Notation*, JSON. One of the main constraints we work with is the hindrance of not to incorporate any software to the log senders as they are usually systems in production that should not be touched. As a consequence, the log format cannot be altered in the source even though some tools like Apache, Nginx or even syslog-ng OSE (which is able to send logs directly to *Elasticsearch*) [36] support *JSON logging*.

As explained previously, systems like *Logstash* or *FluentD* are not fast enough for collecting, nor parsing, the log messages. The resulting conversion throughput is between 15,000 and 30,000 log lines per second depending on the log size.

¹⁵ <https://www.mapr.com/>

¹⁶ <http://storm.apache.org/>

¹⁷ <http://spark.apache.org/>

¹⁸ <https://flink.apache.org/>

2.4 Graphical representation

The visualization system is an essential part of the architecture as it interacts with the final user of the system, hence, the simplicity and flexibility are of fundamental importance to favor user experience. In this regard, charts must be dynamic and flexible enough to meet the needs of the data analysts, yet quick enough to get an overview of the data in a matter of seconds.

Some tools like **D3.js**¹⁹ (a JavaScript library) feature a huge catalogue of different kind of charts (tree maps, box plots, bubble charts, etc.) but require a web developer with good programming skills in order to make any change to the charts, in addition to the fact that they are not able to connect by themselves to the datastore. Hence, a complex middleware must be put together to make them usable.

Moreover, tools like **Grafana**²⁰ or **Kibana** are able to connect to the source of the data and display a large variety of charts without requiring advanced programming skills. In the case of Grafana, it lacks variety of charts, and it is restricted to time series only.

Finally, *Kibana* has both a variety of charts (pie charts, time series, histograms, etc.) and a data browsing API to the *Elasticsearch*²¹ indexes which makes it straightforward to manage or change the chart dashboards. Furthermore, *Kibana* and *Elasticsearch* are in constant development by the *Elastic team*, which recently released *Timelion*²², an specific plugin for *Kibana* that provides time series functionality for the charts (derivatives, moving averages, etc.), thus adding support for some functionalities just seen on *Grafana*. *Kibana*, unlike *Grafana*, also supports embedding dashboards and charts allowing for custom web pages.

All the above-mentioned features makes *Kibana* our graphical tool of choice for plotting the charts and graphs.

3 Our architecture

In *Loginson* we receive all logs in a centralized server called *LogFeeder* that adds information like the timestamp to the logs, to avoid timing problems with logs from different timezones or from desynchronized sources. Then, they are split between several storage nodes, using round robin. The storage consists of flat files of 1 GB size. Such size is large enough for the writes and reads to take advantage of high speed sequential operations on spinning disks. If high availability is required, an active-active *Loginson* system with data replication could be used to avoid loss of data. Such data replication comes at a modest cost, because nodes are highly utilized (vertical scalability), in contrast to the low utilization of pure horizontally scalable systems.

We note that the proposed system provides high performance due to its simplicity and vertical scalability. If higher performance is required, more storage nodes can be added to split the load. The central collection point *LogFeeder* becomes a bottleneck, though, which can be alleviated by the use of load balancer

¹⁹ <https://d3js.org>

²⁰ <http://grafana.org>

²¹ <https://www.elastic.co/products/elasticsearch>

²² <https://www.elastic.co/blog/timelion-timeline>

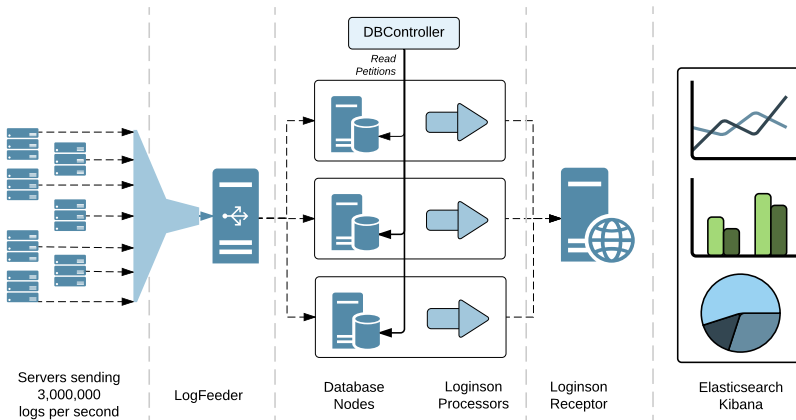


Fig. 3: Loginson architecture

and several *LogFeeder* nodes. In any case, *LogFeeder*'s throughput reaches several million log records per second (see figure 6), which is sufficient for extremely large IT infrastructures.

While the logs are stored on the database nodes, a summary is constructed using the programming language AWK (for instance, moving averages of the data), and sent to a component called *LoginsonReceptor*. Such component centralizes the processed logs and inserts them in an *Elasticsearch* database so *Kibana* can be used to visualize different metrics on the summarized data. Usually, *LoginsonReceptor* could be placed on a different node, such that the access to the summarized data does not interfere with the collection of new data. To further drill down into the data, our solution exposes a web API to access the logs stored in the database nodes. Such component, called *DBController*, is capable of queueing several read requests that can be deleted at any moment. Thus, the user is able to abort and modify the drill-down requests without waiting for them to finish. The logs are then processed in the same way as in real time and sent to the *LoginsonReceptor*.

In what follows, we will provide detailed insight of each of the system components. Figure 3 shows the *Loginson* architecture.

3.1 Message brokering and load balancing for log centralization

We choose to perform load balancing from a central node to the database nodes instead of making the clients send the logs directly to specific database nodes. By doing so, we avoid that some of the database nodes receive more load than others and simplify configuration at the client side. Such load balancing node is called *LogFeeder*. It receives log lines (within UDP) using the high speed HPCAP network driver [37] for Intel[®] Ethernet 10 Gb PCI Express NICs, capable of receiving network traffic at 10 Gbps.

To achieve high speed, *LogFeeder*, as shown in figure 4, uses several threads that share information through a circular queue of buffers. This queue is protected

through a mutex. Buffers are used to avoid locking the mutex for each log, which would decrease performance dramatically.

One of the threads, called *Receiver Thread*, reads the logs from the network interface, performs timestamping and writes them to a buffer in the circular queue. This buffer is then made available for processing, and the Receiver Thread goes on to the next buffer for writing. The rest of the threads, called *Header Threads*, read a filled buffer from the circular queue in a first-come first-serve basis. For each line of log in the buffer, such threads add a header with information about the log source and send them to the database nodes using a Round-Robin approach.

An advantage of using Round-Robin instead of more complex algorithms is that logs are uniformly split between the database nodes. This makes subsequent access to the database faster, as it can take advantage of data parallelism.

3.2 Database nodes

Logs are ordered by timestamp and will be accessed in batches for analysis. Most existing databases are not adequate for the job, as they are optimized for random reads and writes. Instead, each storage node works with sequential flat files. Within such files, logs are preceded by a 64 bytes header that stores the timestamp and type of log.

3.2.1 Writing to the database

Logs are received using TCP from the *LogFeeder* and copied to one of several buffers in a pool. When the buffer is full, it is written to disk asynchronously as a single file. This way the database does not block and continues receiving logs and copying them to another buffer. As mentioned before, the size of these buffers is 1 GB to take advantage of fast sequential writes in spinning disks. Such files are then indexed in a SQL database by storing the first and last timestamps. The former index will be used to get a list of files whenever logs need to be accessed. While this thread is receiving logs, another one is processing them as will be explained on Section 3.3.

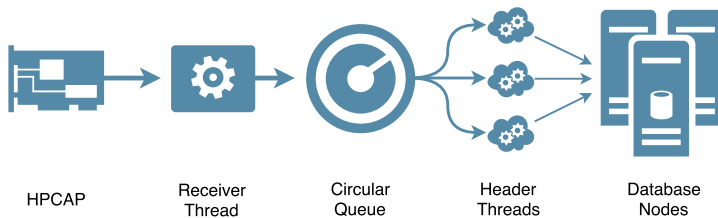


Fig. 4: LogFeeder

3.2.2 Reading logs

The logs are split between several nodes, so we need a single point of access that we called *DBController*. It receives read queries and forwards them to each database node. Read queries are performed through an HTTP API and enqueued until the database nodes are available for reading. We note that the queries can also be canceled even when they are running. When a database node receives a read query from the *DBController*, it first searches in the SQL database for all the files containing logs between the two given timestamps. Each file is loaded to memory and only the logs of the requested type are processed and sent to a program called *LoginsonReceptor* for visualization.

3.3 Pre-processing the logs

To provide a high-level overview of the logs in real-time, a given summary analytics (for example, moving average of HTTP response time) of each kind of log message is sent to its correspondent AWK process through a UNIX pipe. Each of the database nodes use these AWK scripts, one for each type of log, which are utterly flexible allowing for filters, aggregations, or transformations of the relevant log messages as well as its JSON serialization, which is compulsory for *Elasticsearch*.

The result is then sent via TCP to *LoginsonReceptor* (Figure 5), in the graphics visualization server, where the data of each database node is collected and indexed on *Elasticsearch* for subsequent visualization on Kibana and Timelion.

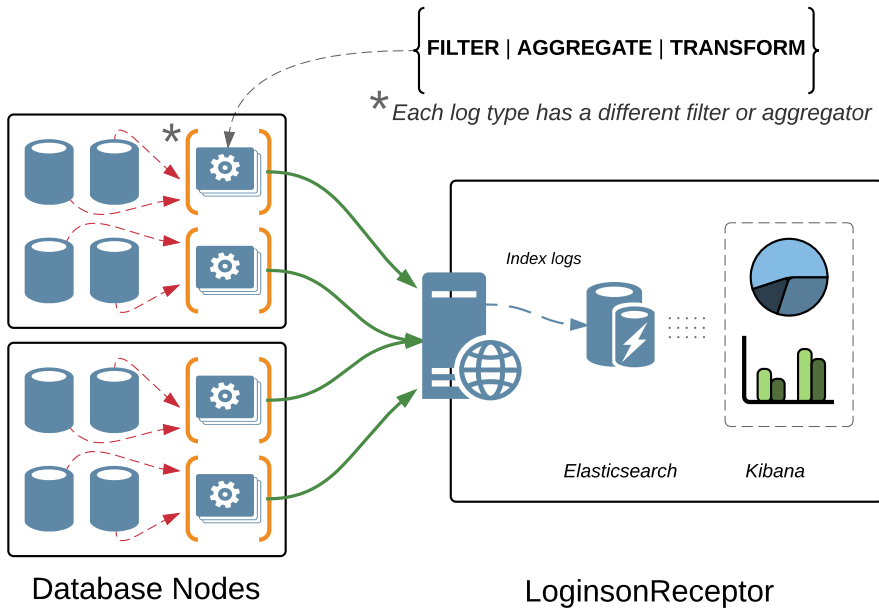


Fig. 5: LoginsonReceptor

Table 2: LogFeeder Performance with logs of 291 Bytes
Number of logs passing through the system every 100 ms.

Number of Header threads	Mean	Median	Standard Deviation
1	512600	516100	17945.03
2	693200	701900	25254
3	1038000	1044000	17588.47
4	1070000	1073000	30756.99

LoginsonReceptor manages the creation of indices and mappings on *Elasticsearch* for each kind of log message making use of the *Elasticsearch Java API*. AWK was chosen for its flexibility and speed as shown on Section 4.3, as well as its high-level capabilities like the use of associative arrays or regular expressions.

In the next section we present a performance evaluation of each component of our architecture as well as the whole system.

4 Performance evaluation

In this section we perform extensive stress testing to ensure that *Loginson* fulfills our performance requirements. We tested each component separately and, then, we ran tests on the whole system.

First, we studied *LogFeeder* performance versus the number of threads used. We also compared the input data with the output and checked that both were equal. Next, we tested the database nodes and performed fine-tuning in order to achieve a constant write rate on a spinning disk. Furthermore, we noticed that log pre-processing did not affect the write performance. Again, our tests showed that the data written to disk was the same as the one inserted in *LogFeeder*.

Afterwards, for the pre-processing stage, we compared different options for the serialization of the log messages as it is a mandatory step before forwarding the logs to *LoginsonReceptor*. These three options were faster than the results obtained with Logstash and Fluentd (Figure 2) but we lastly opted for AWK owing to its speed and flexibility. We also benchmarked the performance of *LoginsonReceptor* indexing logs on *Elasticsearch* in order to check the maximum indexing speed of the latter.

Finally, two tests of the whole system were conducted. To this end, we used four nodes, one of them running *LogFeeder*, two as database nodes and another one running *DBController* and *Elasticsearch*. In the last test only one node with a RAID 0 was used to host *LogFeeder* and the database.

Most importantly, we note that Flume is the only system close to our requirements, thus, the section concludes with a performance comparative between Flume and *Loginson*, leaving aside the rest of alternatives.

4.1 LogFeeder: Message brokering and load balancing for log centralization

We tested *LogFeeder* in a server with two Intel Xeon E5-2630 v2 @ 2.60 GHz CPUs. Each CPU featured six cores and hyperthreading was disabled. The server

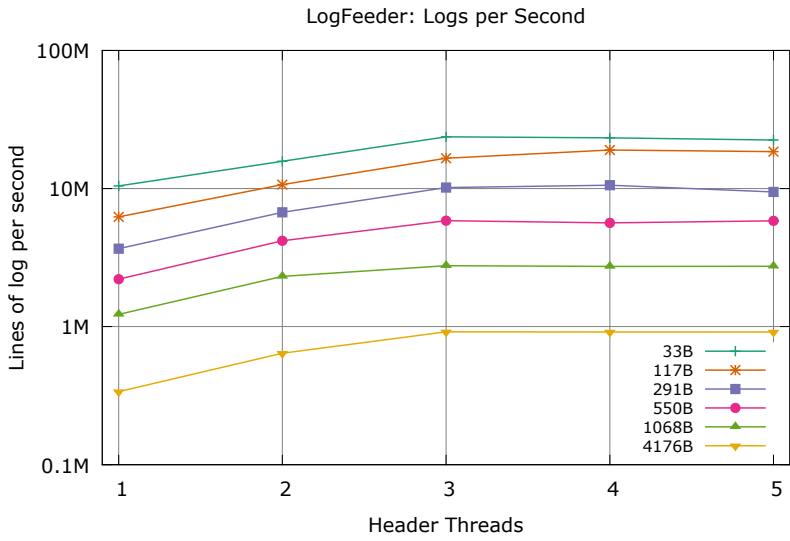


Fig. 6: LogFeeder Speed

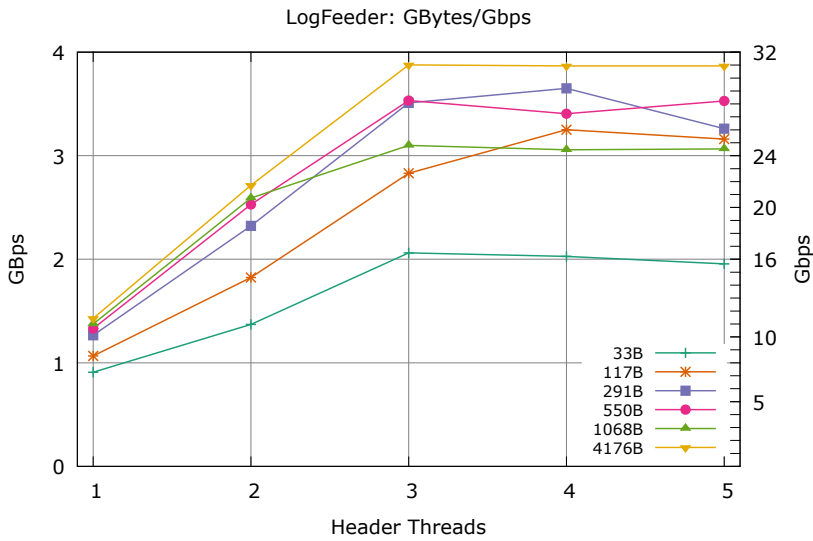


Fig. 7: LogFeeder Gbps/GBps

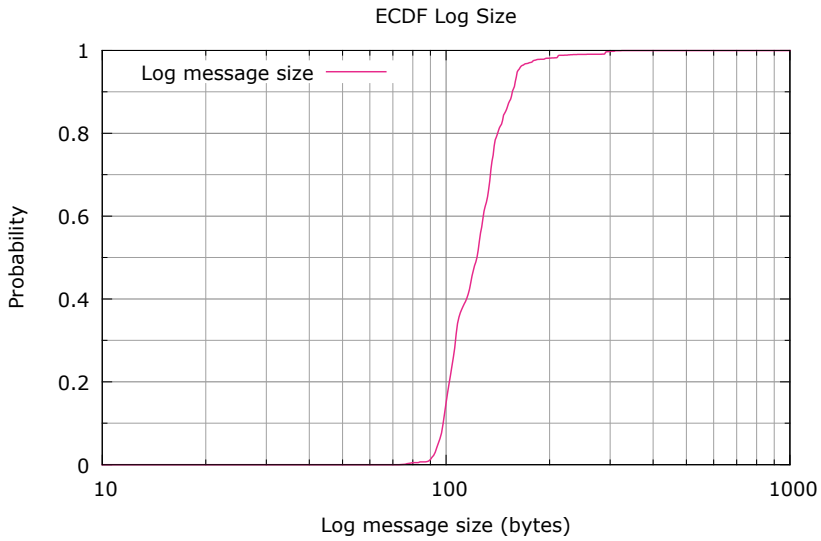


Fig. 8: ECDF of Log message size

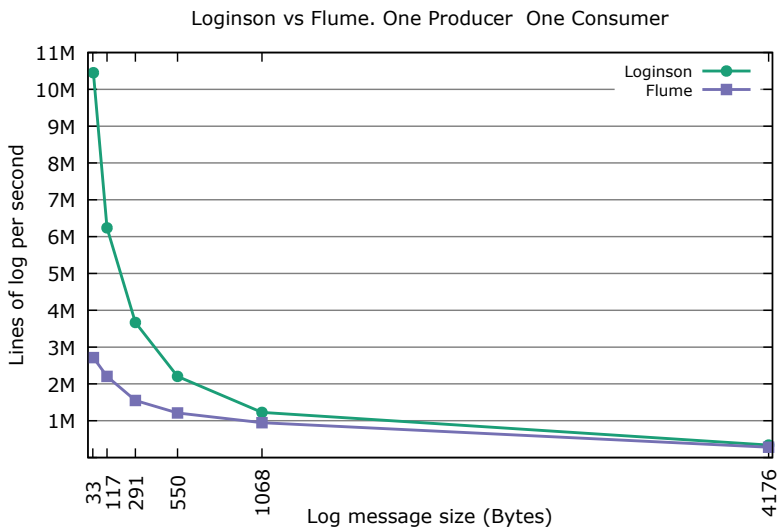


Fig. 9: Loginson vs Flume performance comparison

incorporated 32 GB of RAM memory divided uniformly between two NUMA nodes. The server was running CentOS 7, which was configured to enforce that only the first CPU could be used, by means of *taskset*.

In our first test, we used *LogFeeder* to centralize lines of log of different sizes between 33 and 4176 bytes. The obtained throughput is shown in figures 6 and 7, as well as in Table 2.

We notice that for any log line size, *LogFeeder* provides a throughput higher than 10 Gbps. It can also process more than 10 million log lines per second, if they are shorter than 291 bytes, which, as Figure 8 shows, is the commonest. In order to assess what is the typical size of log lines, we studied a sample from a real production system. We took several days worth of Apache's logs and represented the log line sizes with the Empirical Cumulative Distribution Function (ECDF) of Figure 8. Note that the median log size is around 117 bytes and that 99% of the logs are shorter than 291 bytes, which reinforces the point that a log collection system should be able to work with small log line sizes at very high speed.

We also observe that the highest throughput is achieved when using three or four threads, depending of the log line size. Figure 9 compares Apache Flume and *Loginson* performance. The figure shows the performance of both systems using 2 threads, one for the producer (which we previously called *Receiver Thread* for *Loginson*) and another one for the consumer (*Header Thread* for *Loginson*). As seen, *Loginson* outperforms Flume with small log message sizes, which are the most common as shown in the previous ECDF.

In our second test we wished to assess if the throughput was stable. To this end, we left *LogFeeder* running for more than an hour, measuring the number of logs passing through the system every 100 ms. The results are shown in Figure 10, where we can see that the throughput remains remarkably stable with time.

4.2 Log Database nodes

We tested a log database node using a server with two Intel Xeon E5-2620 v3 @ 2.40 GHz with 6 cores each. The server featured 64 GB of memory and several hard drives of 7200 RPM. The write processes were always ran with the *ionice* command, specifying a real time priority, which is the maximum possible.

The first goal of this test was to check if the database process could take advantage of the full read/write capabilities of a hard drive. To this end, we first wrote 1 GB files directly from memory and determined that the maximum write speed of the drive was 170 MB/s. Then, we launched a database process receiving logs from a *LogFeeder* and the resulting write speed was 160 MB/s (see Figure 11), mainly due to the additional pre-processing.

After that, we wished to find out the maximum write speed to be used so that read operations do not affect the performance. The priority of the read process was set to idle with the *ionice* command. With this priority, I/O operations are only executed if there is no other process accessing the disk.

Consequently, we started the database process by limiting the write speed to 25%, 50%, 75% and 100% of the maximum write speed, which corresponds to 40 MB/s, 80 MB/s, 120 MB/s and 160 MB/s respectively. After the write process is stabilized, we executed a long read operation, while measuring the new write and

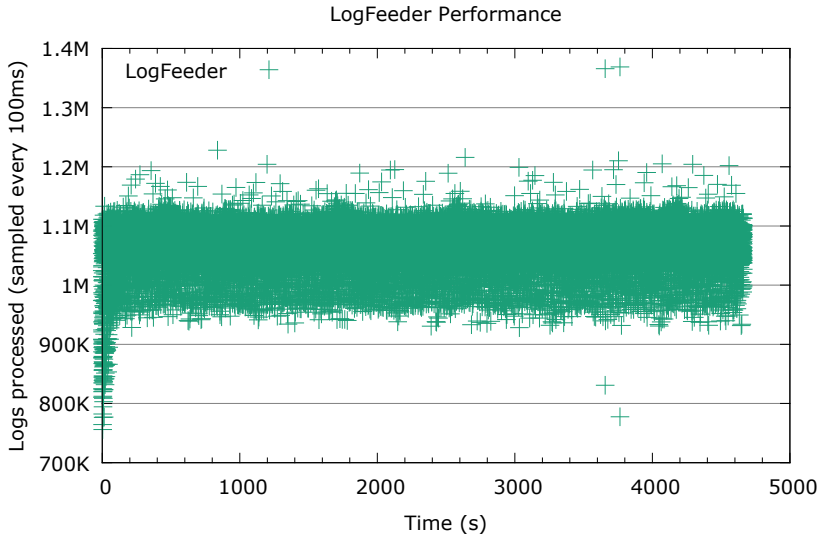


Fig. 10: LogFeeder performance using 4 header threads with logs of 219 Bytes

read speeds. We also tested a use case reading from the database without a write process running at the same time. The results are shown in Figure 11.

We note that using more than 50% (80 MB/s) of the maximum write speed reduces the performance of the write process when a long read is executed. Thanks to this experiments we can determine that the number of hard drives necessary for processing 3 million lines of log per second is equal to 7 when using logs of 117 bytes, and 14 drives for lengths of 291 bytes, plus the 64 bytes header used by our database. The lengths of 117 and 291 bytes are, respectively, the median and the 99th percentile of the log size found in Figure 8.

4.3 Pre-processing

As explained in the previous section, the pre-processing scripts are completely malleable and hence, the performance of each script depends on the process to be performed on each type of log, call it filtering, aggregating and/or transforming the log lines. Notwithstanding, we tried different ways of serializing log lines, (since it is the only mandatory pre-process before the logs are sent to *LoginsonReceptor*), with custom scripts, sending an Apache log file of 1 million lines, 100 bytes long each line, and measuring how long each program took to serialize the corpus. We chose different languages such as Flex, AWK and C, keeping in mind that we do not need them to process 3 million of log entries per second owing to the distributed processing of each kind of log with its corresponding AWK script.

We note that Flex and Bison, a lexical analyzer and parser generator respectively, were used to define the patterns of each log line. Bison defines the structure of the line to be parsed and Flex the regular expressions of each field. These tools

are often used to define custom semantics and develop compilers using patterns and regular expressions. Their major asset is that a pre-knowledge of the log format is defined with rules, patterns and sub-patterns with different priorities, supporting optional fields in a smarter manner. Then, the code can be compiled with `g++`. The resulting parsers are strong but complex due to the pattern-matching performed with the rules. We note that using these tools usually requires training.

A much more straightforward script was done with AWK language, which is fairly easier to use than Flex and Bison. An AWK program is composed of a sequence of pattern-action statements. Each log message is broken up into a series of fields using the given delimiters. AWK is also able to match regular expressions against strings. This flexibility and ease-of-use are good assets that make AWK a good candidate as the code is much simpler to understand and modify. The test script in the experiment simply splits the log with the provided separator and prints the line on JSON format, adding the field names. The code is interpreted without need of compiling. AWK also has other high-level capabilities that make it ideal for the rest of pre-processing operations like transformations, filters, or aggregations thanks to its associative arrays and regular expression features.

Finally, we used C as our third option, with a program in which fields were assigned to a structure and then printed in JSON format, adding the field names too. Although the code does the same than the AWK script, the complexity is higher and the code needs to be compiled as in the Flex option. It also requires a more elaborate programming.

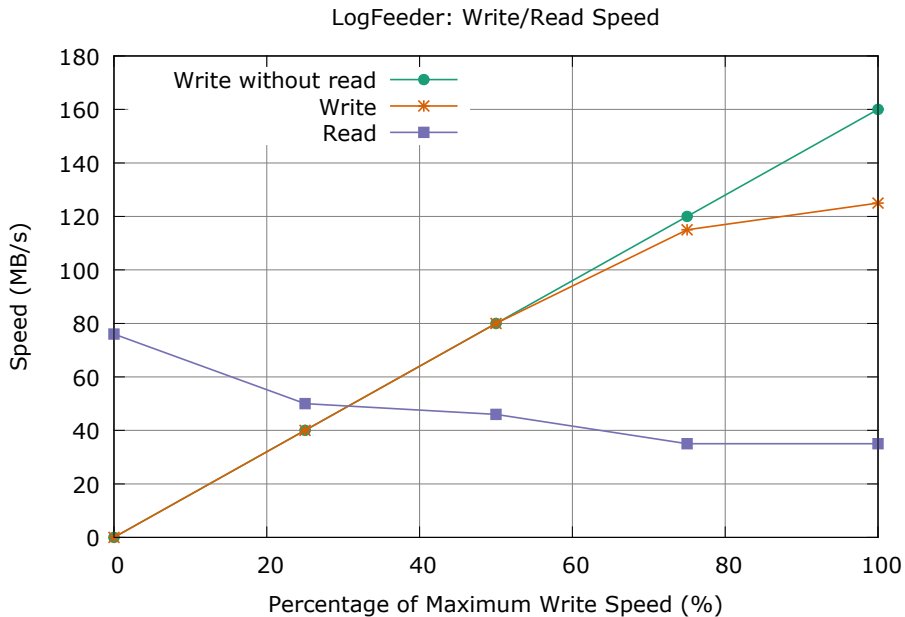


Fig. 11: LogFeeder Write/Read speed

Table 3: Serialization Benchmarks

System	Time (seconds)	Lines per second
Flex	13.9	71,942
AWK	3.5	285,714
C	1.27	787,401

For the latter two programs, a further exercise of coding is necessary in order to consider interdependent and/or optional fields in the log message (if any) but, despite of this, the code is easier to handle for the system programmer, compared to using the Flex and Bison syntaxes.

As shown in the Table 3, the three of them were faster than Logstash (29.5K logs/s/thread) and FluentD (13K log/s/thread) (also Figure 2) at the time of converting logs to JSON. From among the three options tested, Flex is the slowest parser due to the complexity of the rules used to perform pattern-matching. The C ad-hoc parser is more than 10 times faster than the first parser, but in spite of this, the AWK program represents a good balance between flexibility and speed, being 4 times quicker than the lexical parser but also much easier to learn than its counterparts.

4.3.1 LoginsonReceptor

We tested *LoginsonReceptor* and *Elasticsearch* on a machine with an Intel i7 @ 2.4 Ghz, 16GB of RAM and a PCI-e SSD drive. A grand total of 30 million serialized lines were sent to *LoginsonReceptor* on a single *Elasticsearch* node that were indexed at an average speed of 83K lines/s, with Elasticsearch as the bottleneck of the test. Being this an ingest rate 30x lower than the initial 3M log lines received by the system, we consider that the attained throughput is sufficient for a summarized real time statistic which can be used to detect errors, anomalies, or incidents on the system with the relevant messages extracted in previous stages.

4.4 Complete system

We tested the whole system in four servers. *LogFeeder* was run in the same computer as the standalone test in section 4.1, concurrently with 13 database processes, matching the number of hard drives, were executed in two servers like the one used to test the storage speed in section 4.2. A fourth server was used to host *Elasticsearch* and *DBController*.

To match our requirements, *LogFeeder* received 3 million log messages per second and splitted them between the database processes, that were writing to the disks at 50% of their capacity. While *LogFeeder* was inserting logs in the database nodes, we started a long read of the previously stored data. With this test we confirmed that the read operations did not interfere with the write operations. The logs were processed and received correctly in *ElasticSearch*, where they could be visualized in *Kibana*. Figure 12 shows how the test was executed.

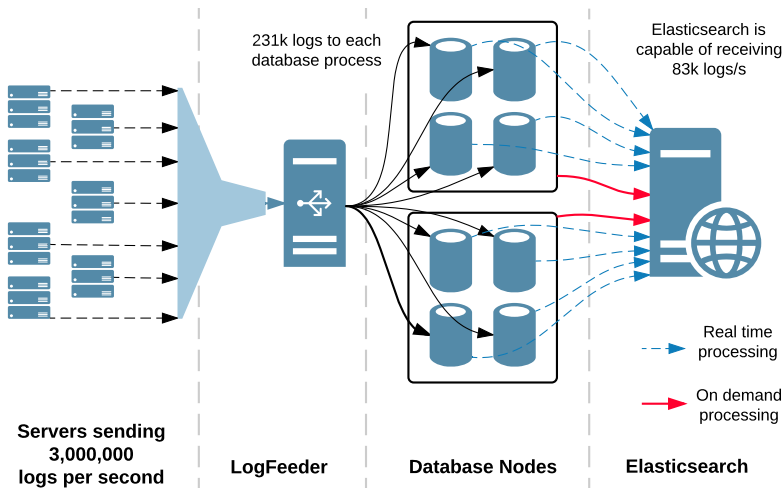


Fig. 12: Complete System

In the last test we executed *LogFeeder* and a database node in the same server used for the standalone test in Section 4.1. Such server has a 10 disk RAID 0 capable of storing data at 10 Gbps. **We assessed that the *Loginson* system was capable of receiving logs, storing them in the RAID and processing them at network rate (also 10 Gbps).** However, in this test the write speed was affected when reading from the RAID, as the upper limit of 50% of the write throughput of the RAID was exceeded, as stated in section 4.2.

5 Conclusions

In this paper we presented a high-performance solution for log collection, storage, processing and visualization called *Loginson*, which tradesoff simplicity with throughput, achieving a remarkable throughput of 3 million logs per second for the most common log size, and up to 10 million logs per second for small logs. Furthermore, all components in the solution are independent, which makes it possible to scale up horizontally for even higher speeds.

As future work, we plan to have a better pre-processing system, as the current one is only capable of transforming the data for *Elasticsearch* consumption. Such system would be capable of analyzing the log data to extract only the relevant information and detect anomalies in a timely manner.

6 Acknowledgments

This work has been partially supported by the *Spanish Ministry of Economy and Competitiveness and the European Regional Development Fund* under the project **TRÁFICA (MINECO/FEDER TEC2015-69417-C2-1-R)**.

We would also like to thank Paloma Dominguez-Tomás for the effort of designing a logo for *Loginson*.

References

1. Chaudhuri, S., Dayal, U., and Narasayya, V. (2011). An overview of business intelligence technology. *Communications of the ACM*, 54(8), 88-98. <https://doi.org/10.1145/1978542.1978562>
2. White, T. (2012). Hadoop: The definitive guide. *O'Reilly Media, Inc.*
3. Shvachko, K., Kuang, H., Radia, S., and Chansler, R. (2010, May). The hadoop distributed file system. In *Mass storage systems and technologies (MSST)*, 2010 IEEE 26th symposium on (pp. 1-10). IEEE. <http://dx.doi.org/10.1109/MSST.2010.5496972>
4. George, L. (2011). HBase: The Definitive Guide: Random Access to Your Planet-Size Data. *O'Reilly Media, Inc.*
5. Ku, R., and Rogoziski, M. (2015). Mastering Elasticsearch. Packt Publishing Ltd.
6. Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M. and Gruber, R. E. (2008). Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2), 4. <https://doi.org/10.1145/1365815.1365816>
7. Peña-López, Ismael and others: *OECD Internet Economy Outlook 2012*, Chapter 4 (2012) <http://dx.doi.org/10.1787/9789264086463-en>
8. Networking, C. V. (2016). *Cisco Global Cloud Index: Forecast and Methodology, 2015-2020*. White paper. <http://www.cisco.com/c/dam/en/us/solutions/collateral/service-provider/global-cloud-index-gci/white-paper-c11-738085.pdf> [Online; accessed 15-Feb-2017].
9. Snyder, J. (2007). Microsoft: datacenter growth defies Moores law. *PC-World*. <http://www.pcworld.com/article/130921/article.html> [Online; accessed 15-Feb-2017].
10. Ashton, K. (2009). That internet of things thing. *RFID Journal*, 22(7), 97-114.
11. Gubbi, J., Buyya, R., Marusic, S., and Palaniswami, M. (2013). Internet of Things (IoT): A vision, architectural elements, and future directions. *Future generation computer systems*, 29(7), 1645-1660. <http://dx.doi.org/10.1016/j.future.2013.01.010>
12. Atzori, L., Iera, A., and Morabito, G. (2010). The internet of things: A survey. *Computer networks*, 54(15), 2787-2805. <http://dx.doi.org/10.1016/j.comnet.2010.05.010>
13. Gerhards, R. (2009). RFC 5424: The *syslog* protocol. <https://tools.ietf.org/html/rfc5424> [Online; accessed 15-Feb-2017].
14. Yang, Fangjin and Tschetter, Eric and Léauté, Xavier and Ray, Nelson and Merlino, Gian and Ganguli, Deep. *Druid: a real-time analytical data store*. Proceedings of the 2014 ACM SIGMOD international conference on Management of data
15. Jacobs, A. (2009). The pathologies of big data. *Communications of the ACM*, 52(8), 36-44. <https://doi.org/10.1145/1536616.1536632>
16. Azarmi, B. (2016). The Big (Data) Problem. In Scalable Big Data Architecture (pp. 1-16). *Apress*. http://dx.doi.org/10.1007/978-1-4842-1326-1_1
17. C. Kalantzis. Revisiting 1 million writes per second. <http://techblog.netflix.com/2014/07/revisiting-1-million-writes-per-second.html>, 2014. [Online; accessed 29-March-2016].
18. BalaBit. How to configure *syslog-ng* pe to cooperate with splunk. <https://www.balabit.com/documents/pdf/syslog-ng-pe-whitepaper-splunk.pdf>, 2015. [Online; accessed 29-March-2016].
19. Scylla. *Scylla vs. cassandra* benchmark. <http://www.scylladb.com/technology/scylla-vs-cassandra-benchmark/>, 2015. [Online; accessed 29-March-2016].
20. J. Kepner, W. Arcand, D. Bestor, B. Bergeron, C. Byun, V. Gadepally, M. Hubbell, P. Michaleas, J. Mullen, A. Prout, et al. Achieving 100,000,000 database inserts per second using accumulo and d4m. In *High Performance Extreme Computing Conference (HPEC), 2014 IEEE*, pages 1-6. IEEE, 2014.
21. Kreps, J., Narkhede, N., and Rao, J. (2011, June). Kafka: A distributed messaging system for log processing. In Proceedings of the NetDB (pp. 1-7).
22. J. Kreps. *Benchmarking Apache Kafka: 2 million writes per second (on three cheap machines)*. <https://engineering.linkedin.com/kafka/benchmarking%2dapache%2dkafka%2d2%2dmillion%2dwrites%2dsecond%2dthree%2dcheap%2dmachines>, 2014. [Online; accessed 29-March-2016].

23. H. Shreedharan. *Using Flume: Flexible, Scalable, and Reliable Data Streaming.* O'Reilly Media, Inc., 2014.
24. E. Silva. *Fluentd: a high performance unified logging layer.* <https://www.linux.com/news/enterprise/high-performance/147-high-performance/847237-fluentd-a-high-performance-unified-logging-layer>, 2015. [Online; accessed 29-March-2016].
25. E. Silva. Unifying events & logs into the cloud. http://events.linuxfoundation.org/sites/events/files/slides/unifying_events.pdf, 2015. [Online; accessed 29-March-2016].
26. FluentD. *Fluentd architecture.* <http://www.fluentd.org/architecture>, 2015. [Online; accessed 29-March-2016].
27. E. POINT. Benchmarking top noSQL databases. http://www.datastax.com/wp-content/themes/datastax-2014-08/files/NoSQL_Benchmarks_EndPoint.pdf, 2015. [Online; accessed 29-March-2016].
28. S. Corp. On-disk microbenchmark. <http://symas.com/mdb/ondisk/>, 2014. [Online; accessed 29-March-2016].
29. Scylla. *Scylla architecture.* <http://www.scylladb.com/technology/architecture/>, 2015. [Online; accessed 29-March-2016].
30. Scylla. *Scylla vs. Cassandra benchmark (cluster).* <http://www.scylladb.com/technology/scylla-vs-cassandra-benchmark-cluster/>, 2015. [Online; accessed 29-March-2016].
31. R. Bodkin. *Mapr releases commercial distributions based on hadoop.* <http://www.infoq.com/news/2011/07/mapr>, 2011. [Online; accessed 29-March-2016].
32. J. Scott. Loading a time series database at 100 million points per second. <https://www.mapr.com/blog/loading-time-series-database-100-million-points-second>, 2014. [Online; accessed 29-March-2016].
33. Splunk. *Splunkit v2.0.2 results and ec2 storage comparisons.* <http://blogs.splunk.com/2013/06/06/splunkit-v2-0-2-results-ec2-storage-comparisons/>, 2013. [Online; accessed 29-March-2016].
34. Splunk. *Splunk sizing and performance: Doing more with more.* <http://blogs.splunk.com/2014/05/07/splunk-sizing-and-performance-doing-more-with-more/>, 2014. [Online; accessed 29-March-2016].
35. Elasticsearch. *Elasticsearch nightly benchmarks* <https://benchmarks.elastic.co/index.html>, 2016 [Online; accessed 16-June-2016]
36. B. SA. Sending messages directly to *Elasticsearch.* <https://www.balabit.com/sites/default/files/documents/syslog-ng-ose-latest-guides/en/syslog-ng-ose-guide-admin/html/configuring-destinations-elasticsearch.html>, 2015. [Online; accessed 29-March-2016].
37. V. VMoreno, P. Santiago del Rio, J. Ramos, D. Muelas, J. Garcia-Dorado, G.-A. F.J., and J. Aracil. *Multi-granular, multi-purpose and multi-Gb/s monitoring on off-the-shelf systems.* <http://onlinelibrary.wiley.com/doi/10.1002/nem.1861/abstract>, 2014. [Online; accessed 29-March-2016].