FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

ASSIGNMENT OF MASTER’S THESIS

Title:	Enterprise Solution for Deploying and Managing H2O Clusters on Hadoop
Student:	Bc. Ondřej Bílek
Supervisor:	Mgr. Jakub Háva
Study Programme:	Informatics
Study Branch:	System Programming
Department:	Department of Theoretical Computer Science
Validity:	Until the end of summer semester 2019/20

Instructions

1) Research existing solutions for deploying H2O clusters on Hadoop.
2) Create an on-premise multi-tenant service for deploying and managing H2O clusters in a Hadoop environment.
3) Expose easy to use web UI that supports authentication via LDAP/AD, SAML, and PAM.
4) Through the UI, securely deploy and monitor H2O clusters on Hadoop (YARN) including Kerberos support and impersonation based on the logged-in user.
5) Create Python and R APIs to start and access the clusters programmatically.
6) Let Hadoop administrators create cluster parameter templates and assign permissions to have control over launched applications.
7) Deploy and manage H2O Sparkling Water clusters as well.
8) Prepare installation and user documentation.
9) Create a Jenkins Pipeline for continuous delivery and release automation.
10) Explore options for automated cluster deployment into the cloud (AWS).

References

Will be provided by the supervisor.

doc. Ing. Jan Janoušek, Ph.D.
Head of Department

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Dean

Prague January 29, 2019

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Master's thesis

Enterprise Solution for Deploying and Managing H_2O Clusters on Hadoop

Bc. Ondřej Bílek

Department of Theoretical Computer Science
Supervisor: Mgr. Jakub Háva

May 8, 2019

Acknowledgements

I want to acknowledge the whole team at H_2O.ai for their support and wisdom they shared with me. Particularly, I would like to thank my thesis supervisor Mgr. Jakub Háva and Mgr. Michal Malohlava, Ph.D. for their feedback and guidance.

Declaration

In Prague on May 8, 2019 . . . .

Citation of this thesis

Bílek, Ondřej. *Enterprise Solution for Deploying and Managing H_2O Clusters on Hadoop.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2019.

Abstrakt

Velké organizace čelí problémům s integrací H_2O softwaru pro big-data strojové učení v prostředí Hadoopu. Hlavní kritika zmiňuje chybějící integraci s podnikovým zabezpečením, chybějící kontrolu nad zdroji (CPU, paměť) a chybějící monitoring. Analýza nasazení H_2O na Hadoopu a podnikových požadavků vytvořila podklad pro hledané řešení.

Na základě výzkumu existujících nástrojů bylo rozhodnuto implementovat aplikaci, která umožňuje Hadoop administrátorům nastavit bezpečnou H_2O platformu pro datové analytiky, kteří zde mohou vytvářet a používat H_2O clustery. Rámec projektu se následně rozšířil o integraci projektu Sparkling Water, který spojuje H_2O a Spark.

Tato práce popisuje implementaci takové aplikace a zaměřuje se na detaily nasazení. Výsledný program, který se jmenuje Enterprise Steam, je nyní v produkčním prostředí několika vysoce postavených amerických firem přes více než rok a dále rozšiřuje svoji funkčnost na základě zpětné vazby a dodatečných požadavků.

Klíčová slova Enterprise Steam, Hadoop, YARN, Spark, H_2O, H_2O.ai, Sparkling Water, cluster, nasazení

Abstract

Large organizations face problems when integrating H_2O software for big-data machine learning in a Hadoop environment. The main criticism mentions the lack of integration with enterprise security, lack of resource control (CPU, memory) and lack of monitoring. Analysis of H_2O deployment on Hadoop along with enterprise specifications formed the basis for a potential solution.

Based on the research of existing tools the conclusion was to implement an application that enables Hadoop administrators to set up a secure H_2O platform for data analysts to start and access H_2O clusters. The scope of the project subsequently expanded to integrate Sparkling Water project that connects H_2O with Spark.

The thesis describes the implementation of the application and focuses on deployment details. The resulting application called Enterprise Steam is now in a production environment of several high-profile American companies for over a year and keeps expanding based on feedback and feature requests.

Keywords Enterprise Steam, Hadoop, YARN, Spark, H_2O, H_2O.ai, Sparkling Water, cluster, deployment

Contents

List of Figures

Introduction

H_2O is widely used open-source software for big data analysis that is developed by the company H_2O.ai from Mountain View, California, U.S. The software is aimed towards all levels of data scientists who can install H_2O and use it on their laptop. However, to tackle the processing of big data, which does not fit in the memory of a single laptop, H_2O needs to be deployed on multiple computers to form a computational cluster.

Typically, companies store their data over numerous on-premise computers that form a Hadoop cluster. H_2O contains a launcher that starts H_2O on the requested number of Hadoop nodes to create an H_2O cluster which is used to analyze data stored on Hadoop's distributed file system.

Every time, the data scientist wants to use H_2O, they need to use the launcher to start their personal H_2O cluster. The launcher is only a CLI (Command-line interface) tool.

Current and prospective customers started to demand a supporting enterprise platform to meet their internal policies such as enterprise security, auditing, and monitoring. They also requested an abstraction layer on top of H_2O launcher to streamline the H_2O cluster creation in environments with multiple users.

In February 2018 I joined H_2O.ai to take over a beta version of the enterprise H_2O service called Enterprise Steam to move it into a full release and add additional features over time.

This thesis initially describes the Hadoop framework and H_2O software. Following chapters introduce us to the Spark cluster computing framework and H_2O's integration with Spark called Sparkling Water. Next, we analyze the solution and formalize requirements for the resulting application. Finally, we describe the implementation and discuss the results.

Chapter 1

Hadoop

Apache Hadoop at its core is a collection of open-source tools written in Java that use a cluster of general purpose computers to tackle computational problems involving large amounts of data. In this chapter, we will introduce Hadoop and all the associated technologies that concern us when developing and deploying applications on Hadoop.

1.1 HDFS

At the Hadoop's core is a distributed storage part known as HDFS (Hadoop Distributed File System). Its main feature apart from being a distributed file system is high fault tolerance. A chance of failure is always present when we store data on general purpose computers, and HDFS handles such situations gracefully.

To support batch processing and streaming workloads HDFS has been build for maximum throughput instead of the low latency of data access. To achieve this speed-up HDFS relaxed some POSIX rules and implemented WORM (write-once-read-many) access model for files. The idea is that the WORM is a very efficient data processing pattern because typically the data is loaded once in full and analyzed multiple times. Thus the time to read the full data is critical compared to the time it takes to fetch the first record. The WORM model simplifies synchronization as it guarantees the readers that the data will not be manipulated once read. Additionally, by moving computation to data rather than the other way around a significant reduction in network congestion was achieved [12].

1.2 Architecture

Hadoop comprises of several general purpose computers called nodes running Hadoop services. The hearth of Hadoop is a group of nodes called Master

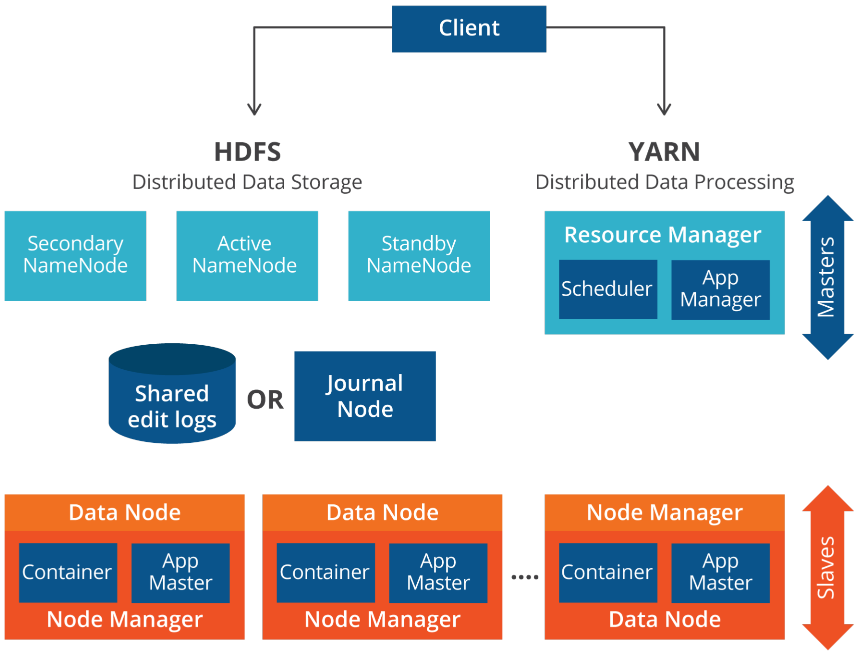


Figure 1.1: Hadoop architecture [1]

Nodes. They run essential services such as NameNodes to manage HDFS storage and Resource Managers for scheduling Hadoop applications. Deploying and configuring several Master Nodes is necessary to achieve redundancy. The single point of failure in a Hadoop cluster is the NameNode.

The largest group of nodes are the Worker Nodes where HDFS data is stored and computation performed. Several services are running on such node. The DataNode service allows the Name Node to save blocks of HDFS data on the node. The Node Manager service, on the other hand, enables the Resource Manager to schedule jobs that can use hardware resources of that node.

A common requirement is to run a MapReduce job or a client application in the Hadoop cluster. Client applications are different from regular Hadoop jobs because they need to run permanently, usually exposing some web server or API endpoint to the user. Placing them on a Worker Node may starve them of resources and mixing them with the critical infrastructure of Master Nodes is not safe for the stability of the cluster. That is why there is another type of node called the Edge Node.

The Edge Nodes, also referred to as Gateway Nodes, run cluster management tools and client applications. They are the gateway to the Hadoop cluster and contain only the essential libraries and Hadoop CLI clients (`hadoop`, `hdfs`, `yarn`) along with the cluster configuration. Edge nodes are not critical to the functionality of the Hadoop cluster and users should access the cluster only through these nodes. Figure 1.1 shows an example of Hadoop architecture.

1.3 MapReduce

So far we have described an array of computers that store data and have resources to perform computation. Hadoop leverages MapReduce framework to process parallelizable problems across large datasets stored in HDFS using the Worker Nodes.

MapReduce framework operates on key-value pairs and composes of three operations. In the Map phase, each worker uses the `map` function to process its data into a set of key-value pairs. In the Shuffle phase, key-value pairs are redistributed based on the key such that all the pairs with the same key are on the same node. Finally, during the Reduce phase, each node applies the `reduce` function, processing each group of output data per key. Map and Reduce steps are performed in parallel allowing large server farm to process petabytes of data in a few hours.

Hadoop exposes this framework via a set of Java interfaces for the user to implement their MapReduce job. Because MapReduce framework and HDFS is running on the same set of nodes it allows to efficiently schedule computation on the nodes where data is already present.

1.4 YARN

YARN (Yet Another Resource Negotiator) was introduced in Hadoop version 2 to help with MapReduce job management. However, it quickly outgrew its original assignment and now supports any custom applications to be submitted to YARN. The main idea is to have a global ResourceManager (RM) daemon and ApplicationMasters (AM) on a per-application basis. NodeManager lives on each node and reports status to the RM.

RM's Scheduler is a pure scheduler without any notion of application status. It allocates resources based only on the resource requirements of the application and available resources reported by the NodeManagers. The Scheduler is configurable with existing or custom policies to spread the work between various applications and queues.

RM's ApplicationsManager accepts job submission and negotiates the first container for execution of submitted application's AM. Negotiation of additional resources from the Scheduler, their status, and progress monitoring is the responsibility of the AM. Figure 1.2 shows the YARN architecture.

1.4.1 YARN queue

Ideally, Resource Manager would grant any request of a YARN application; however in the real world, resources are limited, and applications usually need

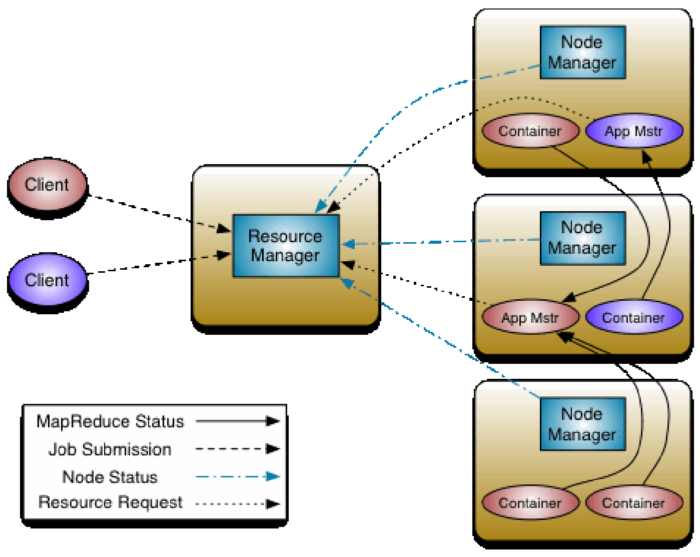


Figure 1.2: YARN architecture [2]

to wait to have their requests fulfilled. YARN ships with three basic schedulers: FIFO, Capacity, and Fair. FIFO (first in, first out) scheduler puts applications in a queue and satisfies their requests in order. Such scheduler is easy to understand but lacks suitability for more massive, shared clusters.

Capacity and Fair schedulers are based on the concept of YARN queues. YARN splits all the resources of the Hadoop cluster between YARN queues in a tree hierarchy where the root queue has access to all the resources. Figure 1.3 shows an example of YARN queue hierarchy.

Hadoop admin defines the capacity split at each level of the hierarchy along with additional configuration depending on the type of the scheduler. Leaf nodes of the hierarchy are the actual queues. The YARN queue name is a path of the queue such as `root.dev.team2`. Additionally, admins can reserve YARN queues for specific users or applications according to the intended usage. When YARN preemption is enabled, higher-priority applications can terminate jobs with lower priority.

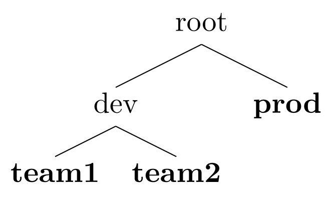


Figure 1.3: Example of YARN queues

1.4.2 YARN Application

Anyone can create a custom application and submit it to YARN. Such applications are usually written in Java thanks to the supporting Hadoop libraries and can implement any custom logic on top of the Hadoop cluster. Application Master can spawn any number of YARN containers running a user process while using resource assigned and governed by YARN.

1.4.2.1 Submitting an Application

1. The flow starts with the Client asking the Resource Manager (RM) for an application ID and available resources. Application state moves to `NEW` and the ID is used later as a crucial identifier for downloading logs, querying or terminating the application.

2. The Client sends another request to the RM with the application details. RM accepts the request and saves it to the local store changing the application status to `SUBMITTED` and passing it along to the Scheduler.

3. The Scheduler checks if the user is authorized to use the provided YARN queue and also if it exceeds any limits of that queue. If everything checks out, the application is scheduled to run, moving it to the `ACCEPTED` status.

4. The Application can stay at this point for an extended period of time if the YARN queue is overloaded or short on resources. When nothing permits the application from running the RM allocates container for the Application Master (AM), starts it and moves the application status to `RUNNING`.

AM is application specific because it contains application logic and its task is to negotiate with RM to launch jobs in separate containers. AM monitors the jobs and reports back to RM. When AM finishes with return code zero, the application status is set to `FINISHED`, otherwise `FAILED`. RM can terminate the application if it fails to respond for a long time or exceeds allocated resources. An authorized user can also terminate the job; then its final status would be set to `KILLED`.

1.5 Kerberos

Before we talk about the data protection inside the Hadoop environment, we first need to mention Kerberos. It is an authentication protocol that allows for communicating over an insecure network in order to prove an identity between each other. It uses tickets and symmetric key cryptography to do so.

Kerberos realm is the domain over which a Kerberos authentication server has the authority and consists of the KDC (Key Distribution Center), Services

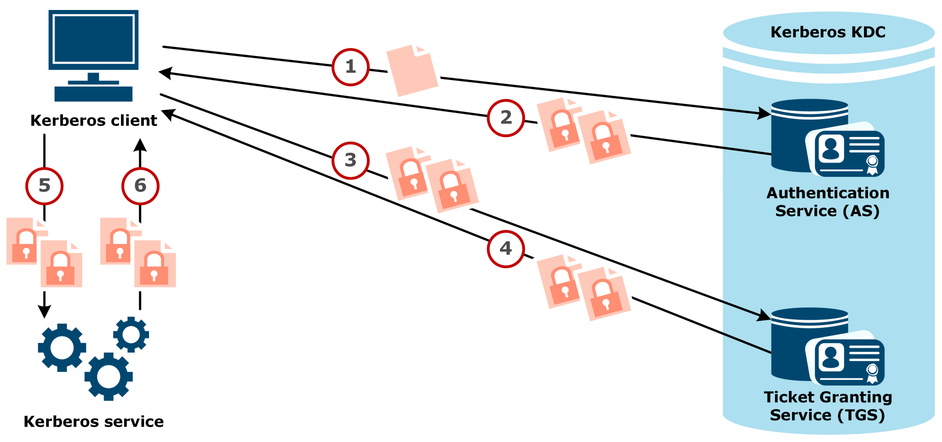


Figure 1.4: Kerberos authentication flow [3]

and Clients. KDC has two main components, an AS (Authentication Server) and TGS (Ticket Granting Server). Unix clients use `kinit` CLI for interaction with the Kerberos realm. It can obtain, renew and cache the TGT (Ticket Granting Ticket) used for Client-Service authorization. Following section is a simple explanation of the Kerberos protocol. Please refer to Figure 1.4 for visual representation.

1.5.1 Kerberos authentication flow

1. When a Client asks for the TGT from the KDC, a plaintext request is made containing the user ID. The request does not contain any secrets. The Client is also prompted for his password which is salted and hashed into the Client Secret Key and kept in memory.

2. KDC generates the Client Session Key and sends back two messages. Message A, containing the Client Session Key encrypted with the Client's hashed password from the KDC's database. Message B is the TGT and contains the Client Session Key in addition to other identifying information such as Client ID, ticket validity and Client network addresses encrypted with the KDC's Secret Key.

   When received by the Client, Message A is decrypted using Secret Key, and Client Session Key extracted. This process is successful as long as the Client's password is correct. The user can not decrypt the TGT from Message B because it was encrypted using KDC's Secret Key. TGT and Client Session Key is enough to negotiate further authorization to services.

3. When accessing Service, the Client sends two messages to the TGS. Message C that contains the TGT and service ID. Message D that consists of client ID and timestamp encrypted by Client Session Key. TGS receives message C and decrypts it using KDC Secret Key and obtains the Client Session Key to decrypt the Message D.

4. If the user ID in both messages match, KDC send another two messages back to the Client. Message E with Client-Service Ticket containing the user ID and newly generated Client-Service Session Key, all encrypted by Service Secret Key. Message F with Client-Service Session Key encrypted by Client Session Key. The client now has all the information to authenticate against the Service.

5. When initiating a connection to Service, the Client sends two messages. Message E with Client-Service Ticket and message G that contains client ID and timestamp encrypted using Client-Service Session Key. Service can decrypt the Client-Service Ticket since it was encrypted using Service's Secret Key. Using the Client-Service Session Key the Service can decrypt Message G to confirm the identity.

6. As a last step, Service sends the same timestamp found in message G back to the client encrypted by Client-Service Session Key. When Client verifies that timestamps match it can then start issuing requests to the Service.

The protocol is designed so that the Client holds tickets it cannot decrypt and sends them along with each ticket granting request. Kerberos's single point of failure is the KDC. When offline, new users cannot log in and compromised KDC allows an attacker to authenticate as any user. Clocks between Client, Service, and KDC has to be in sync as the protocol relies heavily on timestamps.

1.5.2 Keytab

A Kerberos principal is a unique identity to which Kerberos can assign tickets. An authentication is completed only if the principal can provide his password. TGT have short expiration dates and cannot be kept for later use. Not being able to hold tickets poses trouble for using Kerberos in scripts or background tasks as it is unsafe to embed the password in scripts or source code.

Fortunately, Kerberos has files called Keytabs that store Kerberos principals along with the encrypted keys. In other words, it is a store of Client Secret Keys. Keytab can then be used instead of the password to obtain the TGT. Because of that they have to be treated like passwords and securely stored as anyone with read access on that file can use all the keys inside. Keytabs should have minimal Unix permissions. They are not tied to any machine and

```
<property>
   <name>hadoop.proxyuser.steam.hosts</name>
   <value>steam1.company.net,steam2.company.net</value>
</property>

<property>
   <name>hadoop.proxyuser.steam.groups</name>
   <value>steamusers</value>
</property>
```

Listing 1.1: Hadoop impersonation configuration

can be copied around; however if the Kerberos password changes, Keytabs need to be recreated. Kinit utility supports keytabs.

1.6 Data Protection

Hadoop cluster can be configured to run in a secure mode. In such mode, each user and each Hadoop service must authenticate against Kerberos. End-users must authenticate themselves before interacting with any Hadoop service. Usually using the `kinit` CLI providing a password or keytab.

A mapping between Kerberos principals and OS's user accounts has to be set up. The Kerberos principal then authenticates as a particular OS user and access its data on HDFS. Data on HDFS can be optionally encrypted on the wire between the service and the client. HTTP endpoints can be encrypted with TLS as well.

Some applications running under a single user want to access HDFS and submit YARN jobs on behalf of other users without having their Kerberos credentials. Hadoop supports impersonation to enable such use-case [13]. This feature has to be configured in the Hadoop's primary configuration file `core-site.xml` adding two new properties as seen in Listing 1.1.

This configuration lets user `steam` connect from `steam1.company.net` or `steam2.company.net` host and impersonate users belonging to Unix group `steamusers`. Such submitted application can create a proxy user and access HDFS on his behalf. Without this configuration, creation of proxy user would fail.

1.7 Distributions

Managing all of the Hadoop software is no easy task at all. Keeping the configuration and other moving parts in sync across a large cluster is a tough administrative task when armed only with the essential tools included in Apache

Hadoop. That lead to the birth of Hadoop distributions known as Enterprise Hadoop. These products bundle modified versions of open-source Hadoop and build a set of custom tools on top of them to ease the burden of managing the Hadoop stack. In a sense, Hadoop is just a major component in such infrastructure.

Enterprise Hadoop distributions include thorough support and user-friendly interface for managing the cluster. Major Enterprise Hadoop distributors are Cloudera with its CDH (Cloudera Distribution Hadoop), Hortonworks with its HDP (Hortonworks Data Platform) and MapR. Almost all Enterprise customers are split between those distributions.

CHAPTER 2

H_2O

This chapter describes the company H_2O.ai and its main product H_2O, the open source AI and ML platform. We describe the key components and discuss the deployment on Hadoop.

2.1 About H_2O.ai

H_2O.ai is a Silicon Valley-based open source software company. Founded in 2011 by Cliff Click and SriSatish Ambati under the name 0xdata it launched an open-source software for distributed big-data analysis. Over the year H_2O.ai launched additional products such as H_2O 4GPU, Sparkling Water and recently H_2O Driverless AI.

As the company grew to its current more than 150 employees, it raised $73.6M and expanded to more US cities and abroad. The main office is in Mountain View, California and the second largest one in Prague, Czech Republic, currently with more than 20 employees.

The company is active in the open-source community, and its goal is to democratize Artificially Intelligence for everyone. The recent focus being on Explainable AI and Automatic Machine Learning with the Driverless AI platform. H_2O.ai was named a Visionary among the 17 vendors included in Gartner's 2019 Magic Quadrant for Data Science and Machine Learning Platforms. Various machine learning conferences organized by H_2O.ai were held across the world including New York, San Francisco and London.

2.2 H_2O

H_2O is an open-source in-memory platform for distributed, scalable machine learning. It is the original product of H_2O.ai launched in 2011. H_2O-3 is the third incarnation of H_2O and the successor to H_2O-2. The H_2O platform is used by over 18,000 organizations globally including companies like PayPal, Wells Fargo, PwC, Booking.com, Intel, Capital One and Cisco. GitHub repository can be found under the following link `https://github.com/h2oai/h2o-3`. It supports the following algorithms:

- Supervised
  - Cox Proportional Hazards (CoxPH)
  - Deep Learning (Neural Networks)
  - Distributed Random Forest (DRF)
  - Generalized Linear Model (GLM)
  - Gradient Boosting Machine (GBM)
  - Naïve Bayes Classifier
  - Stacked Ensembles
  - XGBoost
- Unsupervised
  - Aggregator
  - Generalized Low Rank Models (GLRM)
  - Isolation Forest
  - K-Means Clustering
  - Principal Component Analysis (PCA)
- Miscellaneous
  - Word2vec

Another feature is a Grid (Hyperparameter) Search and more importantly the AutoML functionality whose goal is to produce a list of best models by running through algorithms and their hyperparameters for a certain amount of time.

H_2O uses many popular programming languages such as Python, R, Scala, Java to interface with it as well as RESTful API and Flow notebook/web interface. An important feature is the ability to run on Cloud, On-Premise Hadoop or on a Private Cluster. Figure 2.1 shows high level architecture of H_2O.

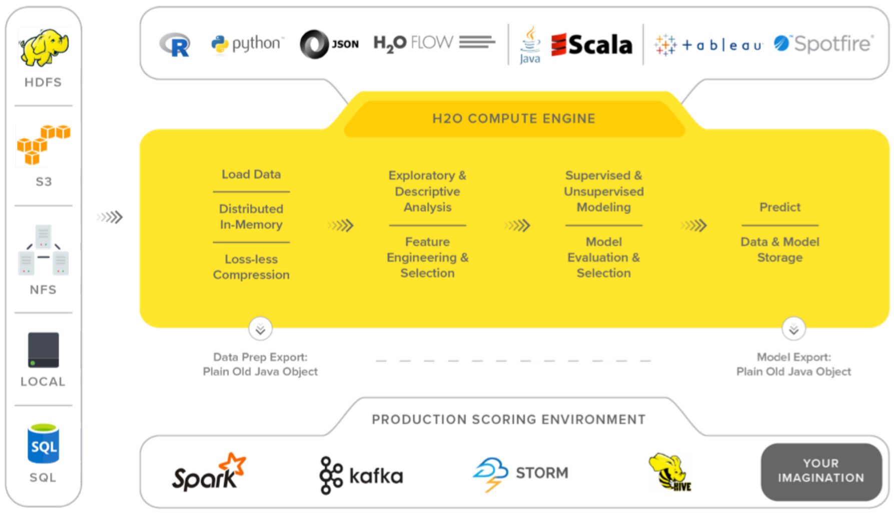

HDFS
S3
NFS
LOCAL
SQL
python
JSON
H2O FLOW
Java
Scala
tableau
Spotfire
H2O COMPUTE ENGINE
Load Data
Distributed In-Memory
Loss-less Compression
Exploratory & Descriptive Analysis
Feature Engineering & Selection
Supervised & Unsupervised Modeling
Model Evaluation & Selection
Predict
Data & Model Storage
Data Prep Export: Plain Old Java Object
Model Export: Plain Old Java Object
PRODUCTION SCORING ENVIRONMENT
Spark
kafka
STORM
HIVE
YOUR IMAGINATION

Figure 2.1: High level architecture of H_2O [4]

2.3 Architecture

An H_2O cluster is made of one or more nodes where each node is a single JVM process. It is recommended to create clusters on fairly symmetric high-performance machines with good network connections between them. Nodes rely on the network for communicating with each other. Leader nodes REST server listens to user commands coming from R, Python or any other supported API library.

2.3.1 Data Frame, Vectors and Chunks

H_2O Data Frame is the basic unit of storage visible to users. It is distributed across all the nodes and is fluid, allowing addition and deletion of columns as opposed to the frame being rigid and immutable. The key ingredient is an atomic Distributed Key/Value store which spreads the data across the cluster.

Importing of user dataset is done in a distributed manner where each node loads a subset of data. After parsing the data a handle to H_2O Frame is returned to the user. Internally an H_2O Frame is a collection of Vectors, each Vector holding a single column of the dataset. Vectors are store compressed (usually 2 to 4 times) with fast random access and on-the-fly decompression. Vector is a collection of roughly thousand of Chunks distributed across the nodes [14]. Operations on a Vector are parallel and distributed with Chunk granularity.

Chunk is home to a single node; thus it is a unit of parallel access and operations on top of it are meant to be single-threaded. It holds around thousand of elements which form a group of contiguous rows and is compressed

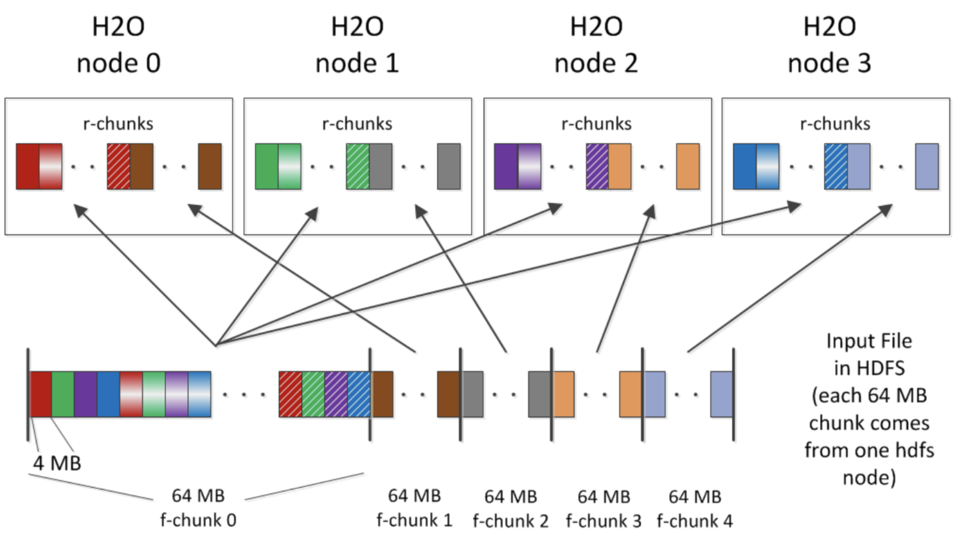


Figure 2.2: Raw data ingestion pattern [5]

as a whole [15]. Chunk can be spilled to disk when needed. When accessing a row that is out of the Chunk's range, it is pulled from the correct Chunk over the network. Figure 2.2 shows how a dataset is ingested from HDFS into the H_2O cluster.

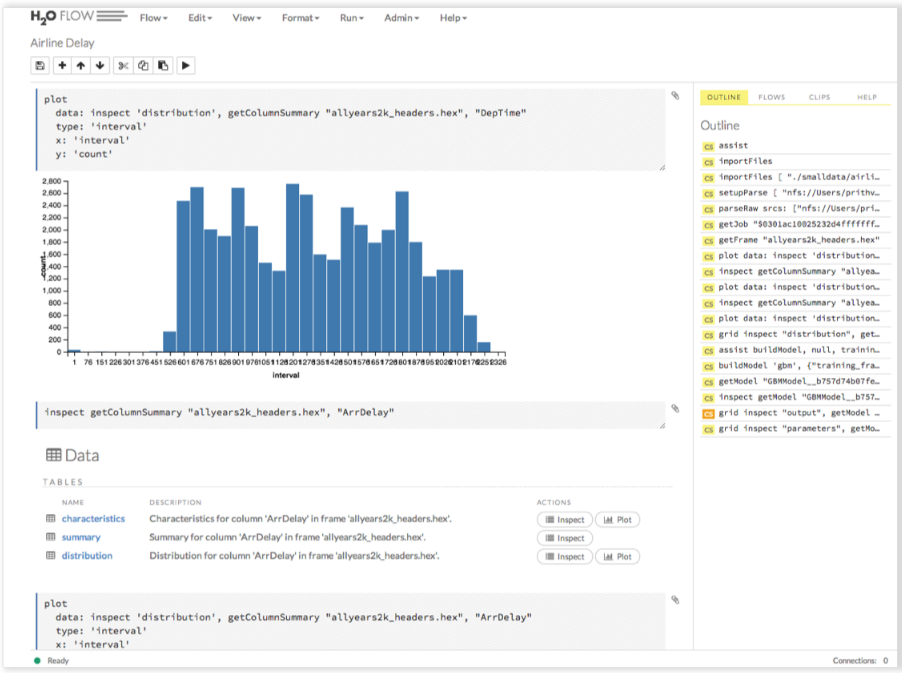
2.3.2 MapReduce

If all rows need to be accessed from a single node, the entire dataset will be fetched, most likely leading to swapping, poor performance or Out Of Memory error. A MapReduce paradigm is used to avoid this. H_2O splits tasks between nodes in a tree pattern. Results are reduced at each node and return with a single value when all sub-tasks are done. On a node level, the task is parallelized over local Chunks using Fork/Join [16]. Listing 2.1 shows an example H_2O MapReduce task on a Vector.

In summary, it is a distributed columnar store which is well optimized for datasets that have a huge number (billions) of rows and a large number of columns (thousands). It is not very suitable for big sparse matrices since it stores only rows sparsely.

2.4 H_2O Flow

H_2O Flow as seen in Figure 2.3 is a web server embedded in every H_2O node and serves as an H_2O's Web UI. It can be used to interactively import files, build and improve models, make predictions, all via the internet browser.



```
plot
  data: inspect 'distribution', getColumnSummary "allyears2k_headers.hex", "DepTime"
  type: 'interval'
  x: 'interval'
  y: 'count'
```

```
inspect getColumnSummary "allyears2k_headers.hex", "ArrDelay"
```

NAME	DESCRIPTION	ACTIONS
characteristics	Characteristics for column 'ArrDelay' in frame 'allyears2k_headers.hex'.	Inspect Plot
summary	Summary for column 'ArrDelay' in frame 'allyears2k_headers.hex'.	Inspect
distribution	Distribution for column 'ArrDelay' in frame 'allyears2k_headers.hex'.	Inspect Plot

```
plot
  data: inspect 'distribution', getColumnSummary "allyears2k_headers.hex", "ArrDelay"
  type: 'interval'
  x: 'interval'
```

```
CS assist
CS importFiles
CS importFiles [ "./smalldata/airli…
CS setupParse [ "nfs://Users/prithv…
CS parseRaw srcs: ["nfs://Users/pri…
CS getJob "$0301ac10025232d4ffffff…
CS getFrame "allyears2k_headers.hex"
CS plot data: inspect 'distribution…
CS inspect getColumnSummary "allyea…
CS plot data: inspect 'distribution…
CS inspect getColumnSummary "allyea…
CS plot data: inspect 'distribution…
CS grid inspect "distribution", get…
CS assist buildModel, null, trainin…
CS buildModel 'gbm', {"training_fra…
CS getModel "GBMModel__b757d74b07fe…
CS inspect getModel "GBMModel__b757…
CS grid inspect "output", getModel …
CS grid inspect "parameters", getMo…
```

Figure 2.3: H_2O Flow

Users can create Flows, which are a series of executable cells. Every command has a user interface to explore all of the parameters to generate commands correctly as well as monitor the progress of a running command. Flows can be exported and imported for use on other clusters since H_2O Flow's lifecycle is tied with the lifecycle of the cluster. Such interface serves as an introduction to H_2O for new users and can be used very productively; however, users are encouraged to use the Python package or R library to explore all the possibilities when building H_2O models.

Apart from machine learning features, H_2O Flow also offers an admin view of the cluster. Users can monitor all the running jobs as well as cluster status with details about every machine. The stack trace can be captured to debug long-running jobs; timeline function lists any communication between nodes and full H_2O logs are downloadable at any point. User can also choose to shut down the cluster from the Flow UI.

2.5 Productionizing

A powerful feature of H_2O is its ability to bring machine learning models into production quickly. Any model trained within the framework can then be exported and is easily embeddable in any Java environment and used for real-

```
double sumY2 = new MRTask2(){
   double map(double x){
      return x*x;
   }

   double reduce(double x, double y){
      return x + y;
   }
}.doAll(vec);
```

Listing 2.1: Sample H_2O MapReduce task

time predictive scoring in the production environment. There are currently two formats of exported models, MOJO and an old format called POJO.

Both format's only compile-time and run-time dependency is the generated `h2o-genmodel.jar`. H_2O also exports the `EasyPredictModelWrapper` API to provide a user-friendly interface in case there is no need for raw speed by accessing the POJO/MOJO directly.

2.5.1 POJO

POJO (Plain Old Java Object) was the first format for representing trained H_2O models. The generated file is a library that supports scoring and also includes the base classes from which the POJO is extended. In a simple example, the POJO is loaded and wrapped in `EasyPrecitModelWrapper` and supplied input from the standard input. The compiled source is then used for scoring.

There are some limitations to POJOs. First, they are not supported for GLRM, Stacked Ensembles or XGBoost models. Second, since the model itself is embedded in the source code, the resulting file cannot be larger than 1 GB because the JAVA compiler will crash with such input.

2.5.2 MOJO

MOJO (Model Object, Optimized) is a second generation and an improved model storage format. By taking the decision tree out of the POJO and navigating it using a conventional tree-walking algorithm it was able to go around the mentioned size issue. New models are also significantly smaller on disk, around 20 times.

Additionally, after JVM's JIT compiler has optimized the execution paths (hot paths), it scores 2 to 3 times faster. During a cold start where the execution path has not been optimized yet, it performs even faster around 10 to 40 times [17]. MOJOs are created from an H_2O model in the very same way as POJOs.

2.6 Deployment

The latest release of H_2O is always posted at the following URL `http://h2o-release.s3.amazonaws.com/h2o/latest_stable.html`. Next steps depend on the deployment type. Only the standard release ZIP file is needed to run H_2O standalone on a single machine or custom cluster of machines. It bundles the R and Python libraries but mainly contains the `h2o.jar`. It is a standard Java JAR and can be run using the following command `java -jar h2o.jar`. The only requirement is Java 7 or higher.

The previous command will start the H_2O with default parameters, such as 1GB of memory, forms H_2O cluster of size one and exposes H_2O Flow on localhost along with the REST API. Flow is then used to interact with the cluster from a browser, or a user can connect to the cluster from Python on R API. Both of the mentioned APIs contain the `h2o.jar` and can initialize it on their own. Thus the step with manually launching the JAR can be skipped if necessary.

H_2O Python package can be obtained directly from H_2O, from PyPI (Python Package Index) or Anaconda Cloud, and the latest version requires Python 2.7 and 3.5 or higher. Similarly, The H_2O R library can be obtained directly from H_2O or CRAN (Comprehensive R Archive Network) and requires R version 3 or later.

It is not necessary to run H_2O on Hadoop unless there is a specific need for it. Steps for H_2O's deployment on Hadoop slightly differ from standalone usage. Every distribution of Enterprise Hadoop differs slightly, and H_2O has to be built against the corresponding Hadoop version because of it. The Hadoop users have to download H_2O distribution ZIP matching their Enterprise Hadoop version.

At the time of writing H_2O supports:

- CDH 4.4 $\rightarrow$ CDH 6.1
- HDP 2.2 $\rightarrow$ HDP 3.
- MapR 4.0 $\rightarrow$ MapR 5.2
- IOP 4.2

The ZIP file contents are the same as in standalone version but contain an `h2odriver.jar` instead of `h2o.jar`. To start H_2O on Hadoop, this new JAR has to be passed to the `hadoop` command like so: `hadoop jar h2odriver.jar` However, this alone is not enough as the driver needs to know how many nodes and how much memory for each node the user needs. H_2O then submits such job to YARN, and it is up to YARN to allocate such YARN containers or reject the job.

Many other launch parameters can be passed to `h2odriver.jar`, but this is the basic. If the Hadoop cluster is Kerberized (running in secure mode), a user has to have a valid Kerberos ticket before submitting the job otherwise it would get rejected. Starting H_2O on Hadoop takes considerably more time because the process is in the hands of YARN.

2.7 Security

For Enterprise customers, security is at the utmost importance, and usually, a product does not touch a production environment without going through a security review. Since H_2O by design works with customer data by it has to adhere to strict data protection policies.

H_2O cluster is a set of H_2O nodes that communicate with each other. The number of nodes that create a single cluster is immutable once the cluster has launched. H_2O node is then a single OS-level process when running standalone or a YARN container when running on top of Hadoop. Each H_2O node exposes two potential attack vectors. One is the H_2O embedded web port which hosts H_2O Flow and the REST API. This port is used by the end-user to access the cluster. The other vector is the internal communication port which is used for communication and data transfer between the nodes. Protocol on this port is proprietary, but an attacker would be able to reverse engineer communication captured on this path.

Enterprises usually deploy H_2O on-premise in their data centers on top of Hadoop. That is the underlying assumption in the threat model. H_2O was not build to withstand a Denial of Service attack because a single cluster is never meant to be exposed on the internet or, more importantly, used by more than one user. Each user needs to start their H_2O cluster. That person also needs to start H_2O correctly, supplying the right configuration as by default H_2O launches with no security and provides the additional options to configure it.

2.7.1 Secure configuration

The initial step to secure H_2O cluster would be to enable TLS to encrypt the communication between H_2O's embedded web port and the client. That is done by passing a Java keystore file containing the certificate as a startup parameter. Clients then connect to the H_2O Flow's HTTPS endpoint. REST API is also protected, so HTTPS endpoint has to be used when connecting using Python or R libraries.

To secure the internode communication an additional parameter has to be passed during H_2O startup. In this mode, the H_2O driver submits the application to YARN along with necessary generated files. Encrypted communication has some performance downsides, slowing operations on the cluster by approximately 10%. Important to note is that H_2O does not support in-memory data encryption. If memory needs to be swapped to disk, it will do so in unencrypted form. It is advised to use encrypted drives as a workaround.

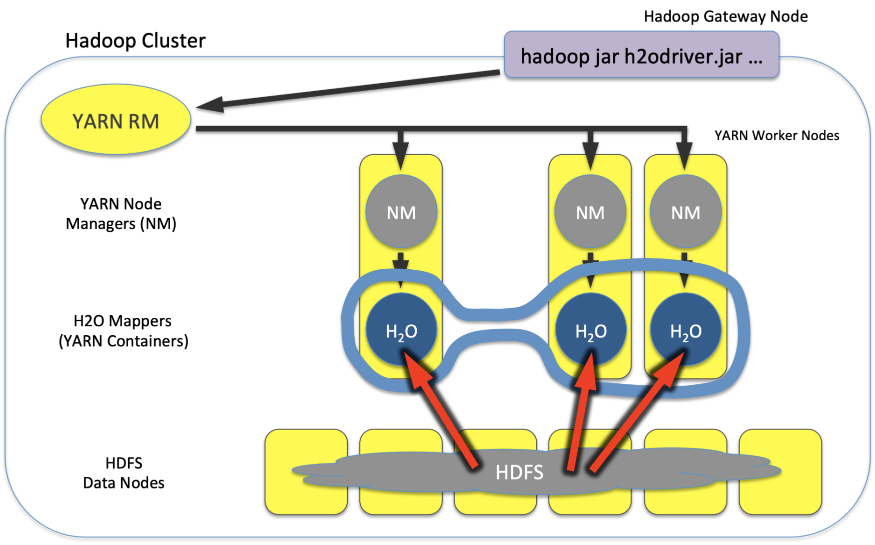


Figure 2.4: H_2O on YARN [6]

Previous steps mention only ways how to secure communication; however, authentication is vital in ensuring only the right user has access to the H_2O Flow or REST API via Python and R. Username and password have to be sent in an HTTP Basic Auth header to those endpoints. As a reminder, only one user should use a single cluster, and there are several ways how to set up H_2O authentication.

First one is a Hash File Authentication where a user provides a username, hashed password and hash used. Only such user can access the cluster. Kerberos can be used as well if the path to KDC and Kerberos's realm is specified. Another authentication schema is LDAP. It is challenging to set up correctly but is another option for the Enterprises. Lastly, PAM authentication is supported.

2.8 Hadoop Integration

As machine learning has progressed from small data to big data a single machine running H_2O is not enough for modern workloads. Most of the enterprise customers use big server farms and run Hadoop distributions on top of them while ingesting Terabytes of data from HDFS. One of the essential features of H_2O is the ability to run inside Hadoop and leverage the computing power inside it. H_2O supports ingesting datasets from HDFS natively by loading them once in parallel into nodes and forming an H_2O DataFrame. Figure 2.4 shows how H_2O is deployed on YARN.

To run H_2O on Hadoop, it essentially means running H_2O as an application on YARN. For simplicity, the H_2O is running as a standard MapReduce job spawning one mapper per node and no reducers. The H_2O cluster does not rely on any Hadoop MapReduce functionality – it has separate architecture mentioned above. It only leverages YARN to do resource management and to ensure nodes can communicate with each other.

2.8.1 Starting H_2O on Hadoop

The command `hadoop jar h2odriver.jar -nodes 4 -mapperXmx 25g` starts the H_2O driver which is used as a bootstrapper that guides the startup process to completion. This particular commands tries to allocate 4 H_2O nodes, each with 25 GB of memory. The H_2O driver is running on the actual machine which is usually a Hadoop Edge Node. The driver first determines its external IP address so it can be contacted from other Hadoop nodes. It has its heuristic for determining the correct network interface if none was specified as a startup parameter.

The driver then determines the RM address, connects to it and submits an application. H_2O JAR (`h2o.jar`) with the driver address and additional configuration is submitted along with it. When the application transitions to the `RUNNING` state its Application Master immediately submits a job request for a number of mappers equal to the number of H_2O nodes requested. Those mappers are separate YARN containers running the H_2O JAR and can be scheduled to run on any Hadoop node. The driver does not know their location and instead listens for H_2O nodes to announce themselves. H_2O nodes register themselves with the driver by asking it for a flatfile.

A flatfile is a text file that describes the topology of the H_2O cluster. Each line contains an IP address and port of a single H_2O node. Not every node has to have an address of every H_2O node as long as the graph of the cluster is connected. That can be checked by constructing a transitive closure of all flatfiles and making sure there is the same number of addresses as requested nodes.

From the nodes that registered the H_2O driver builds a flatfile and sends it back to the nodes. Nodes try to contact other nodes in the flatfile and respond to the driver, indicating the cluster size. This process is repeated when a new node registers until the desired cluster size has been reached. After that, the cluster is locked and not accepting any new members. Following a successful startup, the H_2O driver has an option to disown the cluster and exit, finishing the execution of `hadoop jar` command. The cluster is kept alive and can be terminated using an API request to the H_2O cluster, or a Hadoop admin can kill the whole YARN application.

Chapter 3

Spark

In this chapter, we discuss a different cluster-computing framework called Spark and how it differs from Hadoop's MapReduce.

3.1 Introduction

Apache Spark is an open-source distributed cluster-computing framework with in-memory data processing operating on large volumes of data in motion (streaming) or at rest (batch processing). Rich High-level APIs in Scala, Python, Java, R, and SQL allow for ETL, machine learning, analytics or graph processing.

Thanks to the in-memory nature it excels in complex multi-stage applications. In general, any Spark application loads some input and creates RDDs, applies transformations on the RDD and run actions that trigger computational and return or store the values. Next, we will briefly discuss some fundamental parts of Spark.

3.2 Resilient Distributed Dataset

RDD (Resilient Distributed Dataset) is at the core of Spark as low-level data abstraction. RDD holds a collection of records distributed among one or many partitions residing on multiple nodes in a cluster. They are in-memory and immutable; any transformations create new RDDs while keeping track of all the parent RDDs forming an RDD Lineage used in creating a logical execution plan [7].

Transformations on RDDs are evaluated lazily and triggered only when an action is executed allowing enough time to optimize the logical execution plan. Another benefit of RDD Lineage is fault-tolerance because it stores all the operations applied from last available RDD, so recomputations of damaged partitions are trivial.

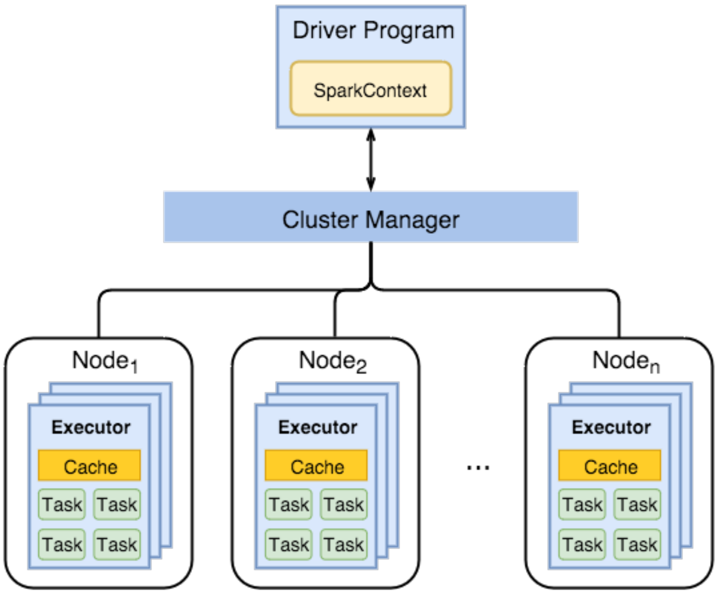


Figure 3.1: Spark architecture [7]

3.3 DAGScheduler

DAGSchduler is a layer of Apache Spark that transform a logical execution plan (RDD Lineage) into a physical execution plan. RDD Action triggers a new Job sent to DAGScheduler. Scheduler computes the execution DAG (Directed Acyclic Graph) consisting of multiple Stages. Each Stage consists of several tasks which are then run in a single thread. Single Stage computes a partial result of Spark Job and can work with partitions from multiple RDDs based on the dependency graph.

3.4 Architecture

Apache Spark uses Master/Worker architecture where one Driver running SparkContext accepts user requests and talks to Cluster Manager that manages Workers in which Executors runs. Figure 3.1 shows basic overview of Spark architecture.

3.4.1 Driver

The Driver is a Java process that contains the SparkContext which is the entry point to the services of Apache Spark. An application can be a Spark application only if it uses the SparkContext at some point. During initialization, Spark Context creates a connection to the Cluster Manager and sets up the

Spark services. Once initialized it is ready to create RDDs, run Spark Jobs, and others, essentially serving as a client of Spark's execution environment.

The Driver also hosts the DAGScheduler and TaskScheduler that split the Spark application into tasks and schedules them to run on Executors. Overall, Driver coordinates workers and their execution of tasks and once it terminates, so does the Spark application.

Driver does not participate in the computation. However, its memory has to be scaled according to the job. Memory requirements are meager if the job uses purely transformers and ends with distributed output action (i.e., save to database). On the other hand, if some Machine Learning algorithm requires results to be materialized before broadcasting them for the next iteration the job depends on the memory of the Driver since `.collect` or `.take` operations deliver data to the Driver.

3.4.2 Executor

Executors are processes responsible for executing tasks in a Spark Job and are running on the Worker node. It communicates directly with the Driver to execute tasks, return results and collect metrics. Executors are usually started at the beginning and run for the entire duration of the application but can be dynamically allocated as well.

3.4.3 Cluster Manager

The driver communicates with the Cluster Manager (aka Master) when it needs to create Executors. Cluster Manager negotiates resources with the cluster and launches Executors. Several Cluster Managers are supporting different cluster types. Currently, Spark can manage clusters on Hadoop YARN, Apache Mesos, Kubernetes as well as a standalone cluster.

3.5 Spark on Hadoop

As we have seen, Spark is just a computing framework and can be embedded in a range of existing clusters. Many enterprises store their data on Hadoop already, and such integration makes Spark a popular option when working with data on HDFS. Figure 3.2 shows basic overview of Spark on Hadoop YARN in the cluster mode.

3.5.1 Hadoop MapReduce vs Spark

Since Hadoop's YARN now supports executing any application and not only the standard MapReduce what are the Spark differences compared to Hadoop?

The main reason would be the speed. Standard MapReduce in Hadoop has to persist data back to HDFS after every Map or Reduce stage. Spark

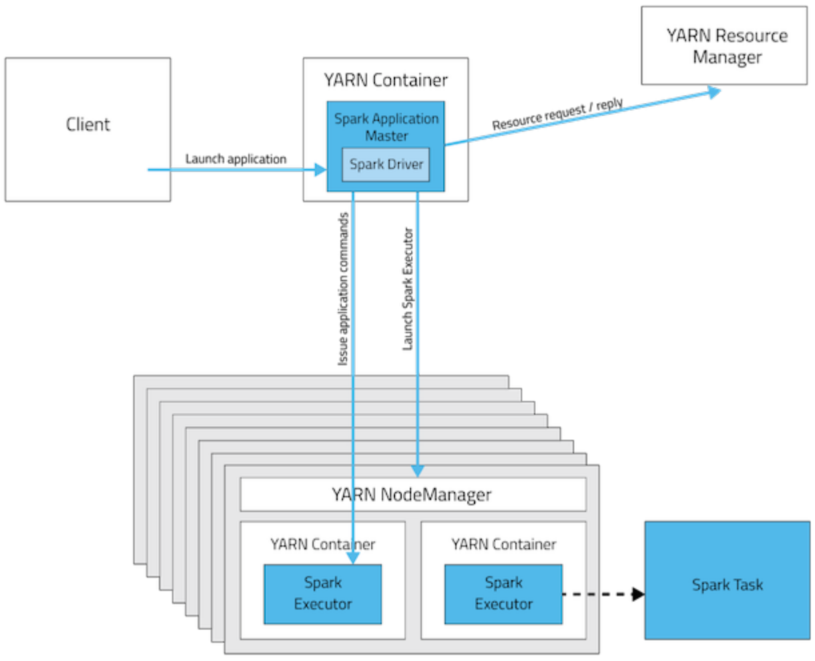


Figure 3.2: Spark on YARN in the cluster mode [8]

keeps data in memory between stages significantly speeding up iterative computations. Hadoop was built for and is generally faster than Spark in simple one-pass ETL jobs.

Another reason is the user-friendly high-level API in Spark that enables data scientists to write data transformation pipelines without an in-depth knowledge of programming. Spark also has a variety of libraries build on top of Spark for Machine Learning, graph analysis or streaming, making its ecosystem very approachable.

3.5.2 Launching Spark on Hadoop

The standard way of launching Spark is through the ”spark-submit” command that executes a script in any supported language or using one of the Spark Shells for Scala, Python, and R (`spark-shell`, `pyspark`, `rspark`). These Spark Shells start REPL (Read-Eval-Print Loop) in the chosen language with pre-initialized Spark Session and Spark Context and allow the user to work with the Spark cluster interactively.

There are two deployment modes available for running Spark on Hadoop. In the client mode, the Driver runs on the same machine that started Spark either using `spark-submit` or any of the Spark Shells. In this mode, Hadoop's Application Manager still requests the resources, and YARN NodeManager starts the Executors. In the cluster mode, the Driver no longer runs on the client machine but instead inside the Application Master on YARN. That is desired to decouple entirely from the client machine; however, in this mode, Spark Shell is unavailable as the Driver is now remote.

When launching through `spark-submit`, the user has to provide necessary configuration such as resource allocation for Driver and Executor (cores and memory). Additionally, YARN queue where the application will be submitted as well as principal and keytab in case of Kerberized clusters. In cluster deployment mode all the dependencies have to be shipped with the job.

Java/Scala job submission has to include the path to the main JAR, paths to JARs containing dependencies and name of the class to run. Python job submission has to include the main Python file and optionally paths to the dependencies, or they can point to a prepared Python environment with R behaving similarly. If a dependency path is specified, it has to be valid on all nodes.

CHAPTER 4

Sparkling Water

As Spark cluster-computing framework gained on popularity and started to be used for machine learning, it became apparent that it lacks a suitable machine learning library. Spark MLlib was eventually released, but it came too late with limited features. This lead H_2O.ai to create Sparkling Water as we will discuss in this chapter.

4.1 Introduction

Sparkling Water is an open-source project that integrates H_2O's machine learning platform with Spark. It was created by Michal Malohlava and Jakub Háva and is available on the following URL `https://github.com/h2oai/sparkling-water`.

The main idea is to run the already proven machine learning framework H_2O alongside Spark and let users take advantage of the powerful Spark transformations, feed the results into H_2O to build a model and make predictions. Typically the end-user would want to use Spark for data munging with the help of powerful Spark API and pass the prepared table to the H_2O algorithm.

4.2 Architecture

Regular initialized Spark application is easily identifiable because it has handles to SparkSession and SparkContext available as global variables in the Scala, Python or R environment.

Sparkling Water is meant to be used as a standard Spark application with added H_2O capabilities. In order to do so, an H_2O cluster needs to be running in the Spark cluster and an H2OContext handle created to access the H_2O functionality.

H2OContext is initialized by passing the SparkContext handle and configuration. H2OContext then starts H_2O nodes in a manner that depends on

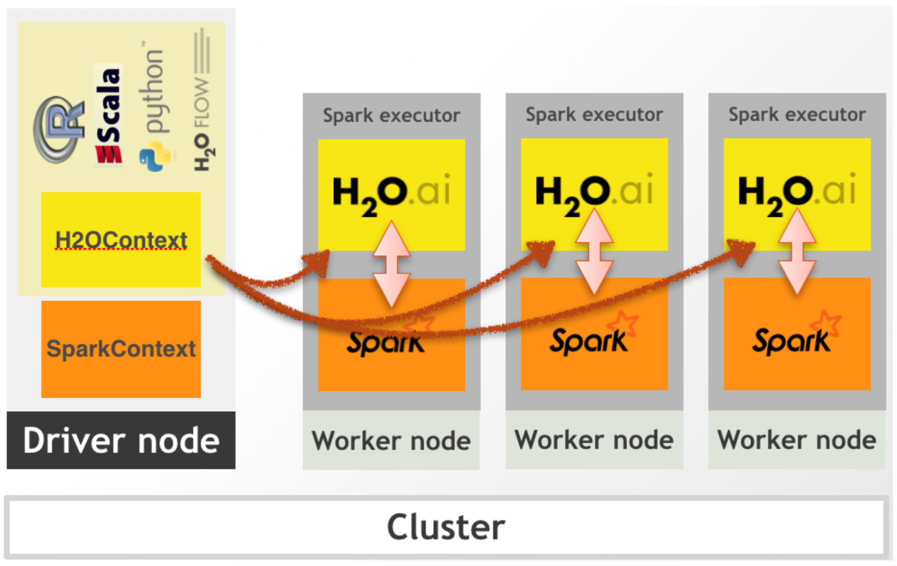


Figure 4.1: Sparkling Water internal backend [9]

the selected backend and orchestrates them into a cluster. In the end, the user has two handles, SparkContext to access Spark API and H2OContext to access the H_2O API.

4.2.1 Backends

Sparkling Water backends refer to the way H_2O cluster is created in the Spark cluster.

4.2.1.1 Internal Backend

The default behavior is to start Sparkling water in the internal backend mode. H_2O context tries to discover all the executors and launch H_2O nodes directly inside them. Spark and H_2O thus share the same JVM instance, and the topology of the H_2O cluster exactly matches the topology of the underlying Spark cluster. Internal backend is easy to deploy and suitable for the majority of the applications. However, it lacks stability for very long running tasks. Figure 4.1 shows architecture of internal backend.

H_2O does not support high availability, therefore if any node crashes, the entire cluster fails. Spark executors are meant to be expendable, and it is not uncommon to see one fail. Such failure does not pose any danger to the Spark application as executor can be restarted and data reconstructed. On the other hand, H_2O cluster cannot recover from the failure.

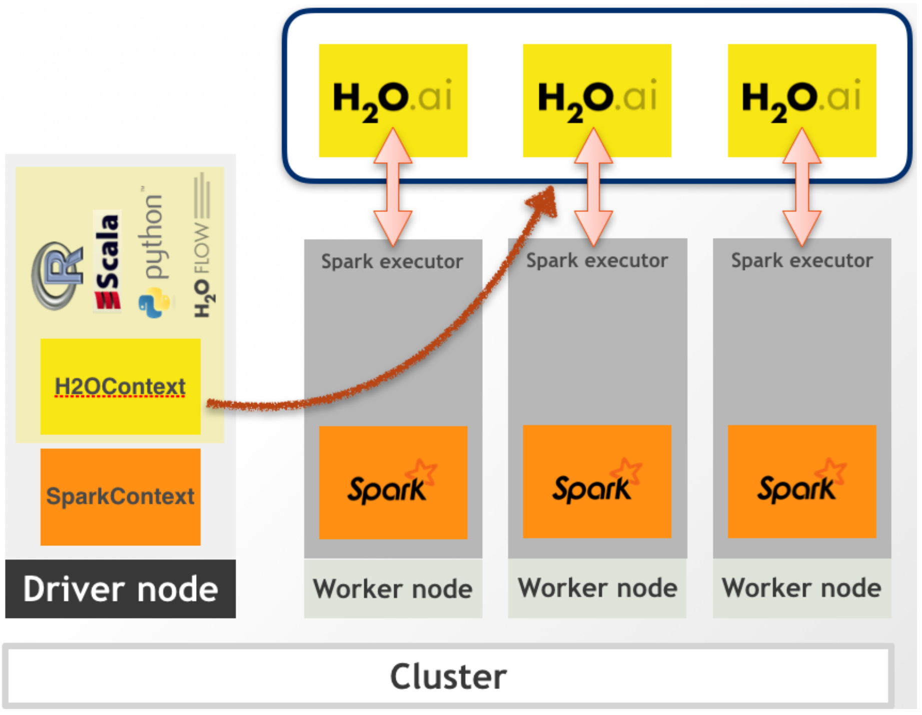


Figure 4.2: Sparkling Water external backend [9]

4.2.1.2 External Backend

External backend exists to solve the stability issues by starting H_2O nodes outside of the Spark executor. Sparkling Water can connect to any existing H_2O cluster or launch its own H_2O cluster. The topology of the H_2O cluster has to be specified as it does not need to be symmetrical with Spark executors. Executors do not run any H_2O functionality and be dynamically scheduled without harm. Figure 4.2 shows architecture of external backend.

There are downsides to the external backend as well. Because Spark and H_2O no longer share the same JVM instance the data transmission can no longer be direct but over the network resulting in communication overhead that can be noticeable on jobs that frequently exchange data between Spark and H_2O.

Lastly, normal distributions of H_2O (`h2o.jar`) does not contain classes required by Sparkling Water or to run on Hadoop. Therefore, users have to download `h2odriver.jar` for standalone Spark or corresponding H_2O version as described in section 2.6.

4.3 Data Sharing

Data imported to Spark are of DataFrame, DataSet or RDD type. Spark can work with such format, but H_2O does not understand it. Data imported to H_2O are of H2OFrame type and stored in chunks inside the H_2O cluster. Sparkling water offers conversion between Spark's DataFrame/RDD and H_2O's H2OFrame.

When converting from H2OFrame to RDD or DataFrame, the H2OFrame is merely wrapped to provide an API that is similar to RDD. Therefore, Spark can work with this data as it was imported through Spark in the first place. No data is duplicated; instead served directly from the underlying H2OFrame [18].

On the other hand, when converting from Spark RDD or DataFrame to H2OFrame, H_2O needs to load the data into the H_2O cluster and parse it to form the H2OFrame. The processing time depends on the Sparkling Water backend. As described above external backend needs to transfer data across the network as opposed to the internal backend where Spark and H_2O share the memory space.

H_2O internally has only a limited amount of types (Numeric, Categorical, String, Time, UIID). Mapping is done between H_2O types and SQL or Scala types in a way that preserves the precision.

4.4 Deployment

Sparkling Water mimics Spark in the way it offers the Spark Shell. Users need to set path to Spark directory and run `sparkling-shell` for Scala environment or `pysparkling` for Python environment. REPL is started in the corresponding language with running SparkSession and Sparkling Water dependencies. User can interact with the cluster normally and at any point start the H2OContext and use it to convert data and run H_2O algorithms.

RSparkling is the R package for Sparkling Water. It is an extension of on top sparklyr package and serves as an interface to H_2O distributed machine learning algorithms on Spark, using R. Sparkling Water does not offer rsparkling shell, but anyone can run it using their R environment with `sparklyr`, `rsparkling` and `h2o` packages installed.

Chapter 5

Analysis

In this chapter, we talk about problems with deploying H_2O on Hadoop in large enterprises, analyze the current state and formulate a desired state. We explore existing software and express formal requirements for an application that could address the problems.

5.1 Current state

Using H_2O on a single computer is easy and valid use-case when the dataset is small enough. To unleash the true potential of H_2O, it needs to run on Hadoop which brings another layer of complexity. The complexity revolves mainly around the infrastructure and configuration. Small teams, power users and longtime H_2O users usually need little time to set up H_2O on Hadoop correctly.

However, over time H_2O found its way into enterprises with a large volume of users and Hadoop resources. These companies soon identified problems with H_2O unique to such environments.

5.1.1 Problems with H_2O on Hadoop

The typical way of using H_2O on Hadoop would be to launch it manually. The data scientist would first need to have a right VPN connection to access the Hadoop cluster and then connect to a Hadoop edge node over SSH. As a next step, the user would have to locate the H_2O driver and go through the documentation to construct a correct configuration for H_2O driver that starts the H_2O cluster.

Getting the command right might take a couple of attempts or require the attention of Hadoop administrator. The amount of resources (CPU/memory) the user requests is limited only through the YARN queue or not at all. Users commonly make mistakes and request an unnecessarily large amount of resources. When the cluster is up, the user finds an IP address and port of the

H_2O node from the output of the H_2O driver and connects to it using the H_2O Python package or R library.

Using default configuration, H_2O does not protect the REST API of the cluster and lets anyone access it. At this point, anyone can log in if they have the IP address and port of the cluster. Users always go with the path of least resistance and do not care about setting authentication unless there is a mechanism that forces them to do so. Leaving the cluster unprotected is dangerous. For example, a user can import a confidential dataset inside his cluster unaware that other users can access his cluster and read the dataset.

When the H_2O cluster starts up, it stays active with allocated cluster resources until it is shut down. Commonly, users forget to shut the cluster down which leads to overloaded Hadoop nodes that are unable to serve other jobs. Hadoop administrators do not know which H_2O nodes are active and which are idle and need to spend much time investigating which clusters can be safely terminated.

As we can see, it is not a trivial process to launch the H_2O cluster, wasting the time of Hadoop users and Hadoop administrators. Some enterprises develop a suite of Bash scripts that automate the process. The scripts are usually quite restrictive and prone to errors because the authors are not experts in H_2O. Authoring such scripts is also a time-consuming process and is a barrier in more widespread adoption.

5.1.2 Problems with Sparkling Water on Hadoop

Sparkling Water in the internal backend is an extension on top of an ordinary Spark cluster. Most of the configurations, apart from securing the H_2O nodes, are inherited, and therefore Sparkling Water on Hadoop does not suffer from all the issues mentioned above. However, with the external backend, we are back at step one.

5.2 Desired state

Soon, it became clear that there is a need for an enterprise-grade tool to support H_2O and Sparkling Water deployments.

The on-premise tool would need to form an abstraction layer on top of H_2O to relief administration efforts and provide a user with an easy way of using H_2O. It would also need to satisfy strict security and privacy requirements of large corporations. The tool has to offer next-level operational efficiency in AI environments with out-of-box security, resource control, and resource monitoring. Data Scientists should be able to freely and safely practice data science in their H_2O cluster.

5.3 Existing solutions

Initially, we have looked over existing solutions to investigate if there is any existing software that could satisfy the requirements.

5.3.1 IBM Spectrum Conductor

IBM Spectrum Conductor [19] is a platform for deploying and managing Apache Spark, Anaconda and other application frameworks and services on a shared cluster. It is made for multi-tenant enterprise environments, both on-premises and in the cloud.

It is a full-featured solution, with all the enterprise capabilities. However, it does not have H_2O support out of the box which can be added by preparing Docker images and launch scripts. On the other hand, Sparkling Water is an officially supported integration.

5.3.2 BlueData

BlueData [20] offers Big-Data-as-a-Service platform with a secure multi-tenant architecture either on-premises, in the public cloud, or a hybrid model. It specializes in containerized environments where applications are spun up as Docker containers. The included App Store contains pre-configured Docker images available for one-click deployment. Customers can create their own Docker images using BlueData templates, in order to add their preferred frameworks and applications to the App Store.

BlueData just added official support for H_2O and Sparkling Water. The functionality was not available when this analysis took place. The BlueData platform is compelling; however, it requires customers to commit to containerization architecture which is a deal breaker for many enterprises.

5.3.3 Qubole

Qubole [21] is a powerful cloud-native data platform, but the absence of on-premise capabilities quickly disqualified the solution.

5.4 Enterprise Steam

After the initial analysis, we have quickly discovered that there is a need to develop an in-house application to support H_2O deployments on Hadoop. The application would help us to maintain full control over the H_2O ecosystem and assist customers with quick H_2O adoption.

The product got the name Enterprise Steam. Appendix A shows a datasheet that expresses the initial application requirements before the implementation began. Next, we will formalize the application requirements.

5.4.1 User roles

There are two main roles described below.

5.4.1.1 Data Scientists

Data Scientists are users that start, monitor and stop H_2O clusters. Their job is to provision the cluster and then use it to practice data science. They are not Hadoop experts and they are not interested in the underlying infrastructure.

5.4.1.2 Hadoop Admins

Hadoop administrators monitor H_2O clusters and make sure the infrastructure can support the workloads of data scientists. They manage the authentication, authorization of data scientists and any other configuration.

5.4.2 Requirements

Following requirements need to be implemented in order for the application to satisfy the required goals. The requirements evolved, but this was the main idea for the application.

5.4.2.1 R1: Start clusters

The user needs to have the functionality to start an H_2O cluster from the application using a minimum set of parameters.

5.4.2.2 R2: Manage clusters

The user and administrators need to be able to stop a cluster when needed.

5.4.2.3 R3: Monitor clusters

Administrators need to see running H_2O clusters, how much resources they are consuming and whether the cluster is idle or not. They also need a way to terminate the cluster.

5.4.2.4 R4: Access clusters

The user needs a way how to access H_2O Flow from the application and connect to the cluster using Python package and R library.

5.4.2.5 R5: Resource control

Administrators should be able to specify which H_2O versions are available and limit the resources users can consume.

5.4.2.6 R6: Manage users

Administrators need to manage authentication and authorization of users from the application.

5.4.2.7 R7: On-premise deployment

Most of the enterprises do not accept any cloud solutions for security or privacy reasons.

5.4.2.8 R8: Web UI

We wanted to move from the command line to a browser. Administrators should be able to perform most of the work from the browser and users have to have the capability to start and manage H_2O clusters, and access H_2O Flow from there as well.

5.4.2.9 R9: Multi-tenancy

We need to prevent users from accessing each other's H_2O clusters (and data).

5.4.2.10 R10: Network isolation

We wanted to isolate users from the Hadoop environment so there would be no need to use VPN or SSH to a Hadoop node. The users should be able to access their clusters even on the outside of the Hadoop firewall/network.

5.4.2.11 R11: Security

Strong emphasis on security. LDAP/AD authentication, TLS encryption, Role-based Access Control, secured H_2O clusters, Kerberos integration.

5.4.2.12 R12: API access

Every user action that can be done inside the Web UI needs to be available from Python and R API as well.

Chapter 6

Implementation

This chapter focuses on the implementation part of Enterprise Steam. We discuss important decisions and software-engineering aspects of the implementation as well as technical challenges encountered on the way. We will also highlight how requirements changed in the face of customers, and the general progression of the solution in time. Represented is the current release of Enterprise Steam 1.4.8, dated April 15th, 2019.

6.1 Technology stack

At the beginning of every application development lies a critical decision on what technologies are going to be used. The decision should be a function of the suitability of that technology in building the application. However, in small teams and startup environments, it is not always the case. Those teams prefer familiar technologies to have the proof of concept available as soon as possible and to have the shortest time to market.

6.1.1 Backend

In the case of this application, a Java backend would be the first choice because H_2O and Spark are written in JVM languages and H_2O team already has much experience with Java and Scala development. On the other hand, this application would be developed by a separate team, and Java usage was not necessary since Hadoop can be interfaced using its CLI (Command-line interface) on the host machine.

6.1.1.1 Go

Go programming language was selected to implement the application backend. It is a statically typed, compiled programming language with a focus on simplicity. Go was designed at Google and is syntactically close to C, but more high-level and with memory safety and garbage collection.

Supported paradigms include imperative programming, object-oriented programming without inheritance and functional programming. The rich standard library supports all aspects of common programming paths. Go also treats concurrency as a first class citizen providing tools to run a concurrent functions in lightweight threads called goroutines and synchronize them with channels as an alternative to locks.

Go is an excellent tool for writing APIs from scratch and hosting small applications. Along with the novelty of emerging language, those were the main reasons in picking Go to implement application backend.

6.1.2 Database

The application has data to persist, and another step is to find a suitable database to store them. The volume of data for such application is low, and the data are very structured and clearly defined by their relationship. There was no reason to venture into the realm of NoSQL databases that thrive with unstructured data.

The application is supposed to be deployed on-premise on the customer's hardware. Usually, that would require the customer to set up a traditional database and provide connection details. This additional setup adds time and complicates cases when the customer wants to try the product before committing.

Traditional databases are stored as files on the file system of the database host, managed with a database process and accessible over the network. In the case of this application, we would like to set up and host the database directly to avoid any setup from application's administrators.

6.1.2.1 SQLite

SQLite was built exactly for that purpose. It is a serverless, self-contained SQL database engine that requires no configuration. SQLite does not have a server process. Instead, it writes directly to the file system as standard disk files. A complete SQL database is contained in a single file in a cross-platform format. SQLite library has bindings in most of the programming languages including Go, and the database can be queried a using an ordinary SQL statements.

6.1.3 Reverse Proxy

The reverse proxy is a type of proxy server that receives requests and retrieves resources from one or more servers on behalf of the user. These resources are then returned to the client and appear as if they originated from the proxy server itself. The main aspect when choosing the right reverse proxy server was reliability.

6.1.3.1 HAProxy

HAProxy is a solution that offers high availability, load balancing and proxying for HTTP and TCP applications. It is widely used and suited for very high traffic applications. Good reputation and reliability influenced the decision to employ HAProxy.

On the other hand, HAProxy might seem like an overkill. The application would not be serving more than hundreds of concurrent users and did not need any other functionality apart from the reverse proxy. The application would also need to manage the HAProxy process through the `haproxy` CLI to supply and reload the configuration. Still, HAProxy was used in the initial implementation because of the reasons stated.

6.1.3.2 Go Reverse Proxy

The standard library of Go Programming language includes reverse proxy as a handler that takes an incoming request and sends it to another server, proxying the response back to the client. It gives absolute control over the request, allowing modification of the payload on the way to the server and also modifying the response on the way back to the client. It was later added to the application as an experimental reverse proxy to potentially eliminate the need for HAProxy.

6.1.4 Frontend

Frontend part of the application was not the main focus of the project. The UI was not supposed to be the state of the art but to only support the functionality of the backend. We will mention the frontend implementation, but the main focus of this chapter will be on the backend part.

6.1.4.1 React

React was chosen as a frontend library mainly because it was familiar and in a mature state. The goal was to build all the components in the initial design and then use them for additional features over time.

React is a JavaScript library used for building user interfaces. It is based around the concept of reusable components that can be stacked together. Components, like HTML elements, receive a list of attributes, called props. At its core, a simple component is just an ordinary JavaScript function that accepts those attributes and returns JSX.

React components return JSX that in turn renders HTML representation of that component. JSX is a JavaScript extension that allows writing

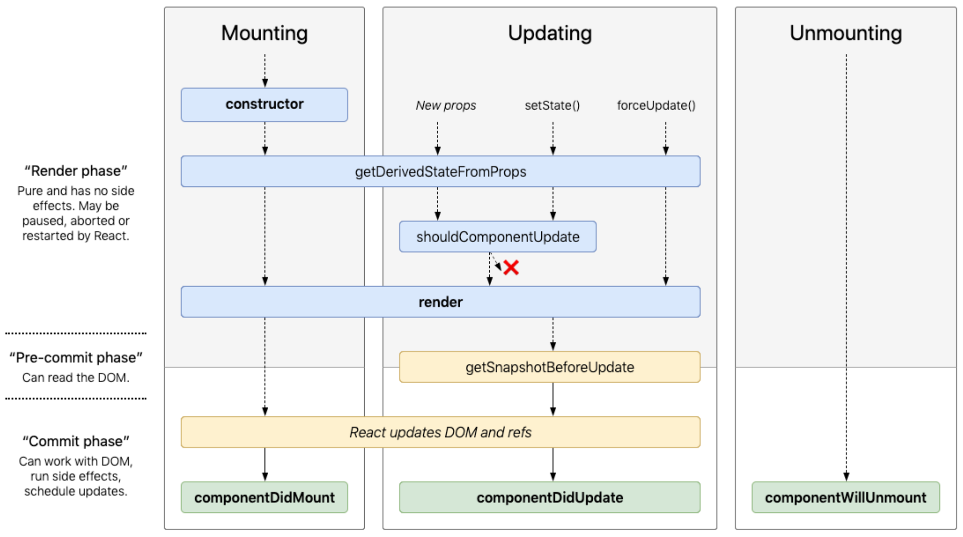


Figure 6.1: React Lifecycle Methods diagram [10]

JavaScript that looks like HTML. At runtime, JSX is translated to regular JavaScipt that calls React functions to create DOM (Document Object Model) elements. The developer can decide not to use JSX and create elements directly; however, that becomes untenable with nesting components. Any JavaScript expressions can be wrapped into curly braces and embedded into JSX. However, control statements like `if` and others can not be used.

Function components are fine for simple uses, but the main benefit of React comes when constructing components as JavaScipt classes. To do so, we only need to define a class that extends `React.Component` and implements the only instance function called `render()`. Every time the class is used in another component's render function, React will instantiate an object from this component and use it to render a DOM element in the DOM tree. We learned that attributes could be passed to components. In the class approach, attributes are received as an instance properties called `props`. Class functions can be defined and used anywhere, including as the returned JSX output in the `render` function.

React components can have private state stored in the `state` property. It cannot be changed directly but only through the React API `this.setState`. The internal state of a component usually changes as a result of a user action, for example, a click of a button triggers function that changes the component state. Setting state using React API has a reason. React needs to keep track of changes inside the component so it can react to them accordingly.

A React component is re-rendered when its state changes or when it receives new props from its parent component. It would be ineffective to construct an entire DOM of the page on every change so React computes the difference between the current components and its previous version to execute efficient DOM operations to synchronize them. The name does not suggest that, but `setState` is an asynchronous request to the React scheduler to update the state and render the component again.

React component class also includes lifecycle functions that fire on important events during the component life. When new props appear, or new state is set `getDerivedStateFromProps` triggers, and the developer has the chance to update the state based on the props received. Next, `shouldComponentUpdate` triggers containing the future props and future state. By comparing them with the current state and props, the developer can decide whether to render the component or not.

Finally, the component is rendered, and `componentDidUpdate` triggers which is used as an opportunity to operate on the DOM when the component has been updated. Figure 6.1 shows a diagram of React lifecycle methods.

6.1.4.2 Redux

A hot topic when implementing React applications is how to manage the state. Components have local state, but they may need to update their parent's state on some occasions. Child components cannot interact with parent components directly. The only thing they receive from them are the props. These props can hold functions and give child components an option to update the parent. In this approach, the state is usually held in the top level component and propagated further using props.

It is a correct application of state store; however, it may cause unnecessary complexity in larger applications. Props need to be drilled through multiple layers of components, are hard to track and become problematic during refactoring. Recently React addressed this issue with their Context API that provides a way to pass data through the component tree without having to pass props down manually at every level. This API was not available at the start of the development.

Instead, Redux was chosen as a state-store for this application. It is a predictable state container for JavaScript apps. It can be used with any framework and provides official bindings for React.

Redux applications have a single source of truth where the state of the application is stored in a single object tree. That makes them easy to debug and visualize. Redux state is read-only, and the only way to update it is to emit an action that describes what happened. Reducers receive the actions and replace the current state with a new version. This way every change is recorded and easily debuggable. Reducers are meant to be pure functions, dependent only on the action and not the current state.

In a React component, the Redux state is mapped to props and trigger a standard update whenever changed. Additionally, component functions can dispatch actions to update the state store.

6.1.4.3 TypeScript

TypeScript is a superset of JavaScript to make it scale in larger applications and repair shortcoming of its design. It offers static typing with compile-time type checking, classes, interfaces, generics, enumerated types, tuples and more. Internet browsers do not understand TypeScript, so TypeScript programs has to be transcompiled to JavaScript. React, and Redux are both JavaScript libraries but offer TypeScript declaration files to make their APIs type safe.

Typescript was chosen for this application to make the developer life more comfortable and to combine nicely with type-safe backend. It is always better for the compiler to catch errors instead of the application failing at runtime.

6.2 Architecture

Before diving down to implementation details let's introduce the high-level architecture of the application. The machine where the application is installed needs to be a Hadoop edge node that has two ports mapped to the outside of the Hadoop firewall and into the corporate network. The application binds to those ports, exposing the API and Web server on one port and cluster connections on the other port. Web UI, Python package and R library are used to authenticate to the application and manage clusters.

All of the started H_2O clusters are proxied behind one of the ports and available only to authenticated and authorized users. Figure 6.2 shows the application architecture for deploying H_2O clusters. Other parts of the application architecture are described later.

6.3 Reverse proxy

The application serves content from multiple H_2O and Sparkling Water clusters started by users. Clusters can have their API endpoints on any Worker Node of the Hadoop cluster and on any port of that node.

One of the applications requirements is to shield users from the Hadoop network. However, the application only has one port available for all the clusters. This requires to employ reverse proxy which proxies different URLs on a single host to different clusters in the Hadoop network. The reverse proxy runs as separate process and is configured by the main application.

The separation of the reverse proxy into a separate process has the benefit of permanent uptime of the cluster connection even though the main service is down for maintenance or upgrades. However, this only affects clients that are already authenticated as fetching the cluster connection details requires the main service to be running.

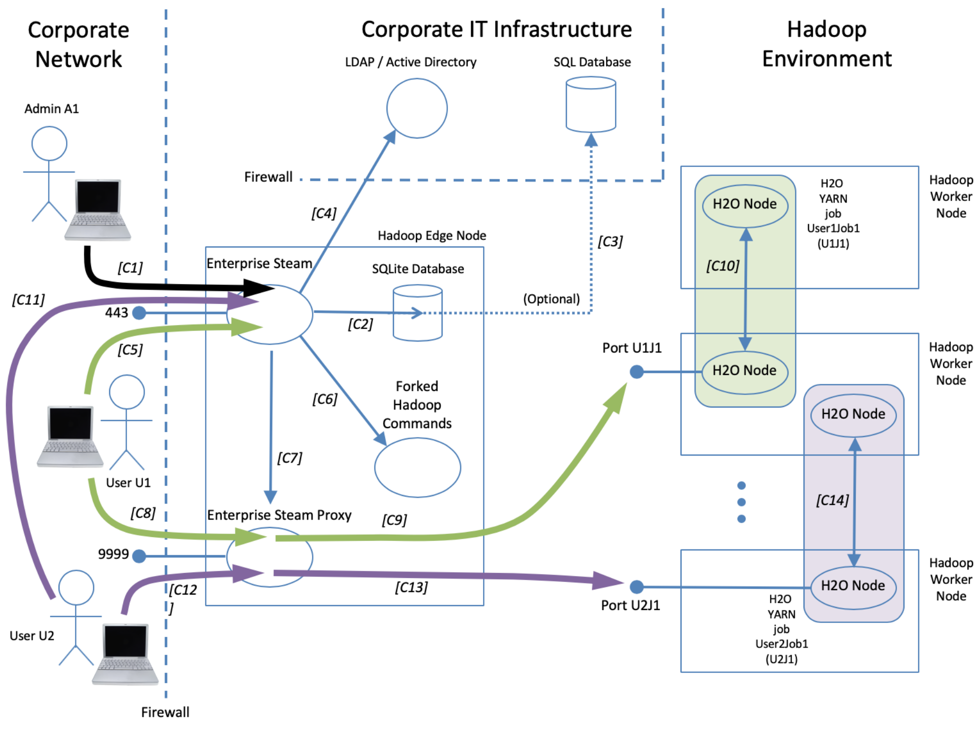


Figure 6.2: Application architecture for H_2O clusters

6.4 Database Abstraction Layer

The database is managed solely by the application logic and has several layers to help the development. Some of the layers are custom, and some use existing libraries.

6.4.1 Database Driver

Go has a package in its standard library that provides a generic interface around SQL-like databases. The package alone is not enough and has to be used along with a database driver. SQLite has several drivers, and for this application, we are using the most prominent one `github.com/mattn/go-sqlite3`.

Standard database drivers only relay the SQL queries to the database process and return results. SQLite is unique because it has no database process as mentioned before. Therefore the entire database logic has to be embedded in the driver. For a project with a scale of SQLite (200,000+ lines), reimplementing the entire library in Go is not an option. Fortunately, Go supports Cgo which lets Go packages call C code. SQLite driver in Go imports the

SQLite C library and appropriately binds the functions. The only downside is more complicated compilation as GCC has to be present on the build system.

6.4.2 Initialization

For the first time, the application has to be started using a special argument that will run the application's one-time setup. The user is prompted for username and password that will be the application's administrator account. SQLite database driver is loaded, and connection string specified pointing at the database file. Database driver creates the database file if it does not exist and imports the schema.

6.4.2.1 Secure Password Ingestion

To securely read the password from the terminal and not print it back to the user we have to set up a safer environment. The first step is to use the system call `SYS_IOCTL` with the file descriptor of standard input to get the terminal instance, modify its configuration and set it back.

There are several terminal flags set in this new terminal configuration. First of all, the `ECHO` flag is turned off so that the input is not echoed to the user. The terminal is also put in canonical mode with the `ICANON` flag so that the input is available only after it was terminated with `EOF` (end of file), new line or similar. The `ISIG` flag aborts the input when a signal is detected, and `ICRNL` flag interprets carriage returns as new lines — altogether creating a safe environment for the user to type their password and have it secured until they choose to submit it.

The terminal configuration is saved before its modification and restored back after the password has been captured.

6.4.3 Schema

Schema is stored as multiple constant strings in the application source code. They are a series of a `CREATE TABLE` statements to prepare the database. The schema itself is not particularly interesting; it makes modest use of foreign key constraints, and only data types are `TEXT`, `INTEGER` and `DATETIME`. Most columns are not nullable. Some foreign key specifications have `ON DELETE CASCADE` to automatically delete corresponding records in the child table.

Along with the schema strings, Go struct types have to be created for every table. Go structs are just typed collections of fields. Struct fields can optionally have string literals placed afterward. Those are called tags, and they add meta information that can be used by the current package or an external one.

Every database table has a corresponding struct definition typed to match the column type and field tags that matches the column name.

6.4.4 Scanning

At some point, fetched raw database rows have to converted to Go structs and passed back to the caller. This action is called scanning and requires the developer to specify which fields hold which columns of the fetched row. Generally, there is one scan function for every table. For a large number of tables, there would have to be much boilerplate code.

The `github.com/variadico/scaneo` tool generates scan functions from the struct definitions. This CLI tool takes the path of a Go file that contains the struct definitions, parses it and generates Go code using Go template.

6.4.4.1 Go parser

Go programming language is self-hosting, meaning the compiler can compile its source code. As a result, the Go parser is included in the standard library. The `scaneo` tool takes advantage of that to infer struct names and fields from the source code.

6.4.4.2 Go templates

Go template is a standard library package to implement data-driven templates for generating textual output. Templates can also be used for generating HTML output that is safe against code injection. An author creates a generic template and applies a Go struct with specific content onto it. The result is a generic text merged with specific text. The `scaneo` tool uses templates to generate the Go source code that performs scanning.

6.4.4.3 Go generate

Programs that write other programs are important elements in software engineering and Go allows us to run them before the build process. It is called `go generate` and it works by looking for special comments in the Go source code and running the commands found there. These commands can generate code that is necessary for the build phase such that `go generate` can be immediately followed by `go build`. As an example, the following comment if present in the source code runs the `scaneo` tools with a parameter that expands to the path of the current source file: `//go:generate scaneo $GOFILE`

6.4.5 CRUD

One of the most common interactions with the database are CRUD (Create, Read, Update, Delete) operations. Rows can be added to a table by providing all the non-nullable columns. Rows can be updated by providing a primary key and updated columns. Rows can be deleted by providing the primary key. Finally, rows can be read, either individually by providing a primary key or in a batch using a condition. Some applications do not need any additional database operations.

The logic with all of the CRUD operation is the same, only the table changes. We can make use of code generation to prepare all the CRUD functions for all the tables. A custom code generator called `crudr` was written to do so. It uses the same approach as `scaneo` and parses the Go source code containing the struct definitions to compose a Go source code that implements CRUD operations using Go templates. To recognize which columns hold primary keys a struct field tag has to be set.

Resulting CRUD function can be used directly in the handlers and code-generation makes it effortless to introduce new columns or tables.

6.4.6 Database migrations

The database schema is not immutable. Over the time updates are introduced that may add tables and columns. Some updates may make more significant changes. In order to have the database synchronized with the application, it has to be properly migrated.

Every time the application is launched, it compares the database version embedded in the source code to the database version stored in the database. If they do not match a migration or series of migrations has to be performed. There is a migration routine for every increment in database version. It contains SQL code that gets executed before the application can start serving requests.

The migration routines are only specified when the version increases. Currently, the application does not support migrating the database down. In the future, it can support it in the very same way as migrating up.

6.5 Authentication

The application requires authenticated access to establish the identity of the users and later serve them data that they are authorized to access. The application supports creating users manually and storing their credentials. However, enterprises have their identity managers and typically want users to access the application with the enterprise credentials instead of creating new ones.

6.5.1 Session token

The application's login page is the only endpoint that is publicly accessible and is used to authenticate users. User credentials are validated, and a secure session token is issued that has to be used for future requests on secured endpoints. This token contains values, namely user ID, and user name. Session tokens cannot be forged because their values are validated using HMAC against a stored signature. That means that anyone can read the values stored, but they cannot be changed because the signature would not match.

Currently, a secure hash key is generated on every application startup thus invalidating any current tokens. By persisting the hash key, we could keep session tokens across application reloads. Session tokens are returned to the user in the form of HTTP cookies. The cookies are issued with the `HttpOnly` flag to prevent access from JavaScript's `Document.cookie` API and `Secure` flag to only transport them over the HTTPS protocol. The cookie also contains an expiration date.

Secure endpoints require a session token to be present inside a cookie of the request. Session token establishes the user identity which is propagated into the handlers.

6.5.2 Local authentication

The application supports local user identities. Credentials for those identities are saved in the database. Passwords are sent in plain text over HTTPS and then hashed and salted on the serverside before storing them in the database. Passwords are hashed using `bcrypt` package that implements Provos and Mazières's bcrypt adaptive hashing algorithm.

6.5.3 LDAP authentication

6.5.3.1 Single sign-on

SSO (Single sign-on) allows a user to log-in with a single user ID and password to multiple related, yet independent software systems. Enterprises make heavy use of SSOs because it has significant benefits for them. It reduces password fatigue from having multiple username and passwords and also lowers risk when integrating 3rd-party applications since passwords are stored externally.

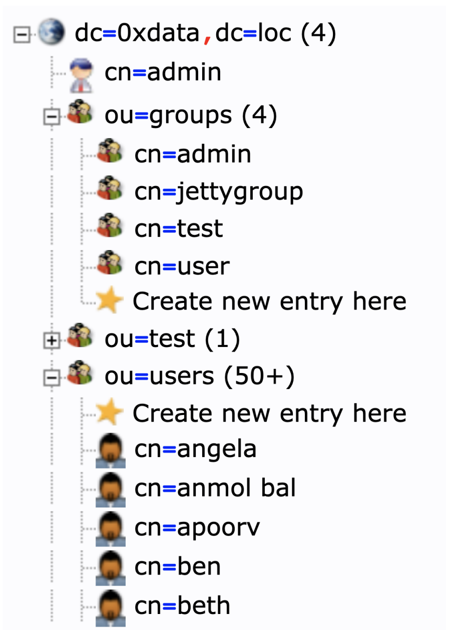
On the other hand, SSO increases impact when the credentials are stolen as they provide access to multiple services. Therefore, SSO should be combined with multi-factor authentication (one-time tokens or smart cards).

6.5.3.2 Active Directory

Most of the enterprises use some central place to store usernames and passwords. Active Directory (AD) provides LDAP/X.500-based directory-based identity-related services and is Microsoft's implementation of a directory service. There are many other implementations of a directory service.

6.5.3.3 Directory service

Directory service consists of objects that are either resources or principals (accounts, groups). Each object represents a single entity, has a set of attributes



```
dc=0xdata,dc=loc (4)
    cn=admin
    ou=groups (4)
        cn=admin
        cn=jettygroup
        cn=test
        cn=user
        Create new entry here
    ou=test (1)
    ou=users (50+)
        Create new entry here
        cn=angela
        cn=anmol bal
        cn=apoorv
        cn=ben
        cn=beth
```

Figure 6.3: Example LDAP domain

and is identified by a canonical name (CN). A schema defines the attributes, which means that there can be objects of different types. For example, a group object has an attribute that holds IDs of group members and an attribute that specifies the group name.

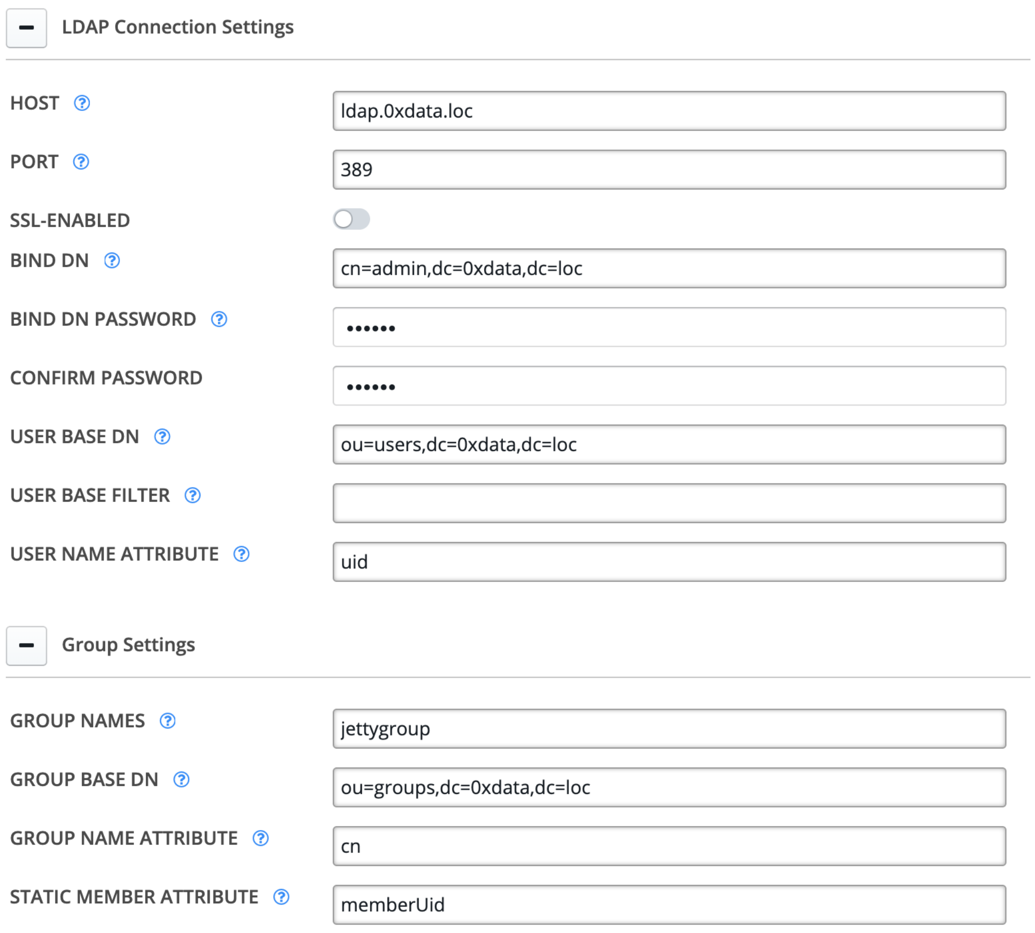
Objects that are stored in a single database represent a domain. Domain names are simar to DNS names such that the distinguished named (DN) of the `0xdata.loc` domain is `dc=0xdata,dc=loc`. DN of an object `obj1` in the first level of `0xdata.loc` domain is `cn=obj1,dc=0xdata,dc=loc`. The DN is a unique identifier of an object within the domain.

Objects within a single domain can be further grouped into organizational units (OU). These bring hierarchy to a domain and can resemble the organization's structure, geography structure or anything else. Figure 6.3 shows an example of a domain.

6.5.3.4 LDAP

Lightweight Directory Access Protocol (LDAP) is an industry standard application protocol to access and manage directory information services over the IP network. LDAP/X.500-based directory services implement this protocol and applications can use the protocol to make queries against the service. All information is transmitted using Basic Encoding Rules (BER) which is a self-describing and self-delimiting format for encoding ASN.1 data structures.

The LDAP client can make several types of requests to the server. First,



LDAP Connection Settings

HOST	ldap.0xdata.loc
PORT	389
SSL-ENABLED	
BIND DN	cn=admin,dc=0xdata,dc=loc
BIND DN PASSWORD	••••••
CONFIRM PASSWORD	••••••
USER BASE DN	ou=users,dc=0xdata,dc=loc
USER BASE FILTER	
USER NAME ATTRIBUTE	uid

Group Settings

GROUP NAMES	jettygroup
GROUP BASE DN	ou=groups,dc=0xdata,dc=loc
GROUP NAME ATTRIBUTE	cn
STATIC MEMBER ATTRIBUTE	memberUid

Figure 6.4: LDAP configuration in Enterprise Steam

the client has to bind to authenticate and specify the protocol version. Then, the client can search the directory or add, modify and delete entries. LDAP can communicate securely using its TLS extension.

6.5.3.5 Configuration

This application uses LDAP to communicate with the customer's LDAP server to check if users are allowed to access the application and validate their password against the LDAP server. The application has to be configured by the LDAP admin who has to supply the relevant information. Figure 6.4 shows a configuration for the domain shown in Figure 6.3.

As a first step, the application needs to bind to the LDAP server. LDAP admin has to provide server **Host** and **Port**. If the LDAP server does not allow anonymous bind requests, the DN of a user that can read entries (**Bind**

DN) and his password (**Bind DN Password**) is required.

When authenticating, the user naturally enters the username, unaware of the DN. The application needs to know the base DN under which the users are located (**User Base DN**). The base DN usually points to an OU that contains all user objects. Additional filter (**User Base Filter**) can be applied to reduce the number of users that can access the application. Since the schema of a user object can differ, the admin has to specify the attribute of the user object that holds the user name (**User Name Attribute**).

The basic configuration allows any users that match the base DN and filter to access the application. In large enterprises that can be undesirable and LDAP admins may want to restrict the application to certain LDAP groups. Name of those groups (**Group Names**) and their base DN (**Group Base DN**) is required to access them. From the group object, we need to know the attribute that holds the group name (**Group Name Attribute**) and attribute that holds the member IDs (**Static Member Attribute**).

6.5.3.6 LDAP login

The LDAP login flow is the following. First, the application receives the user name and password. Application binds to the server using the **Host**, **Port**, **Bind DN** and **Bind DN Password**. Next, a query is executed that tries to find an object in **User Base DN** that matches the **User Base Filter** and the **User Name Attribute** of that object contains the user name. The query must return a single user object.

If the group setting is configured, another query looks under the **Group Base DN** for objects which **Group Name Attribute** equals to **Group Names** and **Static Member Attribute** that contains the user name. Lastly, the DN of the user found in the first step is used to perform a bind using the received password. When the bind is successful, the user's identity is established. Results can be optionally cached for a set amount of time to reduce the load on LDAP servers.

6.5.4 SAML authentication

During the LDAP login flow, the plaintext password is still intercepted by the application because it is entered on the application's login page. To completely decouple authentication from the application a different technology has to be used.

6.5.4.1 SAML

SAML is an open standard to exchange authentication and authorization data between an identity provider (IdP) and a service provider (SP). The SP requests an authentication assertion from the IdP. Based on the assertion, the SP can let the user access the service. Primary use is in the Web SSO when the

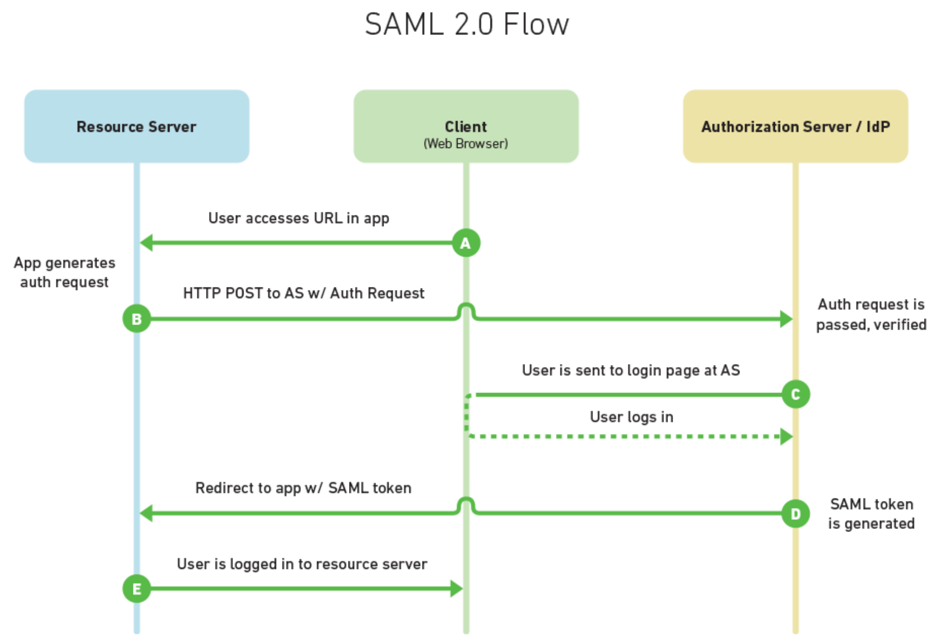


Figure 6.5: SAML 2.0 flow [11]

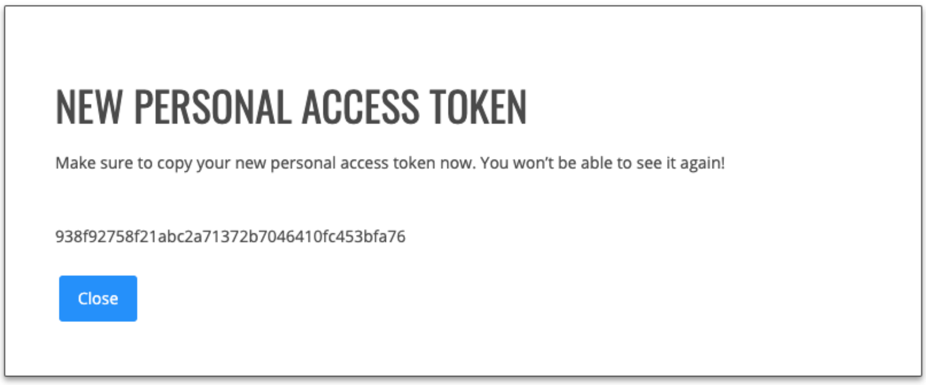
user enters his credentials in a secure website managed by the organization, and the SP only receives the assertion.

There are two types of flows for authorization using SAML. SP initiated SSO begins with a user accessing a protected resource. SP's SSO handler intercepts the request and determines the user needs authentication before accessing the resource. SP is aware of the IdP login page and sends a redirect response back to the user's browser. User is taken to the IdP login page and enters the credentials. IdP creates an assertion and sends the user back to the SP that validates the assertion and grants access. Figure 6.5 shows SP initiated SAML 2.0 flow.

IdP initiated SSO is a little simpler. The flow starts on the IdP portal where a user clicks on a link referencing the protected resource. Still, on the IdP portal a user enters the credentials and after that is sent to the SP with the generated assertion.

6.5.4.2 Configuration

In order to prevent users from forging assertion, the SP has to establish trust with the IdP. The IdP generates metadata that contains its location and public keys among others. SP also needs to provide its public keys. The keys are used to decrypt the assertions and validate the signatures.


NEW PERSONAL ACCESS TOKEN

Make sure to copy your new personal access token now. You won't be able to see it again!

938f92758f21abc2a71372b7046410fc453bfa76

Figure 6.6: User access token

Assertion can contain values and pass any information to the SP. For this application, the assertion has to contain the user name and group name. The attributes are not standardized and need to be specified in the configuration.

6.5.5 Access tokens

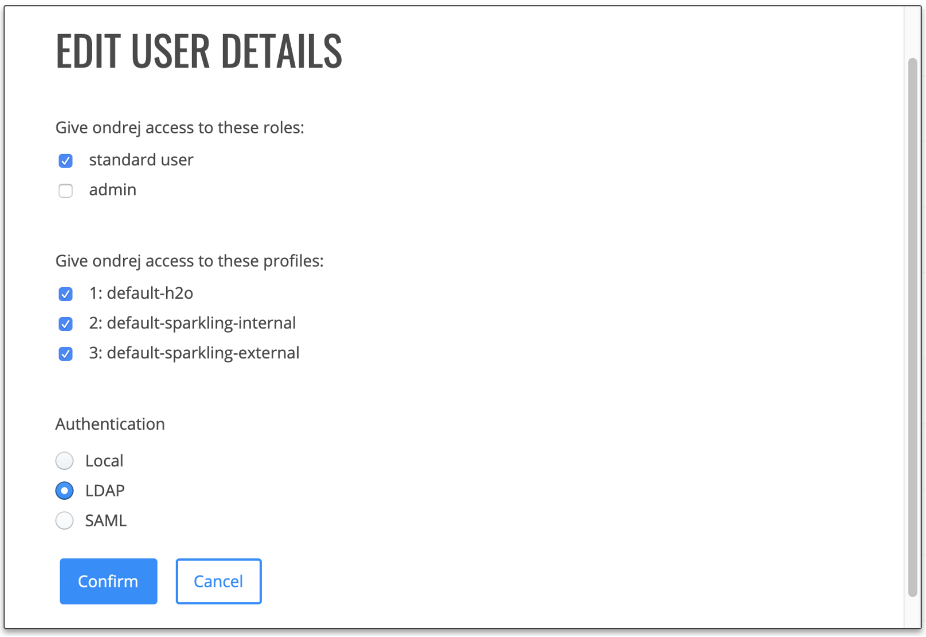
Python and R API clients for this application need to authenticate against the server as well. SAML SSO is for web only and would prevent users from logging in through the API clients. To solve this issue, a user can generate access token after logging in the application through the browser as seen in Figure 6.6. The token can be used instead of the password when authentication through the API clients. Tokens also come in handy in scripts where the user does not want to reveal the real password when LDAP authentication is active.

6.6 Authorization

In systems working with resources, an identified user cannot have access to all resources or actions. Access control systems work by reading the policy definition and enforcing it when users request access to resources. There are many access control models which come with their positives and negatives.

6.6.1 Role-based access control

This application implements Role-based access control (RBAC) to restrict resource access to unauthorized users. RBAC is designed around roles and privileges. Within the application, roles are created for various functions. Roles contain permissions to perform certain operations. Users are assigned roles and through them gain permissions to perform a particular function. Permissions are not assigned directly to users, users gain permissions through their roles which simplifies common administrative operations.


EDIT USER DETAILS

Give ondrej access to these roles:

- [✓] standard user
- [ ] admin

Give ondrej access to these profiles:

- [✓] 1: default-h2o
- [✓] 2: default-sparkling-internal
- [✓] 3: default-sparkling-external

Authentication

Local [✓ LDAP] SAML

Figure 6.7: User management

6.6.2 Role managment

The database is initialized to contain two roles, **admin** with all permissions and the **standard user** with permission suitable for general end-user. Admin can create additional roles with custom permission sets.

The implementation consists of two tables, one for mapping role IDs to permission IDs and another to map identity IDs to role IDs. When handling the request, we can easily check if the user has permissions to perform the action. Additionally, we keep track of resources creates by the user by using a separate table. When accessing said resources, the application automatically checks if the user is authorized to do so.

6.6.3 User managment

When creating local users, the admin can choose which roles to assign them. When LDAP or SAML is active, users are imported automatically and assigned a role that they belong to. On the user management page, users can be manually edited to assign them particular roles, profiles or change the authentication method as shown in Figure 6.7. Users can be disabled to prevent them from logging in and re-enabled later on.

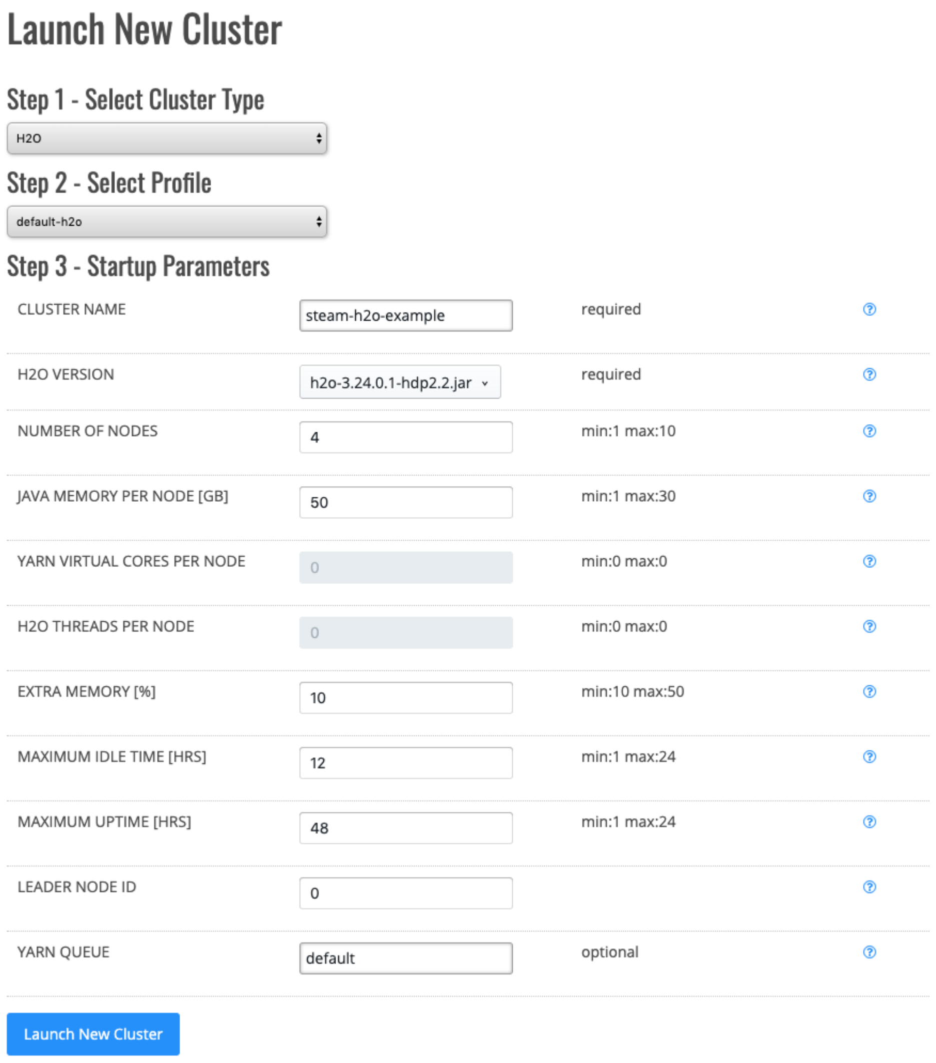
6.7 H_2O integration

The main functionality of this application is to launch H_2O clusters. When a user logs into the application, they see the list of their running clusters and have an option to start a new one.

6.7.1 Startup parameters

As a first step, the user has to select a cluster profile. Profiles are mentioned in section 6.9. As a next step, the user has to fill startup parameters for the cluster. Most of the parameters are related to the resources allocated by the cluster. Figure 6.8 shows the screen where users enters the parameters and starts the H_2O cluster. Following is the list of the parameters and their explanation.

- **Cluster name** is a unique name of the cluster. It can be later used for example to retrieve a connection handle of the cluster in the API.
- **H_2O version** lets users select which version of H_2O to use from the versions that were uploaded by the administrator as mentioned below in subsection 6.7.2.
- The **number of nodes** specifies how many nodes the H_2O cluster needs to have. When launching on Hadoop that many mappers will be started and try to form the H_2O cluster.
- **Java memory per node** specifies how much JVM memory will be allocated for H_2O on each node.
- **YARN virtual cores per node** define how many virtual cores will be assigned to the YARN container running the job. A virtual core is YARN's abstraction of actual cores. One virtual core usually represents one physical core so the application should request the number of cores it can saturate. If the parameter is unset, the YARN will decide and assign a default number of YARN virtual cores to the YARN container.
- **H_2O threads per node** specify how many threads are started for the H_2O's low-priority batch work queue. The parameters can also be left unset to use a default number of threads according to the number of cores.
- **Extra memory** specifies how much more percent of memory will be allocated in the JVM outside of Java heap. The extra native memory is used for algorithms like XGBoost.
- **Maximum idle time** and **maximum uptime** are used to specify the time limits for the cluster. If one of them is exceeded, the cluster is



Launch New Cluster

Step 1 - Select Cluster Type

H2O

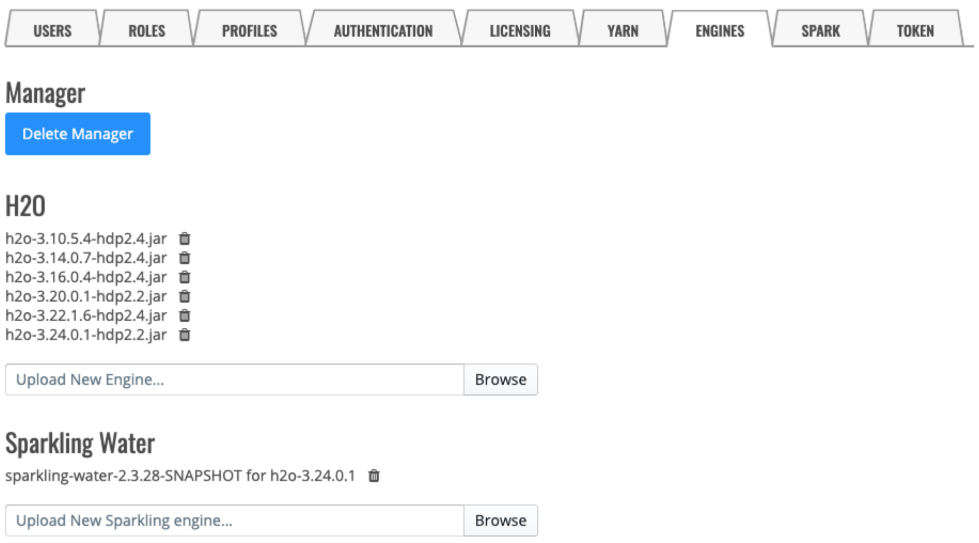
Step 2 - Select Profile

default-h2o

Step 3 - Startup Parameters

CLUSTER NAME	steam-h2o-example	required	?
H2O VERSION	h2o-3.24.0.1-hdp2.2.jar	required	?
NUMBER OF NODES	4	min:1 max:10	?
JAVA MEMORY PER NODE [GB]	50	min:1 max:30	?
YARN VIRTUAL CORES PER NODE	0	min:0 max:0	?
H2O THREADS PER NODE	0	min:0 max:0	?
EXTRA MEMORY [%]	10	min:10 max:50	?
MAXIMUM IDLE TIME [HRS]	12	min:1 max:24	?
MAXIMUM UPTIME [HRS]	48	min:1 max:24	?
LEADER NODE ID	0		?
YARN QUEUE	default	optional	?

Figure 6.8: Screen for launching H_2O clusters



Manager

H2O

h2o-3.10.5.4-hdp2.4.jar
h2o-3.14.0.7-hdp2.4.jar
h2o-3.16.0.4-hdp2.4.jar
h2o-3.20.0.1-hdp2.2.jar
h2o-3.22.1.6-hdp2.4.jar
h2o-3.24.0.1-hdp2.2.jar

Upload New Engine...

Sparkling Water

sparkling-water-2.3.28-SNAPSHOT for h2o-3.24.0.1

Upload New Sparkling engine...

Figure 6.9: H_2O and Sparkling Water engines

forcefully shut down. These measures are needed to stop users from holding Hadoop cluster resources with inactive clusters.

- **Leader node ID** specifies the ID of the node that will receive commands from the user. Usually, the first node should assume the role of the leader, but users may want to change that in order to reproduce models that were trained with a different leader node in the past.

- **YARN queue** specifies the queue name where the job will be submitted. Users may be forced to choose between pre-defined queues or they input the queue name. The parameter can be left unset to submit to a default queue.

6.7.2 Engine

User has the option to choose between multiple versions of H_2O. The admin has control over the available versions in his configuration overview. To register a new H_2O version the admin has to upload the ZIP archive of H_2O distribution built for the corresponding Hadoop version. The application receives the ZIP file and extracts the H_2O driver JAR, Python client, and R client. The H_2O version number is read from the archive and saved into the database along with paths of the other artifacts. Now the engine can be used by any user to launch H_2O clusters. Admins can later delete engines to curate which H_2O versions are being used. Figure 6.9 shows a page for managing H_2O and Sparkling Water engines.

6.7.3 YARN config

H_2O driver and Hadoop accept many startup parameters, but this application only sets the minimal necessary amount when launching the cluster. Sometimes, there are reasons to set other parameters. The application supports specifying additional parameters in the admin configuration. These config entries are set globally, and any new cluster will use these parameters. Existing clusters are not affected. The global parameters can even override the parameters set by the application if necessary.

6.7.4 Starting H_2O clusters

When the user submits a request to launch a cluster the parameters are validated and passed to the command builder. The role of the command builder is to create a Hadoop command that launches the H_2O driver that submits H_2O to YARN.

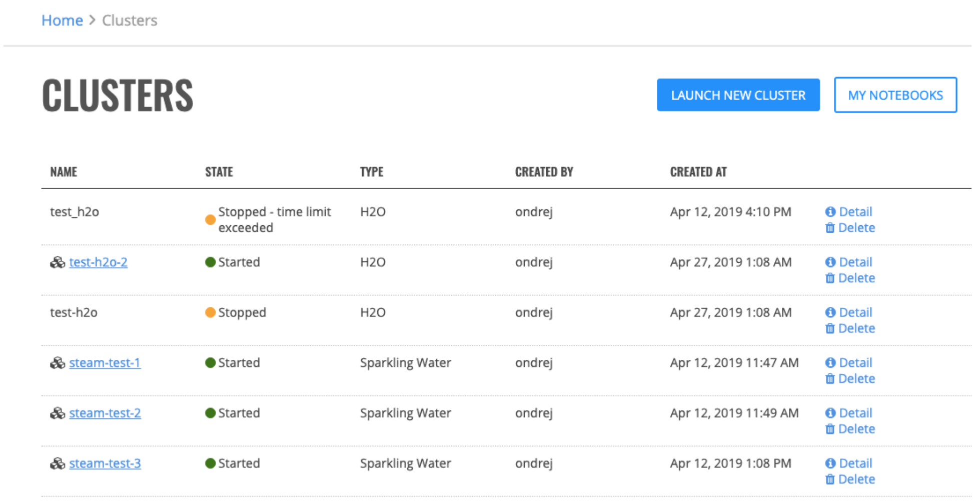
Command builder merges the global configuration with the user configuration, adds the path to the H_2O driver and sets impersonation for the current user. When Kerberos is enabled, principal and keytab file are passed as well. The application generates a password for the H_2O cluster, saves it to the database and exports the hashed password into a hash file. The password is set to protect the cluster from an unauthorized access. Path to the hash file is added to the command making the H_2O cluster password protected after it launches. Finally, the command is executed and monitored.

If there are any errors, they are parsed out of the H_2O driver output and presented to the user. The administrator can see the full log file on the application host machine. If the launch is successful and the H_2O driver exits with status code zero an IP address of one of the nodes is parsed out of the driver output. The application tries to contact the API endpoint of the node to pick up an address of the leader node.

If the node is responding, the cluster has successfully formed, and the IP address of the leader node is added to the reverse proxy configuration. Users can now access H_2O Flow from the cluster page. The cluster is password protected and would normally require a password when connecting to it. However, since the user is authenticated to the application and authorized to access the cluster, the application adds the cluster password it holds in database to basic authentication header when proxying the request thus requiring no additional input from the user.

6.7.5 Monitoring H_2O clusters

The application has two sources of truth when monitoring the H_2O cluster. One option is to monitor the application status through YARN, and the other is to get information directly from the H_2O API. The `yarn` command has to be executed in order to collect application status from YARN. In the case



NAME	STATE	TYPE	CREATED BY	CREATED AT	
test_h2o	● Stopped - time limit exceeded	H2O	ondrej	Apr 12, 2019 4:10 PM	Detail Delete
test-h2o-2	● Started	H2O	ondrej	Apr 27, 2019 1:08 AM	Detail Delete
test-h2o	● Stopped	H2O	ondrej	Apr 27, 2019 1:08 AM	Detail Delete
steam-test-1	● Started	Sparkling Water	ondrej	Apr 12, 2019 11:47 AM	Detail Delete
steam-test-2	● Started	Sparkling Water	ondrej	Apr 12, 2019 11:49 AM	Detail Delete
steam-test-3	● Started	Sparkling Water	ondrej	Apr 12, 2019 1:08 PM	Detail Delete

Figure 6.10: Overview of H_2O and Sparkling water clusters

of Kerberized clusters, a valid ticket is required before executing the `yarn` command. The application has principal and keytab and uses `kinit` to obtain the ticket.

YARN status is the source of truth and if the YARN application is reported to be down, the application will match the internal status and inform the user. When the application is running, the H_2O cluster API endpoint is used to collect information periodically. The information is presented to the user on the cluster detail page and also used to determine if the cluster exceeded the maximum uptime or maximum idle time. Figure 6.10 shows the main screen that has details about H_2O and Sparkling Water clusters launched by the user.

6.7.6 Hadoop helper

Stopping the cluster is an action that cannot be done using the `hadoop` or `yarn` commands. The commands do not support impersonation, preventing the `steam` user from terminating jobs of other users. The only way right now is to use a `hadoop` command to submit a small application that authenticates as the proxy user and shuts down the cluster on his behalf. We call this application a Hadoop helper. Because the Hadoop API keeps changing, the Hadoop helper has to be built for each Hadoop version. The application does not ship with all of them, so they need to be downloaded from the application download page and uploaded through the admin configuration page.

6.7.7 API clients

Majority of the users connect to the H_2O cluster from the Python or R API. Normally they would directly connect to the cluster, but now the application sits between them and the cluster. The `h2osteam` Python package and R library addresses the issue and provides a way to authenticate against the application to access and manage the clusters. The Python package and R library can be downloaded from the application's download page.

A user logs in using the same credentials or optionally using an access token. Then the user has the option to start the cluster or to get a connection handle of an existing cluster using the cluster name as a unique identifier. The connection handle contains the proxy address with an appropriate relative path and user name and password. This config can be passed directly to `h2o.connect` to initiate the connection and users can use the H_2O API to interact with the cluster.

6.8 Sparkling Water integration

The initial version of this application was shipped with the support for H_2O clusters only. After a while, some companies were interested in supporting Sparkling Water. That became the next implementation goal.

6.8.1 Challenges

As we discussed in chapter 4 deploying Sparkling Water cluster is a little bit more complicated. The main difference is that the Sparkling Water API has no `connect` function. Sparkling Water interpreter creates the cluster from a new Spark session and destroys the cluster once the interpreter finishes. There is no notion of a running cluster that can be abandoned and later reconnected. We wanted Sparkling Water clusters to feel the same way as H_2O clusters.

Another challenge was that normally Spark users need to login to the Hadoop edge node and start Sparkling Water cluster from there. The driver runs on the edge node, and users can directly talk to the driver. In this implementation, only the application runs on the edge node and proxies the user request to the cluster. The user is suddenly separated from the driver that now runs in the YARN container, so we needed a way for a user command to reach the Spark driver.

6.8.2 Apache Livy

We found out that the team behind Apache Livy identified the same problem and is solving it with their Livy server. Livy enables multi-tenant submission of Spark jobs from web apps without any Spark client installed. Multiple

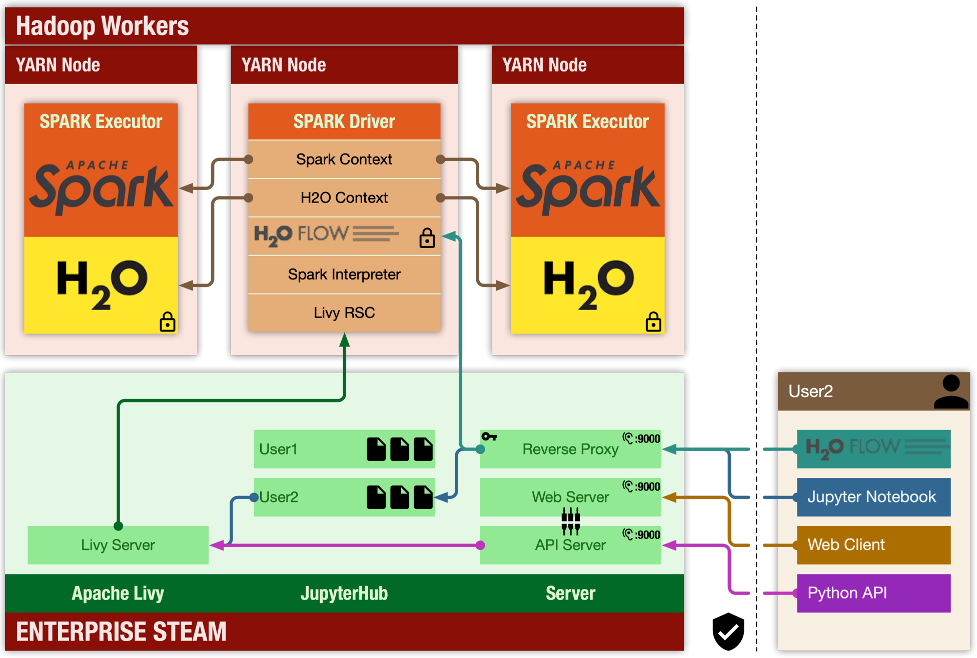


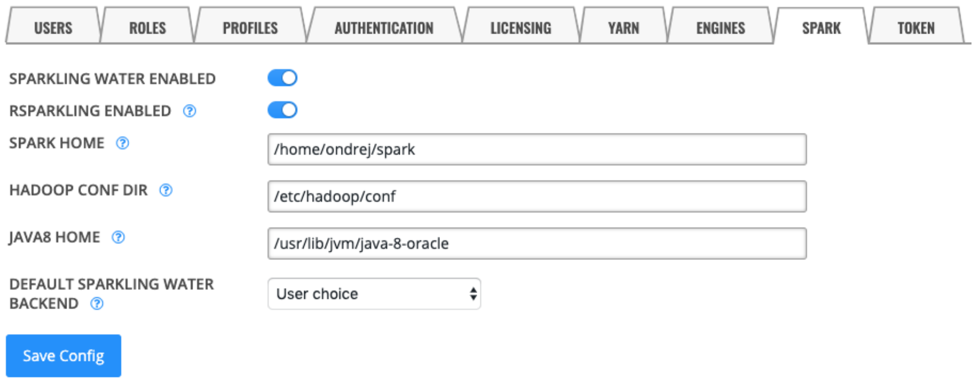
Figure 6.11: Architecture of Sparkling Water integration

users can interact with the Spark cluster remotely, concurrently and in a fault-tolerant fashion.

The Livy server runs on the Hadoop edge node and exposes a REST API to its consumers. This allows us to have long-running Spark Contexts and interact with the Spark Context remotely via a simple REST interface. Behind the scenes when Livy creates a new Spark Session, it initializes a Livy Context in the driver that communicates with the Livy server. User commands are proxied via the Livy server to the Livy Context, executed, and results are sent back.

This application bundles and manages the Livy server. We have implemented the Livy REST API client to communicate with it. This way, we can start a Spark Session and even attach files to make them accessible on the driver. Inside the created Spark session, we can start Sparkling Water to get both Spark Context and H_2O Context ready. Users that connect to the Spark session do not need to import or initialize anything else and can start using the session right away. Figure 6.11 shows the architecture of Sparkling Water integration into the application.

The setup from the application admin perspective is minimal. The application only needs to know the location of the Spark installation, Hadoop configuration and Java Runtime Environment 8. Figure 6.12 shows Sparkling Water configuration screen for administrators.



SPARKLING WATER ENABLED: ON
RSPARKLING ENABLED: ON
SPARK HOME: /home/ondrej/spark
HADOOP CONF DIR: /etc/hadoop/conf
JAVA8 HOME: /usr/lib/jvm/java-8-oracle
DEFAULT SPARKLING WATER BACKEND: User choice

Figure 6.12: Sparkling Water global configuration

6.8.3 Jupyter notebook

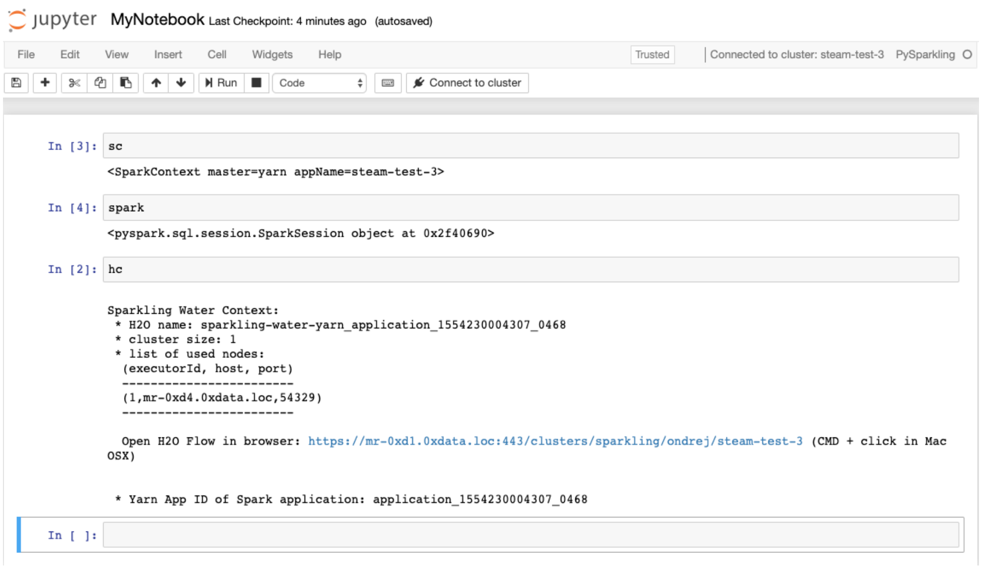
With Livy, we have solved the issue of remote Spark sessions. Now we need an environment that users can access to interact with the cluster. We wanted a notebook environment because it would allow us to have a complete web experience for the user without the need for a command line. We also wanted to support Python, Scala, and R as languages that interact with the Spark cluster.

Jupyter is very familiar notebook environment that allows creating web notebooks that contain live code, equations, visualizations, and narrative text. Jupyter is widely used in the data science world and is very easy to use for the users. We could embed Jupyter notebooks in the application, but we needed a way how to connect the live code in the notebook with the Spark session through the Livy server.

6.8.3.1 Sparkmagic

Sparkmagic project under the Jupyter incubator organization provides a set of Jupyter kernels that can interactively work with remote Spark clusters through Livy. Initially, it looked like a great fit, but it did not offer the cluster persistence feature. Sparkmagic would start a brand new cluster for every new Jupyter notebook and terminate the cluster when the notebook kernel stopped.

In order to get around that, we had to fork the project and make changes to the kernel. Now the cluster does not start with a new notebook, and the cluster does not get terminated when leaving the notebook. Sparkmagic provides an API endpoint that lets us bind one Jupyter notebook to one Spark session. The problem now was how to give that functionality to the user. It could be easily embedded in the applications web UI, but we felt like the user should be able to bind Spark session to the notebook directly in the notebook.


jupyter
MyNotebook Last Checkpoint: 4 minutes ago (autosaved)
File
Edit
View
Insert
Cell
Widgets
Help
Trusted
Connected to cluster: steam-test-3
PySparkling
Run
Code
Connect to cluster

```
In [3]: sc

        <SparkContext master=yarn appName=steam-test-3>

In [4]: spark

        <pyspark.sql.session.SparkSession object at 0x2f40690>

In [2]: hc

        Sparkling Water Context:
         * H2O name: sparkling-water-yarn_application_1554230004307_0468
         * cluster size: 1
         * list of used nodes:
          (executorId, host, port)
          ------------------------
          (1,mr-0xd4.0xdata.loc,54329)
          ------------------------

          Open H2O Flow in browser: https://mr-0xd1.0xdata.loc:443/clusters/sparkling/ondrej/steam-test-3 (CMD + click in Mac
        OSX)


         * Yarn App ID of Spark application: application_1554230004307_0468

In [ ]:
```

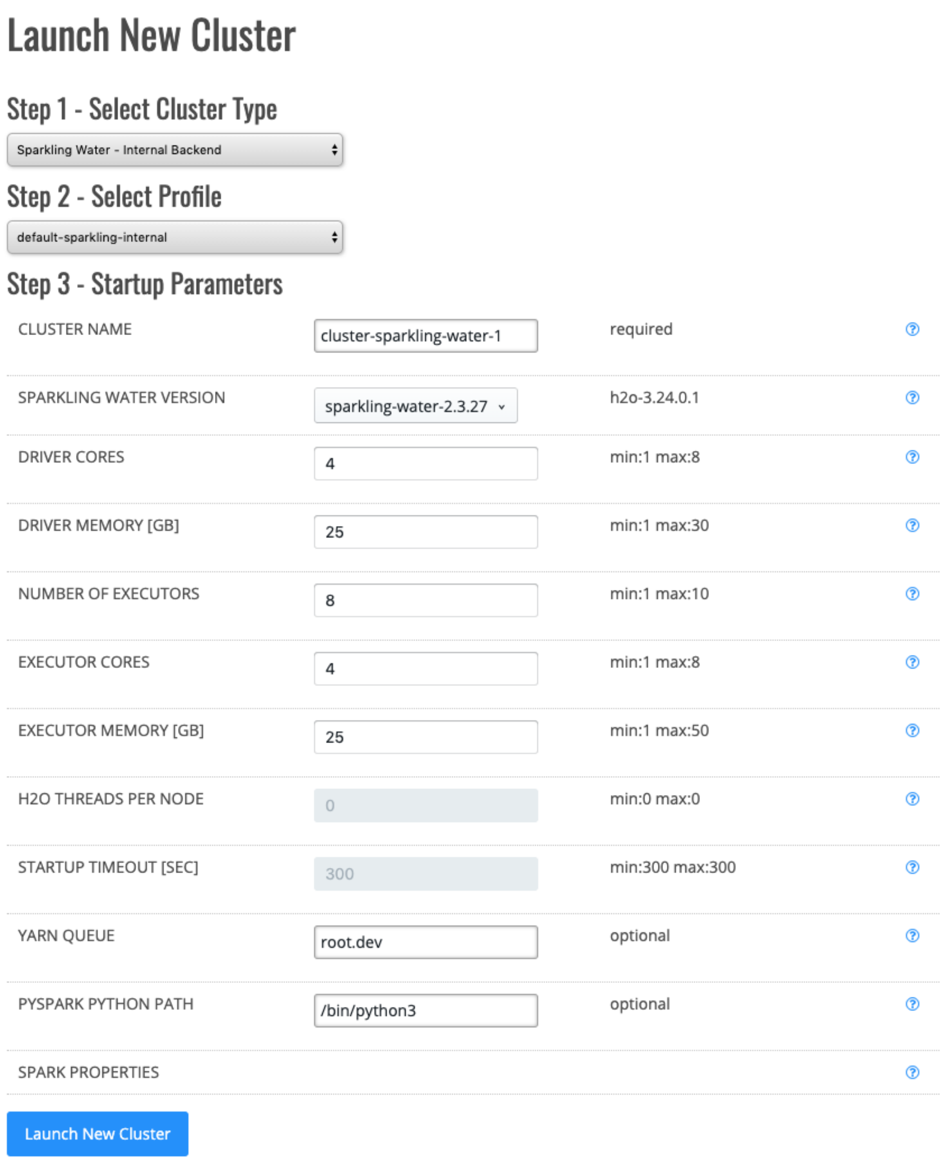
Figure 6.13: Jupyter notebook using Pysparkling kernel

To enable that we had to create an extension to Jupyter notebook that would talk to the application to retrieve the list of running clusters and let the user select which cluster they want to bind the current notebook to. It would also let the user choose which language they want to use in the notebook (Python, Scala or R). We also added a status indicating the cluster connection. Figure 6.13 shows a Jupyter notebook with Pysparkling kernel that is connected to a remote Spark session.

6.8.4 JupyterHub

As a next step, we needed to bring the Jupyter notebook to the multitenant environment of this application. Each user needed to have their environment with notebooks that no one else could access. JupyterHub is a multi-user version of the Jupyter notebook and allows to have separate Jupyter environments on the same server. It is highly customizable and served our needs very well.

We set up JupyterHub to start alongside the main application. Notebooks of each user are persisted on the file system, and the server is managed through the JupyterHub API. The application has administrator access and can manage the users. Since no Jupyter notebook code gets executed on the host as it is all sent to the Livy server for evaluation on a worker node, we did not need to set up any execution environment on the edge node.


Launch New Cluster

Step 1 - Select Cluster Type

Sparkling Water - Internal Backend

Step 2 - Select Profile

default-sparkling-internal

Step 3 - Startup Parameters

CLUSTER NAME	cluster-sparkling-water-1	required
SPARKLING WATER VERSION	sparkling-water-2.3.27	h2o-3.24.0.1
DRIVER CORES	4	min:1 max:8
DRIVER MEMORY [GB]	25	min:1 max:30
NUMBER OF EXECUTORS	8	min:1 max:10
EXECUTOR CORES	4	min:1 max:8
EXECUTOR MEMORY [GB]	25	min:1 max:50
H2O THREADS PER NODE	0	min:0 max:0
STARTUP TIMEOUT [SEC]	300	min:300 max:300
YARN QUEUE	root.dev	optional
PYSPARK PYTHON PATH	/bin/python3	optional
SPARK PROPERTIES		

Figure 6.14: Screen for launching Sparkling Water clusters

6.8.5 Startup parameters

The startup parameters for Sparkling Water cluster are slightly different from H_2O clusters. There are two sets of parameters, one for each Sparkling Water backend. In the internal backend, H_2O inherits most of the executor settings since it runs inside it. In the external backend, H_2O runs outside the executors and needs to be configured separately. Figure 6.14 shows the screen for launching Sparkling Water cluster in internal backend mode. The parameters for the internal backend are shown below.

- **Cluster name** uniquely identifies the Sparkling Water cluster.
- **Sparkling Water version** is a dropdown menu to select which version of Sparkling Water to use.
- **Driver cores** and **driver memory** specifies the resources of the driver.
- The **number of executors**, **executor cores** and **executor memory** specifies the resources of the executors.
- **H_2O threads per node** specify the number of threads for H_2O and can be left unset to use a default value that will match the number of driver/executor cores.
- **Startup timeout** allows the user to select how long to wait for the cluster to start up before failing.
- **YARN queue** lets the user choose or specify which YARN queue to use for this job.
- **PySpark Python path** lets the user choose which Python virtual environment to use. Enterprises usually have multiple Python environments with different packages installed for different purposes.
- **Spark properties** can be used to pass additional configuration to Spark when starting the Spark session.

External backend configuration contains all of the above and adds **H_2O nodes** and **H_2O memory per node** parameters.

6.8.6 Engine

In a similar fashion to H_2O engines, the admin has to first upload the Sparkling water distribution ZIP before the users can launch Sparkling Water clusters. Sparkling Water distribution contains the main Sparkling Water JAR, Python package, and R library. These artifacts are extracted, and their location saved along with Sparkling Water version number to the database.

In order to run Sparkling Water in the internal backend, only the Sparkling Water JAR is needed. The JAR contains a version of H_2O, so no other artifacts are needed. The external backend needs the H_2O engine of the same version as in the Sparkling Water, so both engines have to be present to start Sparkling Water in the external backend.

Both engines also have to present when the user wants an R session to be active. We needed the R libraries from both H_2O and Sparkling Water to properly set up the interpreter.

6.8.7 Starting Sparkling Water clusters

When users send a request to launch a Sparkling Water cluster, the startup parameters are validated and passed along to prepare the Spark session configuration. The configuration contains the cluster name, user impersonation, hash file to secure the H_2O cluster, Kerberos principal and keytab and other details.

A few files have to be uploaded to the Spark session; it is the Sparkling Water JAR, Sparkling Water Python package with dependencies, Sparkling Water R library and the hash file. For external backend, the H_2O driver JAR has to be uploaded and for R session, the H_2O R library as well. The request is prepared and sent to the Livy server.

Livy server starts a new Spark session configured as requested and returns the ID of the session. In this Spark session, we need to start Sparkling Water. We send a Scala code that imports the Sparkling Water JAR and starts the Sparkling Water cluster. This will create a global variable in the session that holds the H_2O Context. Next, we retrieve the IP address of the H_2O node for later use. Now, the Sparkling Water is running, but only the Scala interpreter knows about it. We send a Python code to connect to the Sparkling Water cluster and to get the H_2O Context handle.

Setup for R session is a little more complicated. Sparkling Water R library requires `sparklyr` as backend, and by default, Livy starts only the `rspark` backend. To make this work, we need to initialize the sparklyr gateway in Scala session in a separate thread. In the R session, we load H_2O and Sparkling Water R library and connect to the sparklyr backend and retrieve the H2OContext.

When Scala, Python and R interpreters all hold the Spark Context and H2OContext the Sparkling Water cluster has started. Sparkling Water clusters are not proxied using HAProxy but instead rely on the reverse proxy in the Go standard library. This reverse proxy also handles the JupyterHub access.

6.8.8 API clients

Users that do not prefer the Jupyter notebook can still use the `h2osteam` Python package to start and interact with the Sparkling Water cluster. The authentication and management part is the same as with just the H_2O clusters. The only difference is in the interaction with the cluster. In H_2O the users use the H_2O package to connect to the cluster and use it directly. Here the users enter an interactive session which is a REPL that sends all the command to the Livy server for evaluation and returns the result.

6.9 Cluster profiles

Initially, the application would let the users define any cluster startup parameters. This way users had total freedom and could start a cluster that uses as many resources as the YARN queue would let them. Problems started to arise when users would allocate excessive resources, leave clusters running unused or start new clusters without shutting down the previous ones.

6.9.1 Templates

Feedback was suggesting that a templating feature might be necessary. Users would have access to several templates which contain predefined sets of startup parameters. User would just choose the template and start the cluster. Application admins could set up templates for different use cases (small, medium, large clusters) and assign them to corresponding users.

We felt like templates might be too restrictive as some jobs require fine-tuning of parameters and some power users still wanted to be in control. At the same time, we wanted admins to be able to set up templates in a situation where it is necessary.

6.9.2 Cluster profiles

We came up with cluster profiles. Profiles are set up by the administrator and impose constraints on every single cluster launch parameter. Every parameter has its minimum, maximum and default value. Default values are important for users that do not want to bother or do not have the knowledge to do the fine-tuning. When launching the cluster, users can see the constraints and whether they are exceeding them.

In a special case when minimum and maximum constraint is the same, the user choice is locked to that value mimicking the template feature. In this implementation, some parameters can be locked and other adjustable by the user. We felt like that would give admins the highest level of flexibility.

Profiles are also imposed in the Python package and R library. Users can preview and select which profile to use to launch the cluster. Then users have to specify only the parameters that are different from the default value as long as they are within the constraints of the profile.

6.9.3 Shared parameters

There are three different cluster profile types, one for H_2O and one for each Sparkling Water backend. Each profile type has a different set of parameters, corresponding to what cluster it is launching. However, some meta parameters are shared across them.

A profile can contain a list of YARN queues and force the user to pick one from the dropdown menu. Hadoop administrators are therefore certain where the application gets submitted. Profiles limit the resources consumed by one cluster; however, there was no limitation on the number of running clusters. We added an option to limit the number of running cluster per profile per user.

Shortly after introducing profiles the companies wanted a tool to help them assign profiles to users because they always had to go to the user profile page and manually assign it there. Customers already had users assigned to particular internal groups so it would be appropriate to use this information. We added a feature that would let admins specify which LDAP or SAML groups have access to each profile.

As for preventing the users from leaving their clusters running for an extensive period of time, we have implemented a profile parameter that would impose a limit on the maximum running time of the cluster and on maximum idle time of the cluster. Figure 6.15 shows and example of H_2O cluster profile.

6.10 RPC

The application handles requests from different clients; it can be the web client, Python client or R library. Implementing the clients in TypeScript, Python, R and keeping them synchronized with the server would require much manual effort. We had to look at an automated solution to this problem.

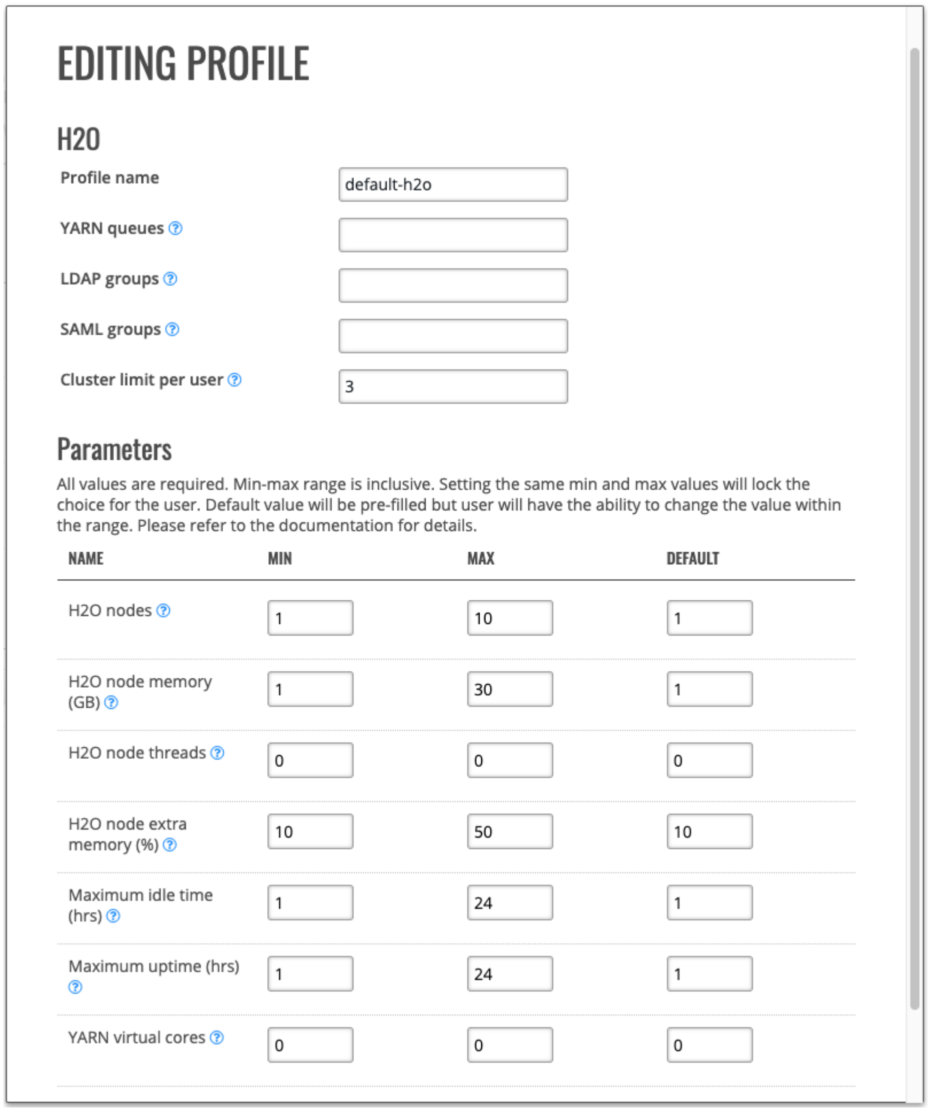
6.10.1 RPC framework

We needed a simple RPC framework that would allow us to generate clients and server with ease. There are many existing solutions. One of the more popular ones is gRPC which is more suitable for service to service communication and uses a custom protocol over HTTP/2. Not all browsers support HTTP/2 which is a dealbreaker for use case within this application. There is also no official support for R. The RPC framework would need to work on HTTP/1.1 with simple client logic.

We decided to go with `gorilla/rpc` package that uses HTTP/1.1 and standard POST requests. The data payload is JSON encoded and based on the method name in the payload routed to the corresponding handler. Clients can quickly implement sending requests just by correctly encoding the data into JSON.

6.10.2 Service definition

The application has to provide a service definition that defines all the messages, their input, and their output. Such definitions serve as a single source of truth and are used to create the clients and server handlers. Popular serialization format Protobuf has its own Proto language to define the messages.



EDITING PROFILE

H2O

Profile name	default-h2o
YARN queues	
LDAP groups	
SAML groups	
Cluster limit per user	3

Parameters

All values are required. Min-max range is inclusive. Setting the same min and max values will lock the choice for the user. Default value will be pre-filled but user will have the ability to change the value within the range. Please refer to the documentation for details.

NAME	MIN	MAX	DEFAULT
H2O nodes	1	10	1
H2O node memory (GB)	1	30	1
H2O node threads	0	0	0
H2O node extra memory (%)	10	50	10
Maximum idle time (hrs)	1	24	1
Maximum uptime (hrs)	1	24	1
YARN virtual cores	0	0	0

Figure 6.15: Editing H_2O cluster profile

Our RPC framework only supports the JSON encoding, so we had to create our service definition format.

The generator would be written in Go, so we wanted the service definition to be easy to work with. We went with Go structure types that can be easily parsed as we tried with other helper tools. All of the messages are stored in one big struct type. Field names define the message names, and the field type holds another struct type with input and output fields. Input and output fields are separated with a dummy field. This gives us a comprehensive service definition that can be used to generate clients and server.

6.10.3 Generating clients and server

To generate clients in Typescript, Python, R and server handlers in Go we created a `piping` utility. This utility does not parse the Go source code as mentioned before. The service definition is more complicated and can contain nested types. Such relations are hard to analyze just from the source code. We had to use reflection to analyze the service definition.

Reflection in Go is an ability to manipulate objects with arbitrary types during run-time. We use reflection to examine the service struct definition and extract all the input and output fields. Once we have a definite structure, we can start generating code. Each language has its own Go template to start with. We have to convert Go types to types of other languages and prepare JSON encoding for every message. The generated server has an interface containing every message, and the application only compiles when all handlers are implemented.

The process of generating clients and server preserves the type safety, and if the schema changes, we can regenerate clients, server and any errors are caught during the compilation. Code generation again saves much time and prevents copy-paste errors.

6.11 Building, packaging and installation

Over the months, the application grew in size and started bundling other services. The main goal was to have a predictable, reproducible build that can be packaged, installed and run on a variety of Linux distributions.

6.11.1 Build

The main distribution target was Red Hat Enterprise Linux (RHEL) as this is the standard Linux distribution in the commercial market. We targeted RHEL 7 and RHEL 6 even though it is approaching the end of life. Another distribution target was Ubuntu, and we chose Ubuntu Precise (12.04) to match the Linux kernel version between minimal supported Linux distributions.

Ubuntu and RHEL have separate Docker images used for the build. Both images are based on the minimal supported version. Each image has to have tools used to compile the entire project. One of the requirements is Node JS version 8 or higher which is used to build the website. We also need a GCC compiler and Go version 1.10 or higher to build the server.

In order to run Jupyter notebook and JupyterHub, we need Python version 3.5 or higher. That poses an issue since RHEL 6 does not contain any Python 3 version and there is no Python 3 downloadable package for RHEL 6. Linux administrator can manually install the required version of Python, but we felt that it would be a very high requirement that could affect the adoption of the product. Instead, we build Python 3.5 from the source inside the Docker image. As the last step, we copy the application source code inside the image.

The new Docker images are then used to build multiple parts of the application. In the first step, the web application is built using NPM and Webpack to produce `bundle.js` and `index.html` files. Next, we build the server and bundle the frontend into the server binary. That allows us to serve the files directly from memory instead of the file system. At this time we have the server binary; however, we are still missing other services that are bundled with the server.

Next step is to build secondary parts of the application. We clone and build the forked `sparkmagic` project and our custom Jupyter notebook extension along with an official release of Apache Livy. We also download and build `configurable-http-proxy` which is a JupyterHub dependency and download the `node` binary to run it.

The application needs to create a Python virtual environment which runs the JupyterHub and Jupyter notebook. At this point, we create a new virtual environment from the previously built Python 3.5, activate it and install JupyterHub and Jupyter notebook inside the environment. We also install our custom Jupyter extension and Sparkling Water JupyterHub kernels. As the last step, we package the Python virtual environment with the application.

6.11.2 Packaging and installation

The application has to be one installable package that the Linux admin can install on the system and upgrade when new releases come out. RHEL uses RPM packages while Ubuntu uses DEB packages. To build packages in both formats, we used FPM which is a tool to help build packages for multiple platforms.

Before actually building the package we had to assemble an archive that has all the parts of the application in the right place. The archive is put together in a way that when installed, it extracts the server binary under `/opt/h2oai/steam` and other parts of the application in the subdirectories. The configuration file is placed in the `/etc/steam` directory. We also specify install scripts that are executed in some stages of the package installation.

Before the installation, we create a new system user and group called `steam` and after installation change ownership of all the application files to that user. Before upgrading, the application is automatically stopped then immediately restarted.

6.11.2.1 SysVinit

SysVinit is the original Linux init system that runs as the first process to bring the Linux system up. On Ubuntu and RHEL 6 we use SysVinit to start and stop the application. We do that by placing an ordinary Bash script in the `/etc/init.d` directory. This Bash script can be used to start, stop and retrieve the status of the application. When the application is started the script places the application PID in the `/var/run` directory in order to monitor the status and to know what process to kill when the application has to stop.

6.11.2.2 systemd

Systemd is a software suite that among other things manages user processes. On RHEL 7 we use systemd to start, stop and monitor the application. In SysVinit we had to prepare a Bash script that had to contain all the logic to start and manage the process. With systemd, we only need to provide a basic configuration that contains the location of the binary and name of the user that starts the process. We can even let systemd restart the application if it shuts down unexpectedly.

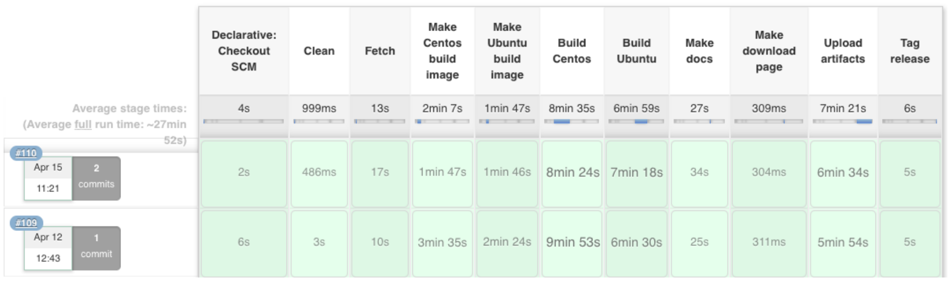
In the end, after building and packaging the application we have RHEL 6 RPM package, RHEL 7 RPM package, and Ubuntu DEB package.

6.12 Documentation

A project of this size has to have good documentation for administrators and end users. We use Sphinx documentation generator which is very popular in the Python community. H_2O uses Sphinx across all its products, and we even have a custom color scheme in the yellow colors of H_2O.

The documentation source uses the RST format which is rich enough to satisfy all the needs of technical documentation. Compiling the documentation source produces PDF documents and HTML representation that we host on our documentation page `http://docs.h2o.ai/`.

Documentation is split into two parts. The first part focuses on installing and setting up the application. It is intended for the administrators that want to install the application for the first time and set it up for the end-users. It goes over installation steps and basic configuration in detail and with pictures where necessary. We continually improve the documentation based on the feedback we receive. The latest version of the installation documentation can



	Declarative: Checkout SCM	Clean	Fetch	Make Centos build image	Make Ubuntu build image	Build Centos	Build Ubuntu	Make docs	Make download page	Upload artifacts	Tag release
Average stage times: (Average full run time: ~27min 52s)	4s	999ms	13s	2min 7s	1min 47s	8min 35s	6min 59s	27s	309ms	7min 21s	6s
#110 Apr 15 11:21 2 commits	2s	486ms	17s	1min 47s	1min 46s	8min 24s	7min 18s	34s	304ms	6min 34s	5s
#109 Apr 12 12:43 1 commit	6s	3s	10s	3min 35s	2min 24s	9min 53s	6min 30s	25s	311ms	5min 54s	5s

Figure 6.16: Jenkins pipleline dashboard

be found on the following page: `http://docs.h2o.ai/enterprise-steam/latest-stable/docs/install-docs/_build/html/index.html`

The second part of the documentation focuses on the end-user. It talks about how to login to the application, start and manage the clusters. It also contains documentation for Python package and R library. The latest version of the user documentation can be found on the following page: `http://docs.h2o.ai/enterprise-steam/latest-stable/docs/user-docs/_build/html/index.html`

6.13 Continous integration

At H_2O we use Jenkins to automate builds and do continuous integration. This application has a Jenkinsfile that allows us to build, package and release the application. Figure 6.16 shows a dashboard of the Jenkins pipeline that automates the release. Whenever we want to trigger a release, we command Jenkins and select a GIT branch from which the release is built.

First, Jenkins prepares the Docker image and then uses it to produce RPM and DEB packages. As a next step Jenkins builds the documentation in the HTML and PDF format. Jenkins also builds the Python package and R library. As the last step, we construct a simple HTML download page that has links to download all the parts of the software and links to the documentation. In the end, all build artifacts are uploaded to a public AWS S3 bucket and link to the download page is published on the H_2O.ai website. When the build finishes successfully it is tagged in the project's GIT repository. The link to the download page of the latest version of this application can be found on this page `https://s3.amazonaws.com/steam-release/enterprise-steam/latest-stable.html`.

Conclusion

The goal of the initial effort was to implement an enterprise platform for deploying and managing H_2O clusters on Hadoop. This was accomplished with a full release of Enterprise Steam in April of 2018.

The release featured a multitenant service secured by LDAP authentication with a web UI and Python/R API client libraries. Through this interface, users could start and monitor H_2O clusters in Kerberos protected Hadoop environment. The application securely submits requests to YARN on behalf of authenticated users (impersonation). The release also included thorough documentation and a Jenkins pipeline for automated and reproducible release.

Over the next year, the platform evolved based on the feedback of the users. Currently, it supports SAML authentication and can launch Sparkling Water clusters as well. Administrators can newly create profiles that control the parameters of launched clusters. Cloud deployment capabilities are currently under development.

Nowadays, Enterprise Steam is used by companies such as Paypal, Mastercard, AT&T, Wells Fargo, Equifax or Aetna. The software also passed a comprehensive security review from Citibank and is being utilized in their environment. In total, approximately hundreds of data scientists and Hadoop administrators rely daily on this application.

Next evolution of Enterprise Steam will focus on better UI/UX experience and efforts to unify the whole H_2O ecosystem by managing deployments of Driverless AI which is a platform for automatic machine learning.

Bibliography

[1] BMC. An Introduction to Hadoop Architecture. Apr 2017, [online]. [Accessed 2019-04-25]. Available from: `https://www.bmc.com/guides/hadoop-architecture.html`

[2] Apache. Apache Hadoop YARN. Nov 2018, [online]. [Accessed 2019-04-25]. Available from: `https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html`

[3] Axway. Kerberos authentication. Jan 2019, [online]. [Accessed 2019-04-25]. Available from: `https://docs.axway.com/bundle/APIGateway_762_IntegrationKerberos_allOS_en_HTML5/page/Content/KerberosIntegration/kerberos_overview.htm`

[4] H2O. Introduction to H2O. Oct 2018, [online]. [Accessed 2019-04-06]. Available from: `https://github.com/h2oai/h2o-meetups/blob/master/2018_10_24_Prague/Praha_MFF.pdf`

[5] Kraljevic, T. H2O on Hadoop. Dec 2013, [online]. [Accessed 2019-04-25]. Available from: `https://github.com/h2oai/h2o-meetups/blob/master/2013_12_12_H2O_on_Hadoop/h2o_on_hadoop_2013_12_12_meetup.pdf`

[6] H2O. How H2O runs on YARN. Apr 2019, [online]. [Accessed 2019-04-25]. Available from: `http://docs.h2o.ai/h2o/latest-stable/h2o-docs/welcome.html`

[7] Kirillov, A. Apache Spark: core concepts, architecture and internals. Mar 2016, [online]. [Accessed 2019-04-26]. Available from: `http://datastrophic.io/core-concepts-architecture-and-internals-of-apache-spark/`

[8] Cloudera. Running Spark Applications on YARN. Dec 2018, [online]. [Accessed 2019-04-26]. Available from: `https://www.cloudera.com/`

documentation/enterprise/5-4-x/topics/cdh_ig_running_spark_on_yarn.html

[9] Háva, J. Sparkling Water design. Apr 2019, [online]. [Accessed 2019-04-26]. Available from: http://docs.h2o.ai/sparkling-water/2.4/latest-stable/doc/design/design.html

[10] Maj, W. React Lifecycle Methods diagram. Apr 2019, [online]. [Accessed 2019-04-26]. Available from: http://projects.wojtekmaj.pl/react-lifecycle-methods-diagram/

[11] Dennis, Z. Choosing an SSO Strategy: SAML vs OAuth2. Jan 2018, [online]. [Accessed 2019-04-26]. Available from: https://www.mutuallyhuman.com/blog/2013/05/09/choosing-an-sso-strategy-saml-vs-oauth2/

[12] Apache. HDFS Architecture Guide. Sep 2018, [online]. [Accessed 2019-04-25]. Available from: http://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html

[13] Apache. Hadoop in Secure Mode. Nov 2018, [online]. [Accessed 2019-04-25]. Available from: https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-common/SecureMode.html

[14] H2O.ai. H2O Architecture. Apr 2019, [online]. [Accessed 2019-04-09]. Available from: http://docs.h2o.ai/h2o/latest-stable/h2o-docs/architecture.html

[15] Click, C. H2O-3 Javadoc for class water.fvec.Vec. Apr 2019, [online]. [Accessed 2019-04-06]. Available from: http://docs.h2o.ai/h2o/latest-stable/h2o-core/javadoc/water/fvec/Vec.html

[16] Nykodym, T.; Maj, P. Fast Analytics on Big Data with H2O. Nov 2014, [online]. [Accessed 2019-04-06]. Available from: http://gotocon.com/dl/goto-berlin-2014/slides/PetrMaj_and_TomasNykodym_FastAnalyticsOnBigData.pdf

[17] H2O. h2o-genmodel version 3.24.0.2 API. Apr 2019, [online]. [Accessed 2019-04-06]. Available from: http://docs.h2o.ai/h2o/latest-stable/h2o-genmodel/javadoc/index.html

[18] Háva, J. Data Sharing. Apr 2019, [online]. [Accessed 2019-04-26]. Available from: http://docs.h2o.ai/sparkling-water/2.4/latest-stable/doc/design/data_sharing.html

[19] IBM. IBM Spectrum Conductor. Apr 2019, [online]. [Accessed 2019-04-30]. Available from: https://www.ibm.com/us-en/marketplace/spark-workload-management

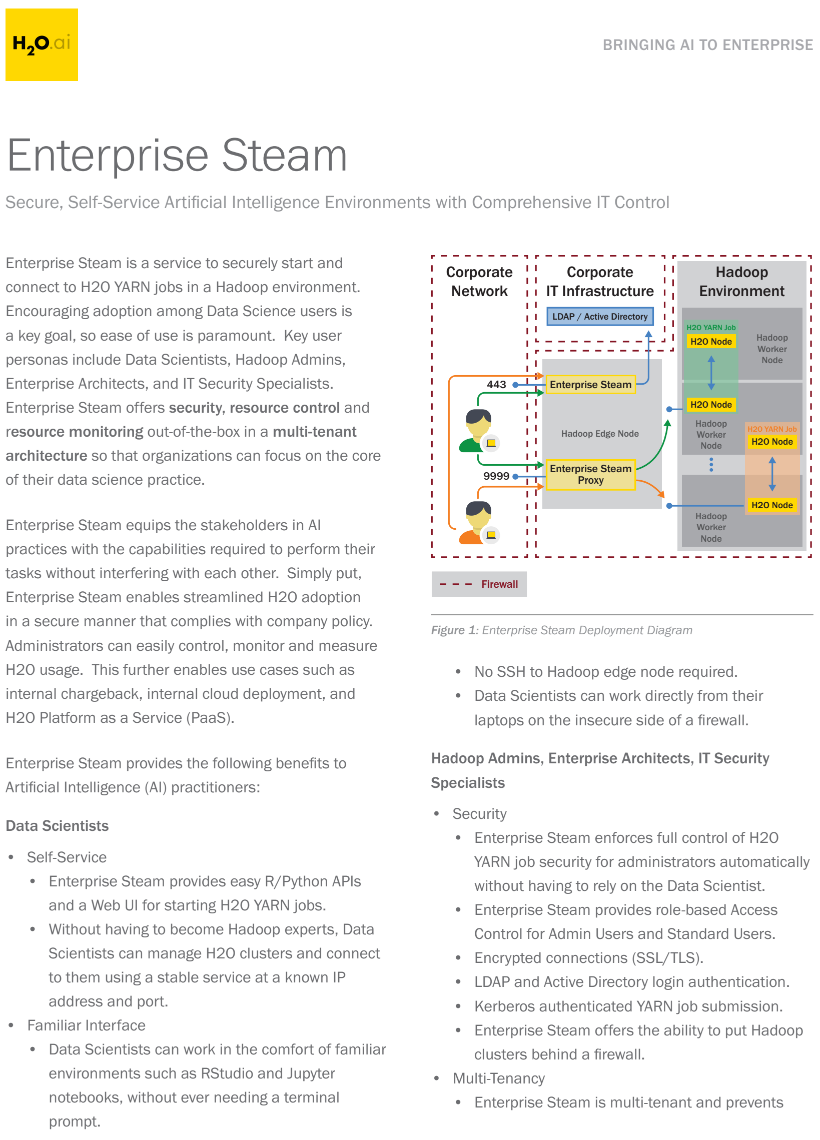
[20] BlueData. The BlueData EPIC. Apr 2019, [online]. [Accessed 2019-04-30]. Available from: `https://www.bluedata.com/product/`

[21] Qubole. Qubole. Apr 2019, [online]. [Accessed 2019-04-30]. Available from: `https://www.qubole.com/`

Appendix A

Enterprise Steam Datasheet

This page is intentionally left blank.



H₂O.ai

BRINGING AI TO ENTERPRISE

Enterprise Steam

Secure, Self-Service Artificial Intelligence Environments with Comprehensive IT Control

Enterprise Steam is a service to securely start and connect to H2O YARN jobs in a Hadoop environment. Encouraging adoption among Data Science users is a key goal, so ease of use is paramount. Key user personas include Data Scientists, Hadoop Admins, Enterprise Architects, and IT Security Specialists. Enterprise Steam offers **security, resource control** and **resource monitoring** out-of-the-box in a **multi-tenant architecture** so that organizations can focus on the core of their data science practice.

Enterprise Steam equips the stakeholders in AI practices with the capabilities required to perform their tasks without interfering with each other. Simply put, Enterprise Steam enables streamlined H2O adoption in a secure manner that complies with company policy. Administrators can easily control, monitor and measure H2O usage. This further enables use cases such as internal chargeback, internal cloud deployment, and H2O Platform as a Service (PaaS).

Enterprise Steam provides the following benefits to Artificial Intelligence (AI) practitioners:

Data Scientists

- Self-Service
  - Enterprise Steam provides easy R/Python APIs and a Web UI for starting H2O YARN jobs.
  - Without having to become Hadoop experts, Data Scientists can manage H2O clusters and connect to them using a stable service at a known IP address and port.
- Familiar Interface
  - Data Scientists can work in the comfort of familiar environments such as RStudio and Jupyter notebooks, without ever needing a terminal prompt.

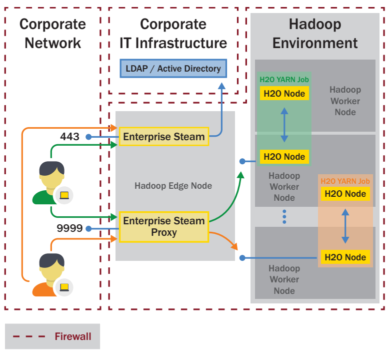


Figure 1: Enterprise Steam Deployment Diagram

  - No SSH to Hadoop edge node required.
  - Data Scientists can work directly from their laptops on the insecure side of a firewall.

Hadoop Admins, Enterprise Architects, IT Security Specialists

- Security
  - Enterprise Steam enforces full control of H2O YARN job security for administrators automatically without having to rely on the Data Scientist.
  - Enterprise Steam provides role-based Access Control for Admin Users and Standard Users.
  - Encrypted connections (SSL/TLS).
  - LDAP and Active Directory login authentication.
  - Kerberos authenticated YARN job submission.
  - Enterprise Steam offers the ability to put Hadoop clusters behind a firewall.
- Multi-Tenancy
  - Enterprise Steam is multi-tenant and prevents

Data Science users from accessing each other's H2O jobs (and data).

- Resource Control – Enterprise Steam allows Hadoop admins to:
  - Control which H2O versions are available.
  - Specify which YARN queue to use.
  - Cap the resources the Data Scientist can use.
  - Stop H2O jobs via a convenient Web UI.
- Resource Monitoring
  - Enterprise Steam provides Hadoop admins with monitoring capabilities to find dormant jobs tying up memory.
  - Enterprise Steam provides mechanisms for H2O usage measurement to enable chargeback and compliance use cases.

Key Benefits

- Next level operational efficiency in your AI environments with out-of-box **security, resource control**, and **resource monitoring**
- Data Scientists can freely and safely practice data science in their own H2O cluster since Enterprise Steam is **multi-tenant**
- Enterprise Steam enables new operation models such as internal chargeback, **internal cloud deployment,** and **H2O Platform as a Service (PaaS)**

License

- Annual Commercial License

System Requirements

- **OS:** Ubuntu 12.04 and up; RHEL 6.7 and up
- **Hadoop Distro:** All CDH, HDP and MAPR distributions from the last 2 years [Adding Support for new versions is typically very fast (within days).]
- Databases: SQLite or Postgres SQL [Support for additional databases to be added upon customer request.]

Appendix B

Acronyms

AI Artificial Intelligence

HDFS Hadoop Distributed File System

YARN Yet Another Resource Negotiator

SW Sparkling Water

MR MapReduce

AM Application Master

RM Resource Manager

K/V Key/Value

CoxPH Cox Proportional Hazards

DRF Distributed Random Forest

DNN Deep Neural Network

GLM Generalized Linear Model

GBM Gradient Boosting Machine

GLRM Generalized Low Rank Models

PCA Principal Component Analysis

POJO Plain Old Java Object

MOJO Model Object Optimized

REST Representational State Transfer

API Application Programming Interface

JVM Java Virtual Machine

JIT Just In Time Compiler

DOS Denial Of Service

DDOS Distributed Denial Of Service

KDC Key Distribution Center

TGS Ticket Granting Server

TGT Ticket Granting Ticket

AS Authentication Server

Appendix C

Contents of enclosed CD

```
README ............................ the file with CD contents description
src ................................. the directory of the source codes
  enterprise-steam ....... the directory of the application source code
  thesis .......... the directory of the LaTeX source codes of the thesis
DP_Bilek_Ondrej_2019.pdf .............. the thesis text in PDF format
```