

**EGI-Engage**

Analysis on Techniques to Manage Big Data on the EGI Accounting System

D3.5

|  |  |
| --- | --- |
| **Date** | 26 January 2016 |
| **Activity** | WP3 |
| **Lead Partner** | STFC |
| **Document Status** | DRAFT |
| **Document Link** | https://documents.egi.eu/document/XXX |

Abstract

This deliverable presents an analysis on techniques to manage big data on the EGI accounting system. The EGI accounting system receives accounting data from sites providing compute, cloud and storage services to EGI. The data is aggregated and the totals across a number of aggregation parameters are sent to an accounting portal for visualisation. The central processing stage for the CPU accounting data takes many hours to complete and operates in a single processor thread. Recent advances in big data tools provide an opportunity to address these limitations: improving the performance and resilience of the central repository. In this document we review the available technologies, discuss how they can be applied to the accounting service and propose a methodology for testing and comparing them.

**COPYRIGHT NOTICE**



This work by Parties of the EGI-Engage Consortium is licensed under a Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/). The EGI-Engage project is co-funded by the European Union Horizon 2020 programme under grant number 654142.

**DELIVERY SLIP**

|  |  |  |  |
| --- | --- | --- | --- |
|  | ***Name*** | ***Partner/Activity*** | ***Date*** |
| **From:** |  |  |  |
| **Moderated by:** |  |  |  |
| **Reviewed by** |  |  |  |
| **Approved by:** |  |  |  |

**DOCUMENT LOG**

|  |  |  |  |
| --- | --- | --- | --- |
| ***Issue*** | ***Date*** | ***Comment*** | ***Author/Partner*** |
| **v0.1** | 2016-01-22 | Document Creation | S. Pullinger / STFC |
| **v0.2** | 2016-01-25 | Updates from S. Pullinger | A. Coveney / STFC |
| **...** |  |  |  |
| **v.n** |  |  |  |

**TERMINOLOGY**

A complete project glossary is provided at the following page: <http://www.egi.eu/about/glossary/>

**Contents**

[1 Introduction 6](#_Toc441571189)

[2 Technologies to manage big data 7](#_Toc441571190)

[2.1 Apache Hadoop 7](#_Toc441571191)

[2.1.1 The Hadoop Distributed File System 7](#_Toc441571192)

[2.1.2 MapReduce 8](#_Toc441571193)

[2.1.3 Enabling Real-Time MySQL to HDFS Integration 8](#_Toc441571194)

[2.1.4 Writing a Hadoop MapReduce program 9](#_Toc441571195)

[2.2 Apache tools that build on the HDFS 9](#_Toc441571196)

[2.2.1 Database like systems 10](#_Toc441571197)

[2.2.2 Non database like systems 10](#_Toc441571198)

[2.2.3 Non Apache tools 12](#_Toc441571199)

[3 Managing big data in the EGI accounting system 13](#_Toc441571200)

[3.1 Why we need to change the APEL technical architecture 13](#_Toc441571201)

[3.2 Metrics to evaluate the different technologies 13](#_Toc441571202)

[3.3 Evaluation of technologies 14](#_Toc441571203)

[3.3.1 Hive 14](#_Toc441571204)

[3.3.2 Pig 15](#_Toc441571205)

[3.3.3 HBase 15](#_Toc441571206)

[3.3.4 Cassandra 15](#_Toc441571207)

[3.3.5 Accumulo 15](#_Toc441571208)

[3.3.6 Flume 15](#_Toc441571209)

[3.3.7 Sqoop 16](#_Toc441571210)

[3.3.8 Cascading 16](#_Toc441571211)

[3.3.9 Storm 16](#_Toc441571212)

[3.3.10 Spark 16](#_Toc441571213)

[3.3.11 ElasticSearch 16](#_Toc441571214)

[4 Roadmap for testing the technologies 17](#_Toc441571215)

[4.1 Resources available 17](#_Toc441571216)

[4.2 Possible configurations of technologies 17](#_Toc441571217)

[4.2.1 Current APEL Configuration 18](#_Toc441571218)

[4.2.2 Hive as Datastore 18](#_Toc441571219)

[4.2.3 Hive as an APEL Backend 19](#_Toc441571220)

[4.2.4 Combined MySQL and Hadoop 19](#_Toc441571221)

[4.2.5 Parallel Stream processing 19](#_Toc441571222)

[5 References 20](#_Toc441571223)

[Appendix I. Summary of big data tools 21](#_Toc441571224)

[Database Tools 21](#_Toc441571225)

[Non Database-like Tools 22](#_Toc441571226)

**Executive summary**

This deliverable presents an analysis on techniques to manage big data on the EGI accounting system. The EGI accounting system receives accounting data from sites providing compute, cloud and storage services to EGI. The data is aggregated and the totals across a number of aggregation parameters are sent to an accounting portal for visualisation. The central processing stage for the CPU accounting data takes many hours to complete and operates in a single processor thread. Recent advances in big data tools provide an opportunity to address these limitations: improving the performance and resilience of the central repository.

The outline of this document is as follows: first we provide a short introduction to the EGI accounting service and the limitations of its current implementation. Then there is an overview of a variety of big data tools that may be useful in addressing these limitations. In the next section, more details are given of the limitations of the current system and of what other motivators there are for change as well as metrics that can be used to evaluate the different tools. A review of the different tools and how suitable they might be follows. Lastly, the resources available for testing are shown and a proposal is made for different configurations of the accounting repository that should be tested.

# Introduction

The EGI Accounting Service receives accounting data from sites providing compute, cloud and storage services to the EGI Federation. Sites send the data to an external message bus. The data is downloaded at a central repository and stored in a MySQL database. The data is aggregated and the totals across a number of aggregation parameters are sent to an accounting portal for visualisation.

The central processing stage for the CPU accounting data takes many hours to complete and operates in a single processor thread. Although the data is backed up regularly, there is no sharding of the data across multiple hosts to provide greater resilience and processing power. Recent advances in Big Data tools provide an opportunity to address these limitations: improving the performance and resilience of the central repository. In this document we review the available technologies, discuss how they can be applied to the accounting service and propose a methodology for testing and comparing them. A future addendum will address the results of these tests and comparisons.

# Technologies to manage big data

## Apache Hadoop

The Apache Hadoop software library is a framework that allows for the distributed processing of large data sets across clusters of computers. It is designed to scale up from single servers to thousands of machines, each offering local computation and storage[[1]](#footnote-1). The two main parts of Apache Hadoop are the Hadoop Distributed File System (HDFS) for storage and the MapReduce programming model.

The HDFS splits files into large blocks and distributes them across nodes in a cluster. To process data, MapReduce transfers packaged code to nodes in order to process the data on that node, taking advantage of data locality[[2]](#footnote-2).

Hadoop can be downloaded from their website[[3]](#footnote-3).

### The Hadoop Distributed File System

Many of the tools covered in this report build upon the HDFS. Therefore the technical benefits and limitations of the HDFS must be understood.

The HDFS is a Java-based file system that provides scalable and reliable data storage. It was designed to span large clusters of commodity servers and has demonstrated production scalability of up to 200 PB[[4]](#footnote-4). The entire size of the APEL database is approximately 0.5 TB, so this should be more than sufficient even allowing for replication and possible expansion of the data once loaded into the HDFS.

The HDFS stores metadata and application data separately. As in other distributed file systems, like PVFS[[5]](#footnote-5), Lustre[[6]](#footnote-6), and Google File System, the HDFS stores metadata on a dedicated server, called the NameNode. Application data are stored on other servers called DataNodes[[7]](#footnote-7). All servers are fully connected and communicate with each other.

Data on DataNodes are broken down into smaller blocks. These blocks are then distributed and replicated throughout the cluster by using TCP-based protocols. Unlike Lustre and PVFS, the DataNodes in the HDFS do not rely on data protection mechanisms such as RAID to make the data contained durable. Instead, the HDFS relies on the file content being replicated on multiple DataNodes for reliability. The default replication setting is to store two additional copies of each block to increase fault tolerance, but this can be changed globally or per file[[8]](#footnote-8). This replication has the added advantage that there are more opportunities for locating computation near the needed data[[9]](#footnote-9).

Version 2.7.1 of the HDFS introduced transparent, end-to-end encryption. Once configured, data read from and written to special HDFS directories is transparently encrypted and decrypted without requiring changes to user application code. This encryption is also end-to-end, which means the data can only be encrypted and decrypted by the client. The HDFS never stores or has access to unencrypted data or unencrypted data encryption keys. This satisfies two typical requirements for encryption: at-rest encryption (meaning data on persistent media, such as a disk) as well as in-transit encryption (e.g. when data is travelling over the network[[10]](#footnote-10)).

### MapReduce

The MapReduce programming model is composed of Map and Reduce procedures. A Map method performs filtering and sorting; then a Reduce method performs a summary operation. An example MapReduce program could contain a Map method for sorting students by first name into queues, one queue for each name; then a Reduce method for counting the number of students in each queue, yielding name frequencies.

The "MapReduce System" manages all communications and data transfers between the various parts of the system, and providing for redundancy and fault tolerance.

### Enabling Real-Time MySQL to HDFS Integration

Hadoop Applier reads from the MySQL binary log and inserts data into the HDFS in real time, applying the events as they happen on the MySQL server[[11]](#footnote-11).

The Hadoop Applier uses an API provided by *libhdfs*, a C library to manipulate files in the HDFS which comes precompiled with Hadoop distributions. Databases are mapped as separate directories, with their tables mapped as sub-directories. A tool, such as Scoop[[12]](#footnote-12), may be needed to transfer the existing APEL data[[13]](#footnote-13).

MySQL Applier can be downloaded from the MySQL website; however it comes with a warning that it is “not fit for production” and is “provided solely for testing purposes”[[14]](#footnote-14).



Figure - MySQL to HDFS Integration (<https://dev.mysql.com/tech-resources/articles/mysql-hadoop-applier.html>)

### Writing a Hadoop MapReduce program

Hadoop MapReduce programs are typically written in Java. However, Apache have released a JAR file that takes non Java programs as inputs and uses them as the Map and Reduce methods. These programs need to take their input from STDIN and put their output to STDOUT. Using this, MapReduce operations using the HDFS can be written in Python[[15]](#footnote-15) or any language capable of reading from STDIN and writing to STDOUT.

## Apache tools that build on the HDFS

As well as the MySQL Applier and the ability to write MapReduce programs in any language, the Apache Foundation provide many tools that build on the HDFS, providing additional functionality. Many of these tools aim to conceal the complexity of the MapReduce model, by being “database-like” and providing a “SQL-like” interface to the stored data. A number of these tools are described in the remainder of this section.

### Database like systems

#### Hive

Apache Hive is a [data warehouse](https://en.wikipedia.org/wiki/Data_warehouse) infrastructure built on top of [Hadoop](https://en.wikipedia.org/wiki/Hadoop) for providing data summarization, query, and analysisR1 of the data stored in the HDFS via an SQL-like interface[[16]](#footnote-16). It has been designed to perform full-table scans across petabyte-scale data sets.

#### Pig

Apache Pig[[17]](#footnote-17) is a high-level platform for creating [MapReduce](https://en.wikipedia.org/wiki/MapReduce) programs used with [Hadoop](https://en.wikipedia.org/wiki/Hadoop). The language for this platform is called Pig Latin. Pig Latin abstracts the programming from the [Java](https://en.wikipedia.org/wiki/Java_%28programming_language%29) MapReduce idiom into a notation which makes MapReduce programming high level, similar to that of [SQL](https://en.wikipedia.org/wiki/SQL) for [RDBMS](https://en.wikipedia.org/wiki/RDBMS) systems. Pig Latin can be extended using User Defined Functions which the user can write in Java, Python, JavaScript, R[uby](https://en.wikipedia.org/wiki/Ruby_%28programming_language%29), or Groovy and then call directly from a Pig Latin program.

#### HBase

Apache HBase is the Hadoop database, a distributed, scalable, big data store providing random, real-time read/write access to data. Apache HBase is an open-source, distributed, versioned, non-relational database modelled after Google's Bigtable[[18]](#footnote-18). Just as Bigtable uses the distributed data storage provided by the Google File System, Apache HBase provides Bigtable-like capabilities on top of Hadoop and HDFS.

#### Cassandra

Apache Cassandra is a NoSQL database providing linear scalability and fault-tolerance on commodity hardware. Cassandra supports replicating across multiple datacentres, providing lower latency for users and the peace of mind of knowing that you can survive regional outages.

### Non database like systems

#### Samza

Samza is a distributed stream processing framework. It uses Apache Kafka[[19]](#footnote-19) for messaging, and Apache Hadoop to provide fault tolerance, processor isolation, security, and resource management[[20]](#footnote-20). The processing that Samza enables is often called stream processing. The expected time to get output from a stream process is usually much lower than batch processing, frequently in the sub-second range[[21]](#footnote-21).

#### Spark

Spark is built on the concept of *distributed datasets*, which contain arbitrary Java or Python objects. You create a dataset from external data, and then apply parallel operations to it. There are two types of operations: *transformations*, which define a new dataset based on previous ones, and *actions*, which kick off a job to execute on a cluster[[22]](#footnote-22).

#### Camel

Apache Camel uses URIs to work directly with any kind of transport or messaging model, such as HTTP, as well as pluggable components and data format options. Apache Camel is a small library with minimal dependencies for easy embedding in any Java application[[23]](#footnote-23).

#### Accumulo

Accumulo is a sparse, distributed, sorted, multi-dimensional map[[24]](#footnote-24) based on Google's BigTable[[25]](#footnote-25) design and is built on top of Apache Hadoop, Zookeeper[[26]](#footnote-26), and Thrift[[27]](#footnote-27). It is designed to scale to trillions of records and tens of petabytes. Apache Accumulo features a few novel improvements on the BigTable design in the form of cell-based access control and a server-side programming mechanism that can modify key/value pairs at various points in the data management process.

#### Flume

Apache Flume is a distributed, reliable, and available service for efficiently collecting, aggregating, and moving large amounts of streaming data into the HDFS. It has a simple and flexible architecture based on streaming data flows; and is robust and fault tolerant with tuneable reliability mechanisms for failover and recovery[[28]](#footnote-28). It uses a simple extensible data model that allows for online analytic application[[29]](#footnote-29).

#### Sqoop

Apache Sqoop is a tool designed for efficiently transferring bulk data between Hadoop and structured datastores such as relational databases. Relational databases are examples of structured data sources with well defined-schema for the data they store. Cassandra, Hbase are examples of semi-structured data sources and HDFS is an example of unstructured data source that Sqoop can support.

#### Cascading

Cascading is a software [abstraction layer](https://en.wikipedia.org/wiki/Abstraction_layer) for [Apache Hadoop](https://en.wikipedia.org/wiki/Apache_Hadoop). Cascading is used to create and execute complex data processing workflows on a Hadoop cluster using any [JVM](https://en.wikipedia.org/wiki/JVM)-based language, hiding the underlying complexity of MapReduce jobs. Cascading also comes with an extension called Lingual. Lingual simplifies application development and integration by providing an ANSI SQL interface for Apache Hadoop. This interface can connect existing SQL codes with Hadoop and accelerate application development with Hadoop[[30]](#footnote-30).

#### Storm

Storm reliably processes unbounded streams of data, doing for real-time processing what Hadoop did for batch processing. Storm can be used with any programming language. It is scalable, fault-tolerant, guarantees your data will be processed, and is easy to set up and operate. Storm uses Thrift, an interface definition language and binary communication protocol[[31]](#footnote-31) that is used to define and create services for numerous languages. Apache Avro also does a similar job, but does not require running a code-generation program when a schema changes[[32]](#footnote-32).

### Non Apache tools

#### ElasticSearch

Elasticsearch for Apache Hadoop (ES-Hadoop) is a two-way connector that solves a top wishlist item for any Hadoop user out there: real-time search. While the Hadoop ecosystem offers a multitude of analytics capabilities, it falls short with fast search. ES-Hadoop bridges that gap, letting you leverage the best of both worlds: Hadoop's big data analytics and the real-time search of Elasticsearch[[33]](#footnote-33).

# Managing big data in the EGI accounting system

## Why we need to change the APEL technical architecture

As mentioned in the introduction, the accounting system pulls sites’ data from a message broker and stores it in a central repository. The data is aggregated over a number of fields to create summaries. These summary totals are then sent on to the accounting portal for display to the users.

The summarising process runs as a query against a large MySQL database of around 500 gigabytes. The process runs in a single thread and takes 8 to 10 hours to complete during which time no data is loaded into the database. It is therefore only practical to run this once per day.

One motivator for investigating alternative tools is to reduce this latency in the system so that summaries arrive at the portal with a shorter delay. Another motivator is the possibility to use multiple cores for the processing thus making better use of the hardware.

Finally, the accounting team have several new types of accounting in development (storage, data-sets and GPGPUs) which, should they all go into production, would increase the volume of data sent to the central accounting repository. Combined with the expected increase in volume of accounting data as more researchers make use of the high performance computing resources, the accounting service will expect to receive greater and greater volumes of data for processing. We therefore need to be prepared to handle larger amounts of data.

## Metrics to evaluate the different technologies

In the context of the accounting system, there are a number of metrics that should be measured for candidate technologies.

An important part of the accounting system is the processing that it does to the data that it receives before the data is sent on to the accounting portal for visualisation. This aggregation or “summarising” currently takes a long time, in the order of hours, and means that is impractical to perform the summarising more than once a day. Ideally this process would be much quicker, allowing data to be sent to the portal on a much more regular basis or even streamed continuously if the technology allows it. This means that the time it takes to process the data is an important metric, measured as the time it takes to perform the summarising process. This can then be compared to the duration of the current summarising process.

As some of the proposed solutions would involve transferring data between the current MySQL database and the interface of a candidate technology, the time it takes to transfer this data may be relevant. Some technologies may allow for incremental transfers once the initial transfer is done and this would reduce the impact of the transfer time, but it should still be measured in case it needs to factor into the processing time.

The MySQL database at the heart of the accounting repository currently takes up about half a terabyte of storage space. When this is converted to another format, such as HDFS, this may increase the storage requirements for the repository. This would be due to the change in data format as well as replication in the case of a solution that uses a cluster.

Different technologies may need different resources, such as VMs or physical hosts, and this should be noted for comparison between the technologies.

An important part of deploying a new technology is supporting it. This has an impact on the level of effort required to use a technology and in some cases it may not be possible to use the technology without some minimum expertise. With this in mind there are at least a couple of areas that should be evaluated when testing a candidate technology. The first is how easy it is to install and configure the technology; if there is clear and comprehensive documentation available, this can help. The second is how easy it is to use the technology; an important aspect of that is if there is an easy way to query the data once it is imported into the tool. A familiar interface, such as a SQL-like query language, would aid this.

## Evaluation of technologies

For one of the Big Data tools listed above to be considered a viable avenue of investigation it must meet certain criteria. First, as APEL is built in Python, the Big Data tool should ideally have a Python API. Secondly, the chosen tool should have a SQL-like query interface for debugging purposes. Thirdly, the order the data processed in must be the same as the data received, as APEL requires this to produce correct summaries. Finally, whether the chosen tool could allow stream processing, the update of summaries on the fly is important.

The Requirement of a Python API excludes Camel and Samza.

A table summarising the tools covered in this section can be found in Appendix I.

### Hive

Hive is written in Java and has a SQL-like (HiveQL) interface for querying data in the HDFS. Such queries have traditionally had high latency, and even small queries could take some time to run because they were transformed into map-reduce jobs and submitted to the cluster to be run in batch mode. Newer versions of Hive have improved this performance by using the Tez execution framework, by using the Optimized Row Columnar (ORC) file format or by enabling cost-based query optimization[[34]](#footnote-34). From the top down, Hive looks much like any other relational database.

###  Pig

Pig is a query language for the HDFS that is not SQL-like. It supports loading directories of data from the HDFS and working with it to get it into the form for querying. Hive is considered friendlier and more familiar to users who are used to using SQL for querying data[[35]](#footnote-35).

### HBase

HBase has a Python API and, unlike Hive, allows write operations into existing tables and the HDFS. It’s query language is not SQL-like and appears relatively simple. However an extension, Apache Phoenix, provides an SQL layer allowing HBase to be a replacement for a MySQL database. HBase is designed to handle large volumes of data (over a few hundred gigabytes) and large number of concurrent clients. As HBase is column oriented, aggregation operations might be quicker than row based databases.

### Cassandra

Cassandra's data model offers the convenience of column indexes with the performance of log-structured updates, strong support for denormalization and materialized views, and powerful built-in caching[[36]](#footnote-36). It has a Python API[[37]](#footnote-37) and an SQL -like query language, which supports inserts and updates. Newer versions support aggregation operations[[38]](#footnote-38), but this may have to be on columns which are part of the primary key.

Cassandra uses its own version of the HDFS, the Cassandra File System (CFS). The CFS, in contrast to the master-slave architecture of HDFS, CFS is based on Cassandra, so the implementation is peer-to-peer and “masterless.” A user is able to create a cluster that seamlessly stores real-time data in Cassandra, performs analytic operations on that same data, and also handles enterprise search operations[[39]](#footnote-39).

### Accumulo

Accumulo does not currently support a SQL-like query language and will not get one any time soon[[40]](#footnote-40). It supports reading and writing into the HDFS and a Python wrapper for Accumulo exists[[41]](#footnote-41).

### Flume

Flume is not a method of querying the data already in the HDFS but rather a method of getting new data into the HDFS. As such, it has not SQL-like query language and would have to be combined with another tool, such as Hive or HBase. It does not provided any ordering guarantees, it only guarantees a single message will be processed exactly once.

### Sqoop

Sqoop allows for transfers from relational databases such as MySQL to Hadoop data stores such as HDFS and Hive and vice versa. The tool could be useful for one-off transfers to create a corpus of test data but also for integration in a system where data exists in a database and Hadoop tool in parallel – allowing for both tools to be used to their best advantage.

### Cascading

Cascading has both a Python API and a SQL interface via Lingual[[42]](#footnote-42). It can query existing data and insert data into the HDFS. One benefit of Cascading is the ability to use Lingual to insert query data using actual SQL, thus requiring less change to the APEL software.

### Storm

Storm has many use cases: real time analytics, online machine learning, continuous computation, distributed RPC, ETL, and more. Storm is fast: a benchmark clocked it at over a million tuples processed per second per node. Storm integrates with the queueing and database technologies you already use. A Storm topology consumes streams of data and processes those streams in arbitrarily complex ways, repartitioning the streams between each stage of the computation however needed. It has a Python wrapper[[43]](#footnote-43) and serves as a means to get data into the HDFS and process it en-route.

### Spark

Spark has both a Python API[[44]](#footnote-44), including examples for non-streaming spark, and a SQL-like for trouble shooting. Spark can also interact either directly with the HDFS, or with other Apache tools, such as Hive[[45]](#footnote-45). It can be used to write data to the HDFS[[46]](#footnote-46).

### ElasticSearch

ElasticSearch is a searching method that can be placed on top of the HDFS. It has a Python interface but not a SQL-like query language. ElasticSearch itself cannot input new data into the HDFS, but it does can become aware of new data when it is inserted into the HDFS via other methods, such as cURL.

# Roadmap for testing the technologies

## Resources available

There are a number of resources available to the APEL project within STFC that can help enable the testing and evaluation of the different big data technologies.

The first is a self-service cloud which provides an internal IaaS cloud resource for STFC users. The cloud is based on OpenNebula for the virtualisation and CEPH for the storage. For the hardware, there are:

* 28x Dell R420 for the hypervisors
* 30x Dell R520 for the storage nodes

This gives a total of 3.5TB of memory, 896 processing cores and a storage capacity of approximately 750TB. A variety of OS images are available, mainly consisting of different versions of Scientific Linux and Ubuntu, although other images can be requested if there is a use-case for them. This cloud allows developers to quickly requisition resources for testing and experimentation.

The second is a small Hadoop cluster of ten nodes that has been used for evaluating cloud storage and compute, and for some development work.

Lastly, the existing accounting data is an excellent resource for creating test data from. To create a corpus of test data, a tool like Apache Sqoop (<http://sqoop.apache.org/>) can be used to extract the data from the current relational database management system, MySQL, into the Hadoop Distributed File System, transform the data in Hadoop MapReduce, and then export the data back into MySQL.

## Possible configurations of technologies

Figure 2 shows the broad categories of technologies and their topology in the APEL system. A message broker operates externally receiving data from sites. The APEL server pulls this data and stores it in a database. Periodically, the data is processed to create summaries. The summaries are then exported from the database and sent on via the same message broker. The database receives ad-hoc queries for the purpose of investigating problems with sites’ data.



Figure - APEL technology topology

### Current APEL Configuration

The current APEL configuration consists of:

* Ingest: APEL SSM receiver and APEL DB Loader

The APEL SSM software pulls messages from the message bus and writes them to the file system. The APEL DB Loader reads the files and loads them into the database.

* Data Store: MySQL
* Export: APEL DB Unloader and APEL SSM Sender

The APEL DB Unloader and SSM Sender operate like the loader and receiver in reverse.

* Ad-hoc queries: MySQL Client
* Aggregator: APEL Summariser

The APEL Summariser runs a one single-threaded query against the MySQL database to produce the summaries. The query takes 8 to 10 hours to complete.

###  Hive as Datastore

* Ingest: Apache Flume

Flume can pull data from a message bus and write directly to Hive.

* Data Store: Apache Hive
* Export: Scripted HiveQL query

Flume does not extract data from Hive. However, it is possible to extract data from Hive to csv using HiveQL. A small script could convert this to the correct format for the APEL SSM software to send to the accounting portal.

* Ad-hoc queries: Apache Hive Beeline

Beeline is the interactive shell for the Hive Server.

* Aggregator: pre-computed Hive query

### Hive as an APEL Backend

The APEL software currently has a MySQL backend. The software could be extended with a new Hive backend replacing the SQL calls with calls to HiveQL.

### Combined MySQL and Hadoop

* Ingest: APEL SSM receiver and APEL DBLoader
* Data Store: MySQL
* Export: APEL DBUnloader and SSM Sender
* Ad-hoc queries: MySQL
* Aggregator: Sqoop could be used to export the data from MySQL to HDFS for later processing by a number of Hadoop Map/Reduce or other technologies built on it. Sqoop could also be used to return the summaries to MySQL. This provides the advantage of multi-core processing without the difficulty of making larger changes to the existing service architecture. The disadvantage of this approach is the overhead of data transfers – though incremental updates could reduce this. Further there could be potential for the 2 copies of the data to go out of sync.

### Parallel Stream processing

The stream processing frameworks provide the opportunity to split the data store and aggregator functions apart so that the aggregation is performed continuously. In this configuration Apache Flume is used to duplicate the data pulled from the message broker: with one copy going to a data store – such as MySQL or Hive – for ad-hoc queries and backup; and the other copy going to a stream processing framework – such as Samza, Spark or Storm – to provide near realtime generation of summaries.

# References

|  |  |
| --- | --- |
| ***No*** | ***Description / Link*** |
| R1 | Venner, Jason (2009). Pro Hadoop. Apress. ISBN 978-1-4302-1942-2 |

1. Summary of big data tools

## Database Tools

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| ***Tool*** | ***Tool language*** | ***API language*** | ***SQL-like interface?*** | ***Data entry into HDFS?*** | ***Searching into HDFS?*** | ***Ordering preserved?*** | ***Batch or streaming?*** | ***Notes*** |
| Hive | Java | Java, Python, PHP | Yes | Yes | Yes | N/A | Batch | Hive only has “INSERT...VALUES” in version 0.14.Python interface. |
| Pig | Pig Latin | Jave, Python, JS, Ruby, Groovy | No (Pig Latin) | No | Yes | N/A | Batch | No append to existing files. |
| HBase | Java | Java, Python | No, see notes | Yes | Yes | N/A | Batch | SQL like query language can be added on top of HBase[[47]](#footnote-47). |
| Cassandra | Java | Java, Python | Yes | Yes, see notes | Yes, see notes | N/A | Batch | Cassandra replaces the HDFS with its own file system[[48]](#footnote-48). |

## Non Database-like Tools

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| ***Tool*** | ***Tool language*** | ***API language*** | ***SQL-like interface?*** | ***Data entry into HDFS?*** | ***Searching into HDFS?*** | ***Ordering preserved?*** | ***Batch or streaming?*** | ***Notes*** |
| Samza | Java, Scala | Java | No[[49]](#footnote-49) | No | Yes | Yes, per stream[[50]](#footnote-50) | Streaming | Stateful stream processing, could possibly compute aggregate on the fly |
| Spark | Java, Scala, Python, R | Java, Scala, Python, R[[51]](#footnote-51) | Yes | Yes | Yes | Yes, per stream | Streaming | Spark/Hadoop cluster already set up |
| Camel | Java | Mostly Java[[52]](#footnote-52), see notes | Yes[[53]](#footnote-53) | Yes[[54]](#footnote-54) | Yes | No[[55]](#footnote-55) | Both | Expressions and Predicates can be written in Python |
| ElasticSearch | Java | Java, Python | No | No, see notes | Yes | N/A | Batch | Data can be inserted using the Linux tool cURL. |
| Accumulo | Java | Java, Python | No | Yes | Yes | N/A | Batch |  |
| Flume | Java | Java, Python, Scala | No | Yes | No | No[[56]](#footnote-56) | Streaming | not a method of querying the data. but a method of putting it into query methods. |
| Cascading | Java | Java, Python, SQL, others[[57]](#footnote-57) | Yes  | Yes | Yes | N/A | Batch | SQL API, Lingual[[58]](#footnote-58) |
| Storm | Clojure, Java | Java, Python | No | Yes | No | Yes, but per stream only[[59]](#footnote-59) | Streaming |  |

1. <https://hadoop.apache.org/> [↑](#footnote-ref-1)
2. <http://www-01.ibm.com/software/data/infosphere/hadoop/hdfs/> [↑](#footnote-ref-2)
3. <https://hadoop.apache.org/releases.html> [↑](#footnote-ref-3)
4. <http://hortonworks.com/hadoop/hdfs/> [↑](#footnote-ref-4)
5. <http://www.pvfs.org/> [↑](#footnote-ref-5)
6. <http://www.lustre.org> [↑](#footnote-ref-6)
7. <http://www.aosabook.org/en/hdfs.html> [↑](#footnote-ref-7)
8. <https://www-01.ibm.com/software/data/infosphere/hadoop/hdfs/> [↑](#footnote-ref-8)
9. <http://www.aosabook.org/en/hdfs.html> [↑](#footnote-ref-9)
10. <https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/TransparentEncryption.html> [↑](#footnote-ref-10)
11. <http://innovating-technology.blogspot.co.uk/2013/04/mysql-hadoop-applier-part-1.html> [↑](#footnote-ref-11)
12. <http://sqoop.apache.org/> [↑](#footnote-ref-12)
13. <https://www.percona.com/blog/2013/07/11/mysql-and-hadoop/> [↑](#footnote-ref-13)
14. <http://labs.mysql.com/> [↑](#footnote-ref-14)
15. <http://www.michael-noll.com/tutorials/writing-an-hadoop-mapreduce-program-in-python/> [↑](#footnote-ref-15)
16. <http://hortonworks.com/hadoop-tutorial/how-to-process-data-with-apache-hive/> [↑](#footnote-ref-16)
17. <http://pig.apache.org/> [↑](#footnote-ref-17)
18. <http://static.googleusercontent.com/media/research.google.com/en//archive/bigtable-osdi06.pdf> [↑](#footnote-ref-18)
19. <http://kafka.apache.org/> [↑](#footnote-ref-19)
20. <http://samza.apache.org/> [↑](#footnote-ref-20)
21. <http://samza.incubator.apache.org/learn/documentation/0.7.0/container/state-management.html> [↑](#footnote-ref-21)
22. <http://spark.apache.org/examples.html> [↑](#footnote-ref-22)
23. <http://camel.apache.org/> [↑](#footnote-ref-23)
24. <https://www.youtube.com/watch?v=nk4yhqHjxOU> [↑](#footnote-ref-24)
25. <http://static.googleusercontent.com/media/research.google.com/en//archive/bigtable-osdi06.pdf> [↑](#footnote-ref-25)
26. <http://zookeeper.apache.org/> [↑](#footnote-ref-26)
27. <http://thrift.apache.org/> [↑](#footnote-ref-27)
28. <https://kzhendev.wordpress.com/2014/04/06/apache-flume-get-logs-out-of-rabbitmq-and-into-hdfs/> [↑](#footnote-ref-28)
29. <https://flume.apache.org/> [↑](#footnote-ref-29)
30. <http://www.cascading.org/projects/lingual/> [↑](#footnote-ref-30)
31. <http://jnb.ociweb.com/jnb/jnbJun2009.html> [↑](#footnote-ref-31)
32. <https://avro.apache.org/> [↑](#footnote-ref-32)
33. <https://www.elastic.co/products/hadoop> [↑](#footnote-ref-33)
34. <http://hortonworks.com/hadoop-tutorial/real-time-data-ingestion-hbase-hive-using-storm-bolt/> [↑](#footnote-ref-34)
35. <http://hortonworks.com/hadoop-tutorial/how-to-process-data-with-apache-hive/> [↑](#footnote-ref-35)
36. <http://cassandra.apache.org/> [↑](#footnote-ref-36)
37. <https://github.com/datastax/python-driver> [↑](#footnote-ref-37)
38. <https://issues.apache.org/jira/browse/CASSANDRA-4914> [↑](#footnote-ref-38)
39. <https://www.datastax.com/wp-content/uploads/2012/09/WP-DataStax-HDFSvsCFS.pdf> [↑](#footnote-ref-39)
40. <https://issues.apache.org/jira/browse/ACCUMULO-1014> [↑](#footnote-ref-40)
41. <https://github.com/accumulo/pyaccumulo> [↑](#footnote-ref-41)
42. <http://www.cascading.org/projects/lingual/> [↑](#footnote-ref-42)
43. <http://github.com/twitter/pycascading/wiki> [↑](#footnote-ref-43)
44. <http://spark.apache.org/examples.html> [↑](#footnote-ref-44)
45. <http://spark.apache.org/sql/> [↑](#footnote-ref-45)
46. <https://spark.apache.org/docs/1.1.1/api/java/org/apache/spark/rdd/RDD.html> [↑](#footnote-ref-46)
47. <https://phoenix.apache.org> [↑](#footnote-ref-47)
48. <https://www.datastax.com/wp-content/uploads/2012/09/WP-DataStax-HDFSvsCFS.pdf> [↑](#footnote-ref-48)
49. <https://samza.apache.org/learn/documentation/0.7.0/container/state-management.html> [↑](#footnote-ref-49)
50. <http://samza.apache.org/learn/documentation/0.7.0/comparisons/spark-streaming.html> [↑](#footnote-ref-50)
51. <http://spark.apache.org/docs/latest/api.html> [↑](#footnote-ref-51)
52. <http://camel.apache.org/scripting-languages.html> [↑](#footnote-ref-52)
53. <http://camel.apache.org/sql-component.html> [↑](#footnote-ref-53)
54. <http://camel.apache.org/hdfs.html> [↑](#footnote-ref-54)
55. <http://camel.apache.org/parallel-processing-and-ordering.html> [↑](#footnote-ref-55)
56. <https://github.com/cloudera/flume/wiki/FAQ> [↑](#footnote-ref-56)
57. <http://www.cascading.org/extensions/> [↑](#footnote-ref-57)
58. <http://www.cascading.org/projects/lingual/> [↑](#footnote-ref-58)
59. <https://samza.apache.org/learn/documentation/0.10/comparisons/storm.html> [↑](#footnote-ref-59)