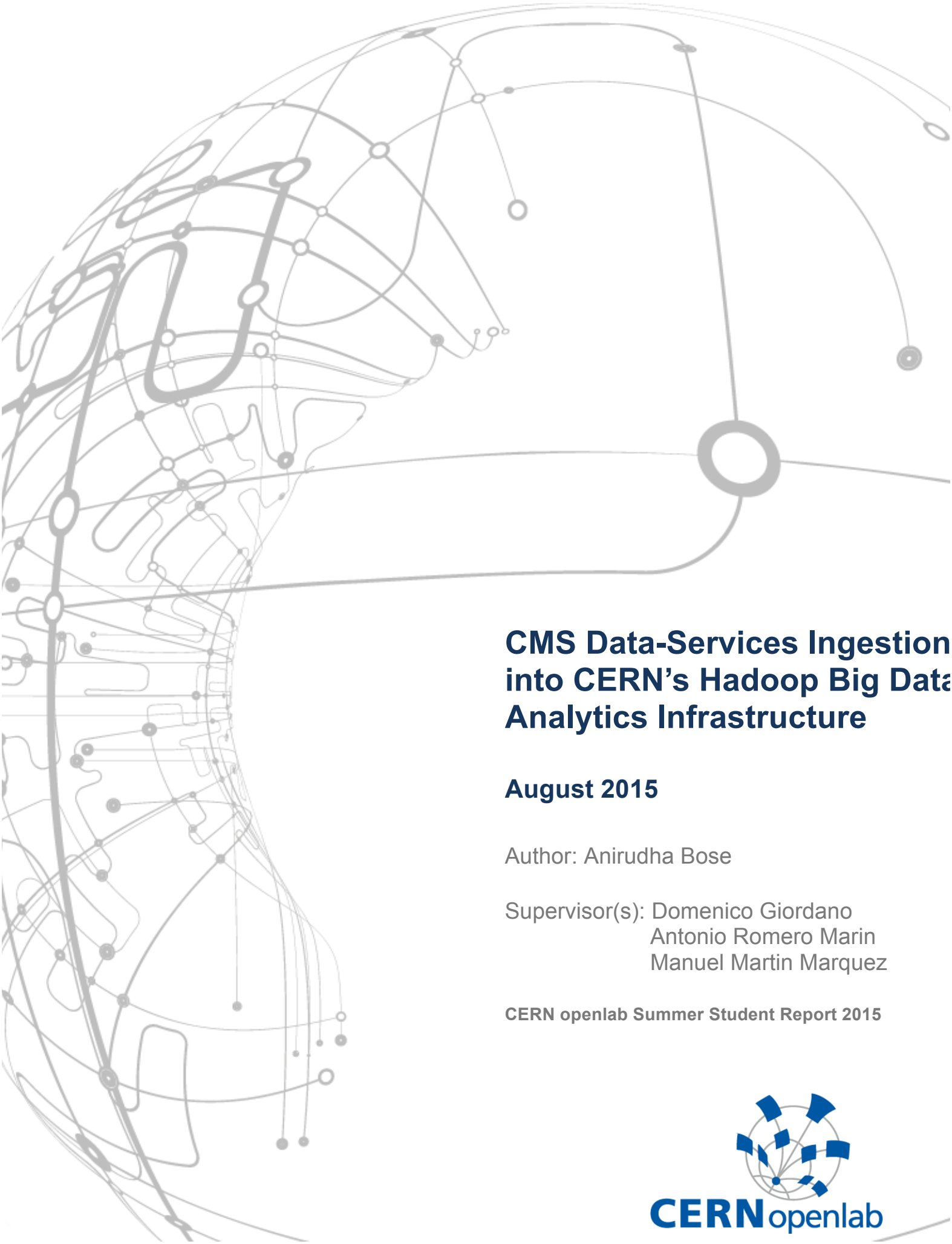


CMS Data-Services Ingestion into CERN's Hadoop Big Data Analytics Infrastructure

August 2015

Author: Anirudha Bose

Supervisor(s): Domenico Giordano
Antonio Romero Marin
Manuel Martin Marquez

CERN openlab Summer Student Report 2015

CERN openlab

Abstract

This document introduces a new data ingestion framework called HLoader, built around Apache Sqoop to perform data ingestion jobs between RDBMS and Hadoop Distributed File System (HDFS). The HLoader framework deployed as a service inside CERN will be used for CMS Data Popularity ingestion into Hadoop clusters. HLoader could also be used for similar use cases like CMS and ATLAS Job Monitoring, ACCLOG databases, etc. The first part of the report describes the architectural details of HLoader, giving some background information about Apache Sqoop. The rest of the report focuses on the HLoader programming API, and is meant to be an entry point for developers describing how HLoader works, and possible directions of extending the framework in future.

Table of Contents

1 Introduction

The processing of enormous amount of data is a fundamental challenge for research in High Energy Physics (HEP) at CERN. Various databases and storage systems are needed to store the large amounts of control, operation and monitoring data in order to run the LHC accelerator and its experiments. The Hadoop ecosystem developed by the Apache Software Foundation provides a powerful set of tools for storing and analysing data in petabyte scale. There is an increasing interest in Hadoop based solutions at CERN in many areas including experiments, accelerator controls, archives, etc. The IT-DB group, in collaboration with IT-DSS is rolling out a Hadoop production platform for big data systems at CERN, integrating it with the current online RDBMS systems.

Most of the data services at CERN are based on relational data storage systems, which are centrally managed by CASTOR. For big data analysis on the data stored in these relational stores, it is important to transfer it to Hadoop-friendly storage systems, like the Hadoop Distributed File System (HDFS), Hive or HBase tables. This data ingestion is particularly complex process for an organization like CERN, and mechanical due to the volume, and frequency of incoming data.

The goal of the project is to automate the process of data ingestion between RDBMS systems and Hadoop clusters by building a framework that acts as a data pipeline, and interfaces with both systems via dedicated tools like Apache Sqoop, and applies all necessary actions in order to deliver ready-to-read data on Hadoop file system for high-end frameworks like Impala or Spark

2 Apache Sqoop

Sqoop is a Top-Level Apache Software Foundation project designed to efficiently transfer bulk data between structured data stores such as relational databases and Hadoop Distributed File System (HDFS), Hive or HBase. Sqoop automates most of the data ingestion process, relying on the database to describe the schema for the data to be imported. Data ingestion using Sqoop is much faster than conventional “query-and-dump”, since it performs the transfers in parallel using Hadoop MapReduce.

A by-product of the import process is a generated Java class, each instance of which can encapsulate one row of the imported table. This class is not only used during the import process by Sqoop, but the Java source code for this class is also provided to the user as well, for use in subsequent MapReduce processing of the data. This allows developers to quickly write MapReduce programs that use the HDFS-stored records in the processing pipeline.

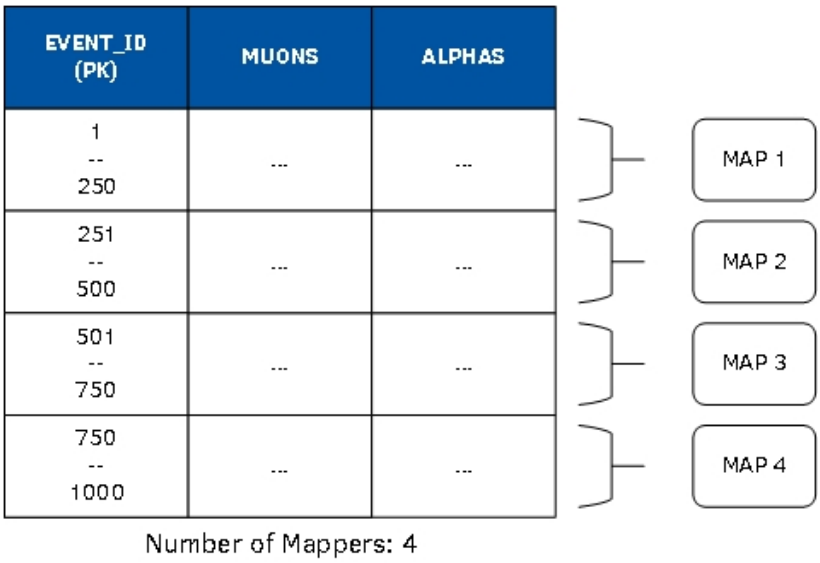


Figure 1: MapReduce job with Sqoop

Sqoop supports data imports to multiple output formats like plain text (default), SequenceFiles, Avro Data files, and Parquet files. While this data on HDFS is uncompressed by default, Sqoop can be set to compress the data using available Hadoop codecs like GZip (default), BZip2, Snappy in order to reduce the overall disk utilization.

2.1 Sqoop Connectors and JDBC Drivers

A "connector" is a plugin that fetches metadata from the database server to optimize the transfers. Apart from the Generic JDBC Connector, Sqoop also ships with specialized fast connectors for popular database systems like MySQL, PostgreSQL, Oracle Database, Microsoft SQL Server, IBM DB2, and Netezza. Additional connectors can likewise be installed and configured with Sqoop to support other database systems or to improve the performance of built-in connectors.

Drivers are database-specific pieces, created and distributed by the various database vendors. Drivers need to be installed on the machine where Sqoop is executed because they are used by connectors to establish the connection with the database server.

3 HLoader Architecture

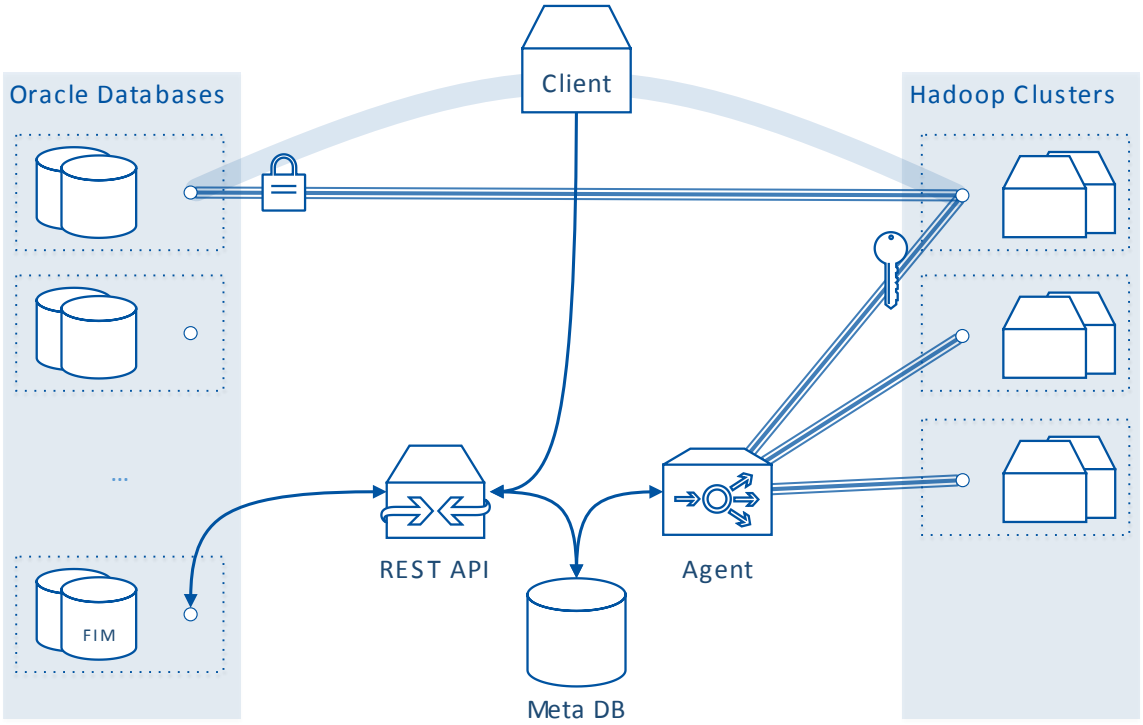


Figure 2: HLoader architecture overview

3.1 Overview

At the core of HLoader is an agent, which is closely coupled with the scheduler, runner, and monitor. The Agent is responsible for polling the meta-database for any newly added jobs, or job modifications, and notify the scheduler of the changes. While adding a new job to the meta database, the user provides all the information required by Sqoop to perform a transfer. If the user requested incremental Sqoop transfers, the job is scheduled to run in specified intervals.

At the requested time, the scheduler fires a transfer and invokes the runner to execute the job on behalf of the user. The transfer initiates a Sqoop MapReduce job which imports data from the source Oracle database to the target Hadoop cluster. The meta-database also stores all the information related to the transfers, including logs and execution status.

The users interact with the HLoader client through a RESTful interface, which exposes some of the data in the meta-database to authenticated users, allowing basic CRUD operations.

3.2 Security

3.2.1 CERN SSO Authentication

In its current deployment at CERN, HLoader uses CERN Single Sign-On to authenticate users, without any exchange of passwords. The Single Sign-On solution allows any web application hosted using CERN Central IT Web Services to authenticate users and receive their information including their group membership to manage authorizations. CERN SSO is powered by the Shibboleth software package.

3.2.2 Authorization

After a user is authenticated with CERN SSO, HLoader queries the FIM database to get a list of all servers for which the user has access rights. HLoader will allow only those database servers to be used which are both in this list, as well as configured in the meta-database. For new Oracle servers to be configured and included in the meta-database, the user can submit a request to the administrators. Use of this authorization mechanism allows HLoader to access the database on behalf of the user, without knowing or having to store their credentials.

3.2.3 Kerberos SSH Tunnelling

HLoader uses a separate CERN service account user to remotely log in to the destination cluster without passwords. This is made possible by using Kerberos tickets for authentication, with all the principles and encrypted keys stored in a keytab file. The keytab file allows automatic authentication using Kerberos, without requiring human interaction or access to password stored in plain-text file.

3.2.4 Secure password input

The password needed by Sqoop to access the source Oracle database is passed securely so that it doesn't appear anywhere in the transfer logs, or command history as plain-text.

3.3 Modularity

3.3.1 Database Connector Agnostic

Database connectors are the access points to the meta-database. Every database connector must implement the `hloader.db.IDatabaseConnector`, which defines a set of functions for the inheriting class to override. This design makes it possible for individual components of HLoader to function without having to know which connector is being used to access the meta-database.

In the initial version of HLoader, the default connector for the PostgreSQL meta-database uses SQLAlchemy as an ORM to communicate with the server. SQLAlchemy is a Python SQL toolkit and Object Relational Mapper that gives application developers the full power and flexibility of SQL. It provides a full suite of well-known enterprise-level persistence patterns, designed for efficient and high-performing database access, adapted into a simple and Pythonic domain language. SQLAlchemy includes dialects for SQLite, PostgreSQL, MySQL, Oracle, MS-SQL, Firebird, Sybase and others, allowing HLoader

to support these dialects with little or no change in the existing codebase. Support for NoSQL databases like MongoDB and CouchDB is also possible.

3.3.2 Interchangeable scheduler

The scheduler is responsible for triggering Sqoop transfers at specific timestamps. However, the type of scheduler would largely depend upon the complexity of the requirements. If there is a need for prioritized scheduling, or special task queues, it could be easily implemented with HLoader without having to change the API of other components.

3.3.3 Interchangeable runner

The runner is responsible for executing the Sqoop commands on the remote cluster, upon activation of a trigger by the scheduler. It is possible to replace the current SSH runner with something else which is more fault tolerant. With the migration to Sqoop 2, it should be possible to use Oozie and remove the runner component entirely by merging it with the scheduler.

3.3.4 REST Interface

HLoader exposes some data in the meta-database through a RESTful interface, making it possible to build a web UI client on top of it. The client cannot directly access the meta-database, but only via the REST API which is protected with CERN SSO.

The modular architecture also makes it possible to logically separate the REST API from the agent, so that they can reside in different servers and function independently. Both the REST API and the Agent use the same mechanism of accessing the meta-database, but with different instances of the same database connector.

3.4 Infrastructure

Meta-database
The meta-database is a PostgreSQL server deployed using the CERN Database on Demand Service with PostgreSQL and SQLAlchemy connector.

Client
REST API is served using Flask microserver framework with Python 2.7, and hosted using CERN Central IT Web Services on Microsoft IIS 8.5 and DFS with FastCGI interface. The web user interface is built on top of the REST API backed with AngularJS.

Agent
The agent can reside on the same Web Services DFS server, but since it is decoupled from the REST interface, it could well be hosted on any locally managed server. Running the agent on a dedicated server is important for performance reasons, since it requires an active thread pool during execution.

As the SSH tunneling is done by Paramiko, a pure python SSH interface (depending on a C low-level cryptography library), it could run virtually anywhere. We will probably use an UNIX OpenStack VM for the agent.

4 API

4.1 Entities

HLoader entities are Python classes, having the ability to represent objects in the meta-database. HLoader has the following base entity classes, and all entities, e.g., SQLAlchemy entities, must derive from them.

```
hloader.entitites.Job.Job
hloader.entitites.HadoopCluster.HadoopCluster
hloader.entitites.OracleServer.OracleServer
hloader.entitites.Log.Log
hloader.entitites.Transfer.Transfer
```

The overriding entity class can also define helper functions and relationships with other entities. Implementation of the overriding class would largely depend upon the type of meta-database connector used.

4.2 Database Manager

The Database Manager provides a meta-connector to connect to a database using a connector of choice. When accessing the meta-database, the meta-connector should be used to access all the member functions of the connector. The Database Manager defines all connectors to be used in the `DatabaseManager.connect_meta()` static method as follows:

```
@staticmethod
def connect_meta(type, address, port, username, password, database):
    DatabaseManager.meta_connector = {
        "PostgreSQLA": PostgreSQLAlchemyConnector(address, port,
username, password, database)
    }.get(type, None)
```

Adding a new connector can be done by updating the above function as:

```
@staticmethod
def connect_meta(type, address, port, username, password, database):
    DatabaseManager.meta_connector = {
        "PostgreSQLA": PostgreSQLAlchemyConnector(address, port,
username, password, database),
        "MongoDB": MongoDBConnector(address, port, username, password,
database)
    }.get(type, None)
```

Initializing the Database Manager with a specific connector can be done as:

```
DatabaseManager.connect_meta("Databasename", "server",
                             port, "user", password, hloader)
```

4.3 Agent

The HLoader Agent runs on a separate thread, and is responsible for initializing and communicating with the Schedule Manager. Since the REST API based client and the agent are decoupled from one another, therefore the agent must poll the meta-database to keep track of any new jobs or modification to existing jobs.

The duration of time for which the agent busy-waits is stored in the `polling_factor` variable. This duration along with the `job_last_update` field in the `Job` entity is used to determine whether any new job has been created, or which jobs were modified while the agent was waiting.

4.3.1 Schedule Manager

The Schedule Manger functions more or less the same way as the Database Manager, by providing a daemon which can be initialized with the scheduler of choice. The Schedule Manager is also responsible to start the daemon upon initialization.

4.3.2 Scheduler

The default transfer scheduler used by HLoader is the Advanced Python Scheduler. Advanced Python Scheduler (APScheduler) is a Python library to schedule Python code to be executed later, either just once or periodically. New jobs can be added or old jobs can be removed on the fly. APScheduler also offers some level of persistence, by storing the jobs in a database, allowing them to survive scheduler restarts and maintain their state. When the scheduler is restarted, it can be configured to run all the jobs it should have run while it was offline.

The Schedule Manager initializes a non-blocking, background scheduler which triggers jobs according to the job description stored in the meta-database. APScheduler offers the following three built-in scheduling systems:

- Cron-style scheduling (with optional start/end times)*
- Interval-based execution (runs jobs on even intervals, with optional start/end times)
- One-off delayed execution (runs jobs once, on a set date/time)

* Not tested with HLoader

`hloader.schedule.schedulers.APScheduler.APScheduler` is the main Scheduler class which implements the `hloader.schedule.ITransferScheduler.ITransferScheduler` interface. The method `tick()` contains the code to be run when a trigger is fired. It must be noted that the `tick()` method must be outside any class for serialization, and getting a textual

reference of the function at runtime. This, however, is an APScheduler specific limitation.

APScheduler also provides some event listeners for the following events:

- `EVENT_SCHEDULER_START`
- `EVENT_SCHEDULER_SHUTDOWN`
- `EVENT_ALL_JOBS_REMOVED`
- `EVENT_JOB_ADDED`
- `EVENT_JOB_REMOVED`
- `EVENT_JOB_MODIFIED`
- `EVENT_JOB_EXECUTED`
- `EVENT_JOB_ERROR`
- `EVENT_JOB_MISSED`

Default APScheduler configuration:

- Job store: `SQLAlchemyJobStore` with local SQLite database at `sqlite:///jobs.sqlite`
- Executor: `ThreadPoolExecutor(20)`
- Coalescing: `False`
- Max number of instances for one transfer: `1`
- Time zone: `Europe/Zurich`

4.3.3 Runner

Runners are the components which execute the Sqoop job remotely on the destination cluster. Every runner in HLoader must implement the `hloader.transfer.ITransferRunner.ITransferRunner` interface, which constructs the Sqoop command to be executed. The Runner spawns a new thread for each transfer for parallelising multiple Sqoop jobs.

HLoader currently uses an SSHRunner, with the tunnelling performed by Paramiko with GSS-API/Kerberos v5 authentication support.

4.3.4 Monitor

Sqoop transfers are fundamentally MapReduce jobs, which can be tracked using YARN API. As soon as the runner detects the tracking URL of a transfer, it invokes the REST Monitor to check the progress of the MapReduce job. For this purpose, the REST Monitor spawns a new thread for tracking each transfer.

5 Recommendation for Future Work

With a view to the original goals of the project, HLoader could be extended in the following directions.

Alternative database connectors

In addition to the current SQLAlchemy based connector for PostgreSQL, more connectors based on other SQLAlchemy dialects could be implemented. The 'classic' dialects like MySQL, Oracle, SQLite, Firebird, and Microsoft SQL Server which are in the SQLAlchemy Core are the prospective candidates.

The modular design of HLoader would also allow the integration of NoSQL database connectors like MongoDB. The partially implemented connector using psycopg2, the PostgreSQL adapter for Python also needs to be completed.

Alternative runners

Apache Oozie is a server based workflow scheduler engine specialized in executing actions that run Hadoop MapReduce jobs. Actions in a workflow job of Oozie are arranged in a control dependency DAG (Direct Acyclic Graph). A "control dependency" from one action to another implies that the second action can't run until the first one has finished.

HLoader's current runner which uses SSH tunnelling to execute Sqoop commands on a remote cluster could be replaced with Oozie, which could also manage the scheduling functions.

Prepare for Sqoop 2

Sqoop 2 is essentially the future of the Apache Sqoop project. It has some interesting features, the most important being the REST API, which is desirable for a project like HLoader. However, since Sqoop 2 lacks some of the crucial features in Sqoop 1, HLoader was built on top of the Sqoop 1 structure. A list of CERN relevant feature differences between Sqoop 1 and Sqoop 2 are given below:

Feature	**Sqoop 1**	**Sqoop 2**
Connectors for all major RDBMS	Support for specialized fast connectors available. Many Oracle DB services used at CERN.	Only generic JDBC Connector supported.
Kerberos Security Integration	Supported. HLoader relies on Kerberos tickets to get access into remote Hadoop clusters.	Not supported.
Data transfer from RDBMS to Hive or HBase	Supported. HLoader should supported this at some point.	Not supported.
REST API	Not supported. HLoader currently uses	Supported.

	Sqoop CLI to submit jobs, but a REST interface would make the process more clean, fault tolerant, and Oozie friendly.	

Support data ingestion into Hive and HBase

HLoader could start supporting imports into Hive and HBase tables with Sqoop 1. The change to the existing codebase of HLoader would be trivial in this case.

Resolve restrictions

Restrictions imposed upon HLoader users are primarily meant for hardened security. Some of them might be implemented differently, or removed entirely with the migration to Sqoop 2.

HLoader as a FOSS

The long-term goal of HLoader is to be available to the developer community as an open-source software. This will be gradually done in a phased manner, after HLoader is mature enough and well tested inside CERN.

6 Workflow tools

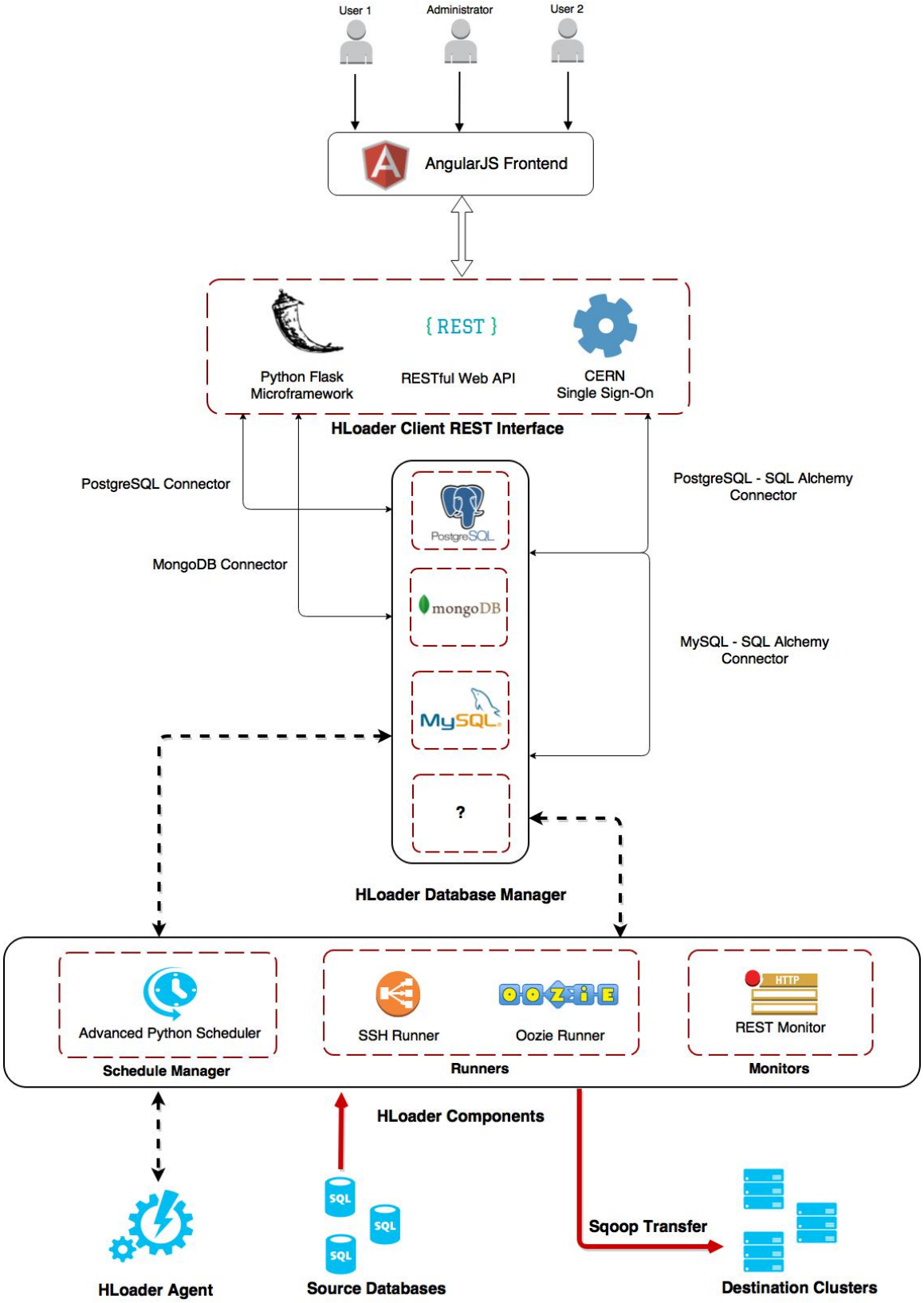
To coordinate a project of this size, we needed some development and workflow tools. We used the local CERN GitLab service and GitHub to collaborate, JIRA as an issue tracker for the project, Slack for everyday communication and Jenkins for continuous integration and testing.

7 Acknowledgements

The author would like to thank all the people at CERN who collaborated in this project. Below is a list of collaborators in no particular order:

- Daniel Stein, Summer Student, IT-DB-DBF
- Antonio Romero Marin, IT-DB-DBF
- Domenico Giordano, IT-SDC-MI
- Kacper Surdy, IT-DB-DBF
- Katarzyna Maria Dziedziniewicz-Wójcik, IT-DB-DBF
- Manuel Martín Márquez, IT-DB-BDF
- Zbigniew Baranowsk, IT-DB-DBF
- Luca Menichetti, IT-DSS-DT

Appendix I: Infrastructure



Appendix II: Database Schema