Problem​ ​Statement

Explain in Brief:

● The workflow of Oozie and its Benefits

Apache Oozie is an open source project based on Java™ technology that simplifies the process of creating workflows and managing coordination among jobs. In principle, Oozie offers the ability to combine multiple jobs sequentially into one logical unit of work. One advantage of the Oozie framework is that it is fully integrated with the Apache Hadoop stack and supports Hadoop jobs for Apache MapReduce, Pig, Hive, and Sqoop. In addition, it can be used to schedule jobs specific to a system, such as Java programs. Therefore, using Oozie, Hadoop administrators are able to build complex data transformations that can combine the processing of different individual tasks and even sub-workflows. This ability allows for greater control over complex jobs and makes it easier to repeat those jobs at predetermined periods.

In practice, there are different types of Oozie jobs:

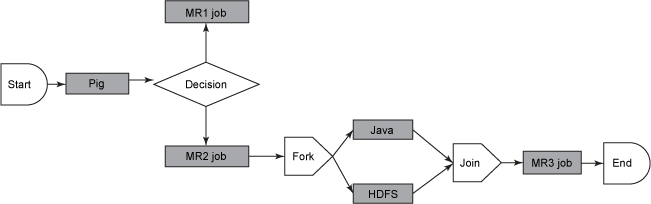
* *Oozie Workflow* jobs — Represented as directed acyclical graphs to specify a sequence of actions to be executed.
* *Oozie Coordinator* jobs — Represent Oozie workflow jobs triggered by time and data availability.
* *Oozie Bundle*— Facilitates packaging multiple coordinator and workflow jobs, and makes it easier to manage the life cycle of those jobs.

## How does Oozie work?

An Oozie workflow is a collection of actions arranged in a directed acyclic graph (DAG). This graph can contain two types of nodes: control nodes and action nodes. *Control nodes*, which are used to define job chronology, provide the rules for beginning and ending a workflow and control the workflow execution path with possible decision points known as fork and join nodes. *Action nodes* are used to trigger the execution of tasks. In particular, an action node can be a MapReduce job, a Pig application, a file system task, or a Java application. (The shell and ssh actions have been deprecated).

Oozie is a native Hadoop stack integration that supports all types of Hadoop jobs and is integrated with the Hadoop stack. In particular, Oozie is responsible for triggering the workflow actions, while the actual execution of the tasks is done using Hadoop MapReduce. Therefore, Oozie becomes able to leverage existing Hadoop machinery for load balancing, fail-over, etc. Oozie detects completion of tasks through callback and polling. When Oozie starts a task, it provides a unique callback HTTP URL to the task, and notifies that URL when it is complete. If the task fails to invoke the callback URL, Oozie can poll the task for completion. Figure 1 illustrates a sample Oozie workflow that combines six action nodes (Pig scrip, MapReduce jobs, Java code, and HDFS task) and five control nodes (Start, Decision control, Fork, Join, and End). Oozie workflows can be also parameterized. When submitting a workflow job, values for the parameters must be provided. If the appropriate parameters are used, several identical workflow jobs can occur concurrently.

##### Figure 1. Sample Oozie workflow



In practice, it is sometimes necessary to run Oozie workflows on regular time intervals, but in coordination with other conditions, such as the availability of specific data or the completion of any other events or tasks. In these situations, Oozie Coordinator jobs allow the user to model workflow execution triggers in the form of the data, time, or event predicates where the workflow job is started after those predicates get satisfied. The Oozie Coordinator can also manage multiple workflows that are dependent on the outcome of subsequent workflows. The outputs of subsequent workflows become the input to the next workflow. This chain is called a *data application pipeline*.

Oozie workflow definition language is XML-based and it is called the *Hadoop Process Definition Language.* Oozie comes with a command-line program for submitting jobs. This command-line program interacts with the Oozie server using REST. To submit or run a job using the Oozie client, give Oozie the full path to your workflow.xml file in HDFS as a parameter to the client. Oozie does not have a notion of global properties. All properties, including the *jobtracker* and the *namenode*, must be submitted as part of every job run. Oozie uses an RDBMS for storing state.

## Oozie in action

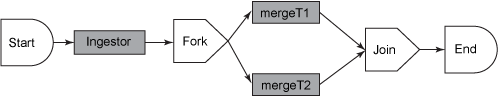
Use an Oozie workflow to run a recurring job. Oozie workflows are written as an XML file representing a directed acyclic graph. Let's look at the following simple workflow example that chains two MapReduce jobs. The first job performs an initial ingestion of the data and the second job merges data of a given type.

##### Listing 1. Simple example of Oozie workflow

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64 | <workflow-app xmlns='uri:oozie:workflow:0.1' name='SimpleWorkflow'>      <start to='ingestor'/>      <action name='ingestor'>          </java>              <job-tracker>${jobTracker}</job-tracker>              <name-node>${nameNode}</name-node>              <configuration>                  <property>                      <name>mapred.job.queue.name</name>                      <value>default</value>                  </property>              </configuration>              <arg>${driveID}</arg>          </java>          <ok to='merging'/>          <error to='fail'/>      </action>      <fork name='merging'>          <path start='mergeT1'/>          <path start='mergeT2'/>      </fork>      <action name='mergeT1'>          <java>              <job-tracker>${jobTracker}</job-tracker>              <name-node>${nameNode}</name-node>              <configuration>                  <property>                      <name>mapred.job.queue.name</name>                      <value>default</value>                  </property>              </configuration>              <arg>-drive</arg>              <arg>${driveID}</arg>              <arg>-type</arg>              <arg>T1</arg>          </java>          <ok to='completed'/>          <error to='fail'/>      </action>      <action name='mergeT2'>          <java>              <job-tracker>${jobTracker}</job-tracker>              <name-node>${nameNode}</name-node>              <configuration>                  <property>                      <name>mapred.job.queue.name</name>                      <value>default</value>                  </property>              </configuration>              <main-class>com.navteq.assetmgmt.hdfs.merge.MergerLoader</main-class>              <arg>-drive</arg>              <arg>${driveID}</arg>              <arg>-type</arg>              <arg>T2</arg>          </java>          <ok to='completed'/>          <error to='fail'/>      </action>      <join name='completed' to='end'/>      <kill name='fail'>          <message>Java failed, error message[${wf:errorMessage(wf:lastErrorNode())}]</message>      </kill>      <end name='end'/>  </workflow-app> |

This simple workflow defines three actions: ingestor, mergeT1, and mergeT2. Each action is implemented as a MapReduce job. As illustrated in Figure 2, the workflow starts with the start node, which transfers control to the ingestor action. Once the ingestor step completes, a fork control node is invoked, an action that starts the execution of mergeT1 and mergeT2 in parallel. Once both actions are completed, the join control node is invoked. On successful completion of join node, the control is passed to the end node, a step that ends the process. The <job-tracker> and <name-node> entities dictate the servers that the Hive job will connect to for executing its script.

##### Figure 2. Illustration of the workflow of Listing 1



Let's now look at another Oozie workflow example that incorporates a Hive job.

##### Listing 2. Oozie workflow incorporating a Hive job

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30 | <workflow-app xmlns='uri:oozie:workflow:0.1' name='Hive-Workflow'>      <start to='hive-job'/>      <action name='hive-job'>          <hive xmlns='uri:oozie:hive-action:0.4'>              <job-tracker>${jobTracker}</job-tracker>              <name-node>${nameNode}</name-node>              <prepare>                  <delete path='${workflowRoot}/output-data/hive'/>                  <mkdir path='${workflowRoot}/output-data'/>              </prepare>              <job-xml>${workflowRoot}/hive-site.xml</job-xml>              <configuration>                  <property>                      <name>oozie.hive.defaults</name>                      <value>${workflowRoot}/hive-site.xml</value>                  </property>              </configuration>  **<script>**hive\_script.q**</script>**              <param>JSON\_SERDE=${workflowRoot}/lib/hive-serdes-1.0-SNAPSHOT.jar</param>              <param>PARTITION\_PATH=${workflowInput}</param>              <param>DATEHOUR=${dateHour}</param>          </hive>          <ok to='end'/>          <error to='fail'/>      </action>      <kill name='fail'>          <message>Hive failed, error message[${wf:errorMessage(wf:lastErrorNode())}]</message>      </kill>      <end name='end'/>  </workflow-app> |
|  |  |

In this workflow, we identify the action as a Hive action with this node: <hive xmlns='uri:oozie:hive-action:0.4'>. The <job-xml> entity is used to specify a configuration file for Hive.

Finally, let's check another Oozie example of scheduling recurring workflows.

##### Listing 3. Oozie recurring workflows

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37 | <coordinator-app name='Hive-workglow' frequency='${coord:hours(1)}'  start='${jobStart}' end='${jobEnd}'  timezone='UTC'  xmlns='uri:oozie:coordinator:0.1'>  <datasets>      <dataset name='InputData' frequency='${coord:hours(1)}'  initial-instance='${initialDataset}' timezone='America/Los\_Angeles'>          <uri-template>              hdfs://hadoop1:8020/user/flume/InputData/${YEAR}/${MONTH}/${DAY}/${HOUR}          </uri-template>          <done-flag></done-flag>      </dataset>  </datasets>  <input-events>      <data-in name='InputData ' dataset='Data'>          <instance>${coord:current(coord:tzOffset() / 60)}</instance>      </data-in>      <data-in name='readyIndicator' dataset='tweets'>          <instance>${coord:current(1 + (coord:tzOffset() / 60))}</instance>      </data-in>  </input-events>  <action>      <workflow>          <app-path>${workflowRoot}/ Hive-Workflow.xml</app-path>          <configuration>              <property>                  <name>workflowInput</name>                  <value>${coord:dataIn('InputData')}</value>              </property>              <property>                  <name>dateHour</name>                  <value>${coord:formatTime(coord:dateOffset(coord:nominalTime(), tzOffset, 'HOUR'), 'yyyyMMddHH')}</value>              </property>          </configuration>      </workflow>  </action>  </coordinator-app> |

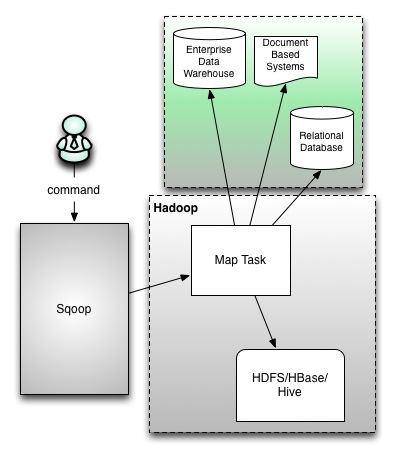
In this example, the Hive workflow of the previous example is configured to be executed on an hourly basis using the coord:hours(1) method. Specify a start time and end time for the job using the codejobStart and jobEndvariables. The datasets entity specifies the location of a set of input data. In this case, there is a dataset called InputData, which is updated every hour, as specified by the frequency. For each execution of the Hive workflow, there will be a separate instance of the input dataset, starting with the initial instance specified by the dataset. YEAR, MONTH, DAY, and HOUR are variables used to parameterize the URI template for the dataset. The done flag specifies a file that determines when the dataset is finished being generated.

● The workflow of Sqoop and its Benefits

[Apache Sqoop (incubating)](http://incubator.apache.org/sqoop/) was created to efficiently transfer bulk data between Hadoop and external structured datastores, such as RDBMS and data warehouses, because databases are not easily accessible by Hadoop. Sqoop is currently undergoing incubation at The Apache Software Foundation. More information on this project can be found at <http://incubator.apache.org/sqoop>.

The popularity of Sqoop in enterprise systems confirms that Sqoop does bulk transfer admirably. That said, to enhance its functionality, Sqoop needs to fulfill data integration use-cases as well as become easier to manage and operate.

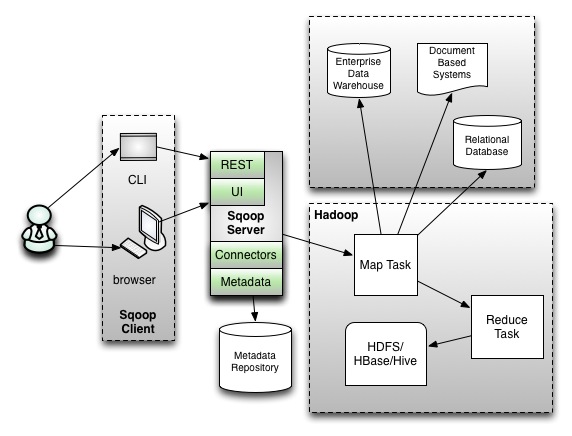
**What is Sqoop?**  
As [described in a previous blog post](https://blogs.apache.org/sqoop/entry/apache_sqoop_overview), Sqoop is a bulk data transfer tool that allows easy import/export of data from structured datastores such as relational databases, enterprise data warehouses, and NoSQL systems. Using Sqoop, you can provision the data from an external system into HDFS, as well as populate tables in Hive and HBase. Similarly, Sqoop integrates with the workflow coordinator Apache Oozie (incubating), allowing you to schedule and automate import/export tasks. Sqoop uses a connector-based architecture which supports plugins that provide connectivity to additional external systems.



**Sqoop's Challenges**  
Sqoop has enjoyed enterprise adoption, and our experiences have exposed some recurring ease-of-use challenges, extensibility limitations, and security concerns that are difficult to support in the original design:  
- Cryptic and contextual command line arguments can lead to error-prone connector matching, resulting in user errors  
- Due to tight coupling between data transfer and the serialization format, some connectors may support a certain data format that others don't (e.g. direct MySQL connector can't support sequence files)  
- There are security concerns with openly shared credentials  
- By requiring root privileges, local configuration and installation are not easy to manage  
- Debugging the map job is limited to turning on the verbose flag  
- Connectors are forced to follow the JDBC model and are required to use common JDBC vocabulary (URL, database, table, etc), regardless if it is applicable

These challenges have motivated the design of Sqoop 2, which is the subject of this post. That said, Sqoop 2 is a work in progress whose design is subject to change.

Sqoop 2 will continue its strong support for command line interaction, while adding a web-based GUI that exposes a simple user interface. Using this interface, a user can walk through an import/export setup via UI cues that eliminate redundant options. Various connectors are added in the application in one place and the user is not tasked with installing or configuring connectors in their own sandbox. These connectors expose their necessary options to the Sqoop framework which then translates them to the UI. The UI is built on top of a REST API that can be used by a command line client exposing similar functionality. The introduction of Admin and Operator roles in Sqoop 2 will restrict 'create' access for Connections to Admins and 'execute' access to Operators. This model will allow integration with platform security and restrict the end user view to only operations applicable to end users.



**Ease of Use**  
Whereas Sqoop requires client-side installation and configuration, Sqoop 2 will be installed and configured server-side. This means that connectors will be configured in one place, managed by the Admin role and run by the Operator role. Likewise, JDBC drivers will be in one place and database connectivity will only be needed on the server. Sqoop 2 will be a web-based service: front-ended by a Command Line Interface (CLI) and browser and back-ended by a metadata repository. Moreover, Sqoop 2's service level integration with Hive and HBase will be on the server-side. Oozie will manage Sqoop tasks through the REST API. This decouples Sqoop internals from Oozie, i.e. if you install a new Sqoop connector then you won't need to install it in Oozie also.

**Ease of Extension**  
In Sqoop 2, connectors will no longer be restricted to the JDBC model, but can rather define their own vocabulary, e.g. Couchbase no longer needs to specify a table name, only to overload it as a backfill or dump operation.

Common functionality will be abstracted out of connectors, holding them responsible only for data transport. The reduce phase will implement common functionality, ensuring that connectors benefit from future development of functionality.

Sqoop 2's interactive web-based UI will walk users through import/export setup, eliminating redundant steps and omitting incorrect options. Connectors will be added in one place, with the connectors exposing necessary options to the Sqoop framework. Thus, users will only need to provide information relevant to their use-case.

With the user making an explicit connector choice in Sqoop 2, it will be less error-prone and more predictable. In the same way, the user will not need to be aware of the functionality of all connectors. As a result, connectors no longer need to provide downstream functionality, transformations, and integration with other systems. Hence, the connector developer no longer has the burden of understanding all the features that Sqoop supports.

**Security**  
Currently, Sqoop operates as the user that runs the 'sqoop' command. The security principal used by a Sqoop job is determined by what credentials the users have when they launch Sqoop. Going forward, Sqoop 2 will operate as a server based application with support for securing access to external systems via role-based access to Connection objects. For additional security, Sqoop 2 will no longer allow code generation, require direct access to Hive and HBase, nor open up access to all clients to execute jobs.

Sqoop 2 will introduce Connections as First-Class Objects. Connections, which will encompass credentials, will be created once and then used many times for various import/export jobs. Connections will be created by the Admin and used by the Operator, thus preventing credential abuse by the end user. Furthermore, Connections can be restricted based on operation (import/export). By limiting the total number of physical Connections open at one time and with an option to disable Connections, resources can be managed.

# Advantages of Sqoop

Below are the advantages of Apache Sqoop, which is also the reason for choosing this technology in this layer.

* Allows the transfer of data with a variety of structured data stores like Postgres, Oracle, Teradata, and so on.
* Since the data is transferred and stored in Hadoop, Sqoop allows us to offload certain processing done in the **ETL** (**Extract**, **Load** and **Transform**) process into low-cost, fast, and effective Hadoop processes.
* Sqoop can execute the data transfer in parallel, so execution can be quick and more cost effective.
* Helps to integrate with sequential data from the mainframe. This helps not only to limit the usage of the mainframe, but also reduces the high cost in executing certain jobs using mainframe hardware. ...