

Azure HDInsight

Developer Guide

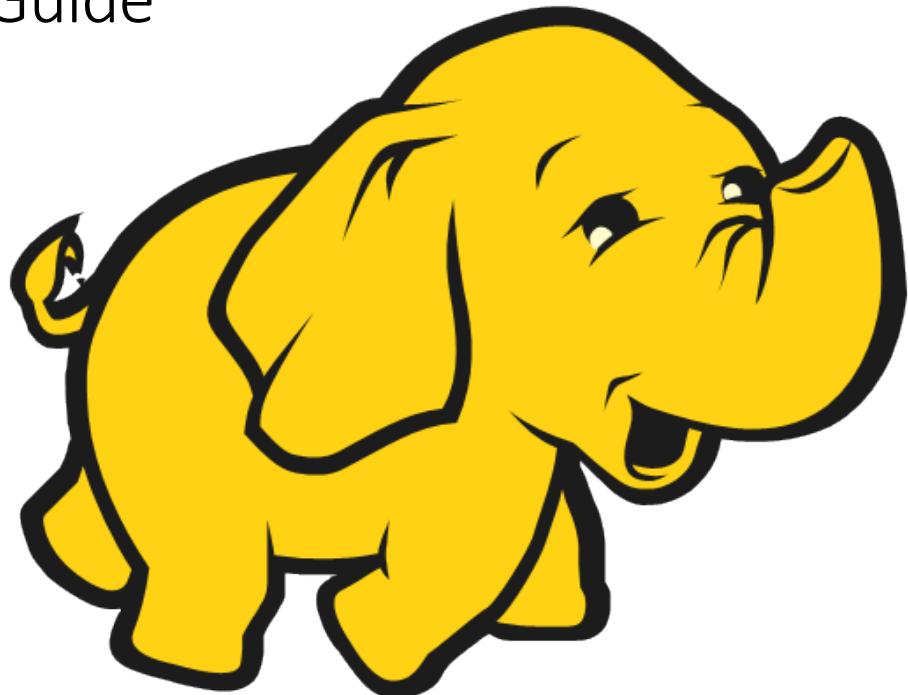


Table of Contents

Overview

[What is Azure HDInsight?](#)

[Iterative data exploration](#)

[Data Warehouse on demand](#)

[ETL at scale](#)

[Streaming at scale](#)

[Machine learning](#)

[Batch & Interactive Processing](#)

[Run Custom Programs](#)

[Upload Data to HDInsight](#)

Azure HDInsight and Hadoop Architecture

[HDInsight Architecture](#)

[Hadoop Architecture](#)

[Lifecycle of an HDInsight Cluster](#)

[High availability model](#)

[Capacity planning](#)

Configuring the Cluster

[Use SSH with HDInsight](#)

[Use SSH tunneling](#)

[Use HDInsight in a Virtual Network](#)

[Scaling best practices](#)

[Configuring Hive and Oozie Metadata Storage](#)

[Configuring Identity and Access Controls](#)

[Manage authorized Ambari users](#)

[Authorize user access to Ranger](#)

[Add ACLs at the file and folder levels](#)

[Sync users from Azure Active Directory to cluster](#)

[Use on-demand HDInsight clusters from Data Factory](#)

Monitoring and managing the HDInsight cluster

Key scenarios to monitor

Monitoring and managing with Ambari

Monitoring with the Ambari REST API

Administering HDInsight using the Azure Portal

Manage configurations with Ambari

Manage cluster logs

Adding storage accounts to a running cluster

Use script actions to customize cluster setup

Develop script actions

OS patching for HDInsight cluster

Developing Hive applications

Hive and ETL Overview

Connect to Hive with JDBC or ODBC

Writing Hive applications using Java

Writing Hive applications using Python

Creating user defined functions

Process and analyze JSON documents with Hive

Hive samples

Query Hive using Excel

Analyze stored sensor data using Hive

Analyze stored tweets using beeline and Hive

Analyze flight delay data with Hive

Analyze website logs with Hive

Developing Spark applications

Spark Scenarios

Use Spark with HDInsight

Use Spark SQL with HDInsight

Run Spark from the Shell

Use Spark with notebooks

Use Zeppelin notebooks with Spark

Use Jupyter notebook with Spark

Use external packages with Jupyter using cell magic

[Use external packages with Jupyter using script action](#)

[Use Spark with IntelliJ](#)

[Create apps using the Azure Toolkit for IntelliJ](#)

[Debug jobs remotely with IntelliJ](#)

[Spark samples](#)

[Analyze Application Insights telemetry with Spark](#)

[Analyze website logs with Spark SQL](#)

[Developing Spark ML applications](#)

[Creating Spark ML Pipelines](#)

[Creating Spark ML models in notebooks](#)

[Deep Learning with Spark](#)

[Use Caffe for deep learning with Spark](#)

[Developing R scripts on HDInsight](#)

[What is R Server?](#)

[Selecting a compute context](#)

[Analyze data from Azure Storage and Data Lake Store using R](#)

[Submit jobs from Visual Studio Tools for R](#)

[Submit R jobs from R Studio Server](#)

[Developing Spark Streaming applications](#)

[What is Spark Streaming \(DStreams\)?](#)

[What is Spark Structured Streaming?](#)

[Use Spark DStreams to process events from Kafka](#)

[Use Spark DStreams to process events from Event Hubs](#)

[Use Spark Structured Streaming to process events from Kafka](#)

[Use Spark Structured Streaming to process events from Event Hubs](#)

[Creating highly available Spark Streaming jobs in YARN](#)

[Creating Spark Streaming jobs with exactly once event processing guarantees](#)

[Optimizing Spark Performance](#)

[Optimizing and configuring Spark jobs for performance](#)

[Configuring Spark settings](#)

[Choosing between Spark RDD, dataframe and dataset](#)

[Use HBase](#)

[What is HBase?](#)

[Understanding the HBase storage options](#)

[Using the HBase shell](#)

[Using the HBase REST SDK](#)

[Configure HBase backup and replication](#)

[Using Spark with HBase](#)

[Monitor HBase with OMS](#)

[HBase - Migrating to a New Version](#)

[Use Phoenix with HBase on HDInsight](#)

[Phoenix in HDInsight](#)

[Get started using Phoenix with SQLLine](#)

[Bulk Loading with Phoenix with psql](#)

[Using Spark with Phoenix](#)

[Using the Phoenix Query Server REST SDK](#)

[Phoenix performance monitoring](#)

[Apache Open Source Ecosystem](#)

[Install HDInsight apps](#)

[Install and use Dataiku](#)

[Install and use Datameer](#)

[Install and use H2O](#)

[Install and use Streamsets](#)

[Install and use Cask](#)

[Advanced Scenarios and Deep Dives](#)

[Advanced Analytics Deep Dive](#)

[ETL Deep Dive](#)

[Operationalize Data Pipelines with Oozie](#)

[Streaming and Business Intelligence](#)

[Troubleshooting](#)

[Troubleshooting a failed or slow HDInsight cluster](#)

[Debug jobs by analyzing HDInsight logs](#)

[Debug Tez jobs using Hive views in Ambari](#)

[Common problems FAQ](#)

Introduction to Azure HDInsight, the Hadoop technology stack, and Hadoop clusters

8/16/2017 • 9 min to read • [Edit Online](#)

This article provides an introduction to Azure HDInsight, a cloud distribution of the Hadoop technology stack. It also covers what a Hadoop cluster is and when you would use it.

What is HDInsight and the Hadoop technology stack?

Azure HDInsight is a cloud distribution of the Hadoop components from the [Hortonworks Data Platform \(HDP\)](#). [Apache Hadoop](#) was the original open-source framework for distributed processing and analysis of big data sets on clusters of computers.

The Hadoop technology stack includes related software and utilities, including Apache Hive, HBase, Spark, Kafka, and many others. To see available Hadoop technology stack components on HDInsight, see [Components and versions available with HDInsight](#). To read more about Hadoop in HDInsight, see the [Azure features page for HDInsight](#).

What is a Hadoop cluster, and when do you use it?

Hadoop is also a cluster type that has:

- YARN for job scheduling and resource management
- MapReduce for parallel processing
- The Hadoop distributed file system (HDFS)

Hadoop clusters are most often used for batch processing of stored data. Other kinds of clusters in HDInsight have additional capabilities: Spark has grown in popularity because of its faster, in-memory processing. See [Cluster types on HDInsight](#) for details.

What is big data?

Big data describes any large body of digital information, such as:

- Sensor data from industrial equipment
- Customer activity collected from a website
- A Twitter newsfeed

Big data is being collected in escalating volumes, at higher velocities, and in a greater variety of formats. It can be historical (meaning stored) or real time (meaning streamed from the source).

Cluster types in HDInsight

HDInsight includes specific cluster types and cluster customization capabilities, such as adding components, utilities, and languages.

Spark, Kafka, Interactive Hive, HBase, customized, and other cluster types

HDInsight offers the following cluster types:

- **Apache Hadoop:** Uses [HDFS](#), [YARN](#) resource management, and a simple [MapReduce](#) programming model to process and analyze batch data in parallel.

- **Apache Spark:** A parallel processing framework that supports in-memory processing to boost the performance of big-data analysis applications, Spark works for SQL, streaming data, and machine learning. See [What is Apache Spark in HDInsight?](#)
- **Apache HBase:** A NoSQL database built on Hadoop that provides random access and strong consistency for large amounts of unstructured and semi-structured data - potentially billions of rows times millions of columns. See [What is HBase on HDInsight?](#)
- **Microsoft R Server:** A server for hosting and managing parallel, distributed R processes. It provides data scientists, statisticians, and R programmers with on-demand access to scalable, distributed methods of analytics on HDInsight. See [Overview of R Server on HDInsight](#).
- **Apache Storm:** A distributed, real-time computation system for processing large streams of data fast. Storm is offered as a managed cluster in HDInsight. See [Analyze real-time sensor data using Storm and Hadoop](#).
- **Apache Interactive Hive preview (AKA: Live Long and Process):** In-memory caching for interactive and faster Hive queries. See [Use Interactive Hive in HDInsight](#).
- **Apache Kafka:** An open-source platform used for building streaming data pipelines and applications. Kafka also provides message-queue functionality that allows you to publish and subscribe to data streams. See [Introduction to Apache Kafka on HDInsight](#).

You can also configure clusters using the following methods:

- **Domain-joined clusters preview:** A cluster joined to an Active Directory domain so that you can control access and provide governance for data.
- **Custom clusters with script actions:** Clusters with scripts that run during provisioning and install additional components.

Example cluster customization scripts

Script actions are Bash scripts on Linux that run during cluster provisioning, and that can be used to install additional components on the cluster.

The following example scripts are provided by the HDInsight team:

- **Hue:** A set of web applications used to interact with a cluster. Linux clusters only.
- **Giraph:** Graph processing to model relationships between things or people.
- **Solr:** An enterprise-scale search platform that allows full-text search on data.

For information on developing your own Script Actions, see [Script Action development with HDInsight](#).

Components and utilities on HDInsight clusters

The following components and utilities are included on HDInsight clusters:

- **Ambari:** Cluster provisioning, management, monitoring, and utilities.
- **Avro** (Microsoft .NET Library for Avro): Data serialization for the Microsoft .NET environment.
- **Hive & HCatalog:** SQL-like querying, and a table and storage management layer.
- **Mahout:** For scalable machine learning applications.
- **MapReduce:** Legacy framework for Hadoop distributed processing and resource management. See [YARN](#).
- **Oozie:** Workflow management.
- **Phoenix:** Relational database layer over HBase.
- **Pig:** Simpler scripting for MapReduce transformations.
- **Sqoop:** Data import and export.
- **Tez:** Allows data-intensive processes to run efficiently at scale.
- **YARN:** Resource management that is part of the Hadoop core library.
- **ZooKeeper:** Coordination of processes in distributed systems.

NOTE

For information on the specific components and version information, see [Hadoop components and versions in HDInsight](#)

Ambari

Apache Ambari is for provisioning, managing, and monitoring Apache Hadoop clusters. It includes an intuitive collection of operator tools and a robust set of APIs that hide the complexity of Hadoop, simplifying the operation of clusters. HDInsight clusters on Linux provide both the Ambari web UI and the Ambari REST API. Ambari Views on HDInsight clusters allow plug-in UI capabilities. See [Manage HDInsight clusters using Ambari](#) and [Apache Ambari API reference](#).

Avro (Microsoft .NET Library for Avro)

The Microsoft .NET Library for Avro implements the Apache Avro compact binary data interchange format for serialization for the Microsoft .NET environment. It defines a language-agnostic schema so that data serialized in one language can be read in another. Detailed information on the format can be found in the [Apache Avro Specification](#). The format of Avro files supports the distributed MapReduce programming model: Files are "splittable", meaning you can seek any point in a file and start reading from a particular block. To find out how, see [Serialize data with the Microsoft .NET Library for Avro](#). Linux-based cluster support to come.

HDFS

Hadoop Distributed File System (HDFS) is a file system that, with YARN and MapReduce, is the core of Hadoop technology. It's the standard file system for Hadoop clusters on HDInsight. See [Query data from HDFS-compatible storage](#).

Hive & HCatalog

[Apache Hive](#) is data warehouse software built on Hadoop that allows you to query and manage large datasets in distributed storage by using a SQL-like language called HiveQL. Hive, like Pig, is an abstraction on top of MapReduce, and it translates queries into a series of MapReduce jobs. Hive is closer to a relational database management system than Pig, and is used with more structured data. For unstructured data, Pig is the better choice. See [Use Hive with Hadoop in HDInsight](#).

[Apache HCatalog](#) is a table and storage management layer for Hadoop that presents you with a relational view of data. In HCatalog, you can read and write files in any format that works for a Hive SerDe (serializer-deserializer).

Mahout

[Apache Mahout](#) is a library of machine learning algorithms that run on Hadoop. Using principles of statistics, machine learning applications teach systems to learn from data and to use past outcomes to determine future behavior. See [Generate movie recommendations using Mahout on Hadoop](#).

MapReduce

MapReduce is the legacy software framework for Hadoop for writing applications to batch process big data sets in parallel. A MapReduce job splits large datasets and organizes the data into key-value pairs for processing. MapReduce jobs run on [YARN](#). See [MapReduce](#) in the Hadoop Wiki.

Oozie

[Apache Oozie](#) is a workflow coordination system that manages Hadoop jobs. It is integrated with the Hadoop stack and supports Hadoop jobs for MapReduce, Pig, Hive, and Sqoop. It can also be used to schedule jobs specific to a system, like Java programs or shell scripts. See [Use Oozie with Hadoop](#).

Phoenix

[Apache Phoenix](#) is a relational database layer over HBase. Phoenix includes a JDBC driver that allows you to query and manage SQL tables directly. Phoenix translates queries and other statements into native NoSQL API calls - instead of using MapReduce - thus enabling faster applications on top of NoSQL stores. See [Use Apache Phoenix](#)

and [SQuirreL with HBase clusters](#).

Pig

[Apache Pig](#) is a high-level platform that allows you to perform complex MapReduce transformations on large datasets by using a simple scripting language called Pig Latin. Pig translates the Pig Latin scripts so they'll run within Hadoop. You can create User-Defined Functions (UDFs) to extend Pig Latin. See [Use Pig with Hadoop](#).

Sqoop

[Apache Sqoop](#) is a tool that transfers bulk data between Hadoop and relational databases such as SQL, or other structured data stores, as efficiently as possible. See [Use Sqoop with Hadoop](#).

Tez

[Apache Tez](#) is an application framework built on Hadoop YARN that executes complex, acyclic graphs of general data processing. It's a more flexible and powerful successor to the MapReduce framework that allows data-intensive processes, such as Hive, to run more efficiently at scale. See "[Use Apache Tez for improved performance](#)" in [Use Hive and HiveQL](#).

YARN

Apache YARN is the next generation of MapReduce (MapReduce 2.0, or MRv2) and supports data processing scenarios beyond MapReduce batch processing with greater scalability and real-time processing. YARN provides resource management and a distributed application framework. MapReduce jobs run on YARN. See [Apache Hadoop NextGen MapReduce \(YARN\)](#).

ZooKeeper

[Apache ZooKeeper](#) coordinates processes in large distributed systems using a shared hierarchical namespace of data registers (znodes). Znodes contain small amounts of meta information needed to coordinate processes: status, location, configuration, and so on. See an example of [ZooKeeper with an HBase cluster and Apache Phoenix](#).

Programming languages on HDInsight

HDInsight clusters - Spark, HBase, Kafka, Hadoop, and other clusters - support many programming languages, but some aren't installed by default. For libraries, modules, or packages not installed by default, [use a script action to install the component](#).

Default programming language support

By default, HDInsight clusters support:

- Java
- Python

Additional languages can be installed using [script actions](#).

Java virtual machine (JVM) languages

Many languages other than Java can run on a Java virtual machine (JVM); however, running some of these languages may require additional components installed on the cluster.

These JVM-based languages are supported on HDInsight clusters:

- Clojure
- Jython (Python for Java)
- Scala

Hadoop-specific languages

HDInsight clusters support the following languages that are specific to the Hadoop technology stack:

- Pig Latin for Pig jobs

- HiveQL for Hive jobs and SparkSQL

HDInsight Standard and HDInsight Premium

HDInsight provides big data cloud offerings in two categories, Standard and Premium. HDInsight Standard provides an enterprise-scale cluster that organizations can use to run their big data workloads. HDInsight Premium builds on Standard capabilities and provides advanced analytical and security capabilities for an HDInsight cluster. For more information, see [Azure HDInsight Premium](#)

Microsoft business intelligence and HDInsight

Familiar business intelligence (BI) tools retrieve, analyze, and report data integrated with HDInsight by using either the Power Query add-in or the Microsoft Hive ODBC Driver:

- [Connect Excel to Hadoop with Power Query](#): Learn how to connect Excel to the Azure Storage account that stores the data from your HDInsight cluster by using Microsoft Power Query for Excel. Windows workstation required.
- [Connect Excel to Hadoop with the Microsoft Hive ODBC Driver](#): Learn how to import data from HDInsight with the Microsoft Hive ODBC Driver. Windows workstation required.
- [Microsoft Cloud Platform](#): Learn about Power BI for Office 365, download the SQL Server trial, and set up SharePoint Server 2013 and SQL Server BI.
- [SQL Server Analysis Services](#)
- [SQL Server Reporting Services](#)

Next steps

- [Get started with Hadoop in HDInsight](#): A quick-start tutorial for provisioning HDInsight Hadoop clusters and running sample Hive queries.
- [Get started with Spark in HDInsight](#): A quick-start tutorial for creating a Spark cluster and running interactive Spark SQL queries.
- [Use R Server on HDInsight](#): Start using R Server in HDInsight Premium.
- [Provision HDInsight clusters](#): Learn how to provision an HDInsight Hadoop cluster through the Azure portal, Azure CLI, or Azure PowerShell.

Iterative Data Exploration

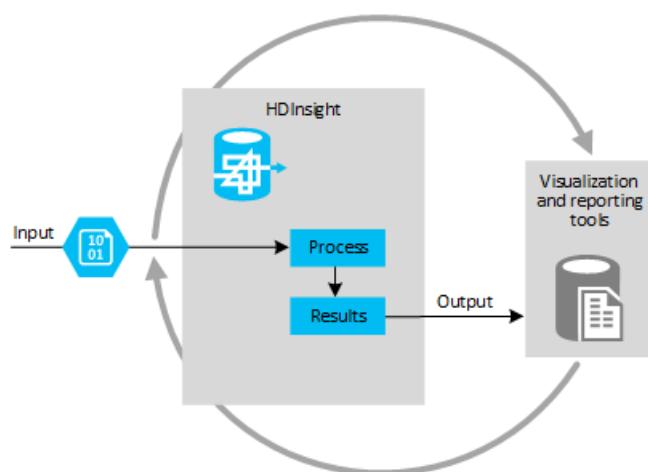
8/15/2017 • 7 min to read • [Edit Online](#)

Traditional data storage and management systems such as data warehouses, data models, and reporting and analytical tools provide a wealth of information on which to base business decisions. However, while traditional BI works well for business data that can easily be structured, managed, and processed in a dimensional analysis model, some kinds of analysis require a more flexible solution that can derive meaning from less obvious sources of data such as log files, email messages, tweets, and more. There's a great deal of useful information to be found in these less structured data sources, which often contain huge volumes of data that must be processed to reveal key data points. This kind of data processing is what big data solutions such as HDInsight were designed to handle. It provides a way to process extremely large volumes of unstructured or semi-structured data, often by performing complex computation and transformation batch processing of the data, to produce an output that can be visualized directly or combined with other datasets.

If you do not intend to reuse the information from the analysis, but just want to explore the data, you may choose to consume it directly in an analysis or visualization tool such as Microsoft Excel.

Use case and model overview

The image below shows an overview of the use case and model for a standalone iterative data exploration and visualization solution using HDInsight. The source data files are loaded into the cluster, processed by one or more queries within HDInsight, and the output is consumed by the chosen reporting and visualization tools. The cycle repeats using the same data until useful insights have been found, or it becomes clear that there is no useful information available from the data — in which case you might choose to restart the process with a different source dataset.



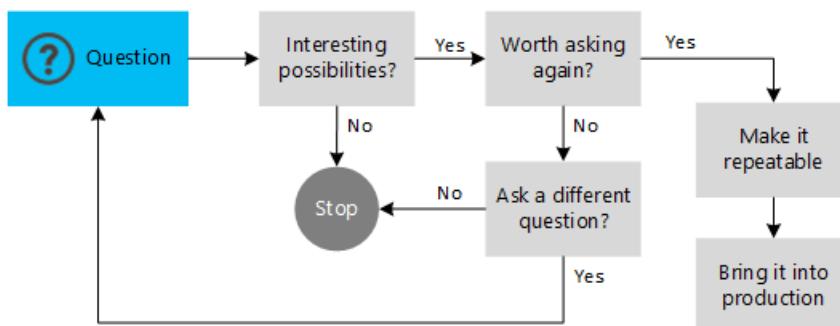
Text files and compressed binary files can be loaded directly into the cluster storage, while stream data will usually need to be collected and handled by a suitable stream capture mechanism (see [Upload data for Hadoop jobs in HDInsight](#) for more information). The output data may be combined with other datasets within your visualization and reporting tools to augment the information and to provide comparisons, as you will see later in this topic.

When using HDInsight for iterative data exploration, you will often do so as an interactive process. For example, you might use the Power Query add-in for Excel to submit a query to an HDInsight cluster and wait for the results to be returned, usually within a few seconds or even a few minutes. You can then modify and experiment with the query to optimize the information it returns.

However, keep in mind that these are batch operations that are submitted to all of the servers in the cluster for parallel processing, and queries can often take minutes or hours to complete when there are very large volumes of source data. For example, you might use Pig to process an input file, with the results returned in an output file some time later—at which point you can perform the analysis by importing this file into your chosen visualization tool.

The decision flow

One of the typical uses of a big data solution such as HDInsight is to explore data that you already have, or data you collect speculatively, to see if it can provide insights into information that you can use within your organization. The decision flow shown below is an example of how you might start with a guess based on intuition, and progress towards a repeatable solution that you can incorporate into your existing BI systems. Or, perhaps, to discover that there is no interesting information in the data, but the cost of discovering this has been minimized by using a "pay for what you use" mechanism that you can set up and then tear down again very quickly and easily.



When to choose this model

The iterative exploration model is typically suited to the following scenarios:

- Handling data that you cannot process using existing systems, perhaps by performing complex calculations and transformations that are beyond the capabilities of existing systems to complete in a reasonable time.
- Collecting feedback from customers through email, web pages, or external sources such as social media sites, then analyzing it to get a picture of customer sentiment for your products.
- Combining information with other data, such as demographic data that indicates population density and characteristics in each city where your products are sold.
- Dumping data from your existing information systems into HDInsight so that you can work with it without interrupting other business processes or risking corruption of the original data.
- Trying out new ideas and validating processes before implementing them within the live system.

Combining your data with datasets available from Azure Marketplace or other commercial data sources can reveal useful information that might otherwise remain hidden in your data.

Data sources

The input data for this model typically includes the following:

- Social data, log files, sensors, and applications that generate data files.
- Datasets obtained from Azure Marketplace and other commercial data providers.
- Internal data extracted from databases or data warehouses for experimentation and one-off analysis.
- Streaming data that is captured, filtered, and pre-processed through a suitable tool or framework (see [Upload data for Hadoop jobs in HDInsight](#)).

Notice that, as well as externally obtained data, you might process data from within your organization's existing database or data warehouse. HDInsight is an ideal solution when you want to perform offline exploration of existing data in a sandbox. For example, you may join several datasets from your data warehouse to create large datasets that act as the source for some experimental investigation, or to test new analysis techniques. This avoids the risk of interrupting existing systems, affecting performance of your data warehouse system, or accidentally corrupting the core data.

The capability to store schema-less data, and apply a schema only when processing the data, may also simplify the task of combining information from different systems because you do not need to apply a schema beforehand, as you would in a traditional data warehouse.

Often you need to perform more than one query on the data to get the results into the form you need. It's not unusual to base queries on the results of a preceding query; for example, using one query to select and transform the required data and remove redundancy, a second query to summarize the data returned from the first query, and a third query to format the output as required. This iterative approach enables you to start with a large volume of complex and difficult to analyze data, and get it into a structure that you can consume directly from an analytical tool such as Excel, or use as input to a managed BI solution.

Output targets

The results from your exploration processes can be visualized using any of the wide range of tools that are available for analyzing data, combining it with other datasets, and generating reports. Typical examples for the iterative exploration model are:

- Interactive analytical tools such as Excel, Power Query, Power Pivot, Power View, and Power Map.
- SQL Server Reporting Services using Report Builder.
- Custom or third party analysis and visualization tools.

Considerations

There are some important points to consider when choosing the iterative exploration model:

- This model is typically used when you want to:
 - Experiment with new types or sources of data.
 - Generate one-off reports or visualizations of external or internal data.
 - Monitor a data source using visualizations to detect changes or to predict behavior.
 - Combine the output with other data to generate comparisons or to augment the information.
- You will usually choose this model when you do not want to persist the results of the query after analysis, or after the required reports have been generated. It is typically used for one-off analysis tasks where the results are discarded after use; and so differs from the other models described in this guide in which the results are stored and reused.
- Very large datasets are likely to preclude the use of an interactive approach due to the time taken for the queries to run. However, after the queries are complete you can connect to the cluster and work interactively with the data to perform different types of analysis or visualization.
- Data arriving as a stream, such as the output from sensors on an automated production line or the data generated by GPS sensors in mobile devices, requires additional considerations. A typical technique is to capture the data using a stream processing technology such as Storm or StreamInsight and persist it, then process it in batches or at regular intervals. The stream capture technology may perform some pre-processing, and might also power a real time visualization or rudimentary analysis tool, as well as feeding it into an HDInsight cluster. A common technique is micro-batch processing, where incoming data is persisted

in small increments, allowing near real-time processing by the big data solution.

- You are not limited to running a single query on the source data. You can follow an iterative pattern in which the data is passed through the cluster multiple times, each pass refining the data until it is suitably prepared for use in your analytical tool. For example, a large unstructured file might be processed using a Pig script to generate a smaller, more structured output file. This output could then be used as the input for a Hive query that returns aggregated data in tabular form.

Next steps

- [HDInsight Architecture](#)
- [Data Warehouse on Demand](#)
- [Upload data for Hadoop jobs in HDInsight](#)

Data Warehouse on Demand

8/16/2017 • 8 min to read • [Edit Online](#)

Hadoop-based big data solutions such as HDInsight can provide a robust, high performance, and cost-effective data storage and parallel job processing mechanism. Data is replicated in the storage system, and jobs are distributed across the nodes for fast parallel processing. In the case of HDInsight, the data is saved in Azure blob storage or Azure Data Lake Store, which is also replicated three times.

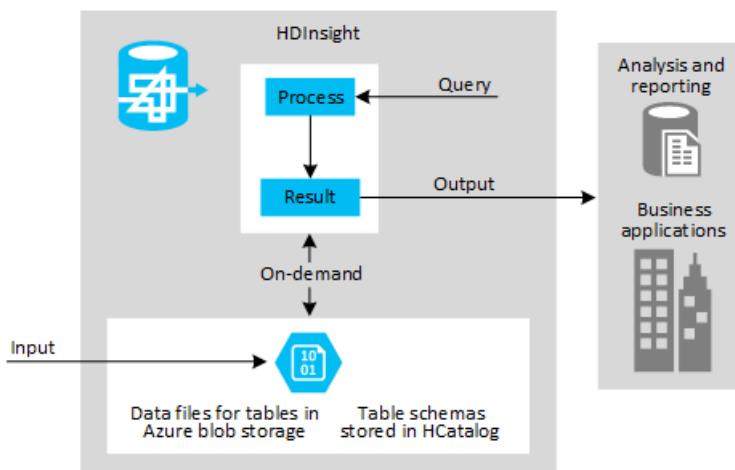
This combination of capabilities means that you can use HDInsight as a basic data warehouse. The low cost of storage when compared to most relational database mechanisms that have the same level of reliability also means that you can use it simply as a commodity storage mechanism for huge volumes of data, even if you decide not to transform the data into Hive tables.

If you need to store vast amounts of data, irrespective of the format of that data, an HDInsight Hadoop-based solution can reduce administration overhead and save money by minimizing the need for the high performance database servers and storage clusters used by traditional relational database systems. With HDInsight, you can also reduce the administration overhead and running costs compared to on-premises deployment of a Hadoop-based cluster.

Use case and model overview

The diagram below shows an overview of the use case and model for a data warehouse on demand solution using HDInsight. The source data files may be obtained from external sources, but are just as likely to be internal data generated by your business processes and applications. For example, you might decide to use this model instead of using a locally installed data warehouse based on the traditional relational database model.

In this scenario, you can store the data both as the raw source data and as Hive tables. Hive provides access to the data in a familiar row and column format that is easy to consume and visualize in most BI tools. The data for the tables is stored in Azure blob storage, and the table definitions can be maintained by [HCatalog](#) (a feature of [Hive](#)).



In this model, when you need to process the stored data, you create an HDInsight Hadoop-based cluster that uses the Azure blob storage container holding that data. When you finish processing the data you can tear down the cluster without losing the original archived data (see [Using external metadata stores](#) for information about preserving or recreating metadata such as Hive table definitions when you tear down and then recreate an HDInsight cluster).

You might also consider storing partly processed data where you have performed some translation or summary of the data, but it is still in a relatively raw form that you want to keep in case it is useful in the future. For example,

you might use a stream capture tool to allocate incoming positional data from a fleet of vehicles into separate categories or areas, add some reference keys to each item, and then store the results ready for processing at a later date. Stream data may arrive in rapid bursts, and typically generates very large files, so using HDInsight to capture and store the data helps to minimize the load on your existing data management systems.

This model is also suitable for use as a data store where you do not need to implement the typical data warehouse capabilities. For example, you may just want to minimize storage cost when saving large tabular format data files for use in the future, large text files such as email archives or data that you must keep for legal or regulatory reasons but you do not need to process, or for storing large quantities of binary data such as images or documents. In this case you simply load the data into the storage associated with cluster, without creating Hive tables for it.

You might, as an alternative, choose to use just an [HBase](#) cluster in this model. HBase can be accessed directly from client applications through the Java APIs and the REST interface. You can load data directly into HBase and query it using the built-in mechanisms.

When to choose this model

The data warehouse on demand model is typically suited to the following scenarios:

- Storing data in a way that allows you to minimize storage cost by taking advantage of cloud-based storage systems, and minimizing runtime cost by initiating a cluster to perform processing only when required.
- Exposing both the source data in raw form, and the results of queries executed over this data in the familiar row and column format, to a wide range of data analysis tools. The processed results can use a range of data types that includes both primitive types (including timestamps) and complex types such as arrays, maps, and structures.
- Storing schemas (or, to be precise, metadata) for tables that are populated by the queries you execute, and partitioning the data in tables based on a clustered index so that each has a separate metadata definition and can be handled separately.
- Creating views based on tables, and creating functions for use in both tables and queries.
- Creating a robust data repository for very large quantities of data that is relatively low cost to maintain compared to traditional relational database systems and appliances, where you do not need the additional capabilities of these types of systems.
- Consuming the results directly in business applications through interactive analytical tools such as Excel, or in corporate reporting platforms such as SQL Server Reporting Services.

Data sources

Data sources for this model are typically data collected from internal and external business processes. However, it may also include reference data and datasets obtained from other sources that can be matched on a key to existing data in your data store so that it can be used to augment the results of analysis and reporting processes. Some examples are:

- Data generated by internal business processes, websites, and applications.
- Reference data and data definitions used by business processes.
- Datasets obtained from Azure Marketplace and other commercial data providers.

If you adopt this model simply as a commodity data store rather than a data warehouse, you might also load data from other sources such as social media data, log files, and sensors; or streaming data that is captured, filtered, and processed through a suitable tool or framework (see [Upload data for Hadoop jobs in HDInsight](#)).

Output targets

The main intention of this model is to provide the equivalent to a data warehouse system based on the traditional relational database model, and expose it as Hive tables. You can use these tables in a variety of ways, such as:

- Combining the datasets for analysis, and using the result to generate reports and business information.
- Generating ancillary information such as “related items” or recommendation lists for use in applications and websites.
- Providing external access to the results through web applications, web services, and other services.
- Powering information systems such as SharePoint server through web parts and the Business Data Connector (BDC).

If you adopt this model simply as a commodity data store rather than a data warehouse, you might use the data you store as an input for any of the models described here.

The data in an HDInsight data warehouse can be analyzed and visualized directly using any tools that can consume Hive tables. Typical examples are:

- SQL Server Reporting Services
- SQL Server Analysis Services
- Interactive analytical tools such as Excel, Power Query, Power Pivot, Power View, and Power Map
- Custom or third party analysis and visualization tools

Considerations

There are some important points to consider when choosing the data warehouse on demand model:

- This model is typically used when you want to:
 - Create a central point for analysis and reporting by multiple users and tools.
 - Store multiple datasets for use by internal applications and tools.
 - Host your data in the cloud to benefit from reliability and elasticity, to minimize cost, and to reduce administration overhead.
 - Store both externally collected data and data generated by internal tools and processes.
 - Refresh the data at scheduled intervals or on demand.
- You can use Hive to:
 - Define tables that have the familiar row and column format, with a range of data types for the columns that includes both primitive types (including timestamps) and complex types such as arrays, maps, and structures.
 - Load data from storage into tables, save data to storage from tables, and populate tables from the results of running a query.
 - Create indexes for tables, and partition tables based on a clustered index so that each has a separate metadata definition and can be handled separately.
 - Rename, alter and drop tables, and modify columns in a table as required.
 - Create views based on tables, and create functions for use in both tables and queries.

The main limitation of Hive tables is that you cannot create constraints such as foreign key relationships that are automatically managed. For more details of how to work with Hive tables, see [Hive Data Definition Language](#) on the Apache Hive website.

- You can store the Hive queries and views within HDInsight so that they can be used to extract data on demand in much the same way as the stored procedures in a relational database. However, to minimize response times you will probably need to pre-process the data where possible using queries within your solution, and store these intermediate results in order to reduce the time-consuming overhead of complex queries. Incoming data may be processed by any type of query, not just Hive, to cleanse and validate the data before converting it to table format.
- You can use the Hive ODBC connector in SQL Server with HDInsight to create linked servers. This allows you to write Transact-SQL queries that join tables in a SQL Server database to tables stored in an HDInsight data warehouse.
- If you want to be able to delete and restore the cluster, as is typically the case for this model, there are additional considerations when creating a cluster. See [Using external metadata stores](#) for more information.

Next steps

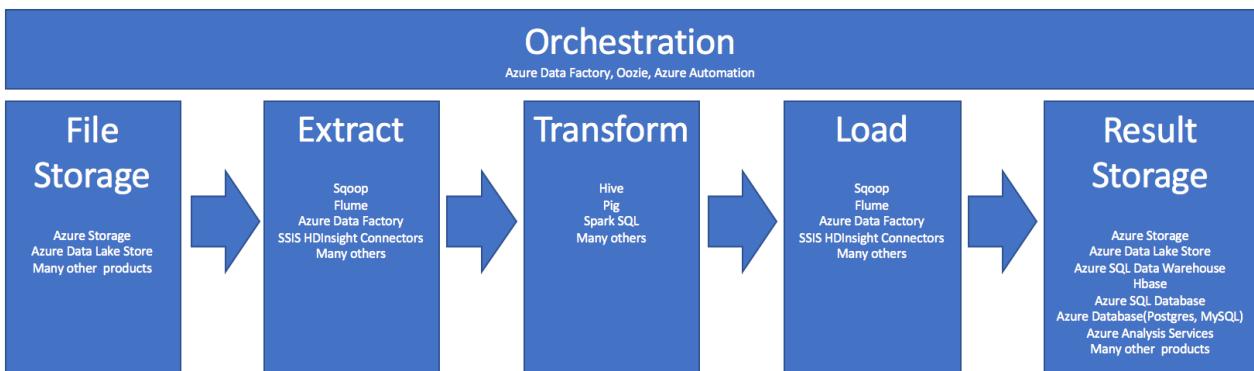
- [HDInsight Architecture](#)
- [Iterative Data Exploration](#)
- [Upload data for Hadoop jobs in HDInsight](#)
- [Using external metadata stores](#)

ETL at Scale

8/16/2017 • 8 min to read • [Edit Online](#)

Extract, Transform and Load (ETL) is the process by which data is acquired from the various sources, collected in a standard location, cleansed and processed and ultimately loaded into a datastore from which it can be queried. Legacy ETL processes import data, clean it in place, and then store it in a relational data engine. With HDInsight, a wide variety of Hadoop ecosystem components are enabled to support performing ETL, and due to the scalable nature of HDInsight in terms of storage and processing, support performing ETL at scale.

The use of HDInsight in the ETL process can be summarized by this pipeline:



The sections that follow explore each of the ETL phases and the components utilized.

Orchestration

Spanning across all phases of the ETL pipeline is orchestration. ETL jobs in HDInsight often involved several different products working in conjunction with each other. You might use Hive to clean some portion of the data, while Pig cleans another portion. You might use Azure Data Factory to load data into Azure SQL Database from Azure Data Lake Store.

Orechestration is needed to run the appropriate job at the appropriate time.

Oozie

Apache Oozie is a workflow/coordination system that manages Hadoop jobs. It runs within an HDInsight cluster and is integrated with the Hadoop stack. It supports Hadoop jobs for Apache MapReduce, Apache Pig, Apache Hive, and Apache Sqoop. It can also be used to schedule jobs that are specific to a system, such as Java programs or shell scripts.

For more information, see [Use Oozie with Hadoop to define and run a workflow on HDInsight](#)

For a deep dive showing how to use Oozie to drive an end-to-end pipeline, see [Operationalize the Data Pipeline](#)

Azure Data Factory

Azure Data Factory provides orchestration capabilities in the form of platform-as-a-service. It is a cloud-based data integration service that allows you to create data-driven workflows in the cloud for orchestrating and automating data movement and data transformation. Using Azure Data Factory, you can create and schedule data-driven workflows (called pipelines) that can ingest data from disparate data stores, process/transform the data by using compute services such as Azure HDInsight Hadoop, Spark, Azure Data Lake Analytics, Azure Batch, and Azure Machine Learning, and publish output data to data stores such as Azure SQL Data Warehouse for business intelligence (BI) applications to consume.

For more information on Azure Data Factory, see the [documentation](#).

Ingest File Storage And Result Storage

Source data files are typically loaded into a location in Azure Storage or Azure Data Lake Store. As stated above, files can be any format, but typically they are flat files like CSVs.

Azure Storage

Azure Storage has [very specific scalability targets](#) that you should familiarize yourself with. For most analytic nodes, Azure Storage scales best when dealing with many smaller files. Azure Storage guarantees the same performance, no matter how many files or how large the files (as long as you are within your limits.) This means that you can store terabytes of data and still get consistent performance, regardless if you are using a subset of the data or all of the data.

Azure Storage also has several different types of blobs. One in particular, append blob, is a great option for storing web logs or sensor data.

Blobs can be distributed across many servers in order to scale out access to them, but a single blob can only be served by a single server. While blobs can be logically grouped in blob containers, there are no partitioning implications from this grouping.

Azure Storage also has a WebHDFS API layer for the blob storage. This means that all of the services in HDInsight can access files in Azure Blob Storage for data cleaning and data processing, similar to how those services would use Hadoop Distributed Files System (HDFS).

Data is typically ingested into Azure Storage using either PowerShell, the Azure Storage SDK, or AZCopy.

Azure Data Lake Store

Azure Data Lake Store (ADLS) is a managed, hyperscale repository for analytics data that is compatible with HDFS. It doesn't actually use HDFS, but its design paradigm is very similar and offers unlimited scalability (in terms of total capacity and the size of individual files). As such, ADLS is very good when working with large files, since a large file can be stored across multiple nodes. The user never needs to think about how to partition data in ADLS, as it's done behind the scenes. You get massive throughput to run analytic jobs with thousands of concurrent executors that efficiently read and write hundreds of terabytes of data.

Data is typically ingested into ADLS using Azure Data Factory, ADLS SDKs, AdlCopy Service, Apache DistCp, or Apache Sqoop. Which of these services to use might largely depend on where the data is. If the data is currently in an existing Hadoop cluster, you might use Apache DistCp, AdlCopy Service, or Azure Data Factory. If it's in Azure Blob Storage, you might use Azure Data Lake Store .NET SDK, Azure PowerShell, or Azure Data Factory.

It is also optimized for event ingestion using Azure Event Hub or Apache Storm.

Considerations with Both Storage options

For uploading datasets that range in several terabytes, network latency can be a major problem, particularly if the data is coming from an on-premise location. In such cases, you can use the options below:

- Azure ExpressRoute: Azure ExpressRoute lets you create private connections between Azure datacenters and infrastructure on your premises. This provides a reliable option for transferring large amounts of data. For more information, see [Azure ExpressRoute documentation](#).
- "Offline" upload of data. If using Azure ExpressRoute is not feasible for any reason, you can use [Azure Import/Export service](#) to ship hard disk drives with your data to an Azure data center. Your data is first uploaded to Azure Storage Blobs. You can then use [Azure Data Factory](#) or [AdlCopy](#) tool to copy data from Azure Storage Blobs to Data Lake Store.

Azure SQL Data Warehouse

Azure SQL DW is a great choice to store cleaned and prepared results for future analytics. Azure HDInsight can be

used to perform those services for Azure SQL DW.

Azure SQL Data Warehouse (SQL DW) is a relational database store that is optimized for analytic workloads. Azure SQL DW scales based on partitioned tables. Tables can be partitioned across multiple nodes. Azure SQL DW nodes are selected at the time of creation. They can scale after the fact, but that's an active process that might require data movement. This is sometimes a time consuming and expensive process, so should be done carefully. See [SQL Data Warehouse - Manage Compute](#) for more information.

HBase

Apache HBase is a key-value store available in Azure HDInsight. Apache HBase is an open-source, NoSQL database that is built on Hadoop and modeled after Google BigTable. HBase provides performant random access and strong consistency for large amounts of unstructured and semistructured data in a schemaless database organized by column families.

Data is stored in the rows of a table, and data within a row is grouped by column family. HBase is a schemaless database in the sense that neither the columns nor the type of data stored in them need to be defined before using them. The open-source code scales linearly to handle petabytes of data on thousands of nodes. It can rely on data redundancy, batch processing, and other features that are provided by distributed applications in the Hadoop ecosystem.

HBase is an excellent destination for Sensor and log data for future analysis.

HBase scalability is dependant on the number of nodes in the HDInsight cluster.

Azure SQL Database and Azure Database

Azure offers three different relational databases as Platform-as-a-Service(PAAS).

- [Azure SQL Database](#) is an implementation of Microsoft SQL Server.
- [Azure Database for MySQL](#) is an implementation of Oracle MySQL.
- [Azure Database for PostgreSQL](#) is an implementation of PostgreSQL.

These products scale up, which means that they are scaled by adding more CPU and memory. You can also choose to use premium disks with the products for better I/O performance.

For more information on performance, see [Tuning Performance in Azure SQL Database](#). Information on tuning Azure Database for MySQL and Azure Database for PostgreSQL will be forthcoming.

Azure Analysis Services

Azure Analysis Services (AAS) is an analytical data engine used in decision support and business analytics, providing the analytical data for business reports and client applications such as Power BI, Excel, Reporting Services reports, and other data visualization tools.

Cubes can scale by changing tiers for each individual cube. For more information, see [Azure Analysis Services Pricing](#).

Extract and Load

Once the data exists in Azure, we can use many services to extract and load it into other products. HDInsight supports Sqoop and Flume.

Sqoop

Apache Sqoop is a tool designed for efficiently transferring data between structured, semi-structured and unstructured data sources.

Sqoop uses MapReduce to import and export the data, which provides parallel operation as well as fault tolerance.

Flume

Apache Flume is a distributed, reliable, and available service for efficiently collecting, aggregating, and moving large amounts of log data. It has a simple and flexible architecture based on streaming data flows. It is robust and fault tolerant with tunable reliability mechanisms and many failover and recovery mechanisms. It uses a simple extensible data model that allows for online analytic application.

Apache Flume cannot be used with Azure HDInsight. An on-premise Hadoop installation can use Flume to send data to either Azure Storage Blobs or Azure Data Lake Store. For more information, see [this blog post](#).

Transform

Once data exists in the chosen location, we need to actually clean it, combine it, or prepare it for a specific usage pattern. Hive, Pig, and Spark SQL are all very good choices for that kind of work. They are all supported on HDInsight.

See [Using Apache Hive as an ETL Tool](#) for more information on Hive.

See [Use Pig with Hadoop on HDInsight](#) for more information on Pig.

Streaming at Scale

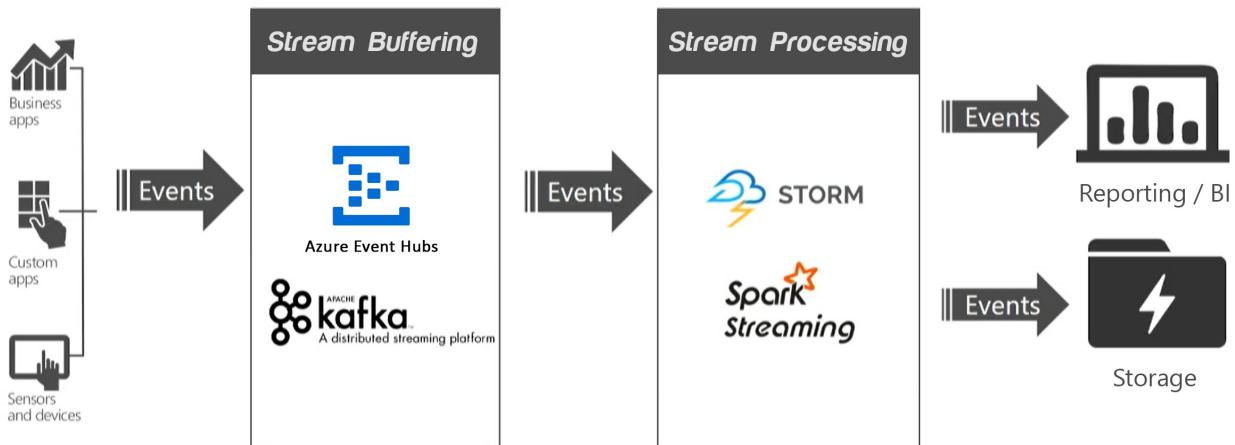
8/16/2017 • 5 min to read • [Edit Online](#)

Many of today's Big Data solutions must act on data in motion at any given point in time. In other words, realtime streaming data. In most cases, this data is most valuable at its time of arrival. Being able to quickly scale your solution by adding nodes on demand can be the difference between missing out on incoming information and making a key decision.

To ease the process of scaling streaming platforms, HDInsight offers an elegant scale model without the need to throttle your resources. All of the necessary steps are handled for you when scaling is performed through the HDInsight utilities.

What does a streaming architecture look like?

From a very high level, you have one or more data sources generating events (sometimes in the millions per second) that need to be ingested very quickly to avoid dropping any useful information. This is handled by the Stream Buffering, or event queuing, layer by a service such as [Kafka](#) or [Event Hubs](#). Once you collect your events, you can then analyze the data using any real-time analytics system within the Stream Processing layer, such as [Storm](#), [Spark Streaming](#), or similar. This processed data can be stored in long-term storage systems, like , or displayed in real-time on a business intelligence dashboard, such as [Power BI](#), Tableau, or a custom web page.



Apache Kafka

Apache Kafka provides high throughput, low-latency message queueing service, originally developed at LinkedIn, and is now part of the Apache suite of Open Source Software (OSS). It uses a publish and subscribe messaging model and stores streams of partitioned data safely in a distributed, replicated cluster. When needed, it scales linearly as throughput increases.

Read [Introducing Apache Kafka on HDInsight \(preview\)](#) for more information.

Apache Storm

Apache Storm is one of the stream processing engines we displayed in the first and second diagram at the top of this article. In summary, it is a distributed, fault-tolerant, open-source computation system that is optimized for processing streams of data in real time with Hadoop. The core unit of data for an event is in the form of a Tuple, which is an immutable set of key/value pairs. An unbounded sequence of these Tuples form a Stream, which is provided by a Spout. The Spout wraps a streaming data source (such as Kafka), and emits Tuples. A storm

Topology is a sequence of transformations on these streams.

Read [What is Apache Storm on Azure HDInsight?](#) for more information.

Deploy a new Azure virtual network with Kafka and Storm clusters:

[Deploy to Azure >](#)

Spark Streaming

The other stream processing engine we displayed in the diagram was Spark Streaming. Since it is an extension to Spark, Spark Streaming allows you to reuse the same code that you use for batch processing, and even allows you to combine both batch and interactive queries in the same application. Unlike Storm, Spark Streaming provides stateful exactly-once processing semantics out of the box. When used in combination with the [Kafka Direct API](#), which ensures that all Kafka data is received by Spark Streaming exactly once, it is possible to achieve end-to-end exactly-once guarantees. One of Spark Streaming's strengths is its fault-tolerant capabilities, recovering faulted nodes rapidly when multiple nodes are being used within the cluster.

Read [What is Spark Streaming?](#) for more information.

Use the following button to deploy a new Azure virtual network, Kafka, and Spark clusters to your Azure subscription:

[Deploy to Azure >](#)

Scale

Although you can specify the number of nodes in your cluster during creation, you may want to grow or shrink the cluster to match workload. All HDInsight clusters allow you to [change the number of nodes in the cluster](#). Also, Spark clusters can be dropped with no loss of data since all the data is stored in Azure Storage or Data Lake Store.

Scaling the Stream Buffering layer

There are advantages to decoupling technologies as we've shown in the first two diagrams. For instance, since Kafka is an event buffering technology, it is very IO-heavy and does not need a lot of processing power. The stream processors such as Spark Streaming, on the other hand, are very compute-heavy, requiring more powerful VMs by comparison. By having these technologies decoupled into different clusters, you can scale them independently and also use correctly sized VMs within those clusters for cost savings.

The two options we've shown for handling the stream buffering tasks, Event Hubs and Kafka, both use partitions, and consumers read from those partitions. Scaling the input throughput means scaling up the number of partitions. Adding partitions means increasing parallelism. In Event Hubs, the partition count cannot be changed after deployment so it is important to start with the target scale in mind. With Kafka, it is possible to [add partitions](#), even while it is processing data. Kafka provides a tool to reassign partitions, called `kafka-reassign-partitions.sh`. As mentioned earlier in this article, HDInsight provides a [partition replica rebalancing tool](#), called `rebalance_rackaware.py`. Under the covers, this tool executes the `kafka-reassign-partitions.sh` tool, but does so in such a way that each replica is in a separate fault domain and update domain, making Kafka rack aware, increasing fault tolerance as well as rebalancing the partitions.

Scaling the Stream Processing layer

Focusing on HDInsight for the topic of scaling the stream processors, both Apache Storm and Spark Streaming support adding worker nodes to their clusters, even while data is being processed.

To take advantage of new nodes added through scaling when using Storm, you need to rebalance any Storm topologies started before the cluster size was increased. This can be performed through the Storm web UI or the Command-line interface (CLI) tool. Refer to the [Apache Storm documentation](#) for more details.

Apache Spark uses three key parameters for configuring its environment, depending on application requirements: `spark.executor.instances`, `spark.executor.cores`, and `spark.executor.memory`. An Executor is a process that is launched for a Spark application. It runs on the worker node and is responsible for carrying out the tasks for the application. The default number of executors and the executor sizes for each cluster is calculated based on the number of worker nodes and the worker node size. These are stored in the `spark-defaults.conf` on the cluster head nodes.

The three configuration parameters can be configured at the cluster level (for all applications that run on the cluster) or can be specified for each individual application as well. Detailed information on these settings and how to manage the configuration can be found within the [Managing resources for Apache Spark cluster on Azure HDInsight](#) article.

Next steps

Learn more about real-time analytics solutions with Storm and Apache Spark on HDInsight:

- [Get started with Apache Storm on HDInsight](#)
- [Example topologies for Apache Storm on HDInsight](#)
- [Introduction to Spark on HDInsight](#)
- [Start with Apache Kafka on HDInsight](#)

Machine learning on HDInsight

8/16/2017 • 4 min to read • [Edit Online](#)

HDInsight enables machine learning against big data, providing the ability to obtain valuable insight from large (petabytes, or even exabytes) of structured, unstructured, and fast-moving data. There are several machine learning options that run in HDInsight:

SparkML and MLlib

[HDInsight Spark](#) is an Azure-hosted offering of [Spark](#), a unified, open source, parallel data processing framework supporting in-memory processing to boost Big Data analytics. The Spark processing engine is built for speed, ease of use, and sophisticated analytics. Spark's in-memory distributed computation capabilities make it a good choice for the iterative algorithms used in machine learning and graph computations. There are two scalable machine learning libraries that bring the algorithmic modeling capabilities to this distributed environment: MLlib and SparkML. MLlib contains the original API built on top of RDDs. SparkML is a newer package that provides a higher-level API built on top of DataFrames for constructing ML pipelines. SparkML does not support all of the same features of MLlib yet, but will eventually replace MLlib as Spark's standard machine learning library.

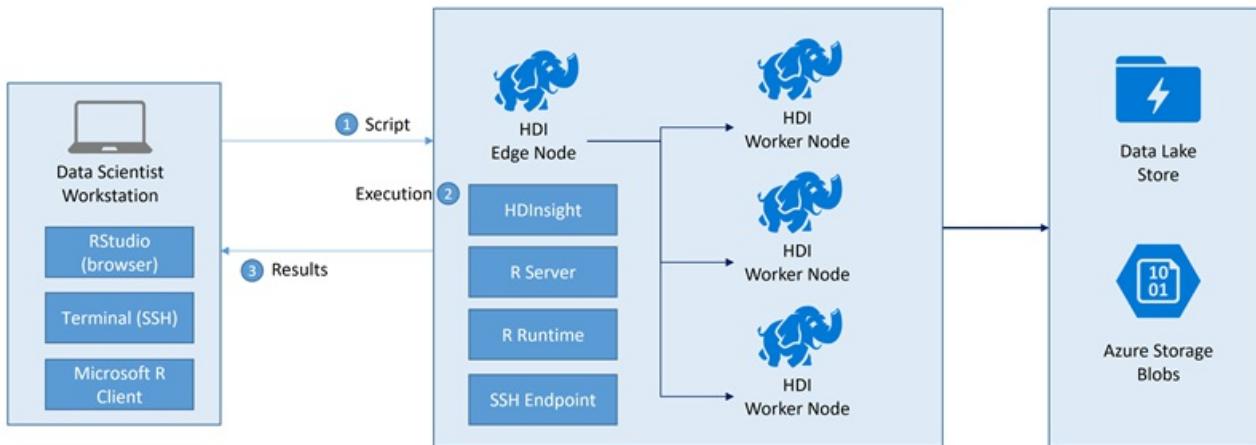
In our effort to contribute to the open source community, helping drive forward innovation in this space, we are excited to announce the Microsoft Machine Learning library for Apache Spark ([MMLSpark](#)). This is a library that is designed to make data scientists more productive on Spark, increase the rate of experimentation, and leverage cutting-edge machine learning techniques, including deep learning, on very large datasets. We've found that many have struggled with SparkML's low-level APIs when building scalable ML models, such as indexing strings, coercing data into a layout expected by machine learning algorithms, and assembling feature vectors. The MMLSpark library simplifies these and other common tasks for building models in PySpark.

R

[R](#) is currently the most popular statistical programming language in the world. It is an open source data visualization tool with a community of over 2.5 million users and growing. Given its thriving user base, and over 8,000 contributed packages, R is the natural choice for many companies who require machine learning. As part of HDInsight, you can now create an HDInsight cluster with R Server ready to be used with massive datasets and models. This new capability provides data scientists and statisticians with a familiar R interface that can scale on-demand through HDInsight, without the overhead of cluster setup and maintenance.

Training for Prediction with R Server

R across the HDInsight cluster

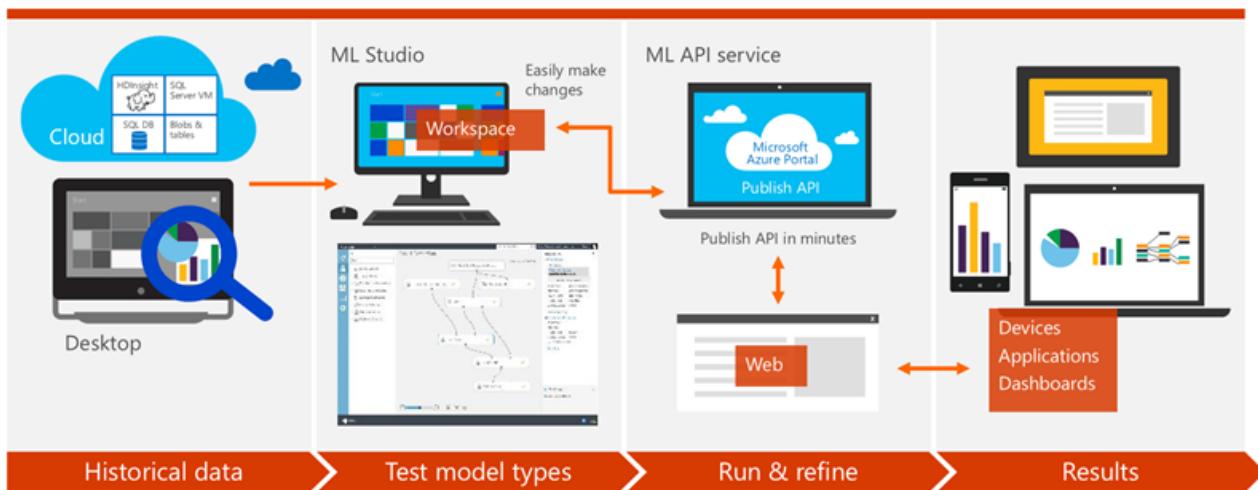


The edge node of a cluster provides a convenient place to connect to the cluster and to run your R scripts. You also have the option to run them across the nodes of the cluster by using ScaleR's Hadoop Map Reduce or Spark compute contexts.

Using R Server on HDInsight with Spark, you can parallelize training across the nodes of a cluster by using a Spark compute context. You can run R scripts directly on the edge node, using all available cores in parallel, if desired. Alternately, you can run your code from the edge node to kick off processing that is distributed across all nodes in the cluster. R Server on HDInsight with Spark also enables parallelizing functions from open source R packages, if desired.

Azure Machine Learning and Hive

Making advanced analytics accessible to Hadoop Microsoft Azure Machine Learning



Azure Machine Learning provides tools to model predictive analytics, as well as a fully managed service you can use to deploy your predictive models as ready-to-consume web services. Azure Machine Learning provides tools for creating complete predictive analytics solutions in the cloud to quickly create, test, operationalize, and manage predictive models. You do not need to buy any hardware nor manually manage virtual machines. Select from a large algorithm library, use a web-based studio for building models, and easily deploy your model as a web service.

Create features for data in an HDInsight Hadoop cluster using [Hive queries](#). Feature engineering attempts to increase the predictive power of learning algorithms by creating features from raw data that help facilitate the learning process. You can run HiveQL queries from Azure ML, and access data processed in Hive and stored in blob storage, by using the [Import Data module](#).

Deep learning

[Deep learning](#) is a branch of machine learning that uses deep neural networks, inspired by the biological processes of the human brain. Many researchers see deep learning as a very promising approach for making artificial intelligence better. Some examples of deep learning are spoken language translators, image recognition systems, and machine reasoning.

To help advance its own work in deep learning, Microsoft has developed the free, easy-to-use, open-source [Microsoft Cognitive Toolkit](#). The toolkit is being used extensively by a wide variety of Microsoft products, by companies worldwide with a need to deploy deep learning at scale, and by students interested in the very latest algorithms and techniques.

See also

Scenarios

- [Spark with Machine Learning: Use Spark in HDInsight for analyzing building temperature using HVAC data](#)
- [Spark with Machine Learning: Use Spark in HDInsight to predict food inspection results](#)
- [Generate movie recommendations with Mahout](#)
- [Hive and Azure Machine Learning](#)
- [Hive and Azure Machine Learning end-to-end](#)
- [Machine learning with Spark on HDInsight](#)

Deep learning resources

- [Deep learning toolkit with Spark](#)
- [Embarrassingly parallel image classification with Cognitive toolkit + Tensorflow on Spark](#)

Achieving Batch & Interactive with Hadoop & Spark

8/16/2017 • 24 min to read • [Edit Online](#)

Hadoop and Spark on HDInsight provide various data processing options, from [real-time stream processing](#), to complicated batch processing that can take from tens of minutes to days to complete. Interactive querying means querying batch data at "human" interactive speeds, meaning results are ready in time frames measured in seconds to minutes. The purpose of this article is to introduce batch and interactive query processing concepts, and how Hadoop and Spark help you achieve those goals.

Batch processing in HDInsight

Batch processing is used for a variety of scenarios, from creating an [Extract, Transform, and Load \(ETL\) pipeline](#), to working with very large data sets where computation takes a significant amount of time. When defined in terms of latency, batch processing are queries or programs that take tens of minutes, hours, or days to complete.

A common example of batch processing is transforming large inbound data, consisting of flat, unstructured files, into a schematized format that can be used for further querying. This is often referred to as *schema on read*, where the schema is applied while loading the data from disk, typically converting the flat file format to a binary format, allowing you to work with the data in an easier, more performant manner.

Another tenet of batch processing is the use of *external* or *managed* (internal) tables for schema on read. The difference between the two being the authority granted to the batch processing solution over the data. With external tables, the underlying data is typically shared between the batch processing solution and other systems, whereas managed tables are owned by the batch processing solution, which effectively controls the lifecycle of the data.

There are three general use cases and corresponding models, described below, that are appropriate for the typical batch processing workloads on an HDInsight cluster. Understanding these use cases will help you to start making decisions on how best to integrate HDInsight with your organization, and with your existing tools.

Iterative exploration

This model is typically chosen for experimenting with data sources to discover if they can provide useful information, and for handling data that you cannot process using existing systems. For example, you might collect feedback from customers through email, web pages, or external sources such as social media sites, then analyze it to get a picture of user sentiment for your products. You might be able to combine this information with other data, such as demographic data that indicates population density and characteristics in each city where your products are sold. See [Iterative data exploration](#) for more information.

Data warehouse on demand

Hadoop-based big data systems such as HDInsight allow you to store both the source data and the results of queries executed over this data. You can also store schemas (or, to be precise, metadata) for tables that are populated by the queries you execute. These tables can be indexed, although there is no formal mechanism for managing key-based relationships between them. However, you can create data repositories that are robust and reasonably low cost to maintain, which is especially useful if you need to store and manage huge volumes of data. See [Data warehouse on demand](#) for more information.

ETL automation

Hadoop-based big data systems such as HDInsight can be used to extract and transform data before you load it into your existing databases or data visualization tools. Such solutions are well suited to performing categorization and normalization of data, and for extracting summary results to remove duplication and redundancy. This is typically referred to as an Extract, Transform, and Load (ETL) process. Refer to the [ETL deep dive](#) for detailed

information.

Batch processing in HDInsight can be achieved using several available services:

Batch processing using Apache Hadoop MapReduce

The MapReduce programming model is a traditional batch processing method, designed for processing big data sets in parallel. Input data is split into independent chunks, which are then processed in parallel across the nodes in your cluster. A MapReduce job consists of two functions:

- **Mapper:** Consumes input data, analyzes it (usually with filter and sorting operations), and emits tuples (key-value pairs)
- **Reducer:** Consumes tuples emitted by the Mapper and performs a summary operation that creates a smaller, combined result from the Mapper data

Apache Hadoop MapReduce implements the MapReduce programming model by splitting data into subsets to be processed in a parallel map phase, followed by reducing the mapped data in aggregate to yield the desired result.

You can implement MapReduce code in Java (which is the native language for MapReduce jobs in all Hadoop distributions), or in a number of other supported languages including JavaScript, Python, C#, and F# through Hadoop Streaming.

Creating map and reduce components

The following code sample shows a commonly referenced JavaScript map/reduce example that counts the words in a source that consist of unstructured text data.

```
var map = function (key, value, context) {
  var words = value.split(/[^a-zA-Z]/);
  for (var i = 0; i < words.length; i++) {
    if (words[i] !== "") {
      context.write(words[i].toLowerCase(), 1);
    }
  }
};

var reduce = function (key, values, context) {
  var sum = 0;
  while (values.hasNext()) {
    sum += parseInt(values.next());
  }
  context.write(key, sum);
};
```

The map function splits the contents of the text input into an array of strings using anything that is not an alphabetic character as a word delimiter. Each string in the array is then used as the key of a new key/value pair with the value set to 1.

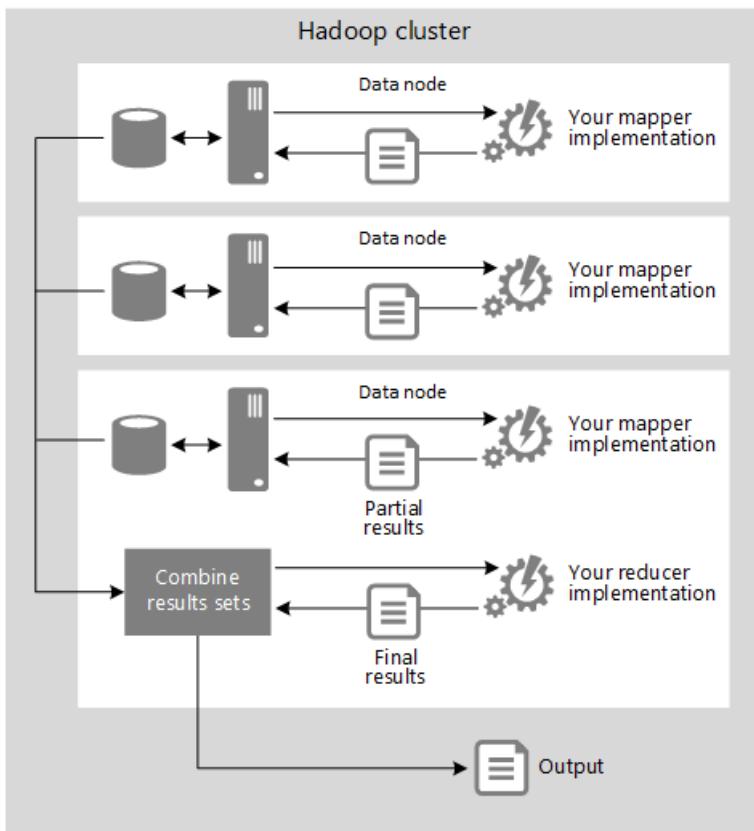
Each key/value pair generated by the map function is passed to the reduce function, which sums the values in key/value pairs that have the same key. Working together, the map and reduce functions determine the total number of times each unique word appeared in the source data, as shown here:

```
Aardvark 2
About 7
Above 12
Action 3
...
```

Using Hadoop streaming

The Hadoop core within HDInsight supports a technology called Hadoop Streaming that allows you to interact with the MapReduce process and run your own code outside of the Hadoop core as a separate executable process. The

below diagram shows a high-level overview of the way that streaming works.



This diagram shows how streaming executes the map and reduce components as separate processes. The schematic does not attempt to illustrate all of the standard map/reduce stages, such as sorting and merging the intermediate results or using multiple instances of the reduce component.

When using Hadoop Streaming, each node passes the data for the map part of the process to a separate process through the standard input (`stdin`), and accepts the results from the code through the standard output (`stdout`), instead of internally invoking a map component written in Java. In the same way, the node(s) that execute the reduce process pass the data as a stream to the specified code or component, and accept the results from the code as a stream, instead of internally invoking a Java reduce component.

Streaming has the advantage of decoupling the map/reduce functions from the Hadoop core, allowing almost any type of components to be used to implement the mapper and the reducer. The only requirement is that the components must be able to read from and write to the standard input and output.

Using the streaming interface does have a minor impact on performance. The additional movement of the data over the streaming interface can marginally increase query execution time. Streaming tends to be used mostly to enable the creation of map and reduce components in languages other than Java. It is quite popular when using Python, and also enables the use of .NET languages such as C# and F# with HDInsight.

For more information, see [Hadoop Streaming](#).

For more information on using Hadoop MapReduce on HDInsight, see [Use MapReduce in Hadoop on HDInsight](#)

For examples of using Hadoop streaming with HDInsight, see the following documents:

- [Develop C# MapReduce jobs](#)
- [Develop Python MapReduce jobs](#)

Batch processing using Hive

Hive uses tables to impose a schema on data, and to provide a query interface for client applications. The key difference between Hive tables and those in traditional database systems, such as SQL Server, is that Hive adopts a

"schema on read" (as mentioned earlier in the article) approach that enables you to be flexible about the specific columns and data types that you want to project onto your data.

Hive supports most of the data types you would expect (`bigint`, `binary`, `boolean`, `char`, `decimal`, `double`, `float`, `int`, `smallint`, `string`, `timestamp`, and `tinyint`), but also has specialized support for arrays, maps, and structs.

You can create multiple tables with different schemas from the same underlying data, depending on how you want to use that data. You can also create views and indexes over Hive tables, and partition tables. Moving data into a Hive-controlled namespace is usually an instantaneous operation.

You can use the Hive command line on the HDInsight cluster to work with Hive tables, and build an automated solution that includes Hive queries by using the HDInsight .NET SDKs and with a range of Hadoop-related tools such Oozie and WebHCat. You can also use the Hive ODBC driver to connect to Hive from any ODBC-capable client application.

In addition to its more usual use as a querying mechanism, Hive can be used to create a simple data warehouse containing table definitions applied to data that you have already processed into the appropriate format. Azure storage is relatively inexpensive, and so this is a good way to create a commodity storage system when you have huge volumes of data.

Hive queries are written in HiveQL (see [HiveQL language reference](#)), which is a query language similar to SQL. Here are a few example queries for different scenarios:

Creating a table

You create tables by using the HiveQL `CREATE TABLE` statement, which in its simplest form looks similar to the equivalent statement in Transact-SQL. You specify the schema in the form of a series of column names and types, and the type of delimiter that Hive will use to delineate each column value as it parses the data. You can also specify the format for the files in which the table data will be stored if you do not want to use the default format (where data files are delimited by an ASCII code 1 (Octal \001) character, equivalent to Ctrl + A). For example, the following code creates a table named **mytable** and specifies that the data files for the table should be tab-delimited.

```
CREATE TABLE mytable (col1 STRING, col2 INT)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '\t';
```

Hive tables are simply metadata definitions imposed on data in underlying files. By default, Hive stores table data in the **user/hive/warehouse/table_name** path in storage (the default path is defined in the configuration property `hive.metastore.warehouse.dir`), so the previous code sample will create the table metadata definition and an empty folder at **user/hive/warehouse/mytable**. When you delete the table by executing the `DROP TABLE` statement, Hive will delete the metadata definition from the Hive database and it will also remove the **user/hive/warehouse/mytable** folder and its contents.

However, you can specify an alternative path for a table by including the `LOCATION` clause in the `CREATE TABLE` statement. The ability to specify a non-default location for the table data is useful when you want to enable other applications or users to access the files outside of Hive. This allows data to be loaded into a Hive table simply by copying data files of the appropriate format into the folder, or downloaded directly from storage. When the table is queried using Hive, the schema defined in its metadata is automatically applied to the data in the files.

An additional benefit of specifying the location is that this makes it easy to create a table for data that already exists in that location (perhaps the output from a previously executed map/reduce job or Pig script). After creating the table, the existing data in the folder can be retrieved immediately with a HiveQL query.

However, one consideration for using managed tables is that, when the table is deleted, the folder it references will also be deleted — even if it already contained other data files when the table was created. If you want to manage

the lifetime of the folder containing the data files separately from the lifetime of the table, you must use the `EXTERNAL` keyword in the `CREATE TABLE` statement to indicate that the folder will be managed externally from Hive, as shown in the following code sample:

```
CREATE EXTERNAL TABLE mytable (col1 STRING, col2 INT)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '\t'
STORED AS TEXTFILE LOCATION '/mydata/mytable';
```

In HDInsight the location shown in this example corresponds to `wabss://[container-name]@[storage-account-name].blob.core.windows.net/mydata/mytable` in Azure storage.

This ability to manage the lifetime of the table data separately from the metadata definition of the table means that you can create several tables and views over the same data, but each can have a different schema. For example, you may want to include fewer columns in one table definition to reduce the network load when you transfer the data to a specific analysis tool, but have all of the columns available for another tool.

As a general guide you should:

- Use `INTERNAL` tables (the default, commonly referred to as managed tables) when you want Hive to manage the lifetime of the table or when the data in the table is temporary; for example, when you are running experimental or one-off queries over the source data.
- Use `INTERNAL` tables and also specify the `LOCATION` for the data files when you want to access the data files from outside of Hive; for example, if you want to upload the data for the table directly into the Azure storage location.
- Use `EXTERNAL` tables when you want to manage the lifetime of the data, when data is used by processes other than Hive, or if the data files must be preserved when the table is dropped. However, notice that cannot use `EXTERNAL` tables when you implicitly create the table by executing a `SELECT` query against an existing table.

Querying tables with HiveQL

After you have created tables and loaded data files into the appropriate locations you can query the data by executing HiveQL `SELECT` statements against the tables. HiveQL `SELECT` statements are similar to SQL, and support common operations such as `JOIN`, `UNION`, `GROUP BY`, and `ORDER BY`. For example, you could use the following code to query the `mytable` table described earlier:

```
SELECT col1, SUM(col2) AS total
FROM mytable
GROUP BY col1;
```

When designing an overall data processing solution with HDInsight, you may choose to perform complex processing logic in custom MapReduce components or Pig scripts and then create a layer of Hive tables over the results of the earlier processing, which can be queried by business users who are familiar with basic SQL syntax. However, you can use Hive for all processing, in which case some queries may require logic that is not possible to define in standard HiveQL functions.

In addition to common SQL semantics, HiveQL supports the use of:

- Custom MapReduce scripts embedded in a query through the `MAP` and `REDUCE` clauses.
- Custom user-defined functions (UDFs) that are implemented in Java, or that call Java functions available in the existing installed libraries.]
- XPath functions for parsing XML data using XPath.

This extensibility enables you to use HiveQL to perform complex transformations on data as it is queried. To help you decide on the right approach, consider the following guidelines:

- If the source data must be extensively transformed using complex logic before being consumed by business users, consider using custom MapReduce components or Pig scripts to perform most of the processing, and create a layer of Hive tables over the results to make them easily accessible from client applications.
- If the source data is already in an appropriate structure for querying and only a few specific but complex transforms are required, consider using MapReduce scripts embedded in HiveQL queries to generate the required results.
- If queries will be created mostly by business users, but some complex logic is still regularly required to generate specific values or aggregations, consider encapsulating that logic in custom UDFs because these will be simpler for business users to include in their HiveQL queries than a custom MapReduce script.

For more information on Hive and HiveQL, see [What is Apache Hive and HiveQL on Azure HDInsight?](#).

Batch processing using Pig

Pig Latin syntax has some similarities to LINQ, and encapsulates many functions and expressions that make it easy to create a sequence of complex data transformations with just a few lines of simple code. Pig Latin is a good choice for creating relations and manipulating sets, and for working with unstructured source data. You can always create a Hive table over the results of a Pig Latin query if you want table format output. However, the syntax of Pig Latin can be complex for non-programmers to master. Pig Latin is not as familiar or as easy to use as HiveQL, but Pig can achieve some tasks that are difficult, or even impossible, when using Hive.

You can run Pig Latin statements interactively in the Hadoop command line window or in a command line Pig shell named Grunt. You can also combine a sequence of Pig Latin statements in a script that can be executed as a single job, and use user-defined functions you previously uploaded to HDInsight. The Pig Latin statements are used by the Pig interpreter to generate jobs, but the jobs are not actually generated and executed until you call either a `DUMP` statement (which is used to display a relation in the console, and is useful when interactively testing and debugging Pig Latin code) or a `STORE` statement (which is used to store a relation as a file in a specified folder).

Pig scripts generally save their results as text files in storage, where they can easily be viewed on demand, perhaps by using the Hadoop command line window. However, the results can be difficult to consume or process in client applications unless you copy the output files and import them into client tools such as Excel.

Executing a Pig script

As an example of using Pig, suppose you have a tab-delimited text file containing source data similar to the following:

```
Value1 1
Value2 3
Value3 2
Value1 4
Value3 6
Value1 2
Value2 8
Value2 5
```

You could process the data in the source file with the following simple Pig Latin script:

```
A = LOAD '/mydata/sourcedata.txt' USING PigStorage('\t') AS (col1, col2:long);
B = GROUP A BY col1;
C = FOREACH B GENERATE group, SUM(A.col2) as total;
D = ORDER D BY total;
STORE D INTO '/mydata/results';
```

This script loads the tab-delimited data into a relation named **A**, imposing a schema that consists of two columns: **col1**, which uses the default byte array data type, and **col2**, which is a long integer. The script then creates a relation named **B**, in which the rows in **A** are grouped by **col1**, and then creates a relation named **C**, in which the

col2 value is aggregated for each group in **B**.

After the data has been aggregated, the script creates a relation named **D**, in which the data is sorted based on the total that has been generated. The relation **D** is then stored as a file in the **/mydata/results** folder, which contains the following text:

```
Value1 7
Value3 8
Value2 16
```

For more information about the Pig Latin syntax, see [Pig Latin Reference Manual 1](#) and [Pig Latin Reference Manual 2](#).

See [Use Pig with Hadoop on HDInsight](#) for more information on using Pig on HDInsight.

Batch processing using Spark

Apache Spark running on HDInsight provides another avenue for batch processing. Spark is a unified framework, meaning that you can use the same code for both batch processing and realtime [stream processing](#). The underlying components of Spark are Spark Core and the RDD API, which let you perform parallel operations on Spark's distributed data. Atop this sits the Spark SQL module, which provides the DataFrame API and DataSet API as well as support for issuing SQL queries directly against the DataFrame API.

A DataFrame is a distributed collection of data organized into named columns. It is important to note that the columns are named only, not typed. DataFrames are conceptually equivalent to a table in a relational database or a data frame in R/Python, but with richer optimizations under the hood. They can be constructed from a wide array of sources such as: structured data files, tables in Hive, external databases, or existing RDDs. The DataFrame API is available in Scala, Java, and Python.

DataFrames and DataSets are higher-level abstractions that improve batch processing efforts due to their expressiveness and higher performance over raw RDD APIs.

Learn more about using [Spark with HDInsight](#), including for batch processing.

Interactive querying in HDInsight

As opposed to batch processing, as described earlier in this article as data processing that takes tens of minutes to days to complete, interactive querying means querying batch data at human interactive speeds. In other words, you want to explore your data by executing multiple queries, but expect a result in seconds or even minutes. In this way, you are interacting with your data in such a way that it is possible to quickly distill its meaning or discover ways in which you can further interact with it through custom programs or BI reports.

Interactive querying using Hive and Tez

Hive is considered the de facto standard for interactive SQL queries over petabytes of data using Hadoop. When you provision a new Hive cluster in HDInsight, it comes preconfigured with settings that optimize Hive for interactive queries, such as storing data in the ORC file format, using vectorized SQL execution, and using the Apache Tez application framework.

Please see [Changing configs via Ambari](#) for specific Hive performance tuning steps that will help ensure the minimal latency for interactive queries.

Generally speaking, here are a few ways to optimize Hive for better interactive query performance:

Indexes

Hive uses indexes to increase the performance of read queries. Index data for a table is stored in a separate table. Here's a sample `CREATE INDEX` query:

```
SET hive.execution.engine=tez;
CREATE INDEX index_total_sales ON TABLE staging_weblogs(purchasetype,paymentamount)
AS 'COMPACT' WITH DEFERRED REBUILD LOCATION '/index_total_sales/';
```

The query creates an index on the `purchasetype` and `paymentamount` columns. The `CompactIndexHandler` is the Java class that implements indexing. The `WITH DEFERRED REBUILD` statement creates an empty index. It has to be rebuilt later.

To manually populate the index, execute the following, using the `ALTER INDEX` and `REBUILD` statements:

```
ALTER INDEX index_total_sales on staging_weblogs REBUILD;
```

Note: The above query populates the `index_total_sales` with data. To verify, navigate to your Azure Storage account on the Azure Portal, and search `index_total_sales` in the bookstore container.

Indexes are also automatically created when using the ORC file format, as recommended earlier. The ORC file format is a column-oriented storage format that is an optimized version of RC file format. It provides much better compression and faster query performance than the RC file format. ORC file format groups row data into stripes, with stripe metadata stored in a file footer. The default stripe size is 250 MB.

The file footer contains a list of stripes, number of rows per stripe, and column data type. It also contains lightweight indexes with column-level aggregations `sum`, `max`, `min`, and `count`. At the end of the file, a postscript contains compression parameters and size of the compressed footer.

Partitioning

Hive allows for splitting data into two or more horizontal partitions based on column values. This improves queries that filter data on specific column values. This, in effect, also helps you to prune large data sets, resulting in faster query times. The reason for this, is that when you partition your data, the ORC files are stored in the filesystem under the partition's respective name. For example, if you partition by `state`, then queries for "California" search within that state's directory, ignoring all of the other directories' worth of data files during the query processing step.

```
SET hive.exec.dynamic.partition = true;
SET hive.exec.dynamic.partition.mode=non-strict;

CREATE TABLE IF NOT EXISTS partitioned_orc_weblogs(
    transactionid varchar(50),
    transactiondate varchar(50) ,
    customerid varchar(50) ,
    bookid varchar(50) ,
    purchasetype varchar(50) ,
    orderid varchar(50) ,
    bookname varchar(50) ,
    categoryname varchar(50) ,
    quantity varchar(50) ,
    shippingamount varchar(50) ,
    invoicenumber varchar(50) ,
    invoicestatus varchar(50) ,
    paymentamount varchar(50)
)
PARTITIONED BY (
    state varchar(10),
    city varchar(50)
)
STORED AS ORC LOCATION '/partitionedorc/' tblproperties ("orc.create.index"="true");
```

The above query creates an ORC format table partitioned on the state and city columns. With dynamic partitioning enabled, Hive automatically creates partitions based on the specified column values. The setting

`hive.exec.dynamic.partition.mode = non-strict` indicates there are no static partitions and all partitions are dynamic. If a table has both dynamic and static partitions, `hive.exec.dynamic.partition.mode` should be set to `strict`.

Configuring YARN queues

For Hadoop, a single interactive queue is recommended. However, there may be cases where two queues are better. If one type of query takes 5 seconds to run, and another type takes around a minute, the shorter-running queries will have to wait for the longer-running queries to complete. The longer-running query types should be in a different queue.

See [Key Scenarios to Monitor](#) for more information on configuring YARN queues in Ambari.

Hive on Tez + LLAP

Live Long and Process (LLAP) added to Hive 2.0, is a hybrid execution model that consists of a long-lived daemon, that promises to bring subsecond query performance to Hive. The daemon processes functionalities such as caching, pre-fetching, and access control. Small or short queries are processed by the daemon whereas heavy queries are processed by the YARN containers.

LLAP is not an execution engine, rather it's an enhancement over the existing execution engine. This is available as Tech Preview in HDInsight 3.5+.

There are two ways to use LLAP at this point. The first way is to provision a new HDInsight cluster, selecting the **Interactive Hive (Preview)** cluster type. See [Use Interactive Hive in HDInsight \(Preview\)](#) for more information.

The other way is to enable LLAP on an existing Hive cluster. To do this, perform the following steps:

1. Open the [Azure portal](#) and select the HDInsight cluster. On the cluster pane, select **dashboard** from the menu. This will open the Ambari UI in a new tab.
2. In Ambari, select **Hive** from the left menu. In the resulting window, navigate to the **Configs** tab.
3. LLAP is enabled by switching on the **Interactive query** option. This is disabled by default.

Interactive Query

Enable Interactive Query (Tech Preview)

No

4. Click on the toggle switch to enable the Interactive Query. Click **Save** on the version control bar to save the changes made.

Please note, this feature is currently in preview and is not officially supported.

Interactive querying using Spark SQL

Like Hive LLAP, Spark executes against an in-memory data cache, potentially returning results within seconds, instead of minutes or hours in order to achieve interactivity. To achieve interactive-level response times, Spark SQL and the DataFrame API support high-performance columnar formats like Parquet and ORC. This enables pruning large data sets at query processing time through table partitioning (as detailed under "Interactive querying using Hive and Tez", above) and predicate pushdown. Through these and in-memory computing, Spark is extremely fast, particularly when the cluster is large enough to hold all of the data in memory. HDFS is optimized for batch processing, focusing on high throughput vs. fast random access to different areas of files, as is required for interactive processing. Spark overcomes this by loading the data in-memory, keeping the working sets in memory,

saving a significant amount of time on disk I/O. For datasets that can fit into a cluster's memory, Spark is fast enough to allow data scientists to interact and explore big data from an interactive shell. This can be performed by using Zeppelin or Jupyter notebooks, an interactive shell, or custom Spark applications that use the same API.

Every Spark for Azure HDInsight cluster has Jupyter and Zeppelin notebooks included. This allows users to do interactive and exploratory analysis in Scala, Python or SQL.

- See [Kernels for Jupyter notebook on Spark clusters in Azure HDInsight](#)
- See [Use Zeppelin notebooks with Apache Spark cluster on Azure HDInsight](#)

Spark SQL enables you to run unmodified Hive queries on existing data. It reuses the Hive frontend and metastore, giving users full compatibility with existing Hive data, queries, and UDFs.

Indexes

Spark SQL does not yet support indexes, at this time. This includes Hive table indexes, meaning if you use a `HiveContext` in Spark to query a Hive table that has explicit indexes, they will not be used in the query processing. However, as with Hive, you can take advantage of the automatically created inline indexes generated by ORC files when you save your data in that format.

Partitions

Partitioning in Spark SQL works in the same way as with Hive (as detailed under "Interactive querying using Hive and Tez" above), creating subfolders named after the column partition with the related values stored within. This effectively prunes the data by excluding entire directories-worth of data when querying by partition. You can partition your DataFrame by using the `partitionBy` method prior to invoking the `save` method:

```
mydataframe.write.format("orc").partitionBy("state").save("mydata")
```

The `partitionBy` method accepts a list of column names by which to partition the data. In this case, we're partitioning by the "state" column.

Next steps

In this article, we covered the various options by which we can perform batch processing and interactive querying, using Hadoop and Spark on HDInsight. Learn more about the batch and interactive processing by following the links below:

- Batch processing through [Iterative data exploration](#)
- Batch processing with [Data warehouse on demand](#)
- Learn more about ETL with the [ETL deep dive](#) and [ETL at scale](#)
- Batch processing with [MapReduce in Hadoop on HDInsight](#)
- More about [Hive and HiveQL on Azure HDInsight](#)
- [Use Pig with Hadoop on HDInsight](#) for batch processing
- Overview of [Spark on HDInsight](#)
- See [Changing configs via Ambari](#) for recommended performance settings
- See [Use Interactive Hive in HDInsight \(Preview\)](#) for LLAP
- Learn more about Apache Spark with [Spark scenarios](#), [Spark SQL with HDInsight](#), and Spark-specific performance enhancements with [Spark settings](#)

Run Custom Programs

8/16/2017 • 6 min to read • [Edit Online](#)

Hadoop-based big data systems such as HDInsight enable data processing using a wide range of tools and technologies. This article provides comparisons between the commonly used tools and technologies to help you choose the most appropriate for your own scenarios, then goes into detail on how to run custom map/reduce programs.

The following table shows the main advantages and considerations for each one.

Tools and technologies

QUERY MECHANISM	ADVANTAGES	CONSIDERATIONS
Hive using HiveQL	<ul style="list-style-type: none">An excellent solution for batch processing and analysis of large amounts of immutable data, for data summarization, and for ad hoc querying. It uses a familiar SQL-like syntax.It can be used to produce persistent tables of data that can be easily partitioned and indexed.Multiple external tables and views can be created over the same data.It supports a simple data warehouse implementation that provides massive scale out and fault tolerance capabilities for data storage and processing.	<ul style="list-style-type: none">It requires the source data to have at least some identifiable structure.It is not suitable for real-time queries and row level updates. It is best used for batch jobs over large sets of data.It might not be able to carry out some types of complex processing tasks.
Pig using Pig Latin	<ul style="list-style-type: none">An excellent solution for manipulating data as sets, merging and filtering datasets, applying functions to records or groups of records, and for restructuring data by defining columns, by grouping values, or by converting columns to rows.It can use a workflow-based approach as a sequence of operations on data.	<ul style="list-style-type: none">SQL users may find Pig Latin is less familiar and more difficult to use than HiveQL.The default output is usually a text file and so it is more difficult to use with visualization tools such as Excel. Typically you will layer a Hive table over the output.

QUERY MECHANISM	ADVANTAGES	CONSIDERATIONS
Custom map/reduce	<ul style="list-style-type: none"> It provides full control over the map and reduce phases and execution. It allows queries to be optimized to achieve maximum performance from the cluster, or to minimize the load on the servers and the network. The components can be written in a range of widely known languages that most developers are likely to be familiar with. 	<ul style="list-style-type: none"> It is more difficult than using Pig or Hive because you must create your own map and reduce components. Processes that require the joining of sets of data are more difficult to implement. Even though there are test frameworks available, debugging code is more complex than a normal application because they run as a batch job under the control of the Hadoop job scheduler.
HCatalog	<ul style="list-style-type: none"> It abstracts the path details of storage, making administration easier and removing the need for users to know where the data is stored. It enables notification of events such as data availability, allowing other tools such as Oozie to detect when operations have occurred. It exposes a relational view of data, including partitioning by key, and makes the data easy to access. 	<ul style="list-style-type: none"> It supports RCFile, CSV text, JSON text, SequenceFile and ORC file formats by default, but you may need to write a custom SerDe if you use other formats. HCatalog is not thread-safe. There are some restrictions on the data types for columns when using the HCatalog loader in Pig scripts. See HCatLoader Data Types in the Apache HCatalog documentation for more details.

QUERY MECHANISM	ADVANTAGES	CONSIDERATIONS
Apache Spark	<ul style="list-style-type: none"> One execution model for multiple tasks: Apache Spark leverages a common execution model for doing multiple tasks like ETL, batch queries, interactive queries, real-time streaming, machine learning, and graph processing on data stored in Azure Storage. In-memory processing for interactive scenarios: Apache Spark persists data in-memory and disk if needed to achieve up to 100x faster queries while processing large datasets in Hadoop. This makes Spark for Azure HDInsight ideal to speed up intensive big data applications. Developer friendly: Spark supports a variety of development languages like Java, Python, and Scala APIs to ease development. You can write sophisticated parallel applications with a collection of over 80 operators, allowing developers to rapidly iterate over data. 	<ul style="list-style-type: none"> Does not handle a large number of small files well. Requires more compute power compared to some other options, due to in-memory processing. The need for extra RAM could cause Spark solutions to be more costly, depending upon the nature of your data.

Typically, you will use the simplest of these approaches that can provide the results you require. For example, it may be that you can achieve these results by using just Hive, but for more complex scenarios you may need to use Pig or even write your own map and reduce components. You may also decide, after experimenting with Hive or Pig, that custom map and reduce components can provide better performance by allowing you to fine tune and optimize the processing.

More about custom map/reduce components

Map/reduce code consists of two separate functions implemented as **map** and **reduce** components. The **map** component is run in parallel on multiple cluster nodes, each node applying it to its own subset of the data. The **reduce** component collates and summarizes the results from all of the map functions (see [Use MapReduce in Hadoop on HDInsight](#) for more details on these two components).

In most HDInsight processing scenarios it is simpler and more efficient to use a higher-level abstraction such as Pig or Hive, although you can create custom map and reduce components for use within Hive scripts in order to perform more sophisticated processing.

Custom map/reduce components are typically written in Java. However, Hadoop provides a streaming interface that allows components to be used that are developed in other languages such as C#, F#, Visual Basic, Python, JavaScript, and more.

- See [Develop Java MapReduce programs for Hadoop on HDInsight](#) for a walkthrough on developing custom Java MapReduce programs.
- To see an example using Python, read [Develop Python streaming MapReduce programs for HDInsight](#).

You might consider creating your own map and reduce components when:

- You want to process data that is completely unstructured by parsing it and using custom logic in order to obtain structured information from it.
- You want to perform complex tasks that are difficult (or impossible) to express in Pig or Hive without resorting to creating a UDF. For example, you might need to use an external geocoding service to convert latitude and longitude coordinates or IP addresses in the source data to geographical location names.
- You want to reuse your existing .NET, Python, or JavaScript code in map/reduce components. You can do this using the Hadoop streaming interface.

Uploading and running your custom MapReduce program

The most common MapReduce programs are written in Java and compiled to a jar file. The steps to upload and run your custom MapReduce program are simple.

Once you have developed, compiled, and tested your MapReduce program, execute the following to upload your jar file to the headnode using the `scp` command:

```
scp mycustomprogram.jar USERNAME@CLUSTERNAME-ssh.azurehdinsight.net
```

Replace **USERNAME** with the SSH user account for your cluster. Replace **CLUSTERNAME** with the cluster name. If you used a password to secure the SSH account, you are prompted to enter the password. If you used a certificate, you may need to use the `-i` parameter to specify the private key file.

Next, connect to the cluster using [SSH](#).

```
ssh USERNAME@CLUSTERNAME-ssh.azurehdinsight.net
```

From the SSH session, execute your MapReduce program through YARN.

```
yarn jar mycustomprogram.jar mynamespace.myclass /example/data/sample.log /example/data/logoutput
```

This command submits the MapReduce job to YARN. The input file is `/example/data/sample.log`, and the output directory is `/example/data/logoutput`. Both the input file and the output file(s) are stored to the default storage for the cluster.

Next steps

This article introduced the landscape of commonly used tools that can be used to process your data, ending off with detailing how to run custom MapReduce programs. Learn more about the various available data processing tools, and methods to create and run custom programs that use them, by following the links below.

- [Use C# with MapReduce streaming on Hadoop in HDInsight](#)
- [Develop Java MapReduce programs for Hadoop on HDInsight](#)
- [Develop Python streaming MapReduce programs for HDInsight](#)
- [Use Azure Toolkit for Eclipse to create Spark applications for an HDInsight cluster](#)
- [Use Python User Defined Functions \(UDF\) with Hive and Pig in HDInsight](#)

Upload data for Hadoop jobs in HDInsight

8/16/2017 • 10 min to read • [Edit Online](#)

Azure HDInsight provides a full-featured Hadoop distributed file system (HDFS) over Azure Blob storage. It is designed as an HDFS extension to provide a seamless experience to customers. It enables the full set of components in the Hadoop ecosystem to operate directly on the data it manages. Azure Blob storage and HDFS are distinct file systems that are optimized for storage of data and computations on that data. For information about the benefits of using Azure Blob storage, see [Use Azure Blob storage with HDInsight](#).

Prerequisites

Note the following requirement before you begin:

- An Azure HDInsight cluster. For instructions, see [Get started with Azure HDInsight](#) or [Provision HDInsight clusters](#).

Why blob storage?

Azure HDInsight clusters are typically deployed to run MapReduce jobs, and the clusters are dropped after these jobs complete. Keeping the data in the HDFS clusters after computations are complete would be an expensive way to store this data. Azure Blob storage is a highly available, highly scalable, high capacity, low cost, and shareable storage option for data that is to be processed using HDInsight. Storing data in a blob enables the HDInsight clusters that are used for computation to be safely released without losing data.

Directories

Azure Blob storage containers store data as key/value pairs, and there is no directory hierarchy. However the "/" character can be used within the key name to make it appear as if a file is stored within a directory structure. HDInsight sees these as if they are actual directories.

For example, a blob's key may be *input/log 1.txt*. No actual "input" directory exists, but due to the presence of the "/" character in the key name, it has the appearance of a file path.

Because of this, if you use Azure Explorer tools you may notice some 0 byte files. These files serve two purposes:

- If there are empty folders, they mark of the existence of the folder. Azure Blob storage is clever enough to know that if a blob called **foo/bar** exists, there is a folder called **foo**. But the only way to signify an empty folder called **foo** is by having this special 0 byte file in place.
- They hold special metadata that is needed by the Hadoop file system, notably the permissions and owners for the folders.

Command-line utilities

Microsoft provides the following utilities to work with Azure Blob storage:

TOOL	LINUX	OS X	WINDOWS
Azure Command-Line Interface	✓	✓	✓
Azure PowerShell			✓

TOOL	LINUX	OS X	WINDOWS
AzCopy			✓
Hadoop command	✓	✓	✓

NOTE

While the Azure CLI, Azure PowerShell, and AzCopy can all be used from outside Azure, the Hadoop command is only available on the HDInsight cluster and only allows loading data from the local file system into Azure Blob storage.

Azure CLI

The Azure CLI is a cross-platform tool that allows you to manage Azure services. Use the following steps to upload data to Azure Blob storage:

[! IMPORTANT] Azure CLI support for managing HDInsight resources using Azure Service Manager (ASM) is **deprecated**, and was removed on January 1, 2017. The steps in this document use the new Azure CLI commands that work with Azure Resource Manager.

Please follow the steps in [Install and configure Azure CLI](#) to install the latest version of the Azure CLI. If you have scripts that need to be modified to use the new commands that work with Azure Resource Manager, see [Migrating to Azure Resource Manager-based development tools for HDInsight clusters](#) for more information.

1. [Install and configure the Azure CLI for Mac, Linux and Windows.](#)
2. Open a command prompt, bash, or other shell, and use the following to authenticate to your Azure subscription.

```
azure login
```

When prompted, enter the user name and password for your subscription.

3. Enter the following command to list the storage accounts for your subscription:

```
azure storage account list
```

4. Select the storage account that contains the blob you want to work with, then use the following command to retrieve the key for this account:

```
azure storage account keys list <storage-account-name>
```

This should return **Primary** and **Secondary** keys. Copy the **Primary** key value because it will be used in the next steps.

5. Use the following command to retrieve a list of blob containers within the storage account:

```
azure storage container list -a <storage-account-name> -k <primary-key>
```

6. Use the following commands to upload and download files to the blob:

- To upload a file:

```
azure storage blob upload -a <storage-account-name> -k <primary-key> <source-file>
<container-name> <blob-name>
```

- To download a file:

```
azure storage blob download -a <storage-account-name> -k <primary-key> <container-name>
<blob-name> <destination-file>
```

NOTE

If you will always be working with the same storage account, you can set the following environment variables instead of specifying the account and key for every command:

- **AZURE_STORAGE_ACCOUNT**: The storage account name
- **AZURE_STORAGE_ACCESS_KEY**: The storage account key

Azure PowerShell

Azure PowerShell is a scripting environment that you can use to control and automate the deployment and management of your workloads in Azure. For information about configuring your workstation to run Azure PowerShell, see [Install and configure Azure PowerShell](#).

IMPORTANT

Azure PowerShell support for managing HDInsight resources using Azure Service Manager is **deprecated**, and was removed on January 1, 2017. The steps in this document use the new HDInsight cmdlets that work with Azure Resource Manager.

Please follow the steps in [Install and configure Azure PowerShell](#) to install the latest version of Azure PowerShell. If you have scripts that need to be modified to use the new cmdlets that work with Azure Resource Manager, see [Migrating to Azure Resource Manager-based development tools for HDInsight clusters](#) for more information.

To upload a local file to Azure Blob storage

1. Open the Azure PowerShell console as instructed in [Install and configure Azure PowerShell](#).
2. Set the values of the first five variables in the following script:

```
$resourceGroupName = "<AzureResourceGroupName>"
$storageAccountName = "<StorageAccountName>"
$containerName = "<ContainerName>

$fileName = "<LocalFileName>"
$blobName = "<BlobName>

# Get the storage account key
$storageAccountKey = (Get-AzureRmStorageAccountKey -ResourceGroupName $resourceGroupName -Name
$storageAccountName)[0].Value
# Create the storage context object
$destContext = New-AzureStorageContext -StorageAccountName $storageAccountName -StorageAccountKey
$storageaccountkey

# Copy the file from local workstation to the Blob container
Set-AzureStorageBlobContent -File $fileName -Container $containerName -Blob $blobName -context
$destContext
```

3. Paste the script into the Azure PowerShell console to run it to copy the file.

For example PowerShell scripts created to work with HDInsight, see [HDInsight tools](#).

AzCopy

AzCopy is a command-line tool that is designed to simplify the task of transferring data into and out of an Azure Storage account. You can use it as a standalone tool or incorporate this tool in an existing application. [Download AzCopy](#).

The AzCopy syntax is:

```
AzCopy <Source> <Destination> [filePattern [filePattern...]] [Options]
```

For more information, see [AzCopy - Uploading/Downloading files for Azure Blobs](#).

Hadoop command line

The Hadoop command line is only useful for storing data into blob storage when the data is already present on the cluster head node.

In order to use the Hadoop command, you must first connect to the headnode using one of the following methods:

- **Windows-based HDInsight:** [Connect using Remote Desktop](#)
- **Linux-based HDInsight:** Connect using SSH ([the SSH command](#) or PuTTY)

Once connected, you can use the following syntax to upload a file to storage.

```
hadoop -copyFromLocal <localFilePath> <storageFilePath>
```

For example, `hadoop fs -copyFromLocal data.txt /example/data/data.txt`

Because the default file system for HDInsight is in Azure Blob storage, /example/data/data.txt is actually in Azure Blob storage. You can also refer to the file as:

```
wasb:///example/data/data.txt
```

or

```
wasb://<ContainerName>@<StorageAccountName>.blob.core.windows.net/example/data/davinci.txt
```

For a list of other Hadoop commands that work with files, see <http://hadoop.apache.org/docs/r2.7.0/hadoop-project-dist/hadoop-common/FileSystemShell.html>

WARNING

On HBase clusters, the default block size used when writing data is 256KB. While this works fine when using HBase APIs or REST APIs, using the `hadoop` or `hdfs dfs` commands to write data larger than ~12GB results in an error. See the [storage exception for write on blob](#) section below for more information.

Graphical clients

There are also several applications that provide a graphical interface for working with Azure Storage. The following is a list of a few of these applications:

CLIENT	LINUX	OS X	WINDOWS
Microsoft Visual Studio Tools for HDInsight	✓	✓	✓
Azure Storage Explorer	✓	✓	✓
Cloud Storage Studio 2			✓
CloudXplorer			✓
Azure Explorer			✓
Cyberduck		✓	✓

Visual Studio Tools for HDInsight

For more information, see [Navigate the linked resources](#).

Azure Storage Explorer

Azure Storage Explorer is a useful tool for inspecting and altering the data in blobs. It is a free, open source tool that can be downloaded from <http://storageexplorer.com/>. The source code is available from this link as well.

Before using the tool, you must know your Azure storage account name and account key. For instructions about getting this information, see the "How to: View, copy and regenerate storage access keys" section of [Create, manage, or delete a storage account](#).

1. Run Azure Storage Explorer. If this is the first time you have run the Storage Explorer, you will be prompted for the **Storage account name** and **Storage account key**. If you have run it before, use the **Add** button to add a new storage account name and key.

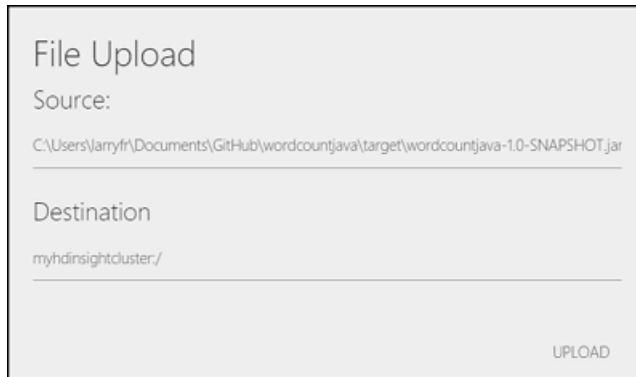
Enter the name and key for the storage account used by your HDInsight cluster and then select **SAVE & OPEN**.



2. In the list of containers to the left of the interface, click the name of the container that is associated with your HDInsight cluster. By default, this is the name of the HDInsight cluster, but may be different if you entered a specific name when creating the cluster.
3. From the tool bar, select the upload icon.



4. Specify a file to upload, and then click **Open**. When prompted, select **Upload** to upload the file to the root of the storage container. If you want to upload the file to a specific path, enter the path in the **Destination** field and then select **Upload**.



Once the file has finished uploading, you can use it from jobs on the HDInsight cluster.

Mount Azure Blob Storage as Local Drive

See [Mount Azure Blob Storage as Local Drive](#).

Services

Azure Data Factory

The Azure Data Factory service is a fully managed service for composing data storage, data processing, and data movement services into streamlined, scalable, and reliable data production pipelines.

Azure Data Factory can be used to move data into Azure Blob storage, or to create data pipelines that directly use HDInsight features such as Hive and Pig.

For more information, see the [Azure Data Factory documentation](#).

Apache Sqoop

Sqoop is a tool designed to transfer data between Hadoop and relational databases. You can use it to import data from a relational database management system (RDBMS), such as SQL Server, MySQL, or Oracle into the Hadoop distributed file system (HDFS), transform the data in Hadoop with MapReduce or Hive, and then export the data back into an RDBMS.

For more information, see [Use Sqoop with HDInsight](#).

Development SDKs

Azure Blob storage can also be accessed using an Azure SDK from the following programming languages:

- .NET
- Java
- Node.js
- PHP
- Python
- Ruby

For more information on installing the Azure SDKs, see [Azure downloads](#)

Troubleshooting

Storage exception for write on blob

Symptoms: When using the `hadoop` or `hdfs dfs` commands to write files that are ~12GB or larger on an HBase cluster, you may encounter the following error:

```
ERROR azure.NativeAzureFileSystem: Encountered Storage Exception for write on Blob :  
example/test_large_file.bin._COPYING_ Exception details: null Error Code : RequestBodyTooLarge  
copyFromLocal: java.io.IOException  
    at com.microsoft.azure.storage.core.Utility.initIOException(Utility.java:661)  
    at com.microsoft.azure.storage.blob.BlobOutputStream$1.call(BlobOutputStream.java:366)  
    at com.microsoft.azure.storage.blob.BlobOutputStream$1.call(BlobOutputStream.java:350)  
    at java.util.concurrent.FutureTask.run(FutureTask.java:262)  
    at java.util.concurrent.Executors$RunnableAdapter.call(Executors.java:471)  
    at java.util.concurrent.FutureTask.run(FutureTask.java:262)  
    at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1145)  
    at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:615)  
    at java.lang.Thread.run(Thread.java:745)  
Caused by: com.microsoft.azure.storage.StorageException: The request body is too large and exceeds the  
maximum permissible limit.  
    at com.microsoft.azure.storage.StorageException.translateException(StorageException.java:89)  
    at com.microsoft.azure.storage.core.StorageRequest.materializeException(StorageRequest.java:307)  
    at com.microsoft.azure.storage.core.ExecutionEngine.executeWithRetry(ExecutionEngine.java:182)  
    at com.microsoft.azure.storage.blob.CloudBlockBlob.uploadBlockInternal(CloudBlockBlob.java:816)  
    at com.microsoft.azure.storage.blob.CloudBlockBlob.uploadBlock(CloudBlockBlob.java:788)  
    at com.microsoft.azure.storage.blob.BlobOutputStream$1.call(BlobOutputStream.java:354)  
    ... 7 more
```

Cause: HBase on HDInsight clusters default to a block size of 256KB when writing to Azure storage. While this works for HBase APIs or REST APIs, it will result in an error when using the `hadoop` or `hdfs dfs` command-line utilities.

Resolution: Use `fs.azure.write.request.size` to specify a larger block size. You can do this on a per-use basis by using the `-D` parameter. The following is an example using this parameter with the `hadoop` command:

```
hadoop -fs -D fs.azure.write.request.size=4194304 -copyFromLocal test_large_file.bin /example/data
```

You can also increase the value of `fs.azure.write.request.size` globally by using Ambari. The following steps can be used to change the value in the Ambari Web UI:

1. In your browser, go to the Ambari Web UI for your cluster. This is <https://CLUSTERNAME.azurehdinsight.net>, where **CLUSTERNAME** is the name of your cluster.
When prompted, enter the admin name and password for the cluster.
2. From the left side of the screen, select **HDFS**, and then select the **Configs** tab.
3. In the **Filter...** field, enter `fs.azure.write.request.size`. This will display the field and current value in the middle of the page.
4. Change the value from 262144 (256KB) to the new value. For example, 4194304 (4MB).

The screenshot shows the Ambari Web UI interface for managing an HDInsight cluster. The left sidebar lists various services: HDFS, MapReduce2, YARN, Tez, Hive, HBase, Pig, Sqoop, Oozie, ZooKeeper, and Ambari Metrics. The main area is focused on the 'Configs' tab, which is highlighted with a red border. At the top right of the main area, there is a search bar containing the text 'fs.azure.write' with a red border around it. Below the search bar, there is a 'Manage Config Groups' section with two items: 'hdinsightwatchdog' (V2) and 'hdinsightwatchdog' (V1). Both items were updated 10 minutes ago and are associated with HDP-2.4. The 'hdinsightwatchdog' (V2) item is currently selected, indicated by a green checkmark. Below this, there are tabs for 'Settings' and 'Advanced', with 'Settings' being the active tab. Under the 'Custom core-site' section, there is a property named 'fs.azure.write.request.size' with a value of '262144'. At the bottom right of the configuration panel, there are 'Discard' and 'Save' buttons.

For more information on using Ambari, see [Manage HDInsight clusters using the Ambari Web UI](#).

Next steps

Now that you understand how to get data into HDInsight, read the following articles to learn how to perform analysis:

- [Get started with Azure HDInsight](#)
- [Submit Hadoop jobs programmatically](#)
- [Use Hive with HDInsight](#)
- [Use Pig with HDInsight](#)

HDInsight Architecture

8/16/2017 • 13 min to read • [Edit Online](#)

An HDInsight cluster consists of several Linux Azure Virtual Machines (nodes) that are used for distributed processing of tasks. Azure HDInsight handles implementation details of installation and configuration of individual nodes, so you only have to provide general configuration information. An HDInsight cluster is deployed by first selecting a cluster type, which determines what components are installed and the specific topology of virtual machines that is deployed.

This article describes all of the available cluster types, their constituent nodes, the services the nodes run, as well the network and data storage architectures.

Cluster types

Azure HDInsight currently provides the following cluster types, each with a set of components that provide certain functionalities.

IMPORTANT

HDInsight clusters are available in various types, each for a single workload or technology. There is no supported method to create a cluster that combines multiple types, such as Storm and HBase on one cluster. If your solution requires technologies that are spread across multiple HDInsight cluster types, an [Azure virtual network](#) can connect the required cluster types.

CLUSTER TYPE	FUNCTIONALITY
Hadoop	Uses HDFS, YARN resource management, and a simple MapReduce programming model to process and analyze data in parallel. Clusters of this type can also be domain-joined , providing enterprise-grade access control for Hive workloads.
HBase	A NoSQL database built on Hadoop that provides random access and strong consistency for large amounts of unstructured and semi-structured data - potentially billions of rows times millions of columns. See What is HBase on HDInsight?
Storm	A distributed, real-time computation system for processing large streams of data quickly. See Analyze real-time sensor data using Storm and Hadoop .
Spark	A parallel processing framework that supports in-memory processing to boost the performance of big-data analytics applications. Spark supports both imperative and SQL based querying, processing of streaming data, and performing machine learning at scale. See What is Apache Spark in HDInsight?

CLUSTER TYPE	FUNCTIONALITY
Kafka (Preview)	A distributed streaming platform that can be used to build real-time streaming data pipelines and applications. Kafka also provides message-queue functionality that allows you to publish and subscribe to data streams. See Introduction to Apache Kafka on HDInsight .
R Server	A server for hosting and managing parallel, distributed R processes for performing machine learning at scale, leveraging the capabilities of the HDInsight cluster. See Overview of R Server on HDInsight .
Interactive Hive (Preview)	In-memory caching for interactive and faster Hive queries, leveraging Live Long and Process (LLAP) technology. See Use Interactive Hive in HDInsight .

Node types

Each cluster type may contain different types of nodes that have a specific purpose in the cluster. The following table summarizes these node types.

TYPE	DESCRIPTION
Head node	For the Hadoop, Interactive Hive, Kafka, Spark, HBase and R Server cluster types, the head nodes hosts the processes that manage execution of the distributed application. In addition, for the Hadoop, Interactive Hive, Kafka, Spark, and HBase cluster types the head node represents the node you can SSH into and execute applications that are then coordinated to run across the cluster resources. The number of head nodes is fixed at two for all cluster types.
Nimbus node	For the Storm cluster type, the Nimbus node provides functionality similar to the Head node. The Nimbus node assigns tasks to other nodes in a cluster through Zookeeper-it coordinates the running of Storm topologies.
ZooKeeper node	Represents the nodes hosting the ZooKeeper process and data, which is used to coordinate tasks between the nodes performing the processing, leader election of the head node, and for keeping track of on which head node a master service is active on. The number of ZooKeeper nodes is fixed at two for all cluster types having ZooKeeper nodes.
R Server Edge node	The R Server Edge node represents the node you can SSH into and execute applications that are then coordinated to run across the cluster resources. An edge node itself does not actively participate in data analysis within the cluster. In addition, this node hosts R Studio Server, enabling you to run R application using a browser.
Worker node	Represents the nodes that support data processing functionality. Worker nodes can be added or removed from the cluster to increase or decrease computing capability and to manage costs.

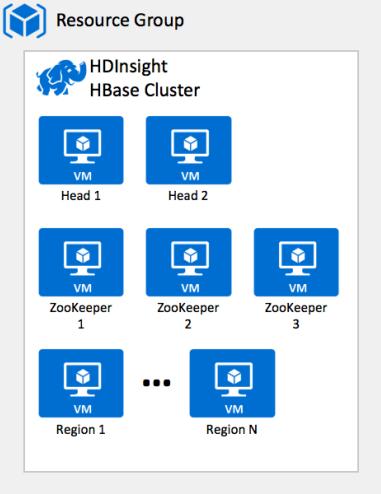
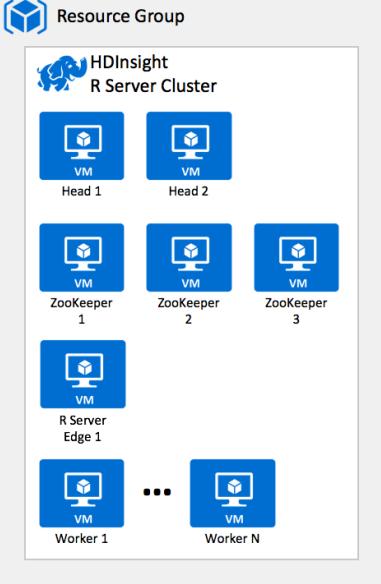
TYPE	DESCRIPTION
Region node	For the HBase cluster type, the region node (also referred to as a Data Node) runs the Region Server that is responsible for serving and managing a portion of the data managed by HBase. Region nodes can be added or removed from the cluster to increase or decrease computing capability and to manage costs.
Supervisor node	For the Storm cluster type, the supervisor node executes the instructions provided by the Nimbus node to performing the desired processing.

Nodes in an HDInsight cluster

Each cluster type has its own number of nodes, terminology for nodes, and default VM size. In the following table, the number of nodes for each node type is in parentheses. The scale-out capabilities (e.g., adding VM nodes to increase processing capabilities) for each cluster type are indicated in the node count as (1+) and in the diagram as Worker 1...Worker N.

TYPE	NODES	DIAGRAM
Hadoop	Head nodes (2), Worker nodes (1+)	<p>The diagram illustrates an HDInsight Hadoop Cluster. It starts with a 'Resource Group' icon at the top. Below it is a box labeled 'HDInsight Hadoop Cluster'. Inside this box, there are two icons labeled 'VM' with the labels 'Head 1' and 'Head 2' underneath. To the right of these, there is a horizontal ellipsis '...', followed by two more 'VM' icons labeled 'Worker 1' and 'Worker N' respectively.</p>
Interactive Hive	Head nodes (2), ZooKeeper nodes (3), Worker nodes (1+)	<p>The diagram illustrates an HDInsight Interactive Hive Cluster. It starts with a 'Resource Group' icon at the top. Below it is a box labeled 'HDInsight Interactive Hive Cluster'. Inside this box, there are two icons labeled 'VM' with the labels 'Head 1' and 'Head 2' underneath. Below them are three icons labeled 'VM' with the labels 'ZooKeeper 1', 'ZooKeeper 2', and 'ZooKeeper 3' underneath. To the right of these, there is a horizontal ellipsis '...', followed by two more 'VM' icons labeled 'Worker 1' and 'Worker N' respectively.</p>

TYPE	NODES	DIAGRAM
Kafka	Head nodes (2), ZooKeeper nodes (3), Worker nodes (1+)	<p>The diagram illustrates an HDInsight Kafka Cluster. It is contained within a light gray box labeled "Resource Group". Inside, there is a "HDInsight Kafka Cluster" section featuring a blue elephant icon. The cluster consists of two "Head" nodes (labeled "Head 1" and "Head 2"), three "ZooKeeper" nodes (labeled "ZooKeeper 1", "ZooKeeper 2", and "ZooKeeper 3"), and a series of "Worker" nodes represented by a blue box with three dots, followed by "Worker N".</p>
Spark	Head nodes (2), Worker nodes (1+)	<p>The diagram illustrates an HDInsight Spark Cluster. It is contained within a light gray box labeled "Resource Group". Inside, there is a "HDInsight Spark Cluster" section featuring a blue elephant icon. The cluster consists of two "Head" nodes (labeled "Head 1" and "Head 2") and a series of "Worker" nodes represented by a blue box with three dots, followed by "Worker N".</p>
Storm	Nimbus node (2), ZooKeeper nodes (3), Supervisor nodes (1+)	<p>The diagram illustrates an HDInsight Storm Cluster. It is contained within a light gray box labeled "Resource Group". Inside, there is a "HDInsight Storm Cluster" section featuring a blue elephant icon. The cluster consists of two "Nimbus" nodes (labeled "Nimbus 1" and "Nimbus 2"), three "ZooKeeper" nodes (labeled "ZooKeeper 1", "ZooKeeper 2", and "ZooKeeper 3"), and a series of "Supervisor" nodes represented by a blue box with three dots, followed by "Supervisor N".</p>

TYPE	NODES	DIAGRAM
HBase	Head server (2), ZooKeeper node (3), Region server (1+)	
Microsoft R Server	Head nodes (2), ZooKeeper nodes (3), R Server Edge node (1), Worker nodes (1+)	

ZooKeeper

[Apache ZooKeeper](#) provides services that are core to the operation of a cluster: a distributed configuration service, a synchronization service and a naming registry for distributed services. Many of the HDInsight cluster types rely on ZooKeeper for the coordination of processes in the cluster, the sharing of configuration and for coordinating the election of the primary head node.

ZooKeeper coordinates processes in large distributed systems using a shared hierarchical namespace of data registers known as znodes. The hierarchical structure is similar to a filesystem consisting of folders and files. Correspondingly, each znode is identified by a path, where the root is "/" and path elements are further separated by a slash (such as /myapplication/settings). Znodes contain small amounts of metadata needed to coordinate processes: status, location, configuration, and so on.

Clients, typically processes running in the Head and Worker nodes of the HDInsight cluster, establish bi-directional TCP connections with ZooKeeper and get high-throughput, high availability and low latency access to the data it manages. ZooKeeper is designed to be fast for read heavy workloads, such as might be expected when nodes in the cluster are retrieving settings or are looking up the name of a service.

ZooKeeper replicates its data in HDInsight over three nodes to ensure there is no single point of failure. The data managed by ZooKeeper is maintained in an in-memory database where updates are logged to disk for

recoverability, and writes are serialized to disk before they are applied. Clients can connect to any of the ZooKeeper nodes and are guaranteed to see the same view of the data regardless of the node to which they connect.

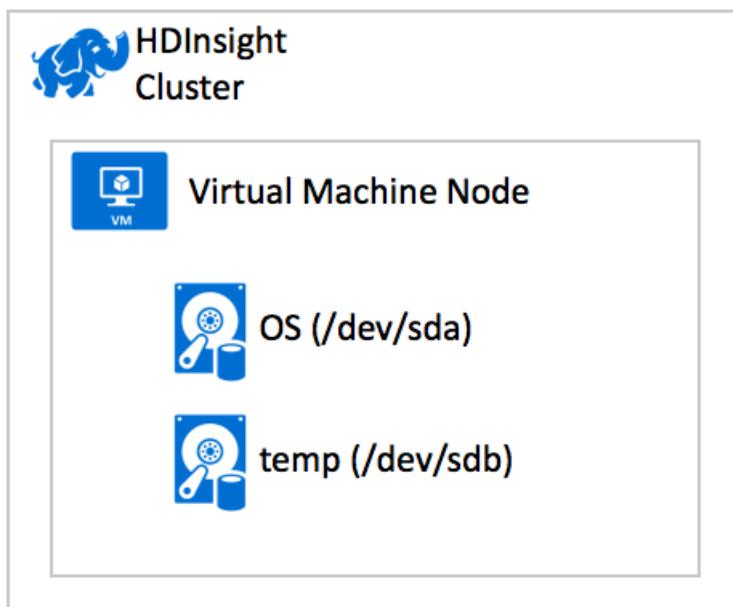
Node Virtual Machines

The Azure Virtual Machines that support each node of an HDInsight cluster are managed by the HDInsight service. This means they are not typically accessed individually (the common exceptions being the Head and Edge nodes that you can access via SSH), and that when enumerating the resources used in your Azure subscription you will see only one resource for the HDInsight cluster, not one resource for each virtual machine within that cluster.

Each virtual machine runs linux with the applicable components from the selected cluster type installed. The amount of RAM and quantity of CPU's are determined by the selected virtual machine size.

A Linux Virtual Machine in Azure, including those used in an HDInsight cluster, has two disks associated with it. The disk at /dev/sda is your OS disk, /dev/sdb is your temporary disk. The OS disk (/dev/sda) contains the operating system and is optimized for fast VM boot time and does not provide good performance for your workloads. While the OS disk is backed by durable storage, as node VM's are managed by HDInsight, the VHD file backing the OS disk is not accessible to you (nor is the Storage Account which contains it).

The temporary disk (/dev/sdb) is not durable, and can be lost if specific events like VM resizing, redeployment, or maintenance forces a restart of your VM. The temporary drive itself is backed by local HDD disk-based storage or SSD based storage depending on the VM type selected. However, as this temporary storage drive is present on the physical machine which is hosting your VM, it can have higher IOPs and lower latency when compared to the persistent storage like a standard data disk. The size of your temporary disk is related to the VM size for the node type you chose at cluster deployment time.



In addition to an OS disk and a temp disk, Azure Virtual Machines support data disks. A data disk is a VHD that's attached to a virtual machine to store application data, or other data you need to keep. In general, the VMs in an HDInsight cluster do not support the attachment of data disks, although there are some exceptions as shown in the table.

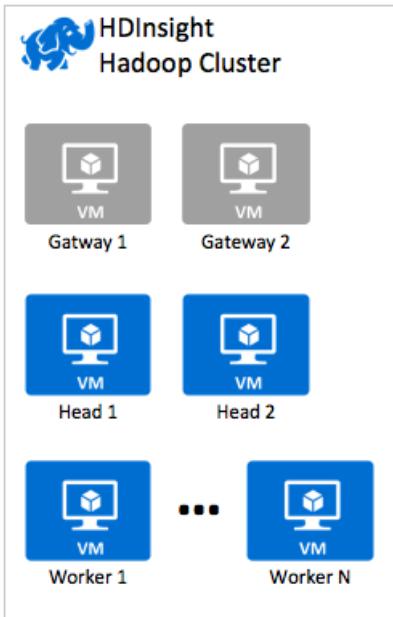
CLUSTER TYPE	DISK CONFIGURATION SUPPORTED
Kafka	Kafka deployments require additional managed disks. Standard or Premium managed disks are supported.
All other cluster types	Only OS and temporary disks. No additional disks supported.

NOTE

For more information on the nodes the virtual machine sizes available to them, see [Default node configuration and virtual machine sizes for clusters](#) in "What are the Hadoop components and versions in HDInsight?"

Gateway Nodes

In addition to the node type previously described, in every HDInsight cluster there are another two nodes that are invisibly supporting communication with the cluster. These are the Gateway nodes. The Gateway nodes provide load balanced HTTPS access to the cluster (such as when accessing Ambari), provide SSL support, handles cluster credential validation, and act as a reverse proxy to communicate with a subset of Hadoop services running on the cluster.



If you are accessing your cluster via a URL of the form:

```
https://<clustername>.azurehdinsight.net
```

Then you are communicating with the cluster through one of the Gateway nodes.

It is important to note that the Gateway nodes do not provide support for SSH access to the cluster, as that is accomplished thru direct access to the head nodes.

Data Storage Architecture

Given that you should not use the local virtual machine storage for your data, where should you store the data managed by your HDInsight cluster?

HDFS in the HDInsight cluster

The Hadoop Distributed File System (HDFS) is a file system that, along with YARN and MapReduce, is the core of Hadoop technology. Hadoop stores data using the HDFS, which spreads the data across multiple servers, and then runs chunks of the processing job on each server, letting big data sets be processed in parallel.

Although an on-premises installation of Hadoop uses the HDFS for storage on the cluster, in Azure you should use external storage services, and not the local disk storage provided by the virtual machines in the cluster.

HDInsight follows a strong separation of compute and storage-- as such the recommendation is to store your data either in Azure Storage blobs and Azure Data Lake Store, or a combination of the two. Both provide an HDFS

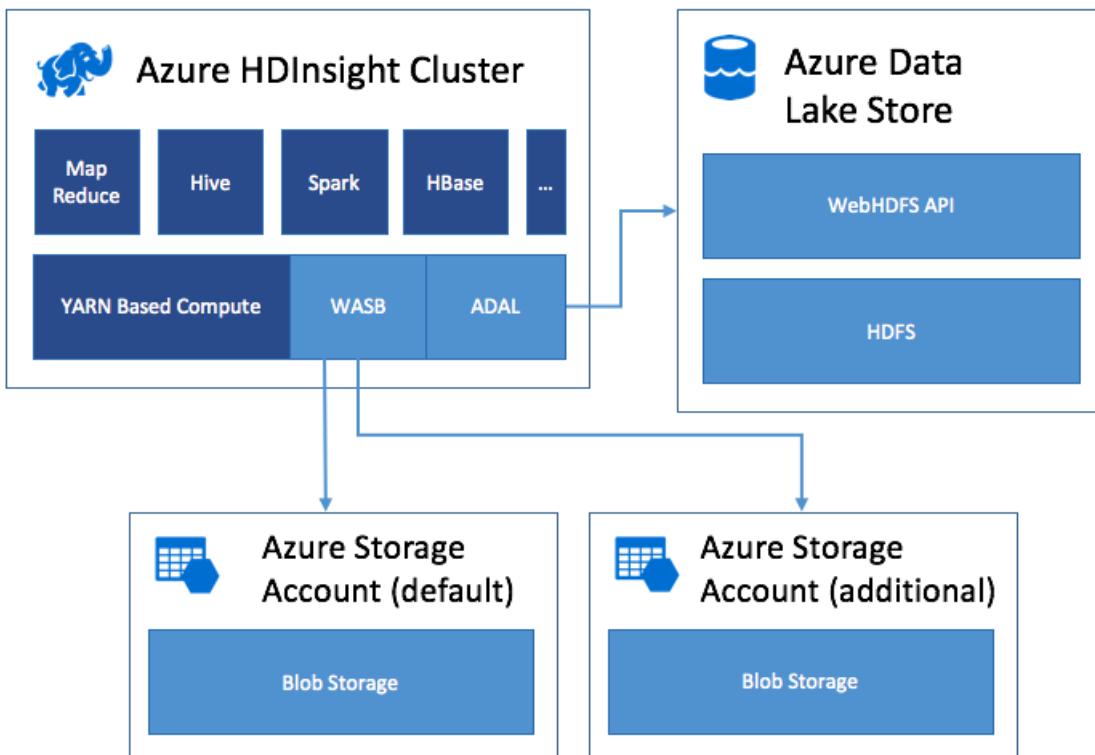
compatible file system that persists data even if the cluster is deleted.

The benefit of this approach is:

- The data is persistent, even after you delete your HDInsight cluster. This means it will also be available without any data transfer effort should you deploy a new cluster to perform additional processing.
- The costs for storing your data are predominantly driven by the volume of data stored and transferred, which can be significantly less than the costs for running a cluster.
- The data is available for multiple clusters to act upon.

Hadoop supports a notion of the default file system. The default file system implies a default scheme and authority. It can also be used to resolve relative paths. During the HDInsight cluster creation process, you can specify a blob container in Azure Storage as the default file system, or with HDInsight 3.5 or later, you can select either Azure Storage or Azure Data Lake Store as the default file system.

In addition to this default file system, you can add additional Azure Storage Accounts or Data Lake Store instances during the cluster creation process or after a cluster has been created.



For instructions about adding additional storage accounts, see [HDInsight using Azure Storage](#), and for details on using Data Lake Store see [HDInsight using Data Lake Store](#).

Metadata Storage

HDInsight supports the use of a custom metastore for Hive and Oozie. The Hive metastore persists the metadata which describes the mappings of Hive tables to their locations in HDFS and the schemas of those tables. The metastore is also used by Spark as it stores metadata for Hive tables created with Spark SQL.

By default, the metastore data is stored internally by the HDInsight cluster. However, you can configure the cluster to use an external Azure SQL Database instance instead by configuring a custom metastore. Custom metastores can only be configured during the cluster creation process.

Oozie uses a metastore to store details about current and completed workflows. To increase performance when using Oozie, you can use SQL Database as a custom metastore. The metastore can also provide access to Oozie job data after you delete your cluster.

By combining use of SQL Database as your metastore with Azure Storage or Data Lake Store for persisting your

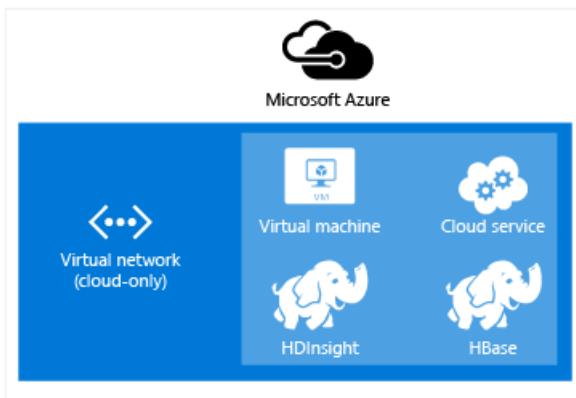
data, you ensure that you can re-create your cluster without first having to reload either the data or the metadata.

Network Architecture

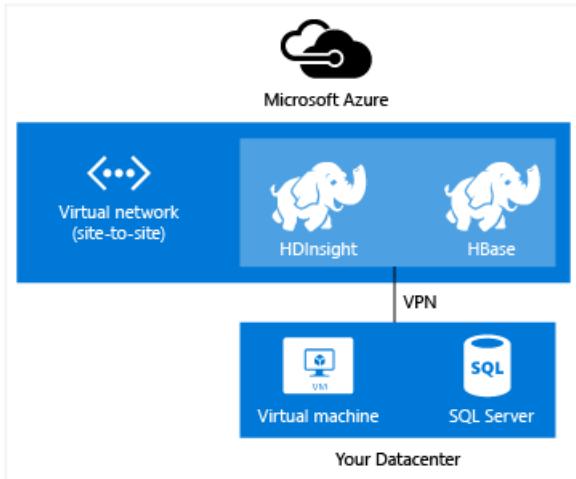
By default when you create an HDInsight cluster, the VM nodes within that cluster are configured to communicate with each other, but Internet access to any of the cluster nodes is restricted to just the Head or Edge nodes (and limited to SSH or HTTPS). There are scenarios that need a greater degree of access into the networking environment of the cluster nodes, for example:

- You need to directly access services on HDInsight that aren't exposed over the Internet. For example, you have consumers or producers that need to directly work with Kafka brokers or clients that need to use the HBase Java API.
- You need to connect on-premises services to HDInsight. For example, use Oozie to import or export data to from an on-premises SQL Server.
- You need to create solutions that involve multiple HDInsight clusters of different types. For example, you might want to use Spark or Storm clusters to analyze data stored in a Kafka cluster.
- You want to restrict access to HDInsight. For example, to prevent inbound traffic from the internet.

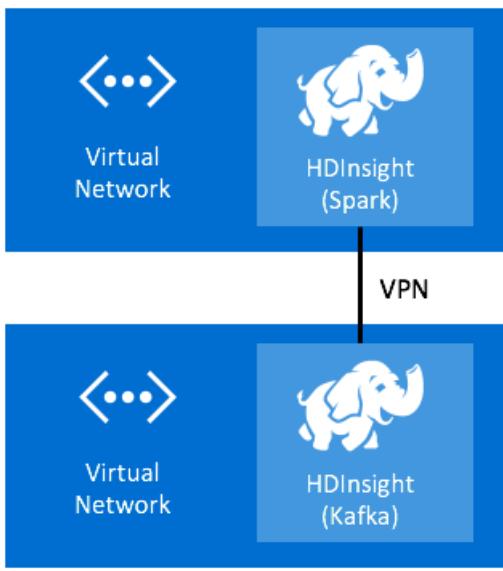
Greater control over the HDInsight networking environment is achieved by deploying your cluster into an Azure Virtual Network. An [Azure Virtual Network](#) allows you to create a secure, persistent network containing the resources you need for your solution. Cloud resources that you want to connect with your HDInsight cluster, such as Virtual Machines and other instances of HDInsight can then be provisioned into the same Virtual Network.



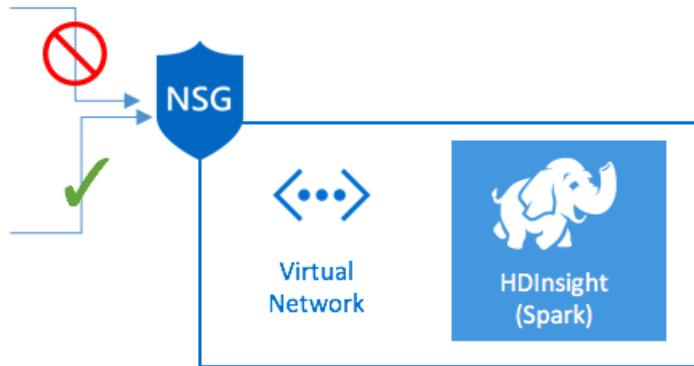
You can create a [site-to-site](#) or [point-to-site](#) VPN connection to enable connectivity between resources in an on-premises network and your HDInsight cluster.



You can also connect two different Virtual Network instances by configuring a [VNET-to-VNET connection](#).



You can also secure the network perimeter by using Network Security Groups to restrict traffic based on protocol, source and destination.



In addition to securing in-bound traffic by applying NSGs to subnet of the Virtual Network, you can also configure user-defined routes and control the flow of network traffic through a virtual firewall appliance by deploying your HDInsight cluster into a Virtual Network.

For more details on using HDInsight within a Virtual Network, see [Use Virtual Network](#)

Next steps

- [Hadoop Components on HDInsight](#): Learn about the Hadoop ecosystem components and versions in Azure HDInsight, as well as the Standard and Premium service levels.

Architecture of Hadoop on HDInsight

8/16/2017 • 6 min to read • [Edit Online](#)

Hadoop includes two core components, the High Density File System (HDFS) which provides storage and Yet Another Resource Negotiator (YARN) which provides processing. With storage and processing capabilities a cluster becomes capable of running MapReduce programs that perform the desired data processing.

IMPORTANT

As described in [HDInsight Architecture](#), HDFS is not typically deployed within the HDInsight cluster to provide storage. Instead, an HDFS compatible interface layer is exposed to Hadoop ecosystem components and the actual storage capability is provided by either Azure Storage or Azure Data Lake Store. In the Hadoop case, MapReduce jobs executing on the HDInsight cluster run as if HDFS were actually present and require no changes to support their storage needs. This simplifies the discussion of the architecture of Hadoop on HDInsight, as storage is outsourced, but the processing which uses YARN remains a core component.

This article introduces YARN and how it coordinates the execution of applications on HDInsight, and then shows how Spark utilizes YARN to run Spark jobs.

YARN basics

YARN is what governs and orchestrates data processing in Hadoop. It is structured as having two core services that run as processes on nodes in the cluster:

- ResourceManager
- NodeManager

The ResourceManager grants cluster compute resources to applications like MapReduce jobs. It grants these resources in the form of containers, which are themselves a way to describe an allocation of CPU cores and RAM memory. If you combined all the resources available in the cluster and then distributed it in blocks of a predefined number of cores and memory, each block of resources is a container. Each node in the cluster has a capacity for a certain number of containers and therefore the cluster has a fixed limit on the number of containers available. The allotment of resources in a container is configurable.

When a MapReduce application needs to run on a cluster, it is the ResourceManager that provides it the containers in which to execute. The ResourceManager tracks the status of running applications, available cluster capacity and tracks applications as they complete and release the resources they utilized.

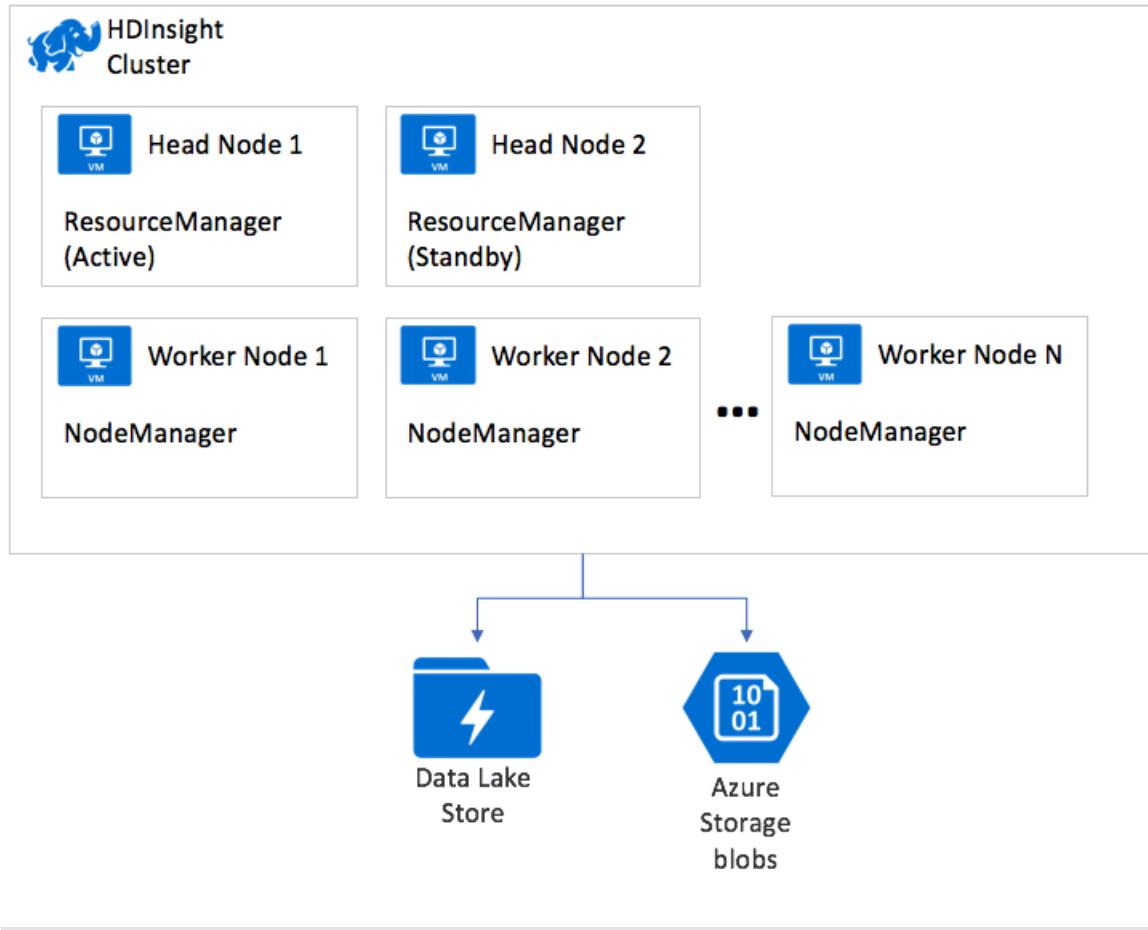
The ResourceManager also runs a web server process that provides a web user interface you can access to monitor the status of applications.

When a user submits a MapReduce application to run on the cluster, it is submitted to the ResourceManager. In turn, the ResourceManager allocates a container on an available NodeManager nodes. The NodeManager nodes are where the application actually executes. In the first container allocated is run a special application called the ApplicationMaster. This ApplicationMaster is responsible for acquiring resources, in the form of subsequent containers, needed to run the submitted application. To do this, the ApplicationMaster examines the stages of the application (e.g., the map stage and reduce stage), factors in how much data needs to be processed and then requests the resources from the ResourceManager on behalf of the application (in a process called "negotiating"). The ResourceManager in turn grants resources from the NodeManagers in the cluster to the ApplicationMaster for it to use in executing the application.

These NodeManagers run the tasks that make up the application and report their progress and status back to ApplicationMaster. The ApplicationMaster, in turn reports the status of the application back to the ResourceManager. The ResourceManager, in turn, returns any results to the client.

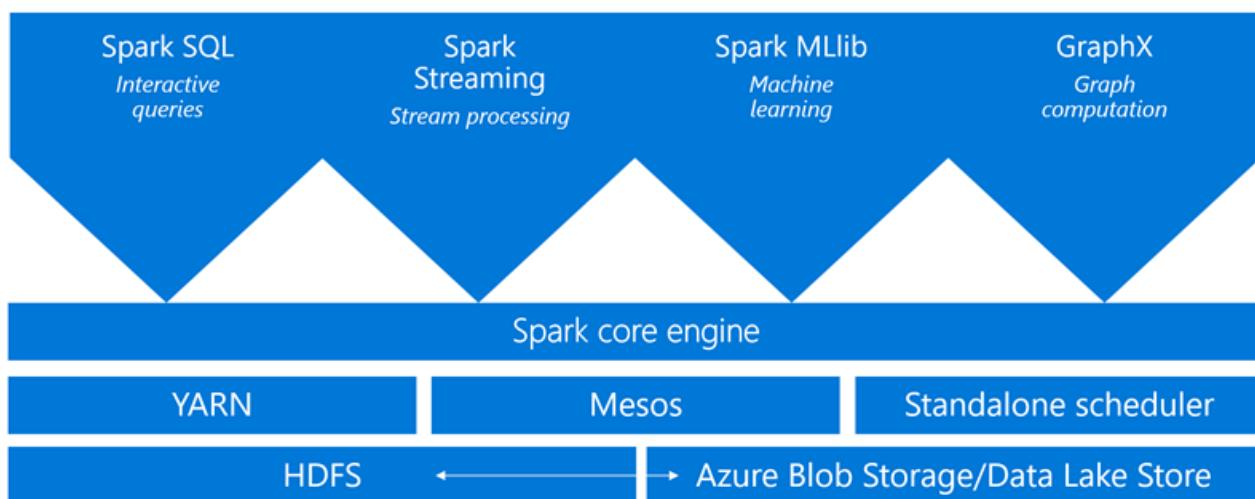
YARN on HDInsight

All HDInsight cluster types deploy YARN. The ResourceManager is deployed in a high-availability fashion having a primary and secondary instance, which run on the first and second head nodes within the cluster respectively. Only the one instance of the ResourceManager is active at a time. The NodeManager instances run across the available Worker Nodes in the cluster.

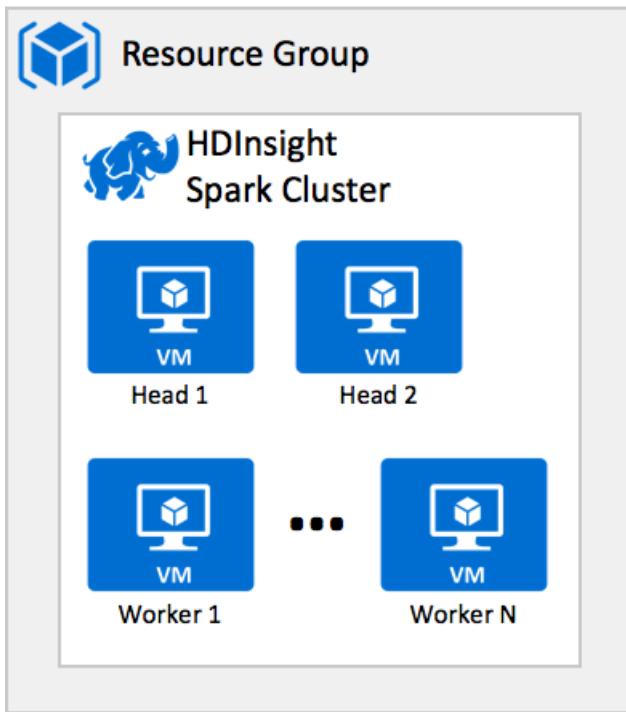


Architecture of Spark on HDInsight

Spark in HDInsight relies on YARN, as described previously, for resource management. The following sections examine in more detail how Spark executes within HDInsight using YARN.

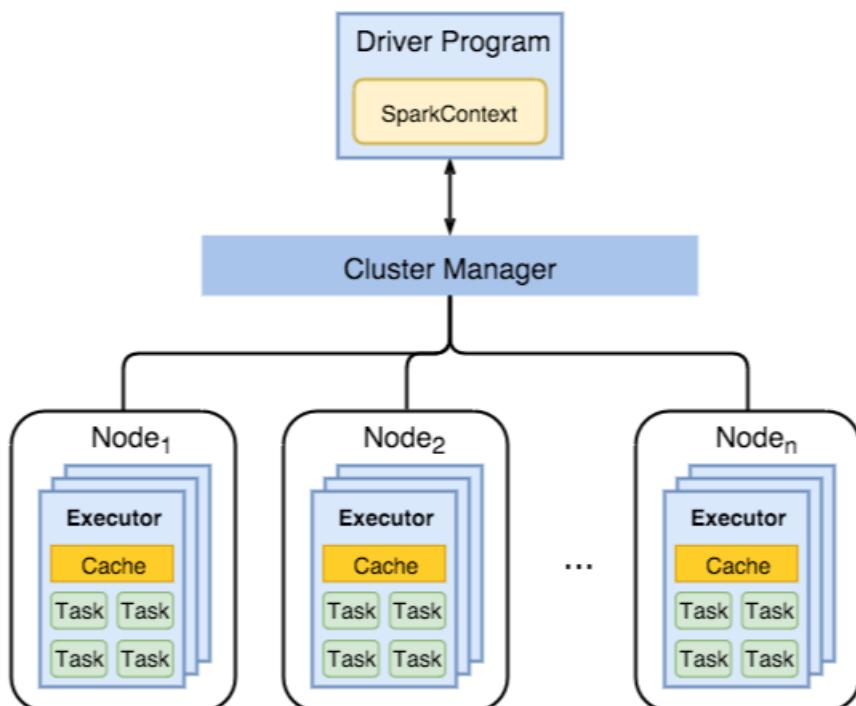


When you setup your HDInsight cluster on Azure, you select a cluster of type **Spark** and then the version of Spark as well as the number and configuration of head and worker nodes for your cluster.



In order to understand the lifecycle of a Spark job, you'll also need to consider Spark objects in your cluster. As shown below, Spark uses a driver process, which runs the `SparkContext` along with the YARN resource manager to schedule and run the Spark jobs submitted to that cluster. When a Spark job is submitted to the cluster, the YARN ResourceManager instantiates an ApplicationMaster that will act as the Spark master process. The Spark driver provides its resource requirements to the ApplicationMaster. The ApplicationMaster subsequently requests YARN containers from the ResourceManager to host the Spark executors. The ApplicationMaster will then be responsible for the YARN containers running the Spark executors for duration of the application. Beyond that, it is the Spark driver that is responsible for coordinating the actual Spark application processing.

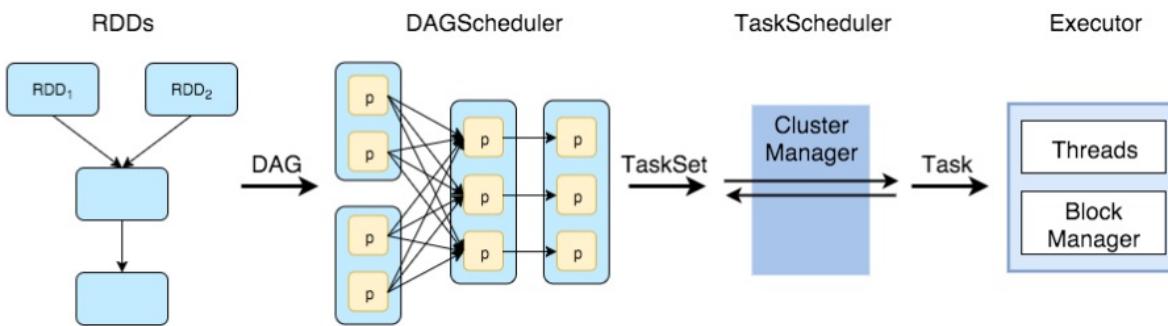
The cluster executes the Spark job steps on the worker nodes. Each worker node has its own Executor, Cache and list of (job) tasks.



Understanding Spark Job Steps

Spark uses an abstraction called a RDD (resilient distributed dataset) to hold the data that it processes during a Spark job. The Spark API has evolved and there are now higher level objects, such as DataFrames and DataSets that operate on top of RDDs and provide more functionality for developers, i.e. strongly-typed objects, etc...

After data is loaded into RDDs on the worker nodes, then the DAG (Directed Acyclic Graph) scheduler coordinates the set of tasks that the Spark job requires and sends that list to the Task Scheduler on the Cluster Manager. Tasks are then distributed to Executors (on various nodes) and run on resources on those nodes. This process is illustrated below.



You can monitor the progress of Spark Jobs via several monitoring UIs that are available for HDInsight. Most commonly, you'd first start by taking a look at the YARN UI to locate the job status and tracking URL for the Spark job(s) of interest. This is shown below.

Screenshot of the Hadoop YARN Application Monitoring UI for application `application_1498242895398_0006`. The UI shows the following details:

Kill Application		Application Overview
User:	ivy	
Name:	remotesparkmagics	
Application Type:	SPARK	
Application Tags:	ivy-session-2-9h4j1hj	
Application Priority:	0 (Higher Integer value indicates higher priority)	
YarnApplicationState:	RUNNING: AM has registered with RM and started running.	
Queue:	default	
FinalStatus Reported by AM:	Application has not completed yet.	
Started:	Fri Jun 23 20:13:03 +0000 2017	
Elapsed:	1hrs, 2mins, 52sec	
Tracking URL:	ApplicationMaster	
Log Aggregation Status:	NOT_START	
Diagnostics:		
Unmanaged Application:	false	
Application Node Label expression:	<Not set>	
AM container Node Label expression:	<DEFAULT_PARTITION>	

Clicking on the **Tracking URL** (shown above), opens the Spark UI. There are a number of views here that allow you to track and monitor the status of your job(s) at a very granular level. This UI opens to the **Jobs** tab. As shown below, here you can see a list of jobs run with Job Ids, Descriptions, Time Submitted, Job Duration, Job Steps and Job Tasks. Here also you can click on a link in the Description column to open a new UI with detailed information about Job step execution overhead.

Spark Jobs (?)

User: yarn

Total Uptime: 1.0 h

Scheduling Mode: FIFO

Completed Jobs: 6

▶ Event Timeline

Completed Jobs (6)

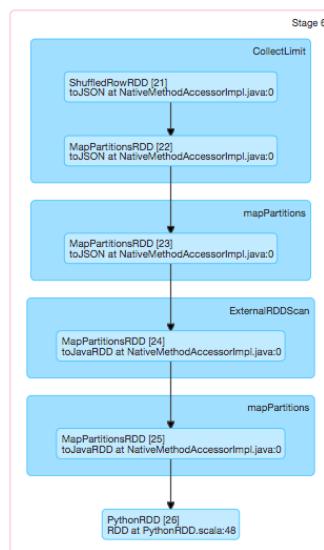
Job Id (Job Group) ▾	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
5 (4)	Job group for statement 4 runJob at PythonRDD.scala:441	2017/06/23 20:13:32	2 s	2/2	2/2
4 (3)	Job group for statement 3 runJob at PythonRDD.scala:441	2017/06/23 20:13:30	0.4 s	1/1	1/1
3 (2)	Job group for statement 2 saveAsTable at NativeMethodAccesso ri.java:0	2017/06/23 20:13:25	2 s	1/1	1/1
2 (2)	Job group for statement 2 csv at NativeMethodAccesso ri.java:0	2017/06/23 20:13:20	2 s	1/1	2/2
1 (2)	Job group for statement 2 csv at NativeMethodAccesso ri.java:0	2017/06/23 20:13:19	1 s	1/1	1/1
0 (2)	Job group for statement 2 csv at NativeMethodAccesso ri.java:0	2017/06/23 20:13:18	1 s	1/1	1/1

In the Spark (Job) Stages UI, you have access to highly detailed information about the process and overhead associated with each task in a Spark job. Shown below is an expanded view of the Spark Stages UI. This includes the [DAG Visualization](#), [Event Timeline](#), [Summary Metrics](#) and [Aggregated Metrics by Executor](#) for a single job stage of a Spark job. These detailed views are quite useful in determining whether and exactly where Spark job performance bottlenecks are occurring on your HDInsight Spark cluster.

Details for Stage 6 (Attempt 0)

Total Time Across All Tasks: 71 ms
 Locality Level Summary: Node local: 1

DAG Visualization

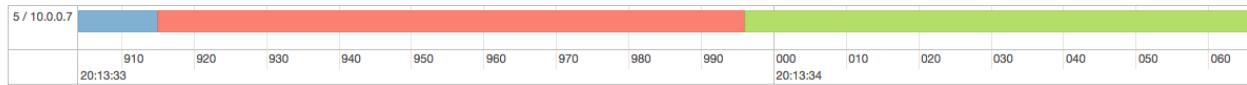


Show Additional Metrics

Event Timeline

Enable zooming

Scheduler Delay Executor Computing Time Getting Result Time
 Task Deserialization Time Shuffle Write Time Result Serialization Time
 Shuffle Read Time



Summary Metrics for 1 Completed Tasks

Metric	Min	25th percentile	Median	75th percentile	Max
Duration	71 ms	71 ms	71 ms	71 ms	71 ms
GC Time	0 ms	0 ms	0 ms	0 ms	0 ms

Aggregated Metrics by Executor

Executor ID ▲	Address	Task Time	Total Tasks	Failed Tasks	Killed Tasks	Succeeded Tasks
5 stdout stderr	10.0.0.7:37315	0.2 s	1	0	0	1

Tasks (1)

Index ▲	ID	Attempt	Status	Locality Level	Executor ID / Host	Launch Time	Duration	GC Time	Errors
0	7	0	SUCCESS	NODE_LOCAL	5 / 10.0.0.7 stdout stderr	2017/06/23 20:13:33	71 ms		

After Spark jobs complete, then job execution information is available in the Spark History Server view. This view is available via a link in the Azure Portal for HDInsight.

Next steps

This article provided an architectural overview of Hadoop on HDInsight.

- To further explore the architecture of HDInsight, see [HDInsight Architecture](#).
- For further details on MapReduce, see [What is MapReduce](#).
- To run a sample MapReduce application on HDInsight, see [Use MapReduce with Hadoop in HDInsight](#).

HDInsight Dynamic Lifecycle

8/16/2017 • 3 min to read • [Edit Online](#)

You can create HDInsight clusters that are either temporary, permanent, or scheduled scale.

Temporary Clusters

Since you only pay for HDInsight when a cluster is up and running, deleting a cluster when it is not in use provides a big opportunity for cost savings. In order to stop the charges, you have to delete the cluster. There is no concept of pausing a cluster.

HDInsight can be primarily used for executing scripts and using compute resources. Storage of all data can be done in less expensive products like Azure SQL Database, Azure Blob Storage, or Azure Data Lake Store.

Most Hadoop jobs are batch jobs. Batch jobs can be used to do a wide variety of things including data aggregation, data cleaning, data organization, or batch analytics that run at night and automatically populate Power BI visualizations or Excel spreadsheets. When implementing a workload like this you would create an HDInsight cluster, run some jobs, save the data, and then completely delete the HDInsight cluster. The process usually resembles the following steps:

1. Create the HDInsight cluster. Clusters that are temporary are typically created in HDInsight using an Azure Resource Manager template. Clusters can be created using [PowerShell](#). If jobs will be running on a regular basis, you would want to schedule cluster creation using PowerShell or Azure CLI. PowerShell scripts and Azure CLI can be scheduled using [Azure Automation](#).

2. Schedule the batch jobs. Batch jobs in HDInsight are popularly run in Apache Hive, but they can be created using several different tools. If you choose to use Apache Hive, there are many ways to run those jobs in HDInsight. There are several ways to schedule a hive job including using [Azure Scheduler with PowerShell](#), using [Oozie](#), or using an HDInsight Hive activity in Azure Data Factory.

3. Pipe your data to a permanent storage location. Costs of storage in Azure are significantly cheaper than the cost of keeping an HDInsight cluster up and running. After you have completed a Hive job, you can export the results to permanent location like Azure SQL database, Azure SQL Data Warehouse, Azure Blob Storage, or Azure Data Lake Store.

Hadoop supports a notion of the default file system. The default file system implies a default scheme and authority. It can also be used to resolve relative paths. During step 1, you can specify a blob container in Azure Storage as the default file system, or with HDInsight 3.5, you can select either Azure Storage or Azure Data Lake Store as the default file system with a few exceptions. For the supportability of using Data Lake Store as both the default and linked storage, see [Availabilities for HDInsight cluster](#).

Since the charges for the cluster are many times more than the charges for storage, it makes economic sense to delete clusters when they are not in use.

4. Tear down the cluster. This can be done automatically using the methods mentioned above, or it can be done manually in the morning after confirmation that the jobs have all completed successfully. See examples using [Azure PowerShell](#), [Azure CLI](#), or the [.NET SDK](#).

Orchestrating the cluster lifecycle using Azure Data Factory

A popular option for automating the creation of and deleting HDInsight clusters, is [Azure Data Factory](#) (ADF). ADF includes its own scheduler you can use to apply various schedules in which your cluster's lifecycle is managed. This can result in significant IT cost-savings, as you have a simple and effective way to ensure your cluster is only up and

running when it is needed. The term for temporary clusters in Azure Data Factory is "on-demand clusters".

Azure Data Factory includes a linked service type for HDInsight, and pipeline activities for [Hive](#), [MapReduce](#), [Pig](#), [Hadoop Streaming](#), and [Spark](#).

Long-Running Clusters

If HDInsight will be used in a more interactive fashion, then it makes sense to keep the HDInsight cluster up and running permanently. Some of these workloads include:

- 1) Using Spark, Kafka or Storm for stream analytics.
- 2) Using Apache Hive, Storm, and Pig for interactive querying and analytics.
- 3) Using ETL and data cleaning tools permanently for real-time data processing.

! [NOTE]

When you create an HDInsight cluster, the processing can begin as soon as the nodes become available. You do not need to wait for all nodes in the cluster to be ready before using it to process jobs.

Scheduled Scale

The cost of HDInsight clusters is determined by the number of nodes and the virtual machines sizes for the nodes.

You can use PowerShell to select the number of nodes and virtual machine sizes for those nodes. You can scale them up during heavy usage and scale them down for light usage. Scheduling the scaling of nodes and virtual machine size can represent a significant cost savings.

Availability and reliability of Hadoop clusters in HDInsight

8/16/2017 • 9 min to read • [Edit Online](#)

HDInsight clusters provide two head nodes to increase the availability and reliability of Hadoop services and jobs running.

Hadoop achieves high availability and reliability by replicating services and data across multiple nodes in a cluster. However standard distributions of Hadoop typically have only a single head node. Any outage of the single head node can cause the cluster to stop working. HDInsight provides two headnodes to improve Hadoop's availability and reliability.

IMPORTANT

Linux is the only operating system used on HDInsight version 3.4 or greater. For more information, see [HDInsight retirement on Windows](#).

Availability and reliability of nodes

Nodes in an HDInsight cluster are implemented using Azure Virtual Machines. The following sections discuss the individual node types used with HDInsight.

NOTE

Not all node types are used for a cluster type. For example, a Hadoop cluster type does not have any Nimbus nodes. For more information on nodes used by HDInsight cluster types, see the Cluster types section of the [Create Linux-based Hadoop clusters in HDInsight](#) document.

Head nodes

To ensure high availability of Hadoop services, HDInsight provides two head nodes. Both head nodes are active and running within the HDInsight cluster simultaneously. Some services, such as HDFS or YARN, are only 'active' on one head node at any given time. Other services such as HiveServer2 or Hive MetaStore are active on both head nodes at the same time.

Head nodes (and other nodes in HDInsight) have a numeric value as part of the hostname of the node. For example, `hn0-CLUSTERNAME` or `hn4-CLUSTERNAME`.

IMPORTANT

Do not associate the numeric value with whether a node is primary or secondary. The numeric value is only present to provide a unique name for each node.

Nimbus Nodes

Nimbus nodes are available with Storm clusters. The Nimbus nodes provide similar functionality to the Hadoop JobTracker by distributing and monitoring processing across worker nodes. HDInsight provides two Nimbus nodes for Storm clusters

Zookeeper nodes

ZooKeeper nodes are used for leader election of master services on head nodes. They are also used to insure that services, data (worker) nodes, and gateways know which head node a master service is active on. By default, HDInsight provides three ZooKeeper nodes.

Worker nodes

Worker nodes perform the actual data analysis when a job is submitted to the cluster. If a worker node fails, the task that it was performing is submitted to another worker node. By default, HDInsight creates four worker nodes. You can change this number to suit your needs both during and after cluster creation.

Edge node

An edge node does not actively participate in data analysis within the cluster. It is used by developers or data scientists when working with Hadoop. The edge node lives in the same Azure Virtual Network as the other nodes in the cluster, and can directly access all other nodes. The edge node can be used without taking resources away from critical Hadoop services or analysis jobs.

Currently, R Server on HDInsight is the only cluster type that provides an edge node by default. For R Server on HDInsight, the edge node is used test R code locally on the node before submitting it to the cluster for distributed processing.

For information on using an edge node with cluster types other than R Server, see the [Use edge nodes in HDInsight](#) document.

Accessing the nodes

Access to the cluster over the internet is provided through a public gateway. Access is limited to connecting to the head nodes and (if one exists) the edge node. Access to services running on the head nodes is not effected by having multiple head nodes. The public gateway routes requests to the head node that hosts the requested service. For example, if Ambari is currently hosted on the secondary head node, the gateway routes incoming requests for Ambari to that node.

Access over the public gateway is limited to port 443 (HTTPS), 22, and 23.

- Port **443** is used to access Ambari and other web UI or REST APIs hosted on the head nodes.
- Port **22** is used to access the primary head node or edge node with SSH.
- Port **23** is used to access the secondary head node with SSH. For example,
`ssh username@mycluster-ssh.azurehdinsight.net` connects to the primary head node of the cluster named **mycluster**.

For more information on using SSH, see the [Use SSH with HDInsight](#) document.

Internal fully qualified domain names (FQDN)

Nodes in an HDInsight cluster have an internal IP address and FQDN that can only be accessed from the cluster. When accessing services on the cluster using the internal FQDN or IP address, you should use Ambari to verify the IP or FQDN to use when accessing the service.

For example, the Oozie service can only run on one head node, and using the `oozie` command from an SSH session requires the URL to the service. This URL can be retrieved from Ambari by using the following command:

```
curl -u admin:PASSWORD "https://CLUSTERNAME.azurehdinsight.net/api/v1/clusters/CLUSTERNAME/configurations?&type=oozie-site&tag=TOPOLOGY_RESOLVED" | grep oozie.base.url
```

This command returns a value similar to the following command, which contains the internal URL to use with the `oozie` command:

```
"oozie.base.url": "http://hn0-CLUSTERNAME-randomcharacters.cx.internal.cloudapp.net:11000/oozie"
```

For more information on working with the Ambari REST API, see [Monitor and Manage HDInsight using the Ambari REST API](#).

Accessing other node types

You can connect to nodes that are not directly accessible over the internet by using the following methods:

- **SSH:** Once connected to a head node using SSH, you can then use SSH from the head node to connect to other nodes in the cluster. For more information, see the [Use SSH with HDInsight](#) document.
- **SSH Tunnel:** If you need to access a web service hosted on one of the nodes that is not exposed to the internet, you must use an SSH tunnel. For more information, see the [Use an SSH tunnel with HDInsight](#) document.
- **Azure Virtual Network:** If your HDInsight cluster is part of an Azure Virtual Network, any resource on the same Virtual Network can directly access all nodes in the cluster. For more information, see the [Extend HDInsight using Azure Virtual Network](#) document.

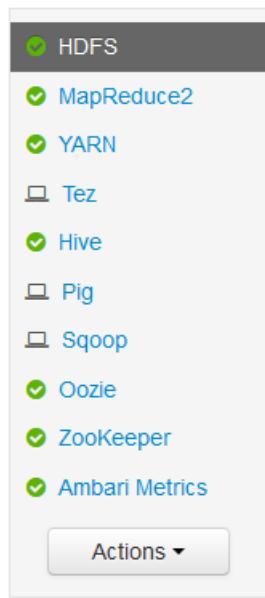
How to check on a service status

To check the status of services that run on the head nodes, use the Ambari Web UI or the Ambari REST API.

Ambari Web UI

The Ambari Web UI is viewable at <https://CLUSTERNAME.azurehdinsight.net>. Replace **CLUSTERNAME** with the name of your cluster. If prompted, enter the HTTP user credentials for your cluster. The default HTTP user name is **admin** and the password is the password you entered when creating the cluster.

When you arrive on the Ambari page, the installed services are listed on the left of the page.



There are a series of icons that may appear next to a service to indicate status. Any alerts related to a service can be viewed using the **Alerts** link at the top of the page. You can select each service to view more information on it.

While the service page provides information on the status and configuration of each service, it does not provide information on which head node the service is running on. To view this information, use the **Hosts** link at the top of the page. This page displays hosts within the cluster, including the head nodes.

Name	IP Address	Rack	Cores	RAM	Disk Usage	Load Avg	Versions	Components
<input type="checkbox"/> Any	<input type="text"/> Any	<input type="text"/> Any	<input type="text"/> Any	<input type="text"/> Any	<input type="text"/> Any	<input type="text"/> Any	Filter	Filter
<input checked="" type="checkbox"/> headnode0....	10.0.0.40	/default-...	4 (4)	6.81GB	<div style="width: 100px; height: 10px; background-color: #ccc;"></div>	<div style="width: 10px; height: 10px; background-color: #ccc;"></div>	HDP-2.2.4.9-1 (Current)	▶ 20 Components
<input checked="" type="checkbox"/> headnode1....	10.0.0.42	/default-...	4 (4)	6.81GB	<div style="width: 100px; height: 10px; background-color: #ccc;"></div>	<div style="width: 10px; height: 10px; background-color: #ccc;"></div>	HDP-2.2.4.9-1 (Current)	▶ 12 Components

Selecting the link for one of the head nodes displays the services and components running on that node.

headnode0.CLUSTERNAME-ssh.d9.internal.cloudapp.net

[◀ Back](#)

[Summary](#) [Configs](#) [Alerts 0](#) [Versions](#)

Components	+ Add
<input checked="" type="checkbox"/> App Timeline Server / YARN	Started ▾
<input checked="" type="checkbox"/> History Server / MapReduce2	Started ▾
<input checked="" type="checkbox"/> Hive Metastore / Hive	Started ▾
<input checked="" type="checkbox"/> HiveServer2 / Hive	Started ▾
<input checked="" type="checkbox"/> Metrics Collector / Ambari Metrics	Started ▾
<input checked="" type="checkbox"/> Standby NameNode / HDFS	Started ▾
<input checked="" type="checkbox"/> Oozie Server / Oozie	Started ▾
<input checked="" type="checkbox"/> Standby ResourceManager / YARN	Started ▾
<input checked="" type="checkbox"/> WebHCat Server / Hive	Started ▾
<input checked="" type="checkbox"/> Metrics Monitor / Ambari Metrics	Started ▾

For more information on using Ambari, see [Monitor and manage HDInsight using the Ambari Web UI](#).

Ambari REST API

The Ambari REST API is available over the internet. The HDInsight public gateway handles routing requests to the head node that is currently hosting the REST API.

You can use the following command to check the state of a service through the Ambari REST API:

```
curl -u admin:PASSWORD https://CLUSTERNAME.azurehdinsight.net/api/v1/clusters/CLUSTERNAME/services/SERVICENAME?
fields=ServiceInfo/state
```

- Replace **PASSWORD** with the HTTP user (admin) account password.
- Replace **CLUSTERNAME** with the name of the cluster.
- Replace **SERVICENAME** with the name of the service you want to check the status of.

For example, to check the status of the **HDFS** service on a cluster named **mycluster**, with a password of **password**, you would use the following command:

```
curl -u admin:password https://mycluster.azurehdinsight.net/api/v1/clusters/mycluster/services/HDFS?  
fields=ServiceInfo/state
```

The response is similar to the following JSON:

```
{  
  "href" : "http://hn0-  
CLUSTERNAME.randomcharacters.cx.internal.cloudapp.net:8080/api/v1/clusters/mycluster/services/HDFS?  
fields=ServiceInfo/state",  
  "ServiceInfo" : {  
    "cluster_name" : "mycluster",  
    "service_name" : "HDFS",  
    "state" : "STARTED"  
  }  
}
```

The URL tells us that the service is currently running on a head node named **hn0-CLUSTERNAME**.

The state tells us that the service is currently running, or **STARTED**.

If you do not know what services are installed on the cluster, you can use the following command to retrieve a list:

```
curl -u admin:PASSWORD https://CLUSTERNAME.azurehdinsight.net/api/v1/clusters/CLUSTERNAME/services
```

For more information on working with the Ambari REST API, see [Monitor and Manage HDInsight using the Ambari REST API](#).

Service components

Services may contain components that you wish to check the status of individually. For example, HDFS contains the NameNode component. To view information on a component, the command would be:

```
curl -u admin:PASSWORD  
https://CLUSTERNAME.azurehdinsight.net/api/v1/clusters/CLUSTERNAME/services/SERVICE/components/component
```

If you do not know what components are provided by a service, you can use the following command to retrieve a list:

```
curl -u admin:PASSWORD  
https://CLUSTERNAME.azurehdinsight.net/api/v1/clusters/CLUSTERNAME/services/SERVICE/components/component
```

How to access log files on the head nodes

SSH

While connected to a head node through SSH, log files can be found under **/var/log**. For example, **/var/log/hadoop-yarn/yarn** contain logs for YARN.

Each head node can have unique log entries, so you should check the logs on both.

SFTP

You can also connect to the head node using the SSH File Transfer Protocol or Secure File Transfer Protocol (SFTP), and download the log files directly.

Similar to using an SSH client, when connecting to the cluster you must provide the SSH user account name and the SSH address of the cluster. For example, `sftp username@mycluster-ssh.azurehdinsight.net`. Provide the password

for the account when prompted, or provide a public key using the `-i` parameter.

Once connected, you are presented with a `sftp>` prompt. From this prompt, you can change directories, upload, and download files. For example, the following commands change directories to the **/var/log/hadoop/hdfs** directory and then download all files in the directory.

```
cd /var/log/hadoop/hdfs  
get *
```

For a list of available commands, enter `help` at the `sftp>` prompt.

NOTE

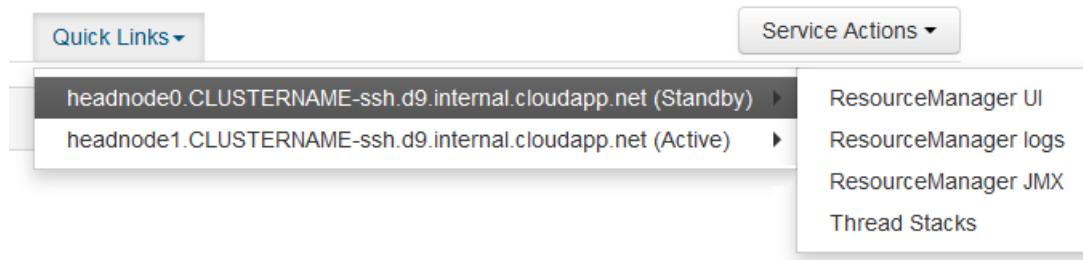
There are also graphical interfaces that allow you to visualize the file system when connected using SFTP. For example, [MobaXTerm](#) allows you to browse the file system using an interface similar to Windows Explorer.

Ambari

NOTE

To access log files using Ambari, you must use an SSH tunnel. The web interfaces for the individual services are not exposed publicly on the Internet. For information on using an SSH tunnel, see the [Use SSH Tunneling](#) document.

From the Ambari Web UI, select the service you wish to view logs for (for example, YARN). Then use **Quick Links** to select which head node to view the logs for.



How to configure the node size

The size of a node can only be selected during cluster creation. You can find a list of the different VM sizes available for HDInsight on the [HDInsight pricing page](#).

When creating a cluster, you can specify the size of the nodes. The following information provides guidance on how to specify the size using the [Azure portal](#), [Azure PowerShell](#), and the [Azure CLI](#):

- **Azure portal:** When creating a cluster, you can set the size of the nodes used by the cluster:

The screenshot shows two windows side-by-side. The left window is titled 'Node Pricing Tiers' and displays configuration options for a cluster. It includes fields for 'Number of Worker nodes' (set to 4), 'Worker Nodes Pricing Tier' (selected as D12 (4 nodes)), and 'Head Node Pricing Tier' (selected as D12 (2 nodes)). Below these are cost calculations: 'WORKER NODES' at 0.40 x 4 = 1.61, 'HEAD NODES' at 0.40 x 2 = 0.81, and a total 'TOTAL COST' of 2.42 USD/HOUR (ESTIMATED). A note states 'Using 119 of 260 total cores in South Central US'. The right window is titled 'Choose your pricing tier' and lists three pricing tiers: D12 Standard, D13 Standard, and D14 Standard. Each tier is defined by its core count, RAM, and disk storage. The D12 Standard tier is highlighted with a blue border and has a price of 0.40 USD/HOUR (ESTIMATED). The D13 Standard tier has a price of 0.73 USD/HOUR (ESTIMATED), and the D14 Standard tier has a price of 1.31 USD/HOUR (ESTIMATED). Both windows have a 'Select' button at the bottom.

Pricing Tier	Cores	RAM	Disk Storage	Price (USD/HOUR ESTIMATED)
D12 Standard	4	28 GB RAM	8 Disks 200 GB Local SSD	0.40
D13 Standard	8	56 GB RAM	16 Disks 400 GB Local SSD	0.73
D14 Standard	16	112 GB RAM	32 Disks 800 GB Local SSD	1.31

- **Azure CLI:** When using the `azure hdinsight cluster create` command, you can set the size of the head, worker, and ZooKeeper nodes by using the `--headNodeSize`, `--workerNodeSize`, and `--zookeeperNodeSize` parameters.
- **Azure PowerShell:** When using the `New-AzureRmHDInsightCluster` cmdlet, you can set the size of the head, worker, and ZooKeeper nodes by using the `-HeadNodeVmSize`, `-WorkerNodeSize`, and `-ZookeeperNodeSize` parameters.

Next steps

Use the following links to learn more about things mentioned in this document.

- [Ambari REST Reference](#)
- [Install and configure the Azure CLI](#)
- [Install and configure Azure PowerShell](#)
- [Manage HDInsight using Ambari](#)
- [Provision Linux-based HDInsight clusters](#)

Capacity Planning

8/16/2017 • 5 min to read • [Edit Online](#)

Capacity planning is an important first step in deploying your HDInsight cluster. A good plan can help you optimize your costs while delivering high performance and usability to the users of your cluster. In addition, when provisioning your cluster there are a series of decisions you need to make that are either difficult or impossible to change later- thinking thru these decisions can save you from having to tear down your cluster and setup a new one to address the issue.

The key questions to ask when doing your capacity planning are:

- In which geographic region should you deploy your cluster?
- What cluster type should you deploy?
- What size and type of virtual machine should your cluster nodes use?
- How many worker nodes should your cluster have?

The sections that follow provide guidance on each of these topics.

Choosing a region

The region determines where your cluster is actually provisioned. The primary consideration in selecting a region is data locality- you want your cluster physically near the data it will process to minimize latency of reads and writes. There are a few scenarios to consider.

Availability of HDInsight in the region desired

HDInsight is available in most Azure regions, but not all. Be sure to check the HDInsight Linux entry in [Azure Products Available by Region](#) when selecting the location for your cluster deployment.

Location of default storage

When provisioning your cluster, the default storage selected (e.g., the Azure Storage Account or Data Lake Store) must be in the same location as your cluster. This means your storage choice dictates the location you must select for your cluster. When using Data Lake Store for default storage, this may further narrow the locations available to your cluster as the Data Lake Store is currently only available in three locations globally (Central US, East US 2 and North Europe). Azure Storage is available in all location, so this same consideration does not apply.

Location of existing data

If you already have a Storage Account or Data Lake Store deployed and loaded with data and intend to use this storage as your cluster's default storage, then you will need to deploy your cluster into the same location as where the data is stored. Once you have an HDInsight cluster deployed, you can attach additional Azure Storage Accounts or access other Data Lake Stores. When using a Storage Account as an additional storage location, this account must reside in the same location as your cluster. When using a Data Lake Store, however, the cluster and the Data Lake Store can reside in different locations (although consider the latency consequences of doing so as the data has further to travel in reaching your cluster).

Choosing a cluster type

The cluster type dictates the workload your HDInsight cluster is configured to run, such as Hadoop, Storm, Kafka or Spark. For a detailed description of the available cluster types, see [HDInsight Architecture](#). Each cluster type brings with it a specific deployment topology along with requirements specific to the type of nodes, namely the size and number of nodes.

Choosing the VM size and type

When planning your cluster deployment, you need to consider the VM size and type. Each cluster type brings with it a set of node types, and each node type has specific options for the VM size and type. You typically select different VM sizes and types for the different node types. What your particular cluster will need is something you will need to benchmark using either simulated workloads (where you run your expected workloads on different size clusters, gradually increasing the size until the desired performance is reached) or canary queries (where you have specific queries you run periodically among the other production queries that can indicate if the cluster has enough resources or not). In general terms, the selection of the VM size and type boils down to selecting amongst CPU, RAM Memory and Network:

- CPU: The VM size dictates the number of cores. The more cores, the greater the degree of parallel computation each node can achieve. Additionally, some VM types have cores that are faster than the standard offering.
- RAM: The VM size also dictates the amount of RAM available in the VM. With many workloads choosing to store data in memory for processing, as opposed to reading from disk, ensuring your worker nodes have enough memory to fit the data is one way to drive your selection of VM size.
- Network: For most cluster types, the data processed by the cluster lives in an external storage service (e.g., Data Lake Store or Azure Storage) and not on the local disks. Therefore, the network bandwidth and throughput connecting the node VM and this storage service is an important consideration. Typically, the network bandwidth available to a VM increases with larger sizes. For details, see [VM sizes overview](#).

Choosing the cluster scale

Related to the size and type of the VM nodes, is the quantity of VM nodes in your cluster. This is often referred to as the cluster scale. For all cluster types, there are node types that have a specific scale (e.g., they require exactly three ZooKeeper nodes or two Head nodes) and node types that support scale out. The latter typically applies to Worker nodes that do the processing in a distributed fashion, where the processing can benefit from scaling out (e.g., adding additional Worker nodes). Depending on your cluster type, increasing the number of Worker nodes will add additional computational capacity (e.g., more cores), but may also add to the total amount of memory available cluster wide to support in-memory storage of data being processed. As for the choice of VM size and type, the approach for selecting the right cluster scale is typically reached empirically either by using simulated workloads or canary queries.

Quotas

A very important consideration when planning your deployment, after you have identified your target cluster VM size, scale and type is to confirm your subscription has enough quota capacity remaining. When you reach a quota limit, you may be blocked from deploying new clusters or from scaling out existing clusters by adding more worker nodes. The most common quota limit reached is the CPU Cores quota which exists at the subscription, region and VM series levels. For example, your subscription may have a 200 core limit and you may have a 30 core limit in West US 2 and a 30 core limit on Dv2 instances. These quota limits are soft in that you can [contact support to request a quota increase](#). However, there are some hard limits that cannot be changed with a call to support. When it comes to CPU cores, a single Azure subscription can have at most 10,000 cores, and this is a hard limit. For details on these limits, see [Azure subscription and service limits, quotas, and constraints](#).

Next steps

- [Set up clusters in HDInsight with Hadoop, Spark, Kafka, and more](#): Learn how to set up and configure clusters in HDInsight with Hadoop, Spark, Kafka, Interactive Hive, HBase, R Server, or Storm.
- [Key Scenarios to monitor in HDInsight](#): Learn about key scenarios to monitor for your HDInsight cluster that might affect your cluster's capacity.

Connect to HDInsight (Hadoop) using SSH

8/16/2017 • 9 min to read • [Edit Online](#)

Learn how to use [Secure Shell \(SSH\)](#) to securely connect to Hadoop on Azure HDInsight.

HDInsight can use Linux (Ubuntu) as the operating system for nodes within the Hadoop cluster. The following table contains the address and port information needed when connecting to Linux-based HDInsight using an SSH client:

ADDRESS	PORT	CONNECTS TO...
<clustername>-ed-ssh.azurehdinsight.net	22	Edge node (R Server on HDInsight)
<edgenodename>. <clustername>-ssh.azurehdinsight.net	22	Edge node (any other cluster type, if an edge node exists)
<clustername>-ssh.azurehdinsight.net	22	Primary headnode
<clustername>-ssh.azurehdinsight.net	23	Secondary headnode

NOTE

Replace <edgenodename> with the name of the edge node.

Replace <clustername> with the name of your cluster.

If your cluster contains an edge node, we recommend that you **always connect to the edge node** using SSH. The head nodes host services that are critical to the health of Hadoop. The edge node runs only what you put on it.

For more information on using edge nodes, see [Use edge nodes in HDInsight](#).

SSH clients

Linux, Unix, and macOS systems provide the `ssh` and `scp` commands. The `ssh` client is commonly used to create a remote command-line session with a Linux or Unix-based system. The `scp` client is used to securely copy files between your client and the remote system.

Microsoft Windows does not provide any SSH clients by default. The `ssh` and `scp` clients are available for Windows through the following packages:

- [Azure Cloud Shell](#): The Cloud Shell provides a Bash environment in your browser, and provides the `ssh`, `scp`, and other common Linux commands.
- [Bash on Ubuntu on Windows 10](#): The `ssh` and `scp` commands are available through the Bash on Windows command line.
- [Git \(<https://git-scm.com/>\)](#): The `ssh` and `scp` commands are available through the GitBash command line.
- [GitHub Desktop \(<https://desktop.github.com/>\)](#) The `ssh` and `scp` commands are available through the

GitHub Shell command line. GitHub Desktop can be configured to use Bash, the Windows Command Prompt, or PowerShell as the command line for the Git Shell.

- [OpenSSH](https://github.com/PowerShell/Win32-OpenSSH/wiki/Install-Win32-OpenSSH) (<https://github.com/PowerShell/Win32-OpenSSH/wiki/Install-Win32-OpenSSH>): The PowerShell team is porting OpenSSH to Windows, and provides test releases.

WARNING

The OpenSSH package includes the SSH server component, `sshd`. This component starts an SSH server on your system, allowing others to connect to it. Do not configure this component or open port 22 unless you want to host an SSH server on your system. It is not required to communicate with HDInsight.

There are also several graphical SSH clients, such as [PuTTY](http://www.chiark.greenend.org.uk/~sgtatham/putty/) (<http://www.chiark.greenend.org.uk/~sgtatham/putty/>) and [MobaXterm](http://mobaxterm.mobatek.net/) (<http://mobaxterm.mobatek.net/>). While these clients can be used to connect to HDInsight, the process of connecting is different than using the `ssh` utility. For more information, see the documentation of the graphical client you are using.

Authentication: SSH Keys

SSH keys use [Public-key cryptography](#) to authenticate SSH sessions. SSH keys are more secure than passwords, and provide an easy way to secure access to your Hadoop cluster.

If your SSH account is secured using a key, the client must provide the matching private key when you connect:

- Most clients can be configured to use a **default key**. For example, the `ssh` client looks for a private key at `~/.ssh/id_rsa` on Linux and Unix environments.
- You can specify the **path to a private key**. With the `ssh` client, the `-i` parameter is used to specify the path to private key. For example, `ssh -i ~/.ssh/id_rsa sshuser@myedge.mycluster-ssh.azurehdinsight.net`.
- If you have **multiple private keys** for use with different servers, consider using a utility such as [ssh-agent](#) (<https://en.wikipedia.org/wiki/Ssh-agent>). The `ssh-agent` utility can be used to automatically select the key to use when establishing an SSH session.

IMPORTANT

If you secure your private key with a passphrase, you must enter the passphrase when using the key. Utilities such as `ssh-agent` can cache the password for your convenience.

Create an SSH key pair

Use the `ssh-keygen` command to create public and private key files. The following command generates a 2048-bit RSA key pair that can be used with HDInsight:

```
ssh-keygen -t rsa -b 2048
```

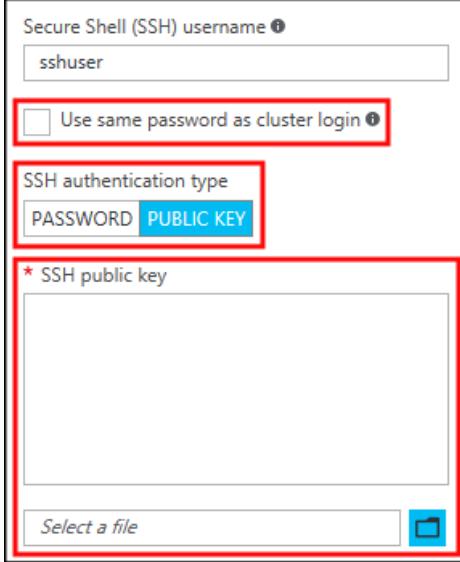
You are prompted for information during the key creation process. For example, where the keys are stored or whether to use a passphrase. After the process completes, two files are created; a public key and a private key.

- The **public key** is used to create an HDInsight cluster. The public key has an extension of `.pub`.
- The **private key** is used to authenticate your client to the HDInsight cluster.

IMPORTANT

You can secure your keys using a passphrase. A passphrase is effectively a password on your private key. Even if someone obtains your private key, they must have the passphrase to use the key.

Create HDInsight using the public key

CREATION METHOD	HOW TO USE THE PUBLIC KEY
Azure portal	<p>Uncheck Use same password as cluster login, and then select Public Key as the SSH authentication type. Finally, select the public key file or paste the text contents of the file in the SSH public key field.</p> 
Azure PowerShell	<p>Use the <code>-SshPublicKey</code> parameter of the <code>New-AzureRmHdinsightCluster</code> cmdlet and pass the contents of the public key as a string.</p>
Azure CLI 1.0	<p>Use the <code>--sshPublicKey</code> parameter of the <code>azure hdinsight cluster create</code> command and pass the contents of the public key as a string.</p>
Resource Manager Template	<p>For an example of using SSH keys with a template, see Deploy HDInsight on Linux with SSH key. The <code>publicKeys</code> element in the <code>azuredeploy.json</code> file is used to pass the keys to Azure when creating the cluster.</p>

Authentication: Password

SSH accounts can be secured using a password. When you connect to HDInsight using SSH, you are prompted to enter the password.

WARNING

We do not recommend using password authentication for SSH. Passwords can be guessed and are vulnerable to brute force attacks. Instead, we recommend that you use [SSH keys for authentication](#).

Create HDInsight using a password

CREATION METHOD	HOW TO SPECIFY THE PASSWORD
Azure portal	<p>By default, the SSH user account has the same password as the cluster login account. To use a different password, uncheck Use same password as cluster login, and then enter the password in the SSH password field.</p>
Azure PowerShell	<p>Use the <code>--SshCredential</code> parameter of the <code>New-AzureRmHdinsightCluster</code> cmdlet and pass a <code>PSCredential</code> object that contains the SSH user account name and password.</p>
Azure CLI 1.0	<p>Use the <code>--sshPassword</code> parameter of the <code>azure hdinsight cluster create</code> command and provide the password value.</p>
Resource Manager Template	<p>For an example of using a password with a template, see Deploy HDInsight on Linux with SSH password. The <code>linuxOperatingSystemProfile</code> element in the <code>azuredeploy.json</code> file is used to pass the SSH account name and password to Azure when creating the cluster.</p>

Change the SSH password

For information on changing the SSH user account password, see the **Change passwords** section of the [Manage HDInsight](#) document.

Authentication: Domain-joined HDInsight

If you are using a **domain-joined HDInsight cluster**, you must use the `kinit` command after connecting with SSH. This command prompts you for a domain user and password, and authenticates your session with the Azure Active Directory domain associated with the cluster.

For more information, see [Configure domain-joined HDInsight](#).

Connect to nodes

The head nodes and edge node (if there is one) can be accessed over the internet on ports 22 and 23.

- When connecting to the **head nodes**, use port **22** to connect to the primary head node and port **23** to connect to the secondary head node. The fully qualified domain name to use is `clusternode-ssh.azurehdinsight.net`, where `clusternode` is the name of your cluster.

```
# Connect to primary head node  
# port not specified since 22 is the default  
ssh sshuser@clustername-ssh.azurehdinsight.net  
  
# Connect to secondary head node  
ssh -p 23 sshuser@clustername-ssh.azurehdinsight.net
```

- When connecting to the **edge node**, use port 22. The fully qualified domain name is `edgenodename.clustername-ssh.azurehdinsight.net`, where `edgenodename` is a name you provided when creating the edge node. `clustername` is the name of the cluster.

```
# Connect to edge node  
ssh sshuser@edgnodename.clustername-ssh.azurehdinsight.net
```

IMPORTANT

The previous examples assume that you are using password authentication, or that certificate authentication is occurring automatically. If you use an SSH key-pair for authentication, and the certificate is not used automatically, use the `-i` parameter to specify the private key. For example,

```
ssh -i ~/.ssh/mykey sshuser@clustername-ssh.azurehdinsight.net .
```

Once connected, the prompt changes to indicate the SSH user name and the node you are connected to. For example, when connected to the primary head node as `sshuser`, the prompt is `sshuser@hn0-clustername:~$`.

Connect to worker and Zookeeper nodes

The worker nodes and Zookeeper nodes are not directly accessible from the internet. They can be accessed from the cluster head nodes or edge nodes. The following are the general steps to connect to other nodes:

- Use SSH to connect to a head or edge node:

```
ssh sshuser@myedge.mycluster-ssh.azurehdinsight.net
```

- From the SSH connection to the head or edge node, use the `ssh` command to connect to a worker node in the cluster:

```
ssh sshuser@wn0-myhd1
```

To retrieve a list of the domain names of the nodes in the cluster, see the [Manage HDInsight by using the Ambari REST API](#) document.

If the SSH account is secured using a **password**, enter the password when connecting.

If the SSH account is secured using **SSH keys**, make sure that SSH forwarding is enabled on the client.

NOTE

Another way to directly access all nodes in the cluster is to install HDInsight into an Azure Virtual Network. Then, you can join your remote machine to the same virtual network and directly access all nodes in the cluster.

For more information, see [Use a virtual network with HDInsight](#).

Configure SSH agent forwarding

IMPORTANT

The following steps assume a Linux or UNIX-based system, and work with Bash on Windows 10. If these steps do not work for your system, you may need to consult the documentation for your SSH client.

1. Using a text editor, open `~/.ssh/config`. If this file doesn't exist, you can create it by entering `touch ~/.ssh/config` at a command line.

2. Add the following text to the `config` file.

```
Host <edgenodename>.<clusternode>-ssh.azurehdinsight.net
  ForwardAgent yes
```

Replace the **Host** information with the address of the node you connect to using SSH. The previous example uses the edge node. This entry configures SSH agent forwarding for the specified node.

3. Test SSH agent forwarding by using the following command from the terminal:

```
echo "$SSH_AUTH_SOCK"
```

This command returns information similar to the following text:

```
/tmp/ssh-rfSUL1ldCldQ/agent.1792
```

If nothing is returned, then `ssh-agent` is not running. For more information, see the agent startup scripts information at [Using ssh-agent with ssh \(http://mah.everybody.org/docs/ssh\)](http://mah.everybody.org/docs/ssh) or consult your SSH client documentation.

4. Once you have verified that `ssh-agent` is running, use the following to add your SSH private key to the agent:

```
ssh-add ~/.ssh/id_rsa
```

If your private key is stored in a different file, replace `~/.ssh/id_rsa` with the path to the file.

5. Connect to the cluster edge node or head nodes using SSH. Then use the SSH command to connect to a worker or zookeeper node. The connection is established using the forwarded key.

Copy files

The `scp` utility can be used to copy files to and from individual nodes in the cluster. For example, the following command copies the `test.txt` directory from the local system to the primary head node:

```
scp test.txt sshuser@clusternode-ssh.azurehdinsight.net:
```

Since no path is specified after the `:`, the file is placed in the `sshuser` home directory.

The following example copies the `test.txt` file from the `sshuser` home directory on the primary head node to the local system:

```
scp sshuser@clusternode-ssh.azurehdinsight.net:test.txt .
```

IMPORTANT

`scp` can only access the file system of individual nodes within the cluster. It cannot be used to access data in the HDFS-compatible storage for the cluster.

Use `scp` when you need to upload a resource for use from an SSH session. For example, upload a Python script and then run the script from an SSH session.

For information on directly loading data into the HDFS-compatible storage, see the following documents:

- [HDInsight using Azure Storage](#).
- [HDInsight using Azure Data Lake Store](#).

Next steps

- [Use SSH tunneling with HDInsight](#)
- [Use a virtual network with HDInsight](#)
- [Use edge nodes in HDInsight](#)

Use SSH Tunneling to access Ambari web UI, JobHistory, NameNode, Oozie, and other web UIs

8/15/2017 • 5 min to read • [Edit Online](#)

Linux-based HDInsight clusters provide access to Ambari web UI over the Internet, but some features of the UI are not. For example, the web UI for other services that are surfaced through Ambari. For full functionality of the Ambari web UI, you must use an SSH tunnel to the cluster head.

Why use an SSH tunnel

Several of the menus in Ambari only work through an SSH tunnel. These menus rely on web sites and services running on other node types, such as worker nodes. Often, these web sites are not secured, so it is not safe to directly expose them on the internet.

The following Web UIs require an SSH tunnel:

- JobHistory
- NameNode
- Thread Stacks
- Oozie web UI
- HBase Master and Logs UI

If you use Script Actions to customize your cluster, any services or utilities that you install that expose a web UI require an SSH tunnel. For example, if you install Hue using a Script Action, you must use an SSH tunnel to access the Hue web UI.

What is an SSH tunnel

[Secure Shell \(SSH\) tunneling](#) routes traffic sent to a port on your local workstation. The traffic is routed through an SSH connection to your HDInsight cluster head node. The request is resolved as if it originated on the head node. The response is then routed back through the tunnel to your workstation.

Prerequisites

- An SSH client. For more information, see [Use SSH with HDInsight](#).
- A web browser that can be configured to use a SOCKS5 proxy.

WARNING

The SOCKS proxy support built into Windows does not support SOCKS5, and does not work with the steps in this document. The following browsers rely on Windows proxy settings, and do not currently work with the steps in this document:

- Microsoft Edge
- Microsoft Internet Explorer

Google Chrome also relies on the Windows proxy settings. However, you can install extensions that support SOCKS5. We recommend [FoxyProxy Standard](#).

Create a tunnel using the SSH command

Use the following command to create an SSH tunnel using the `ssh` command. Replace **USERNAME** with an SSH user for your HDInsight cluster, and replace **CLUSTERNAME** with the name of your HDInsight cluster:

```
ssh -C2qTnNF -D 9876 USERNAME@CLUSTERNAME-ssh.azurehdinsight.net
```

This command creates a connection that routes traffic to local port 9876 to the cluster over SSH. The options are:

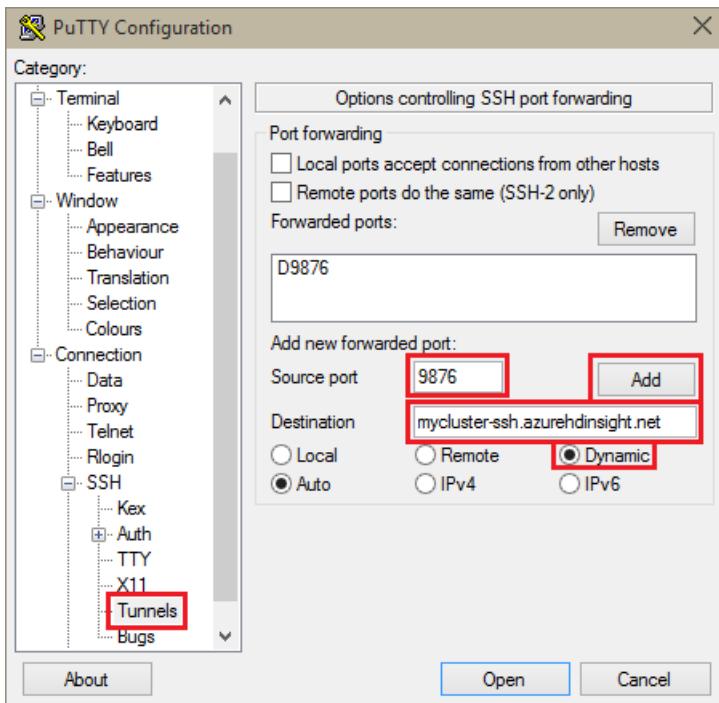
- **D 9876** - The local port that routes traffic through the tunnel.
- **C** - Compress all data, because web traffic is mostly text.
- **2** - Force SSH to try protocol version 2 only.
- **q** - Quiet mode.
- **T** - Disable pseudo-tty allocation, since we are just forwarding a port.
- **n** - Prevent reading of STDIN, since we are just forwarding a port.
- **N** - Do not execute a remote command, since we are just forwarding a port.
- **f** - Run in the background.

Once the command finishes, traffic sent to port 9876 on the local computer is routed to the cluster head node.

Create a tunnel using PuTTY

PuTTY is a graphical SSH client for Windows. Use the following steps to create an SSH tunnel using PuTTY:

1. Open PuTTY, and enter your connection information. If you are not familiar with PuTTY, see the [PuTTY documentation](http://www.chiark.greenend.org.uk/~sgtatham/putty/docs.html) (<http://www.chiark.greenend.org.uk/~sgtatham/putty/docs.html>).
2. In the **Category** section to the left of the dialog, expand **Connection**, expand **SSH**, and then select **Tunnels**.
3. Provide the following information on the **Options controlling SSH port forwarding** form:
 - **Source port** - The port on the client that you wish to forward. For example, **9876**.
 - **Destination** - The SSH address for the Linux-based HDInsight cluster. For example, **mycluster-ssh.azurehdinsight.net**.
 - **Dynamic** - Enables dynamic SOCKS proxy routing.



4. Click **Add** to add the settings, and then click **Open** to open an SSH connection.

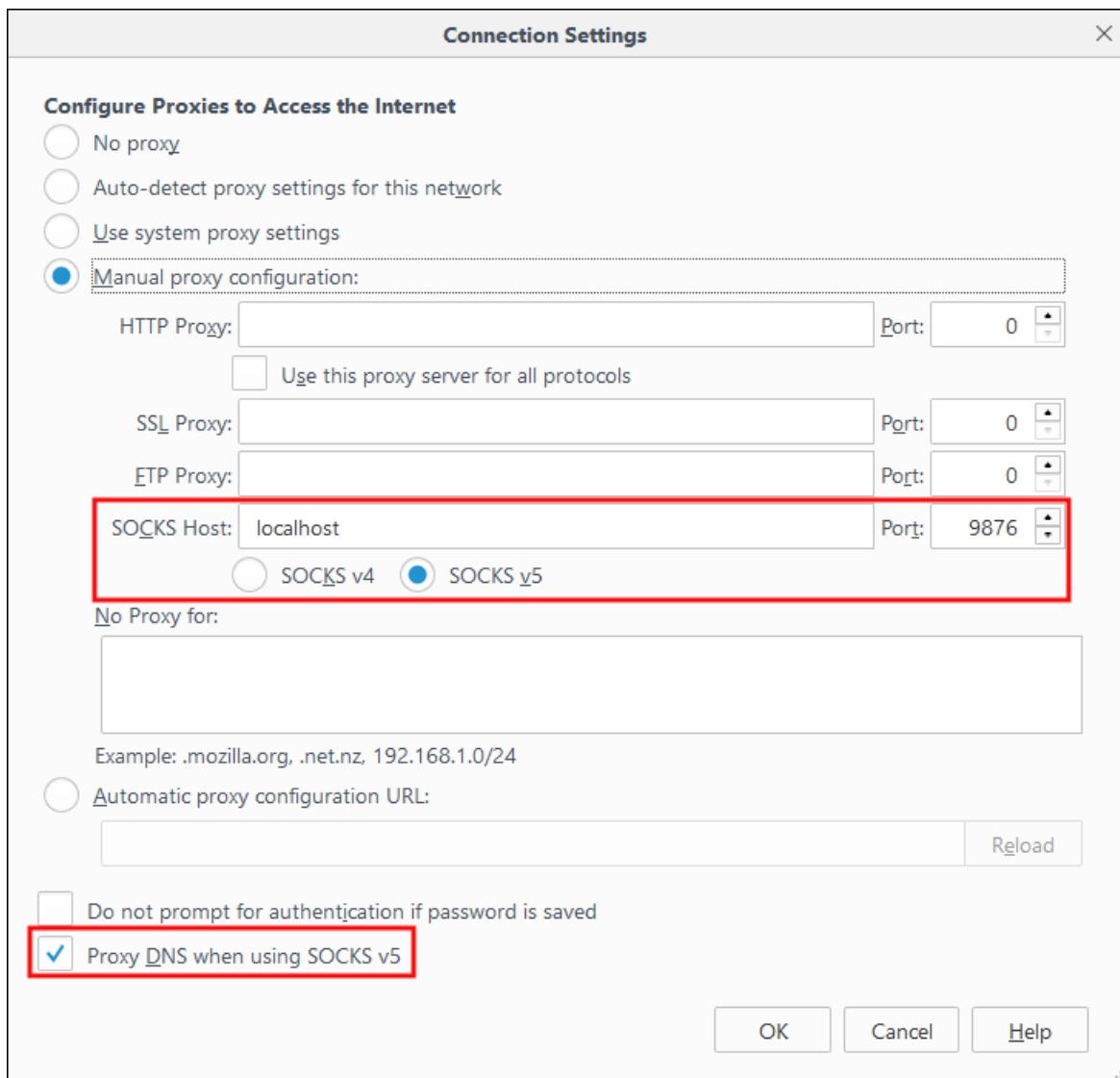
5. When prompted, log in to the server.

Use the tunnel from your browser

IMPORTANT

The steps in this section use the Mozilla Firefox browser, as it provides the same proxy settings across all platforms. Other modern browsers, such as Google Chrome, may require an extension such as FoxyProxy to work with the tunnel.

1. Configure the browser to use **localhost** and the port you used when creating the tunnel as a **SOCKS v5** proxy. Here's what the Firefox settings look like. If you used a different port than 9876, change the port to the one you used:



NOTE

Selecting **Remote DNS** resolves Domain Name System (DNS) requests by using the HDInsight cluster. This setting resolves DNS using the head node of the cluster.

2. Verify that the tunnel works by visiting a site such as <http://www.whatismyip.com/>. If the proxy is correctly configured, the IP address returned is from a machine in the Microsoft Azure datacenter.

Verify with Ambari web UI

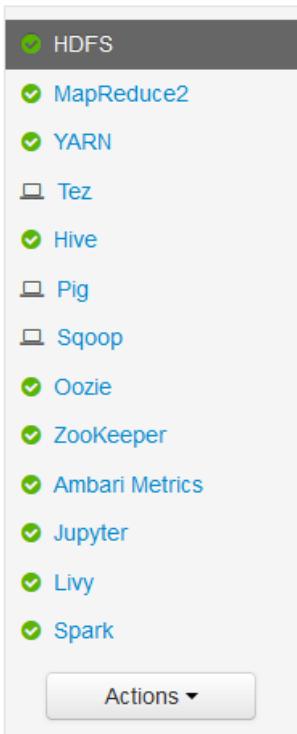
Once the cluster has been established, use the following steps to verify that you can access service web UIs from the Ambari Web:

1. In your browser, go to <http://headnodehost:8080>. The `headnodehost` address is sent over the tunnel to the cluster and resolve to the headnode that Ambari is running on. When prompted, enter the admin user name (admin) and password for your cluster. You may be prompted a second time by the Ambari web UI. If so, reenter the information.

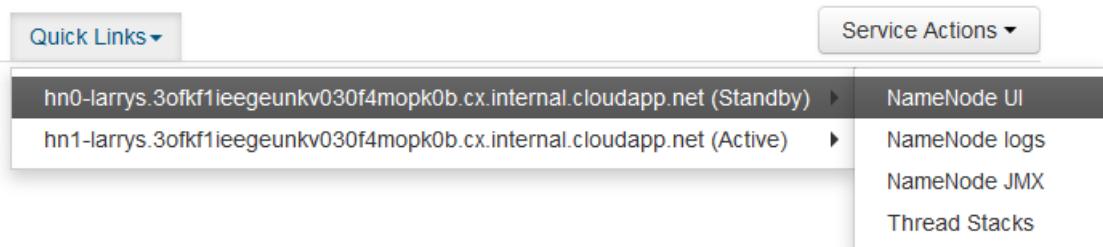
NOTE

When using the <http://headnodehost:8080> address to connect to the cluster, you are connecting through the tunnel. Communication is secured using the SSH tunnel instead of HTTPS. To connect over the internet using HTTPS, use <https://CLUSTERNAME.azurehdinsight.net>, where **CLUSTERNAME** is the name of the cluster.

2. From the Ambari Web UI, select HDFS from the list on the left of the page.



3. When the HDFS service information is displayed, select **Quick Links**. A list of the cluster head nodes appears. Select one of the head nodes, and then select **NameNode UI**.



NOTE

When you select **Quick Links**, you may get a wait indicator. This can happen if you have a slow internet connection. Wait a minute or two for the data to be received from the server, then try the list again.

Some entries in the **Quick Links** menu may be cut off by the right side of the screen. If so, expand the menu using your mouse and use the right arrow key to scroll the screen to the right to see the rest of the menu.

4. A page similar to the following image is displayed:

The screenshot shows the Ambari web interface with the title "Namenode information". The URL in the address bar is "hn0-larrys.3ofkf1ieegeunkv030f4mopk0b.cx.internal.cloudapp.net:8020". The main navigation bar at the top includes links for Overview, Datanodes, Datanode Volume Failures, Snapshot, Startup Progress, and Utilities. The "Overview" tab is currently selected. Below the navigation bar, the word "Hadoop" is displayed in a large, bold font. The main content area is titled "Overview" and contains the text "'hn0-larrys.3ofkf1ieegeunkv030f4mopk0b.cx.internal.cloudapp.net:8020' (active)". A table below provides detailed information about the cluster:

Namespace:	mycluster
Namenode ID:	nn1
Started:	Wed Jul 20 09:15:44 UTC 2016
Version:	2.7.1.2.4.2.0-258, r13debf893a605e8a88df18a7d8d214f571e05289
Compiled:	2016-04-25T05:44Z by jenkins from (HEAD detached at 13debf8)
Cluster ID:	CID-7754bffb-d6ac-4d3d-a7fa-cccd94732c90e
Block Pool ID:	BP-2095951953-10.0.0.20-1468529251487

NOTE

Notice the URL for this page; it should be similar to <http://hn1-CLUSTERNAME.randomcharacters.cx.internal.cloudapp.net:8088/cluster>. This URI is using the internal fully qualified domain name (FQDN) of the node, and is only accessible when using an SSH tunnel.

Next steps

Now that you have learned how to create and use an SSH tunnel, see the following document for other ways to use Ambari:

- [Manage HDInsight clusters by using Ambari](#)

For more information on using SSH with HDInsight, see [Use SSH with HDInsight](#).

Extend HDInsight capabilities by using Azure Virtual Network

8/16/2017 • 14 min to read • [Edit Online](#)

Learn how to use Azure Virtual Networks with HDInsight to enable the following scenarios:

- Restrict access to HDInsight. For example, prevent inbound traffic from the internet.
- Directly access services on HDInsight that aren't exposed over the Internet. For example, directly work with Kafka brokers or use the HBase Java API.
- Directly connect services to HDInsight. For example, use Oozie to import or export data to a SQL Server within your data center.
- Create solutions that involve multiple HDInsight clusters. For example, use Spark or Storm to analyze data stored in Kafka.

Prerequisites

- Azure CLI 2.0: For more information, see [Install and Configure Azure CLI 2.0](#).
- Azure PowerShell: For more information, see [Install and Configure Azure PowerShell](#).

NOTE

The steps in this document require the latest version of the Azure CLI and Azure PowerShell. If you are using an older version, the commands may be different. For best results, use the previous links to install the latest versions.

What is Azure Virtual Network

[Azure Virtual Network](#) allows you to create a secure, persistent network containing the resources you need for your solution.

The following are a list of considerations when using HDInsight in a virtual network:

- **Virtual networks:** Install your cluster on a Resource Manager virtual network.
To access resources in an incompatible virtual network (classic virtual network), join the two networks. For more information on connecting classic and Resource Manager Virtual Networks, see [Connecting classic VNets to new VNets](#).
- **Custom DNS:** If you need to enable name resolution between HDInsight and resources in your local network, you must use a custom DNS server. You do not need a custom DNS server to access resources that are publicly available on the internet.
For more information on using a custom DNS server, see the [Name Resolution for VMs and Role Instances](#) document.
- **Forced tunneling:** HDInsight does not support forced tunneling.
- **Restricted virtual network:** You can install HDInsight into a virtual network that restricts inbound and outbound traffic. You must allow access to specific IP addresses for the Azure region that you are using.

- **Network Security Groups:** If you use Network Security Groups, you must allow unrestricted access to several Azure IPs. For the list of IPs, see the [required IP addresses](#) section.

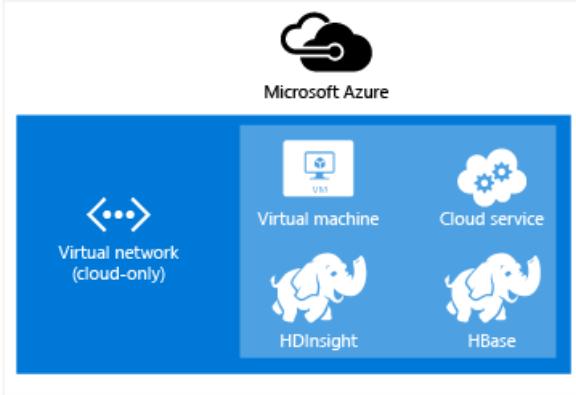
For more information, see the [Network Security Groups](#) section.

- **User-defined routes:** If you use user-defined routes, you must define routes to several Azure IP addresses. For the list of IPs, see the [required IP addresses](#) section.

For more information, see the [User-defined routes](#) section.

- **Network Virtual Appliance:** If you use a virtual appliance firewall, see the [Virtual appliance firewall](#) section for a list of ports that must be allowed through the firewall.

Connect cloud resources together in a private network (cloud-only)

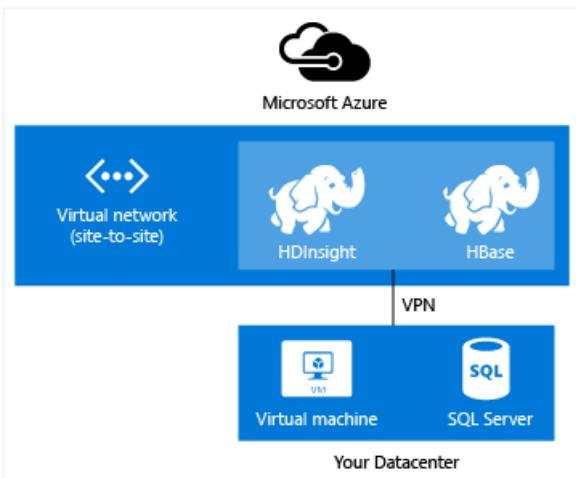


Using Virtual Network to link Azure services with Azure HDInsight enables the following scenarios:

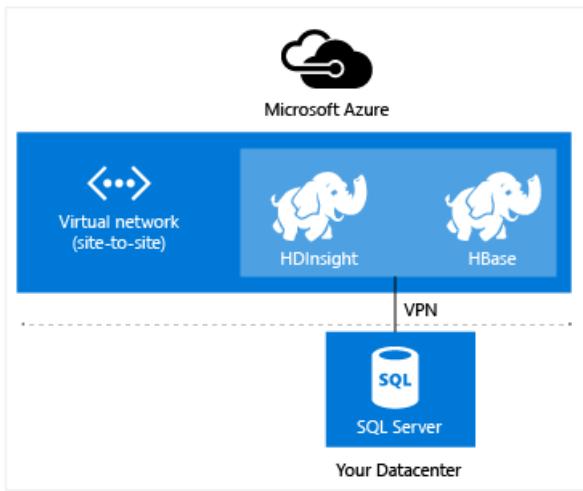
- **Invoking HDInsight services or jobs** from Azure websites or services running in Azure virtual machines.
- **Directly transferring data** between HDInsight and Azure SQL Database, SQL Server, or another data storage solution running on a virtual machine.
- **Combining multiple HDInsight servers** into a single solution. There are several types of HDInsight clusters, which correspond to the workload or technology that the cluster is tuned for. There is no supported method to create a cluster that combines multiple types, such as Storm and HBase on one cluster. Using a virtual network allows multiple clusters to directly communicate with each other.

Connect cloud resources to a local datacenter network

Site-to-site configuration allows you to connect multiple resources in your datacenter to the Azure virtual network. The connection can be made using a hardware VPN device or the Routing and Remote Access service.



Point-to-site configuration allows you to connect a specific resource to the Azure virtual network by using software VPN.



Using Virtual Network to link the cloud and your datacenter enables similar scenarios to the cloud-only configuration. But instead of being limited to working with resources in the cloud, you can also work with resources in your datacenter.

- **Directly transferring data** between HDInsight and your datacenter. An example is using Sqoop to transfer data to or from SQL Server or reading data generated by a line-of-business (LOB) application.
- **Invoking HDInsight services or jobs** from an LOB application. An example is using HBase Java APIs to store and retrieve data from an HDInsight HBase cluster.

For more information on Virtual Network features, benefits, and capabilities, see the [Azure Virtual Network overview](#).

NOTE

Create the Azure Virtual Network before provisioning an HDInsight cluster, then specify the network when creating the cluster. If you plan on using a custom DNS server, it must be added to the virtual network before HDInsight. For more information, see [Virtual Network configuration tasks](#).

Required IP addresses

The HDInsight service is a managed service, and requires access to Azure management services during provisioning and while running. Azure management performs the following services:

- Monitor the health of the cluster
- Initiate failover of cluster resources
- Change the number of nodes in the cluster through scaling operations
- Other management tasks

NOTE

These operations do not require full access to the internet. When restricting internet access, allow inbound access on port 443 for the following IP addresses. This allows Azure to manage HDInsight:

If you restrict access to the virtual network you must allow access to the management IP addresses. The IP addresses that should be allowed are specific to the region that the HDInsight cluster and Virtual Network reside in. Use the following table to find the IP addresses for the region you are using.

NOTE

The appropriate region IP addresses listed below should be set as the *source IP* when defining inbound Network Security Group rules for the subnet that contains HDInsight.

COUNTRY	REGION	ALLOWED IP ADDRESSES	ALLOWED PORT
Brazil	Brazil South	191.235.84.104 191.235.87.113	443
Canada	Canada East	52.229.127.96 52.229.123.172	443
	Canada Central	52.228.37.66 52.228.45.222	443
Germany	Germany Central	51.4.146.68 51.4.146.80	443
	Germany Northeast	51.5.150.132 51.5.144.101	443
India	Central India	52.172.153.209 52.172.152.49	443
Japan	Japan East	13.78.125.90 13.78.89.60	443
	Japan West	40.74.125.69 138.91.29.150	443
United Kingdom	UK West	51.141.13.110 51.141.7.20	443
	UK South	51.140.47.39 51.140.52.16	443
United States	West Central US	52.161.23.15 52.161.10.167	443
	West US 2	52.175.211.210 52.175.222.222	443

If your region is not listed in the table, allow traffic to port **443** on the following IP addresses:

- 168.61.49.99
- 23.99.5.239
- 168.61.48.131
- 138.91.141.162

IMPORTANT

HDInsight doesn't support restricting outbound traffic, only inbound traffic. When defining Network Security Group rules for the subnet that contains HDInsight, **only use inbound rules**.

NOTE

If you use a custom DNS server with your virtual network, you must also allow access from **168.63.129.16**. This is the address of Azure's recursive resolver. For more information, see the [Name resolution for VMs and Role instances](#) document.

Network Security Groups

If you block internet access using Network Security Groups (NSG), you cannot use HDInsight services that are normally exposed through the public gateway for a cluster. These include Ambari and SSH. Instead, you must access services using the internal IP address of the cluster head nodes.

To find the internal IP address of the head nodes, use the scripts in the [Internal IPs and FQDNs](#) section.

Example: Secured virtual network

The following examples demonstrate how to create a Network Security Group that allows inbound traffic on port 443 from the following IP addresses:

- 168.61.49.99
- 23.99.5.239
- 168.61.48.131
- 138.91.141.162

IMPORTANT

These addresses are for regions that do not have specific IP addresses listed. To find the IP addresses for your region, use the information in the [Secured Virtual Networks](#) section.

These steps assume that you have already created a Virtual Network and subnet that you want to install HDInsight into. See [Create a virtual network using the Azure portal](#).

WARNING

Rules are tested against network traffic in order by **priority**. Once a rule matches the test criteria, it is applied and no more rules are tested for that request. If you have a rule that broadly blocks inbound traffic (such as a **deny all** rule), it **must** come after the rules that allow traffic.

For more information on Network Security Group rules, see the [What is a Network Security Group](#) document.

Example: Azure Resource Management template

Use the following Resource Management template from the [Azure QuickStart Templates](#) to create an HDInsight cluster in a VNet with the secure network configurations:

[Deploy a secured Azure VNet and an HDInsight Hadoop cluster within the VNet](#)

Example: Azure PowerShell

```
$vnetName = "Replace with your virtual network name"
$resourceGroupName = "Replace with the resource group the virtual network is in"
$subnetName = "Replace with the name of the subnet that HDInsight will be installed into"
# Get the Virtual Network object
$vnet = Get-AzureRmVirtualNetwork ` 
    -Name $vnetName ` 
    -ResourceGroupName $resourceGroupName
# Get the region the Virtual network is in.
$location = $vnet.Location
```

```

# Get the subnet object
$subnet = $vnet.Subnets | Where-Object Name -eq $subnetName
# Create a Network Security Group.
# And add exemptions for the HDInsight health and management services.
$nsg = New-AzureRmNetworkSecurityGroup `

    -Name "hdisecure" `

    -ResourceGroupName $resourceGroupName `

    -Location $location `

    | Add-AzureRmNetworkSecurityRuleConfig `

        -Name "hdirule1" `

        -Description "HDI health and management address 168.61.49.99" `

        -Protocol "*" `

        -SourcePortRange "*" `

        -DestinationPortRange "443" `

        -SourceAddressPrefix "168.61.49.99" `

        -DestinationAddressPrefix "VirtualNetwork" `

        -Access Allow `

        -Priority 300 `

        -Direction Inbound `

    | Add-AzureRmNetworkSecurityRuleConfig `

        -Name "hdirule2" `

        -Description "HDI health and management 23.99.5.239" `

        -Protocol "*" `

        -SourcePortRange "*" `

        -DestinationPortRange "443" `

        -SourceAddressPrefix "23.99.5.239" `

        -DestinationAddressPrefix "VirtualNetwork" `

        -Access Allow `

        -Priority 301 `

        -Direction Inbound `

    | Add-AzureRmNetworkSecurityRuleConfig `

        -Name "hdirule3" `

        -Description "HDI health and management 168.61.48.131" `

        -Protocol "*" `

        -SourcePortRange "*" `

        -DestinationPortRange "443" `

        -SourceAddressPrefix "168.61.48.131" `

        -DestinationAddressPrefix "VirtualNetwork" `

        -Access Allow `

        -Priority 302 `

        -Direction Inbound `

    | Add-AzureRmNetworkSecurityRuleConfig `

        -Name "hdirule4" `

        -Description "HDI health and management 138.91.141.162" `

        -Protocol "*" `

        -SourcePortRange "*" `

        -DestinationPortRange "443" `

        -SourceAddressPrefix "138.91.141.162" `

        -DestinationAddressPrefix "VirtualNetwork" `

        -Access Allow `

        -Priority 303 `

        -Direction Inbound

# Set the changes to the security group
Set-AzureRmNetworkSecurityGroup -NetworkSecurityGroup $nsg
# Apply the NSG to the subnet
Set-AzureRmVirtualNetworkSubnetConfig `

    -VirtualNetwork $vnet `

    -Name $subnetName `

    -AddressPrefix $subnet.AddressPrefix `

    -NetworkSecurityGroup $nsg

```

Example: Azure CLI

1. Use the following command to create a new network security group named `hdisecure`. Replace **RESOURCEGROUPNAME** with the resource group that contains the Azure Virtual Network. Replace **LOCATION** with the location (region) that the group was created in.

```
az network nsg create -g RESOURCEGROUPNAME -n hdisecure -l LOCATION
```

Once the group has been created, you receive information on the new group.

2. Use the following to add rules to the new network security group that allow inbound communication on port 443 from the Azure HDInsight health and management service. Replace **RESOURCEGROUPNAME** with the name of the resource group that contains the Azure Virtual Network.

```
az network nsg rule create -g RESOURCEGROUPNAME --nsg-name hdisecure -n hdirule1 --protocol "*" --source-port-range "*" --destination-port-range "443" --source-address-prefix "168.61.49.99/24" --destination-address-prefix "VirtualNetwork" --access "Allow" --priority 300 --direction "Inbound"  
az network nsg rule create -g RESOURCEGROUPNAME --nsg-name hdisecure -n hdirule2 --protocol "*" --source-port-range "*" --destination-port-range "443" --source-address-prefix "23.99.5.239/24" --destination-address-prefix "VirtualNetwork" --access "Allow" --priority 301 --direction "Inbound"  
az network nsg rule create -g RESOURCEGROUPNAME --nsg-name hdisecure -n hdirule3 --protocol "*" --source-port-range "*" --destination-port-range "443" --source-address-prefix "168.61.48.131/24" --destination-address-prefix "VirtualNetwork" --access "Allow" --priority 302 --direction "Inbound"  
az network nsg rule create -g RESOURCEGROUPNAME --nsg-name hdisecure -n hdirule4 --protocol "*" --source-port-range "*" --destination-port-range "443" --source-address-prefix "138.91.141.162/24" --destination-address-prefix "VirtualNetwork" --access "Allow" --priority 303 --direction "Inbound"
```

3. Once the rules have been created, use the following to retrieve the unique identifier for this network security group:

```
az network nsg show -g RESOURCEGROUPNAME -n hdisecure --query 'id'
```

This command returns a value similar to the following text:

```
"/subscriptions/SUBSCRIPTIONID/resourceGroups/RESOURCEGROUPNAME/providers/Microsoft.Network/networkSecurityGroups/hdisecure"
```

Use double-quotes around id in the command if you don't get the expected results.

4. Using the following command to apply the network security group to a subnet. Replace the **GUID** and **RESOURCEGROUPNAME** values with the ones returned from the previous step. Replace **VNETNAME** and **SUBNETNAME** with the virtual network name and subnet name that you want to use when creating an HDInsight cluster.

```
az network vnet subnet update -g RESOURCEGROUPNAME --vnet-name VNETNAME --name SUBNETNAME --set networkSecurityGroup.id="/subscriptions/GUID/resourceGroups/RESOURCEGROUPNAME/providers/Microsoft.Network/networkSecurityGroups/hdisecure"
```

Once this command completes, you can successfully install HDInsight into the secured Virtual Network on the subnet used in these steps.

IMPORTANT

Using the preceding steps only open access to the HDInsight health and management service on the Azure cloud. Any other access to the HDInsight cluster from outside the Virtual Network is blocked. To enable access from outside the virtual network, you must add additional Network Security Group rules.

The following example demonstrates how to enable SSH access from the Internet:

```
Add-AzureRmNetworkSecurityRuleConfig -Name "SSH" -Description "SSH" -Protocol "*" -SourcePortRange "*" -DestinationPortRange "22" -SourceAddressPrefix "*" -DestinationAddressPrefix "VirtualNetwork" -Access Allow -Priority 304 -Direction Inbound
```

```
az network nsg rule create -g RESOURCEGROUPNAME --nsg-name hdisecure -n hdirule5 --protocol "*" --source-port-range "*" --destination-port-range "22" --source-address-prefix "*" --destination-address-prefix "VirtualNetwork" --access "Allow" --priority 304 --direction "Inbound"
```

For more information on Network Security Groups, see [Network Security Groups overview](#). For information on controlling routing in an Azure Virtual Network, see [User-defined Routes and IP forwarding](#).

User-defined routes

If you use user-defined routes (UDR) to secure the virtual network, you must add routes for the HDInsight management IP addresses for your region. For a list of IP addresses by region, see the [Required IP addresses](#) section.

The routes to the required IP addresses must set the **Next Hop** type to **Internet**. The following image is an example of how the routes appear in the Azure portal:

NAME	ADDRESS PREFIX	NEXT HOP	
HDIRoute_1	168.61.49.99/32	Internet	...
HDIRoute_2	23.99.5.239/32	Internet	...
HDIRoute_3	168.61.48.131/32	Internet	...
HDIRoute_4	138.91.141.162/32	Internet	...

For more information on user-defined routes, see the [user-defined routes and IP forwarding](#) document.

Virtual appliance firewall

If you are using a virtual appliance firewall to secure the virtual network, you must allow outbound traffic on the following ports:

- 53
- 443
- 1433
- 11000-11999

- 14000-14999

For more information on firewall rules for virtual appliances, see the [virtual appliance scenario](#) document.

Forced tunneling

Forced tunneling is not supported with HDInsight.

Retrieve internal IPs and FQDNs

When connecting to HDInsight using a virtual network, you can connect directly to the nodes in the cluster. Use the following scripts to determine the internal IP address and fully qualified domain names (FQDN) for the nodes in the cluster:

Azure PowerShell

```
$resourceGroupName = Read-Input -Prompt "Enter the resource group that contains the virtual network used with HDInsight"

$clusterNICs = Get-AzureRmNetworkInterface -ResourceGroupName $resourceGroupName | where-object {$_._Name -like "*node*"}

$nodes = @()
foreach($nic in $clusterNICs) {
    $node = new-object System.Object
    $node | add-member -MemberType NoteProperty -name "Type" -value $nic.Name.Split('-')[1]
    $node | add-member -MemberType NoteProperty -name "InternalIP" -value
    $nic.IpConfigurations.PrivateIpAddress
    $node | add-member -MemberType NoteProperty -name "InternalFQDN" -value $nic.DnsSettings.InternalFqdn
    $nodes += $node
}
$nodes | sort-object Type
```

Azure CLI

```
az network nic list --resource-group <resourcegroupname> --output table --query "[?contains(name, 'node')].{NICname:name,InternalIP:ipConfigurations[0].privateIpAddress,InternalFQDN:dnsSettings.internalFqdn}"
```

IMPORTANT

In the Azure CLI 2.0 example, replace `<resourcegroupname>` with the name of the resource group that contains the virtual network.

The scripts work by querying the virtual network interface cards (NICs) for the cluster. The NICs exist in the resource group that contains the virtual network used by HDInsight.

Static Internal Private IP (DIP) addresses

There are times when a static IP address is required for your cluster nodes. For instance, when you require cluster-to-cluster communication across virtual networks, such as a Spark cluster communicating with an HBase cluster, they would need to address each other by IP that you wouldn't want to randomly change. Another example would be [replication between HBase clusters across virtual networks](#), for high availability and minimal downtime.

Configure static IP addresses through the portal

1. From the left-hand menu in the [Azure portal](#), click **Resource Groups**.
2. Select the resource group that contains your HDInsight cluster.

3. In the list of resources, you will see the virtual network that contains your cluster. For example, click **vnet-xxxx**. Once open, you will see a list of network interfaces (NIC) for the cluster node VMs. For instance, your ZooKeeper nodes will have NICs with the following naming convention: **nic-zookeepermode-xxxx**.
4. Click one of the NICs you wish to configure.
5. Click **IP configurations**.
6. Click **ipConfig1** from the list.
7. Click **Static**. This sets the IP assignment from dynamic to static. Make note of the IP address for future reference.

ipConfig1
nic-gateway-1-17e897bb7a804aa39f3c4007f4f49df1

Public IP address settings

Public IP address

Private IP address settings

Virtual network/subnet

[vnet-hbase-westus2/default](#)

Assignment

* IP address

10.1.0.38

8. Repeat these steps to set the static IP address for each of the cluster nodes as required.

Configure static IP addresses using PowerShell

PowerShell can be used to mark a DIP as "static" while creating a new Network Interface, by setting the `PrivateIpAddress` parameter:

```
New-AzureRmNetworkInterface -Name InterfaceName -ResourceGroupName $gname -Location $PrimaryLocation -Subnet $vnet1.Subnets[1] -PublicIpAddress $dcvip `

-PrivateIpAddress "10.1.1.4" -InternalDnsNameLabel $dnsuffix -DnsServer "10.1.1.5", "10.1.1.4"
```

When you specify an explicit IP value as in the example above, Azure will consider that address as static and will never change.

Communicating between virtual networks

Cluster-to-cluster communication across virtual networks, where each cluster resides in a different virtual network in the same region, requires you to peer the networks. Once peered, both virtual networks are able to communicate with each other over the Azure backbone network through their private IP addresses. In other words, the virtual machine resources in both virtual networks, such as your HDInsight cluster nodes, communicate as if they are part of the same network.

Read more about how to [create a virtual network peering](#)

If your virtual networks are in different regions, you will need to [configure a VNet-to-VNet VPN gateway](#)

connection.

Next steps

The following examples demonstrate how to use HDInsight with Azure Virtual Network:

- [HBase clusters in Azure Virtual Network](#)
- [Analyze sensor data with Storm and HBase in HDInsight](#)
- [Provision Hadoop clusters in HDInsight](#)
- [Use Sqoop with Hadoop in HDInsight](#)

To learn more about Azure virtual networks, see the [Azure Virtual Network overview](#).

Scaling - Best Practices

8/16/2017 • 12 min to read • [Edit Online](#)

HDIInsight offers elasticity by giving administrators the option to scale up and scale down the number of Worker Nodes in the clusters. This allows you to shrink a cluster during after hours or on weekends, and grow it during peak business demands.

For example, if you have some batch processing that happens once a week or once a month, the HDIInsight cluster can be scaled up a few minutes prior to that scheduled event so that there is plenty of memory and CPU compute power. You can automate that with the PowerShell cmdlet [Set-AzureRmHDInsightClusterSize](#). Later, after the processing is done, and usage is expected to go down again, the administrator can scale down the HDIInsight cluster to fewer worker nodes.

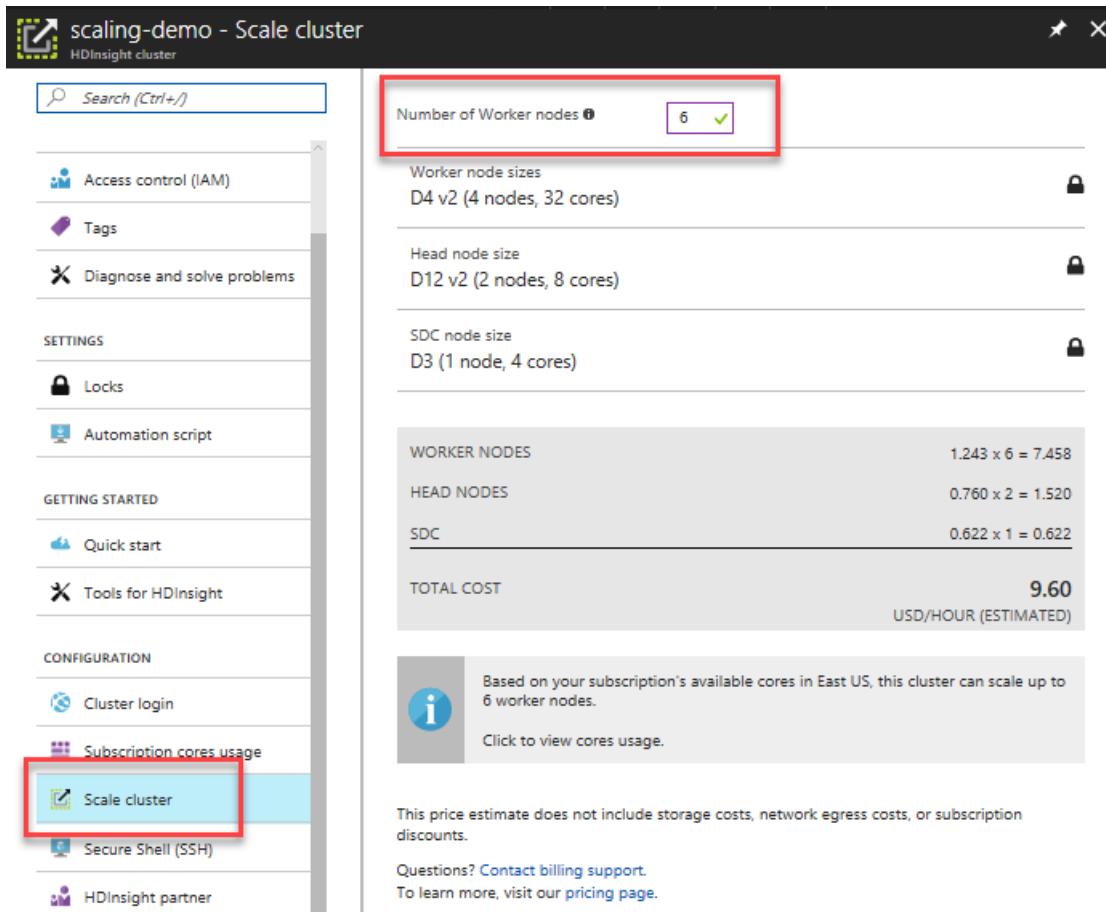
Scaling your cluster through [PowerShell](#):

```
Set-AzureRmHDInsightClusterSize -ClusterName <Cluster Name> -TargetInstanceCount <NewSize>
```

Scaling your cluster through the [Azure CLI](#):

```
azure hdinsight cluster resize [options] <clusterName> <Target Instance Count>
```

To scale your cluster through the [Azure Portal](#), open your HDIInsight cluster blade, select **Scale cluster** on the left-hand menu, then on the Scale cluster blade, type in the number of worker nodes, and click save.



The screenshot shows the Azure Portal interface for scaling a HDIInsight cluster. The main title is "scaling-demo - Scale cluster". On the left, a sidebar lists various options: Access control (IAM), Tags, Diagnose and solve problems, SETTINGS (Locks, Automation script), GETTING STARTED (Quick start, Tools for HDIInsight), and CONFIGURATION (Cluster login, Subscription cores usage). The "Scale cluster" option is highlighted with a red box. The main content area shows the current configuration: Number of Worker nodes set to 6, Worker node sizes D4 v2 (4 nodes, 32 cores), Head node size D12 v2 (2 nodes, 8 cores), SDC node size D3 (1 node, 4 cores). Below this, a summary table provides costs: WORKER NODES (1.243 x 6 = 7.458), HEAD NODES (0.760 x 2 = 1.520), SDC (0.622 x 1 = 0.622), and a TOTAL COST of 9.60 USD/HOUR (ESTIMATED). A note indicates the cluster can scale up to 6 worker nodes. At the bottom, a disclaimer states the price estimate does not include storage costs, network egress costs, or subscription discounts, and provides links for billing support and pricing information.

Using any of these methods, you can scale your HDIInsight cluster up or down within minutes.

Scaling impacts on running jobs

When you **add** nodes to your running HDInsight cluster, any pending or running jobs will not be impacted. In addition, new jobs can be safely submitted while the scaling process is running. If the scaling operations fails for any reason, the failure is gracefully handled, leaving the cluster in a functional state.

However, if you are scaling down your cluster by **removing** nodes, any pending or running jobs will fail when the scaling operation completes. This is due to some of the services restarting during the process.

To address this, you can wait for the jobs to complete before scaling down your cluster, manually terminate them, or simply resubmit the jobs after the scaling operation has concluded.

To see a list of pending and running jobs, you can use the YARN ResourceManager UI, following these steps:

1. Sign in to [Azure portal](#).
2. On the left menu, click **Browse**, click **HDInsight Clusters**, select your cluster.
3. From your HDInsight cluster blade, click **Dashboard** on the top menu to open the Ambari UI. You'll be prompted for your cluster login credentials.
4. Click **YARN** on the list of services on the left-hand menu. On the YARN page, click **Quick Links** hover over the active head node, then click **ResourceManager UI**.



The screenshot shows the Ambari dashboard for a cluster named 'scaling-demo'. The 'YARN' service is selected in the sidebar. The 'Quick Links' dropdown is open, and the 'ResourceManager UI' option is highlighted with a red box.

You may directly access the ResourceManager UI by navigating to:

<https://<HDInsightClusterName>.azurehdinsight.net/yarnui/hn/cluster>

You will see a list of jobs, along with their current State. In the screenshot below, we can see that we have one job currently running:

Cluster Metrics																			
Apps Submitted	Apps Pending	Apps Running	Apps Completed	Containers Running	Memory Used	Memory Total	Memory Reserved	Vcores Used	Vcores Total	Vcores Reserved									
3	0	1	2	1	1.50 GB	96 GB	0 B	1	32	0									
Scheduler Metrics																			
Scheduler Type																			
Capacity Scheduler				Scheduling Resource Type				Minimum Allocation											
<memory:1536, vCores:1>																			
Show 20 entries																			
ID	User	Name	Application Type	Queue	Application Priority	StartTime	FinishTime	State	FinalStatus	Running Containers									
application_1499348398273_0003	sshuser	HIVE-9d3618a4-33a7-4e1d-99cb-0a82adb8f54f	TEZ	default	0	Thu Jul 6 15:13:01 -0400 2017	N/A	RUNNING	UNDEFINED	1									
application_1499348398273_0002	sshuser	word count	MAPREDUCE	default	0	Thu Jul 6 15:11:39 -0400 2017	Thu Jul 6 15:12:23 -0400 2017	FINISHED	SUCCEEDED	N/A									
application_1499348398273_0001	root	HIVE-6448d43a-b36d-44d5-8301-5606d433ae88	TEZ	default	0	Thu Jul 6 09:43:35 -0400 2017	Thu Jul 6 09:43:54 -0400 2017	FINISHED	SUCCEEDED	N/A									
Showing 1 to 3 of 3 entries																			

We can manually kill that running application by executing the following command from the SSH shell:

```
yarn application -kill "application_1499348398273_0003"
```

The syntax for this command is:

```
yarn application -kill <application_id>
```

Rebalancing an HBase cluster

Region servers are automatically balanced within a few minutes after completion of the scaling operation. To manually balance region servers, use the following steps:

1. Connect to the HDInsight cluster using SSH. For more information, see [Use SSH with HDInsight](#).
2. Use the following to start the HBase shell:

```
hbase shell
```

3. Once the HBase shell has loaded, use the following to manually balance the region servers:

```
balancer
```

Scale down implications

As mentioned above, any pending or running jobs will be terminated at the completion of a scaling down operation. However, there are other potential implications to scaling down that may occur.

HDInsight Name Node stays in Safe mode after a Scale Down

scaling-demo - Scale cluster

Number of Worker nodes 1 ✓

Worker node sizes
D4 v2 (4 nodes, 32 cores)

Head node size
D12 v2 (2 nodes, 8 cores)

SDC node size
D3 (1 node, 4 cores)

WORKER NODES $1.243 \times 1 = 1.243$
HEAD NODES $0.760 \times 2 = 1.520$
SDC $0.622 \times 1 = 0.622$

TOTAL COST **3.38**
USD/HOUR (ESTIMATED)

Based on your subscription's available cores in East US, this cluster can scale up to 6 worker nodes.
Click to view cores usage.

This price estimate does not include storage costs, network egress costs, or subscription discounts.
Questions? [Contact billing support](#).
To learn more, visit our [pricing page](#).

If you shrink your cluster down to the minimum of 1 worker node, as shown above, from several nodes, it is

possible for HDFS to become stuck in safe mode when worker nodes are rebooted due to patching, or immediately after the scaling operation.

The primary cause of this is that Hive uses a few `scratchdir` files, and by default, expects 3 replicas of each block, but there is only 1 replica possible if you scale down to the minimum 1 worker node. As a consequence, the files in the `scratchdir` become under replicated. This could cause HDFS to stay in Safe mode when the services are restarted after the scale operation.

When a scale down attempt happens, HDInsight relies upon the Ambari management interfaces to first decommission the extra unwanted worker nodes, which replicates the HDFS blocks to other online worker nodes, and then safely scale the cluster down. HDFS goes into a safe mode during the maintenance window, and is supposed to come out once the scaling is finished. It is at this point that HDFS can become stuck in safe mode.

HDFS is configured with the `dfs.replication` setting of 3. Thus, the blocks of the scratch files are under replicated whenever there are fewer than 3 worker nodes online, because there are not the expected 3 copies of each file block available.

One way to bring HDFS out of safe mode is to execute a command to leave safe mode (ignoring the cause, assuming it is benign). For example, if you know that the only reason safe mode is on is because the temporary files are under replicated (discussed in detail below), then you can safely leave safe mode. This is primarily because the under-replicated files are Hive temporary scratch files.

```
hdfs dfsadmin -D 'fs.default.name=hdfs://mycluster/' -safemode leave
```

After leaving safe mode, you may remove the problematic temp files or wait for Hive to eventually clean them up automatically.

Example errors when safe mode is turned on

ERROR 1

H070 Unable to open Hive session.

```
org.apache.hadoop.ipc.RemoteException(org.apache.hadoop.ipc.RetrifiableException):
org.apache.hadoop.hdfs.server.namenode.SafeModeException: Cannot create directory
/tmp/hive/hive/819c215c-6d87-4311-97c8-4f0b9d2adcf0. Name node is in safe mode. The reported blocks 75
needs additional 12 blocks to reach the threshold 0.9900 of total blocks 87. The number of live datanodes 10 has
reached the minimum number 0. Safe mode will be turned off automatically once the thresholds have been
reached.
```

ERROR 2

```
H100 Unable to submit statement show databases: org.apache.thrift.transport.TTransportException:
org.apache.http.conn.HttpHostConnectException: Connect to hn0-
clusternode.servername.internal.cloudapp.net:10001 [hn0-clusternode.servername. internal.cloudapp.net/1.1.1.1]
failed: Connection refused
```

ERROR 3

```
H020 Could not establish connection to hn0-hdisrv.servername.bx.internal.cloudapp.net:10001:
org.apache.thrift.transport.TTransportException: Could not create http connection to http://hn0-hdisrv.servername.bx.internal.cloudapp.net:10001/. org.apache.http.conn.HttpHostConnectException: Connect to
hn0-hdisrv.servername.bx.internal.cloudapp.net:10001 [hn0-hdisrv.servername.bx.internal.cloudapp.net/10.0.0.28]
failed: Connection refused: org.apache.thrift.transport.TTransportException: Could not create http connection to
http://hn0-hdisrv.servername.bx.internal.cloudapp.net:10001/. org.apache.http.conn.HttpHostConnectException:
Connect to hn0-hdisrv.servername.bx.internal.cloudapp.net:10001 [hn0-
hdisrv.servername.bx.internal.cloudapp.net/10.0.0.28] failed: Connection refused
```

ERROR 4 – from the Hive logs

```
WARN [main]: server.HiveServer2 (HiveServer2.java:startHiveServer2(442)) – Error starting HiveServer2 on attempt 21, will retry in 60 seconds java.lang.RuntimeException: Error applying authorization policy on hive configuration: org.apache.hadoop.ipc.RemoteException(org.apache.hadoop.ipc.RetriableException): org.apache.hadoop.hdfs.server.namenode.SafeModeException: Cannot create directory /tmp/hive/hive/70a42b8a-9437-466e-acbe-da90b1614374. Name node is in safe mode. The reported blocks 0 needs additional 9 blocks to reach the threshold 0.9900 of total blocks 9. The number of live datanodes 10 has reached the minimum number 0. Safe mode will be turned off automatically once the thresholds have been reached. at org.apache.hadoop.hdfs.server.namenode.FSNamesystem.checkNameNodeSafeMode(FSNamesystem.java:1324)
```

You can review the Name Node logs from the `/var/log/hadoop/hdfs/` folder, near the time when the cluster was scaled to see when it entered safe mode. The log files are named after the following pattern:

```
Hadoop-hdfs-namenode-hn0-clustername.*
```

The gist of the above errors is that Hive depends on temporary files in HDFS while running queries. When HDFS enters "Safe Mode", Hive cannot run queries since it cannot write to HDFS. The temp files in HDFS are located in the local drive mounted to the individual worker node VMs, and replicated amongst the worker nodes at 3 replicas, minimum.

The `hive.exec.scratchdir` parameter in Hive is configured within `/etc/hive/conf/hive-site.xml` as shown:

```
<property>
  <name>hive.exec.scratchdir</name>
  <value>hdfs://mycluster/tmp/hive</value>
</property>
```

View the health and state of your HDFS file system

You can view a status report from each name node to see whether nodes are in safe mode. To view the report, SSH into each head node to run the following command:

```
hdfs dfsadmin -D 'fs.default.name=hdfs://mycluster/' -safemode get
```

```
sshuser@hn0-scalin:~$ hdfs dfsadmin -D fs.default.
Safe mode is OFF in hn0-scalin.cxei5na03ppe1cvcujs
Safe mode is OFF in hn1-scalin.cxei5na03ppe1cvcujs
sshuser@hn0-scalin:~$
```

Next, you can view a report that shows the details of the HDFS state:

```
hdfs dfsadmin -D 'fs.default.name=hdfs://mycluster/' -report
```

The above command will result in the following on a healthy cluster where all blocks are replicated to the expected degree:

```

sshuser@hn0-scalin:~$ hdfs dfsadmin -D fs.default.name=hdfs://mycluster/ -report
Configured Capacity: 1686190620672 (1.53 TB)
Present Capacity: 1569026465792 (1.43 TB)
DFS Remaining: 1569026252800 (1.43 TB)
DFS Used: 212992 (208 KB)
DFS Used%: 0.00%
Under replicated blocks: 0
Blocks with corrupt replicas: 0
Missing blocks: 0
Missing blocks (with replication factor 1): 0

-----
Live datanodes (4):

Name: 10.0.0.13:30010 (10.0.0.13)
Hostname: 10.0.0.13
Decommission Status : Normal
Configured Capacity: 421547655168 (392.60 GB)
DFS Used: 57344 (56 KB)
Non DFS Used: 29422284800 (27.40 GB)
DFS Remaining: 392125313024 (365.20 GB)
DFS Used%: 0.00%

```

Note: The `-D` switch is used in these queries, because the default file system in HDInsight is either Azure Storage or Azure Data Lake Store. The switch specifies that the commands need to execute against the local HDFS file system.

HDFS supports the `fsck` command to check for various inconsistencies with various files, for example, missing blocks for a file or under-replicated blocks. To run the `fsck` command against the `scratchdir` (temporary scratch disk) files, execute the following:

```
hdfs fsck -D 'fs.default.name=hdfs://mycluster/' /tmp/hive/hive
```

When executed on a healthy HDFS file system with no under-replicated blocks, you will see an output similar to the following:

```

Connecting to namenode via http://hn0-scalin.cxei5na03ppe1cvcujsey5ynd.bx.internal.cloudapp.net:30070/fsck?
ugi=sshuser&path=%2Ftmp%2Fhive%2Fhive
FSCK started by sshuser (auth:SIMPLE) from /10.0.0.21 for path /tmp/hive/hive at Thu Jul 06 20:07:01 UTC 2017
..Status: HEALTHY
Total size:    53 B
Total dirs:    5
Total files:   2
Total symlinks:          0 (Files currently being written: 2)
Total blocks (validated): 2 (avg. block size 26 B)
Minimally replicated blocks: 2 (100.0 %)
Over-replicated blocks:    0 (0.0 %)
Under-replicated blocks:   0 (0.0 %)
Mis-replicated blocks:    0 (0.0 %)
Default replication factor: 3
Average block replication: 3.0
Corrupt blocks:          0
Missing replicas:         0 (0.0 %)
Number of data-nodes:     4
Number of racks:          1
FSCK ended at Thu Jul 06 20:07:01 UTC 2017 in 3 milliseconds

```

The filesystem under path '/tmp/hive/hive' is **HEALTHY**

In contrast, when the command is executed on an HDFS file system with some under-replicated blocks, the output will be similar to the following:

```
Connecting to namenode via http://hn0-scalin.cxei5na03ppe1cvctjsey53ynd.bx.internal.cloudapp.net:30070/fsck?  
ugi=sshuser&path=%2Ftmp%2Fhive%2Fhive  
FSCK started by sshuser (auth:SIMPLE) from /10.0.0.21 for path /tmp/hive/hive at Thu Jul 06 20:13:58 UTC 2017  
. .  
/tmp/hive/hive/4f3f4253-e6d0-42ac-88bc-90f0ea03602c/inuse.info: Under replicated BP-1867508080-10.0.0.21-  
1499348422953:blk_1073741826_1002. Target Replicas is 3 but found 1 live replica(s), 0 decommissioned  
replica(s) and 0 decommissioning replica(s).  
. .  
/tmp/hive/hive/e7c03964-ff3a-4ee1-aa3c-90637a1f4591/inuse.info: CORRUPT blockpool BP-1867508080-10.0.0.21-  
1499348422953 block blk_1073741825  
  
/tmp/hive/hive/e7c03964-ff3a-4ee1-aa3c-90637a1f4591/inuse.info: MISSING 1 blocks of total size 26 B.Status:  
CORRUPT  
Total size: 53 B  
Total dirs: 5  
Total files: 2  
Total symlinks: 0 (Files currently being written: 2)  
Total blocks (validated): 2 (avg. block size 26 B)  
*****  
UNDER MIN REPL'D BLOCKS: 1 (50.0 %)  
dfs.namenode.replication.min: 1  
CORRUPT FILES: 1  
MISSING BLOCKS: 1  
MISSING SIZE: 26 B  
CORRUPT BLOCKS: 1  
*****  
Minimally replicated blocks: 1 (50.0 %)  
Over-replicated blocks: 0 (0.0 %)  
Under-replicated blocks: 1 (50.0 %)  
Mis-replicated blocks: 0 (0.0 %)  
Default replication factor: 3  
Average block replication: 0.5  
Corrupt blocks: 1  
Missing replicas: 2 (33.333332 %)  
Number of data-nodes: 1  
Number of racks: 1  
FSCK ended at Thu Jul 06 20:13:58 UTC 2017 in 28 milliseconds
```

The filesystem under path '/tmp/hive/hive' is CORRUPT

You can also view the HDFS status in Ambari UI by selecting the **HDFS** service on the left (direct link:

<https://<HDInsightClusterName>.azurehdinsight.net/#/main/services/HDFS/summary>)

HDFS 1

Summary Heatmaps Configs Quick Links Service Actions ▾

Restart Required: 0 Components on 0 Hosts Restart

Summary 1 alert

Active NameNode Started 1 alert Disk Remaining 365.4 GB / 392.6 GB (93.08%)
ZKFailoverController Started No alerts Blocks (total) 2
Standby NameNode Started 1 alert Block Errors 0 corrupt replica / 0 missing / 2 under replicated
ZKFailoverController Started No alerts Total Files + Directories 17
DataNodes 1/1 Started Upgrade Status No pending upgrade
DataNodes Status 4 live / 0 dead / 3 decommissioning Safe Mode Status Not in safe mode
JournalNodes 3/3 JournalNodes Live
NFSGateways 0/0 Started
NameNode Uptime 6.58 hours
NameNode Heap 165.8 MB / 1004.0 MB (16.5% used)
Disk Usage (DFS Used) 208.0 KB / 392.6 GB (0.00%)
Disk Usage (Non DFS Used) 27.2 GB / 392.6 GB (6.92%)

Metrics Actions ▾ Last 1 hour ▾

NameNode GC count: 1 ms
NameNode GC time: 0.5 ms
NN Connection Load: No Data Available for the time period.
NameNode Heap: 1000 MB
NameNode Host Load: 50 %
NameNode RPC: No Data Available for the time period.
Failed disk volumes: 0
Blocks With Corrupted Replicas: 0
Under Replicated Blocks: 2
HDFS Space Utilization: 0 %

You may also see one or more critical errors on the active or standby NameNodes. Click the NameNode link next to the alert to view the NameNode Blocks Health.

Ambari scaling-demo 0 ops 2 alerts Dashboard Services Hosts Alerts Admin admin ▾

NameNode Blocks Health OK (1) CRIT (1)

Back

Configuration

Description: This service-level alert is triggered if the number of corrupt or missing blocks exceeds the configured critical threshold. The threshold values are in blocks.

Check Interval: 5 Minute

Thresholds:

- OK: Total Blocks:[{1}], Missing Blocks:[{0}]
- WARNING: 1 Blocks Total Blocks:[{1}], Missing Blocks:[{0}]
- CRITICAL: 1 Blocks Total Blocks:[{1}], Missing Blocks:[{0}]

Connection Timeout: CRITICAL 5 Seconds

Instances

Service	Host	Status	24-Hour	Response
HDFS	hn0-scalin.cxei5na03ppe1cvvujsey53ynd.bx.internal.cloudapp.net	CRIT for 3 minutes	2	Total Blocks:[2], Missing Blocks:[1]
HDFS	hn1-scalin.cxei5na03ppe1cvvujsey53ynd.bx.internal.cloudapp.net	OK for 7 hours	1	Total Blocks:[2], Missing Blocks:[0]

Show: 10 1 - 2 of 2

To clean up the scratch files, removing the block replication errors, SSH into each head node to run the following command:

```
hadoop fs -rm -r -skipTrash hdfs://mycluster/tmp/hive/
```

Warning: This can break Hive if some jobs are still running.

How to Prevent HDInsight from getting stuck in safe mode due to under-replicated blocks

There are some ways in which you can prevent HDInsight from being left in safe mode. A few options are:

- Stop all Hive jobs before scaling HDInsight down. Alternately, schedule the scale down process to avoid conflicting with running Hive jobs.
- Manually clean up Hive's scratch Tmp directory files in HDFS before scaling down.
- Only scale down HDInsight to 3 worker nodes, minimum. Avoid going as low as 1 worker node.
- Run the command to leave safe mode, if needed.

Let's go through these possibilities in more detail:

Stop all Hive jobs

Stop all Hive jobs before scaling down to 1 worker node. If you know your workload is scheduled, then execute your scale down after Hive work is done.

This will help minimize the number of scratch files in the tmp folder (if any).

Manually clean up Hive's scratch files

If Hive has left behind temporary files before you scale down, then you can manually clean up the tmp files before scaling down to avoid safe mode.

You should stop Hive services to be safe, and at a minimum, make sure all queries and jobs are completed.

You can list the contents of the `hdfs://mycluster/tmp/hive/` directory to see if it contains any files:

```
hadoop fs -ls -R hdfs://mycluster/tmp/hive/hive
```

Sample output when files exist:

```
sshuser@hn0-scalin:~$ hadoop fs -ls -R hdfs://mycluster/tmp/hive/hive
drwx-----  - hive hdfs          0 2017-07-06 13:40 hdfs://mycluster/tmp/hive/hive/4f3f4253-e6d0-42ac-88bc-
90f0ea03602c
drwx-----  - hive hdfs          0 2017-07-06 13:40 hdfs://mycluster/tmp/hive/hive/4f3f4253-e6d0-42ac-88bc-
90f0ea03602c/_tmp_space.db
-rw-r--r--   3 hive hdfs         27 2017-07-06 13:40 hdfs://mycluster/tmp/hive/hive/4f3f4253-e6d0-42ac-88bc-
90f0ea03602c/inuse.info
-rw-r--r--   3 hive hdfs         0 2017-07-06 13:40 hdfs://mycluster/tmp/hive/hive/4f3f4253-e6d0-42ac-88bc-
90f0ea03602c/inuse.lck
drwx-----  - hive hdfs          0 2017-07-06 20:30 hdfs://mycluster/tmp/hive/hive/c108f1c2-453e-400f-ac3e-
e3a9b0d22699
-rw-r--r--   3 hive hdfs         26 2017-07-06 20:30 hdfs://mycluster/tmp/hive/hive/c108f1c2-453e-400f-ac3e-
e3a9b0d22699/inuse.info
```

If you know Hive is done with these files, you can remove them. You should stop Hive services beforehand.

As stated earlier, this can break Hive if some jobs are still running, so refrain from cleaning up the scratch files unless all Hive jobs are done. Be sure that Hive does not have any queries running by looking in the Yarn ResourceManager UI page before doing so, as detailed at the beginning of the article.

Example command line to remove files from HDFS:

```
hadoop fs -rm -r -skipTrash hdfs://mycluster/tmp/hive/
```

Only scale down HDInsight to 3 worker nodes, minimum

If being stuck in safe mode is a persistent problem, and the previous steps are not options, then you may want to avoid the problem by only scaling down to 3 worker nodes, minimum. This may not be optimal, due to cost constraints compared to scaling down to only 1 node. However, with 1 worker node, HDFS cannot guarantee 3 replicas of the data will be made available to the cluster.

Having 3 worker nodes is the safest minimum for HDFS with the default replication factor.

Run the command to leave safe mode, if needed

The final option is to watch for the rare occasion in which HDFS enters safe mode, then execute the leave safe mode command. Once you have proven that the reason HDFS has entered safe mode is due to the Hive files being under-replicated, simply execute the following commands to leave safe mode:

HDInsight on Linux:

```
hdfs dfsadmin -D 'fs.default.name=hdfs://mycluster/' -safemode leave
```

HDInsight on Windows:

```
hdfs dfsadmin -fs hdfs://headnodehost:9000 -safemode leave
```

Next steps

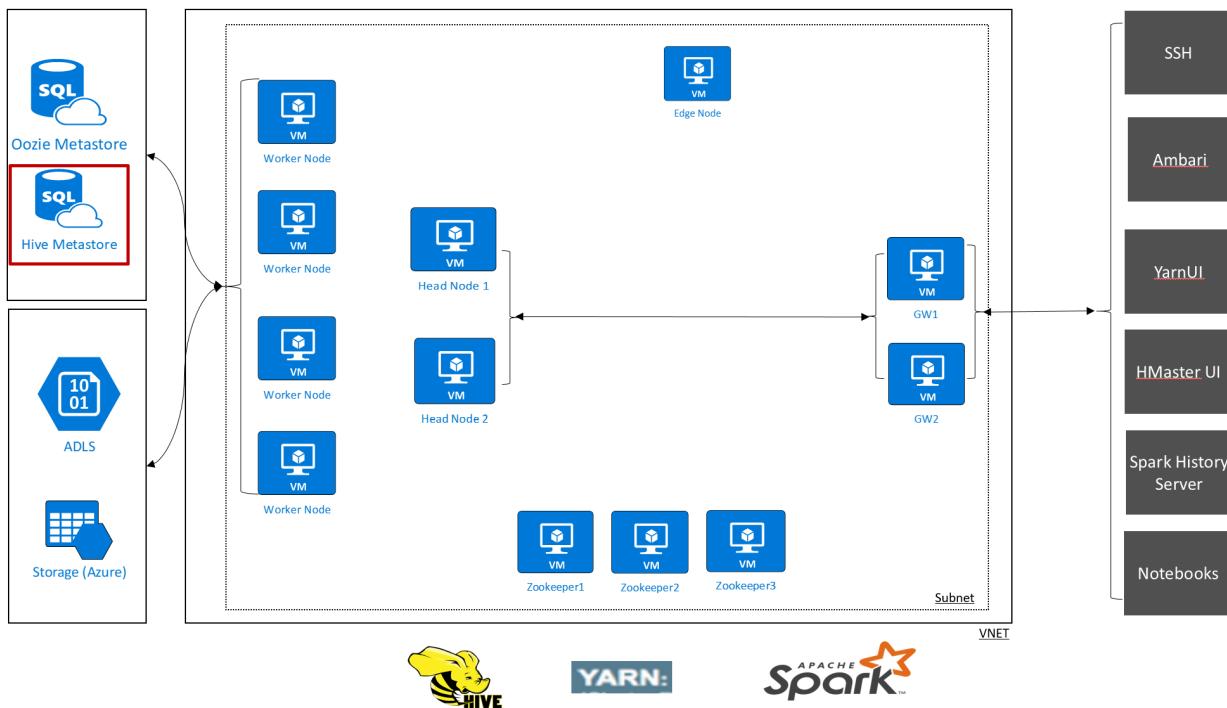
In this article, we covered the various ways in which you can scale the number of nodes in your HDInsight cluster, as well as some potential impacts scaling down can have, particularly while executing jobs. These issues may not occur often, but following the exercise of scaling down when there are no running jobs is a practice that should prevent issues during the process, as well as further down the road. Learn more about the HDInsight cluster architecture, using Ambari, and scaling your cluster by following the links below.

- [HDInsight Architecture](#)
- [Scale clusters](#)
- [Manage HDInsight clusters by using the Ambari Web UI](#)

Using External Metadata Stores

8/16/2017 • 3 min to read • [Edit Online](#)

Hive Metastore is critical part of Hadoop architecture as it acts as a central schema repository which can be used by other access tools like Spark, Interactive Hive (LLAP), Presto, Pig, and many other Big Data engines.



In HDInsight, we use Azure SQL Database as the Hive Metastore.

There are two ways you can setup Metastore for your HDInsight clusters

HDInsight default Metastore – If you don't provide a custom Metastore option, HDInsight will provision Metastore with every cluster type. Here are some key considerations with default Metastore

- No additional cost. HDInsight provisions Metastore with every cluster type without any additional cost to you.
- Default Metastore is tied to the cluster life, when you delete the cluster your Metastore and metadata is also deleted
- You cannot share the Default Metastore with additional clusters.
- The default Metastore use Basic Azure SQL DB which gives you 5 DTU [Database Transaction limit]

This is generally good option for relatively simple workload where you don't have multiple clusters and don't need metadata preserved beyond the life cycle of the cluster.

Custom – HDInsight lets you pick custom Metastore. It's a recommended approach for production clusters due to number reasons such as

- You bring your own Azure SQL Database as Metastore
- As the lifecycle of the Metastore is not tied to a cluster lifecycle, you can create and delete clusters without worrying about the metadata loss- meaning metadata like your Hive schemas will persist even when you delete and re-create the HDInsight cluster.
- Custom Metastore lets you attach multiple clusters and cluster types to the same Metastore. For example, a single Metastore can be shared across Interactive Hive, Hive and Spark clusters in HDInsight
- You pay for the cost of Metastore (Azure SQL DB) according to the performance level you choose.

- You can scale up the metastore as needed.

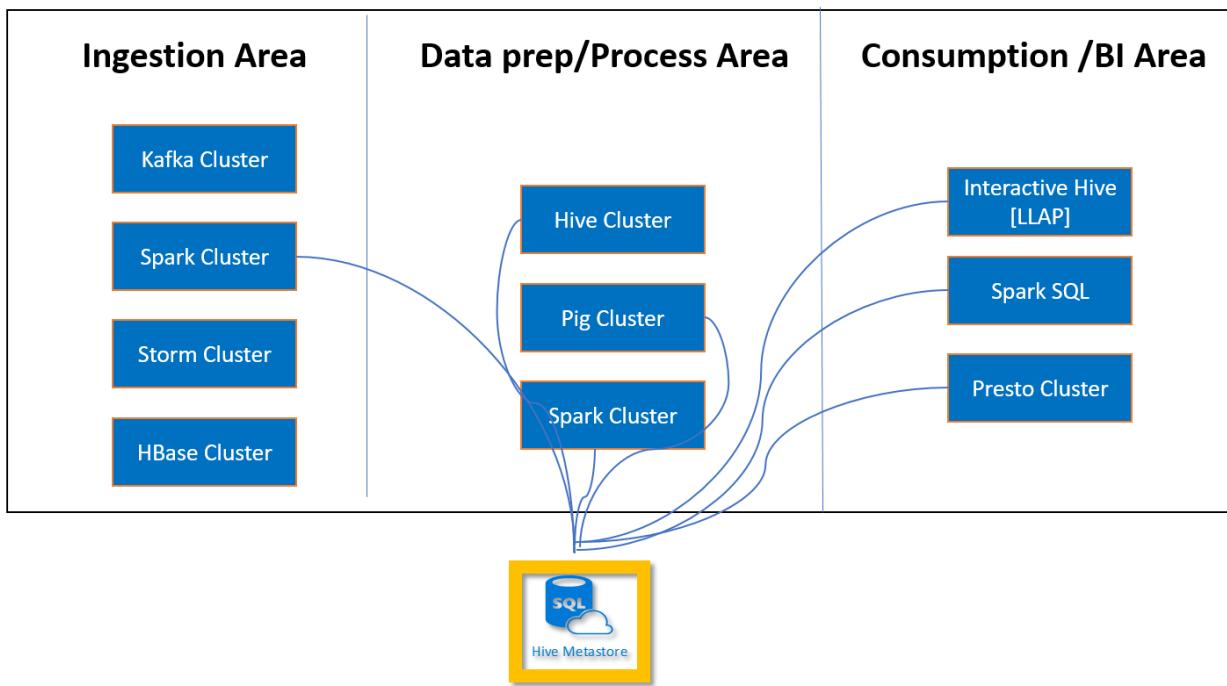


Image – Typical shared custom Metastore scenario in HDInsight

Here are general HDInsight Hive Metastore best practices that you should consider

- Use a custom Metastore whenever possible, this will help you separate Compute (your running cluster) and Metadata (stored in the Metastore).
- Start with S2 tier which will give you 50 DTU and 250 GB of storage, you can always scale the database up in case you see bottlenecks.
- Ensure that the Metastore created for one HDInsight cluster version is not shared across different HDInsight cluster versions. This is due to different Hive versions have different schemas. Example – Hive 1.2 and Hive 2.1 clusters trying to use same Metastore.
- Back-up your custom Metastore periodically.
- Keep your Metastore and HDInsight cluster in same region.
- Monitor your Metastore for performance and availability with Azure SQL Database Monitoring tools (such as in the Azure Portal or by using Azure Log Analytics).

Selecting a custom Metastore during cluster creation

You can point your cluster to a pre-created Azure SQL Database during cluster creation, or you can configure it after the cluster is created. The option is under storage → Metastore settings while creating a new Hadoop, Spark or Interactive Hive cluster from Azure portal

The cluster will use this data source as the primary location for most data access, such as job input and log output.

Primary storage type

- Azure Storage
- Data Lake Store

Selection method

- My subscriptions
- Access key

Select a Storage account

orcbuilder6397

Create new

Default container

sdsdsdcv-2017-03-24t18-54-10-933z

Additional storage accounts

Optional

Data Lake Store access

Optional

Metastore Settings (optional)

Filtered to location and subscription of cluster.

To preserve your metadata outside this cluster, link a SQL database to this account.

Select a SQL database for Hive

ecelmdmora/gswhowmaterialsHDIMetastore

Authenticate SQL Database

Not configured

Select a SQL database for Oozie

Select a database, type to filter.

Next

Select

Additionally, You can add additional clusters to the Custom Metastore for Azure Portal as well as from Ambari configurations (Hive → Advanced)

Hive Metastore

Hive Metastore hosts

hn0-ashish.3naucmplkbeh1guj2w05pyje.cx.internal.cloudapp.net and 1 other

Hive Database

- New MySQL Database
- Existing MySQL / MariaDB Database
- Existing PostgreSQL Database
- Existing Oracle Database
- Existing SQL Anywhere Database

Database Name

hive

Database Username

ashish

Database Password

.....

JDBC Driver Class

com.microsoft.sqlserver.jdbc.SQLServerDriver

Database URL

jdbcsqllserver://ashishmetaserver.database.secure.windows.net;database=LLAP100TBDBASHISH;

Hive Database Type

mssql

As discussed above Hive Metastore is critical component of Hadoop and Spark architecture and picking up right Metastore strategy will certainly help you with right Architecture and user experience.

Oozie Metastore

Apache Oozie is a workflow/coordination system that manages Hadoop jobs. It supports Hadoop jobs for Apache MapReduce, Pig, Hive and others. Oozie uses a Metastore to store details about current and completed workflows. To increase performance when using Oozie, you can use Azure SQL Database as a custom metastore. The metastore can also provide access to Oozie job data after you delete your cluster.

For instructions on creating an Oozie metastore with Azure SQL Database, see [Use Oozie for workflows](#).

See Next

- [Operationalize Data Pipelines with Oozie](#): Learn how to build a data pipeline that uses Hive to summarize CSV flight delay data, stage the prepared data in Azure Storage blobs and then use Sqoop to load the summarized data into Azure SQL Database. A Metastore is created for both Hive and Oozie.

Manage Ambari - Authorize Users to Ambari

8/16/2017 • 5 min to read • [Edit Online](#)

Domain-joined HDInsight clusters provide enterprise-grade capabilities, including Azure Active Directory-based authentication. You can synchronize new users added to Azure AD groups that have been provided access to the cluster, allowing those specific users to perform certain actions. Currently, working with users, groups, and permissions in Ambari is only supported when using a domain-joined HDInsight cluster.

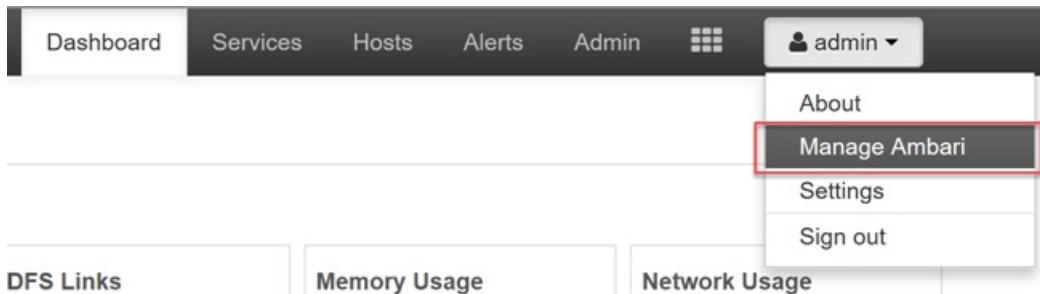
Active Directory users can log on to the cluster nodes using their domain credentials. They can also use their domain credentials to authenticate with other approved endpoints like Hue, Ambari Views, ODBC, JDBC, PowerShell and REST APIs to interact with the cluster.

WARNING

Do not change the password of the Ambari watchdog (`hdinsightwatchdog`) on your Linux-based HDInsight cluster. Changing the password breaks the ability to use script actions or perform scaling operations with your cluster.

If you have not already done so, follow [these instructions](#) to provision a new domain-joined cluster.

Most of the actions in this article will be performed from the **Ambari Management Page** on the [Ambari Web UI](#). To get there, browse to `https://<YOUR CLUSTER NAME>.azurehdinsight.net`, substituting `<YOUR CLUSTER NAME>`. Enter your cluster administrator username and password that you defined when creating your cluster, when prompted. Then, from the Ambari dashboard, select **Manage Ambari** underneath the **admin** menu.



Grant permissions to Hive views

Ambari comes with view instances for, among other things, Hive and Tez. To grant access to one or more Hive view instances, go to the **Ambari Management Page** (following the steps above).

- From the management page, select the **Views** link under the **Views** menu heading on the left.



- On the Views page, expand the **HIVE** row. By default, you will see a Hive view that is auto-created when the Hive services is added to the cluster. **Select** the listed Hive view. Notice that you have the option to create more Hive view instances if desired.

View Name	Instances
➤ CAPACITY-SCHEDULER	1.0.0 (1)
▼ HIVE	2.0.0 (1)
Hive View 2.0	2.0.0

[**+ Create Instance**](#)

1. Scroll toward the bottom of the View page. Under the *Permissions* section, you have two options for granting domain users permissions to the view:

Grant permission to these users

Permissions		
Permission	Grant permission to these users	Grant permission to these groups
Use	<input type="button" value="Add User"/> Add User	<input type="button" value="Add Group"/> Add Group

Grant permission to these groups

Permissions		
Permission	Grant permission to these users	Grant permission to these groups
Use	<input type="button" value="Add User"/> Add User	<input type="button" value="Add Group"/> Add Group

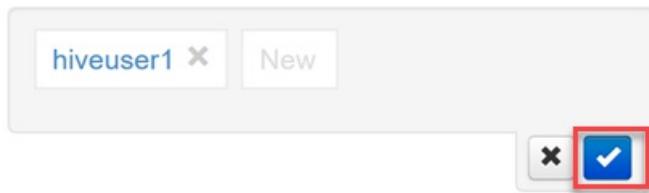
1. To add a user, click the **Add User** button.

- Start typing the user name. As you type, you will see a dropdown list of matching names.



- Select, or finish typing, the user name. There's a box named **New** next to the user's name you added. This is to add additional users, if desired. When finished, **click the blue checkbox** to save your changes.

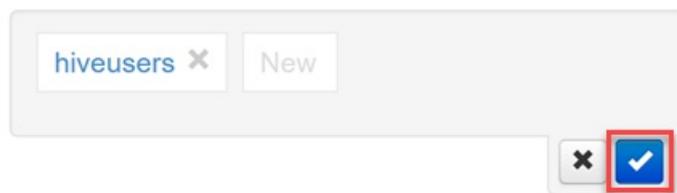
Grant permission to these users



2. To add a group, click the **Add Group** button.

- Start typing the group name. As with adding a user, when you start to type, you will see a dropdown list of matching names.
- The process of selecting a suggested group name, or typing the whole name, as well as adding more than one group, is the same as for adding users. **Click the blue checkbox** to save your changes when done.

Grant permission to these groups



Adding users directly to a view is useful when you want to assign permissions to a user to use that view, but do not necessarily want them to be a member of a group that has other permissions the user should not have. In practice, however, it is preferable to assign permissions to groups, as it reduces the amount of administrative overhead managing user access.

Grant permissions to Tez views

The Tez view instances allow the users to monitor and debug all Tez jobs, submitted by Hive queries and Pig scripts. The default Tez view instance is also auto-generated when the cluster is provisioned.

To assign users and groups to a Tez view instance, expand the **TEZ** row on the Views page, as detailed in the steps for granting permissions to Hive views.

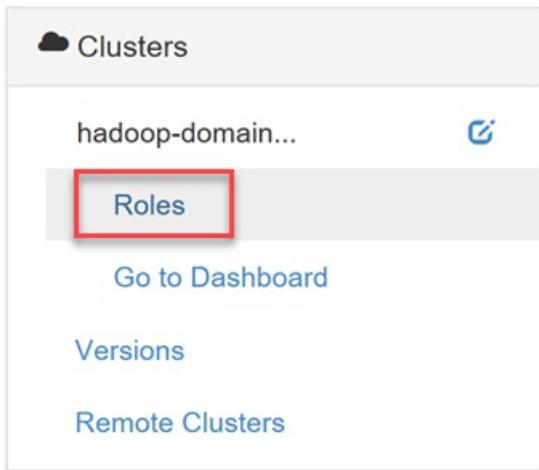
SMART SENSE	1.4.0.2.0.0.10-0 (0)
▼ TEZ	0.7.0.2.6.0.10-20 (1)
Tez View	0.7.0.2.6.0.10-20
+ Create Instance	

Repeat steps 3 - 5 in the previous section to add users or groups.

Assign users to Roles

There are five security roles to which you can assign users and groups: Cluster Administrator, Cluster Operator, Service Administrator, Service Operator, and Cluster User.

To manage roles, go to the **Ambari Management Page**, then select the **Roles** link within the *Clusters* menu group on the left.

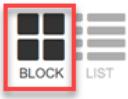


To see the list of permissions provided by each role, click on the blue question mark next to the **Roles** table header on the Roles page.

	Cluster User	Service Operator	Service Administrator	Cluster Operator	Cluster Administrator	Ambari Administrator
Service-level Permissions						
View metrics	✓	✓	✓	✓	✓	✓
View status information	✓	✓	✓	✓	✓	✓
View configurations	✓	✓	✓	✓	✓	✓
Compare configurations	✓	✓	✓	✓	✓	✓
View service-level alerts	✓	✓	✓	✓	✓	✓
Start/Stop/Restart Service	✓	✓	✓	✓	✓	✓
Decommission/recommission	✓	✓	✓	✓	✓	✓
Run service checks	✓	✓	✓	✓	✓	✓
Turn on/off maintenance mode	✓	✓	✓	✓	✓	✓
Perform service-specific tasks	✓	✓	✓	✓	✓	✓
Modify configurations		✓	✓	✓	✓	✓
Manage configuration groups			✓	✓	✓	✓
Move service to another host				✓	✓	✓

On this page, there are two different views you can use to manage roles for users and groups: Block and List.

Block displays each role in its own role, providing the familiar **Assign roles to these users** and **Assign roles to these groups** options.



Roles	Assign roles to these users	Assign roles to these groups
Cluster Administrator	<button>Add User</button>	<button>Add Group</button>
Cluster Operator	<button>Add User</button>	<button>Add Group</button>
Service Administrator	<button>Add User</button>	<button>Add Group</button>
Service Operator	<button>Add User</button>	<button>Add Group</button>
Cluster User	<button>Add User</button>	<button>Add Group</button>

List provides quick editing capabilities in two categories: Users and Groups.

The Users category of the List view displays a list of all users, allowing you to select a role for each user in the dropdown list.

Name	Role
admin	Ambari Administrator
hdinsightwatchdog	Ambari Administrator
hiveadmin	Ambari Administrator
hiveuser1	None
hiveuser2	None
hiveuser3	None
hiveuser4	None
hiveuser5	None
ike	None
joel	None

10 of 11 users showing - [clear filters](#)

10 ▾ Previous 1 2 Next

The Groups category of the List view displays all of the groups, and the role assigned to each group. In our example, the list of groups are synchronized from our Azure AD groups that were specified in the **Access user group** property of the Domain settings during cluster creation. Please see the [Create HDInsight cluster](#) section of the [Configure Domain-joined HDInsight clusters](#) article for reference.

Name	Role
aad dc administrators	Cluster Operator
hiveusers	Cluster User

2 of 2 groups showing - [clear filters](#)

10 ▾ Previous 1 Next

In the screenshot above, the **hiveusers** group is assigned the *Cluster User* role. This is essentially a read-only role that allows the users of that group to view service configurations and cluster metrics, without being able to alter any related settings.

Logging in to Ambari as a user only assigned to views

We have assigned our Azure AD domain user, **hiveuser1**, permissions to Hive and Tez views. When we launch the Ambari Web UI and enter this user's domain credentials (using the Azure AD user name in email format and

password), the user is redirected to the Ambari Views page. From here, they can select each view to which they have access. They cannot visit any other part of the site, such as the dashboard, services, hosts, alerts, and admin pages.

The screenshot shows the 'Your Views' section of the Ambari Views page. It lists three views:

- YARN Queue Manager** (1.0.0)
Manage YARN Capacity Scheduler Queues
- Hive View 2.0** (2.0.0)
This view instance is auto created when the Hive service is added to a cluster.
- Tez View** (0.7.0.2.6.0.10-20)
Monitor and debug all Tez jobs, submitted by Hive queries and Pig scripts (auto-created)

Logging in as a user assigned to the Cluster User role

We have assigned our Azure AD domain user, **hiveuser2**, to the *Cluster User* role. When we log in to the Ambari Web UI with this user, we are able to access the dashboard and all of the menu items. However, not all of the same options available to an admin-level user are available to this one. For instance, **hiveuser2** can view configurations for each of the services, but cannot edit them.

The screenshot shows the Ambari Dashboard. On the left, there is a sidebar with service icons and names. The main area displays various metrics in cards:

- HDFS Disk Usage**: 7%
- DataNodes Live**: 1/1
- HDFS Links**: Active NameNode, Standby NameNode, 1 DataNodes
- Memory Usage**: 9.3 GB
- Network Usage**: 97.6 KB, 48.8 KB
- CPU Usage**: 100%, 50%
- Cluster Load**: 5
- NameNode Heap**: 18%
- NameNode RPC**: 0.57 ms
- NameNode CPU WIO**: n/a
- NameNode Uptime**
- ResourceManager**
- ResourceManager**
- YARN Memory**
- NodeManagers**

Next steps

In this article, we learned how to assign domain users to Views and Roles in Ambari. Please use the links below to find out more about Domain-joined HDInsight clusters, and operations available to domain users.

- Configure Hive policies in Domain-joined HDInsight
- Manage Domain-joined HDInsight clusters
- Synchronize Azure AD users to the cluster
- Use the Hive View with Hadoop in HDInsight

Manage Domain-joined HDInsight clusters (Preview)

8/16/2017 • 3 min to read • [Edit Online](#)

Learn the users and the roles in Domain-joined HDInsight, and how to manage domain-joined HDInsight clusters.

Users of Domain-joined HDInsight clusters

An HDInsight cluster that is not domain-joined has two user accounts that are created during the cluster creation:

- **Ambari admin:** This account is also known as *Hadoop user* or *HTTP user*. This account can be used to log on to Ambari at <https://<clustername>.azurehdinsight.net>. It can also be used to run queries on Ambari views, execute jobs via external tools (i.e. PowerShell, Templeton, Visual Studio), and authenticate with the Hive ODBC driver and BI tools (i.e. Excel, PowerBI, or Tableau).
- **SSH user:** This account can be used with SSH, and execute sudo commands. It has root privileges to the Linux VMs.

A domain-joined HDInsight cluster has three new users in addition to Ambari Admin and SSH user.

- **Ranger admin:** This account is the local Apache Ranger admin account. It is not an active directory domain user. This account can be used to setup policies and make other users admins or delegated admins (so that those users can manage policies). By default, the username is *admin* and the password is the same as the Ambari admin password. The password can be updated from the Settings page in Ranger.
- **Cluster admin domain user:** This account is an active directory domain user designated as the Hadoop cluster admin including Ambari and Ranger. You must provide this user's credentials during cluster creation. This user has the following privileges:
 - Join machines to the domain and place them within the OU that you specify during cluster creation.
 - Create service principals within the OU that you specify during cluster creation.
 - Create reverse DNS entries.

Note the other AD users also have these privileges.

There are some end points within the cluster (for example, Templeton) which are not managed by Ranger, and hence are not secure. These end points are locked down for all users except the cluster admin domain user.

- **Regular:** During cluster creation, you can provide multiple active directory groups. The users in these groups will be synced to Ranger and Ambari. These users are domain users and will have access to only Ranger-managed endpoints (for example, Hiveserver2). All the RBAC policies and auditing will be applicable to these users.

Roles of Domain-joined HDInsight clusters

Domain-joined HDInsight have the following roles:

- Cluster Administrator
- Cluster Operator
- Service Administrator
- Service Operator
- Cluster User

To see the permissions of these roles

1. Open the Ambari Management UI. See [Open the Ambari Management UI](#).
2. From the left menu, click **Roles**.
3. Click the blue question mark to see the permissions:

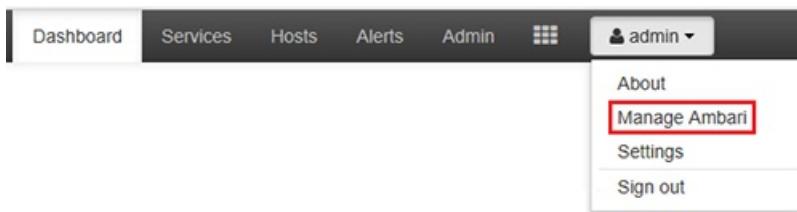
Role Based Access Control

	Cluster User	Service Operator	Service Administrator	Cluster Operator	Cluster Administrator	Ambari Administrator
Service-level Permissions						
View metrics	✓	✓	✓	✓	✓	✓
View status information	✓	✓	✓	✓	✓	✓
View configurations	✓	✓	✓	✓	✓	✓
Compare configurations	✓	✓	✓	✓	✓	✓
View service-level alerts	✓	✓	✓	✓	✓	✓
Start/Stop/Restart Service		✓	✓	✓	✓	✓
Decommission/recommission		✓	✓	✓	✓	✓
Run service checks		✓	✓	✓	✓	✓
Turn on/off maintenance mode		✓	✓	✓	✓	✓
Perform service-specific tasks		✓	✓	✓	✓	✓
Modify configurations			✓	✓	✓	✓
Manage configuration groups			✓	✓	✓	✓
Move service to another host				✓	✓	✓

[Close](#)

Open the Ambari Management UI

1. Sign on to the [Azure portal](#).
2. Open your HDInsight cluster in a blade. See [List and show clusters](#).
3. Click **Dashboard** from the top menu to open Ambari.
4. Log on to Ambari using the cluster administrator domain user name and password.
5. Click the **Admin** dropdown menu from the upper right corner, and then click **Manage Ambari**.



The UI looks like:

The screenshot shows the Apache Ambari Management UI homepage. The top navigation bar includes the Ambari logo, a user icon labeled "admin", and a "Clusters" dropdown. The left sidebar has sections for "Clusters" (with "jgshhd1011" selected), "Views" (with "Views" and "View URLs"), and "User + Group Management" (with "Users" and "Groups"). The main content area features a "Welcome to Apache Ambari" message, a "Operate Your Cluster" section with a cloud icon, "Manage Roles" and "Go to Dashboard" buttons, and two smaller boxes: "Manage Users + Groups" (with a user icon) and "Deploy Views" (with a grid icon).

List the domain users synchronized from your Active Directory

1. Open the Ambari Management UI. See [Open the Ambari Management UI](#).
2. From the left menu, click **Users**. You shall see all the users synced from your Active Directory to the HDInsight cluster.

The screenshot shows the "Users" page in the Ambari Management UI. The top navigation bar includes the Ambari logo, a user icon labeled "admin", and a "Clusters" dropdown. The left sidebar has sections for "Clusters" (with "jgshhd1011" selected), "Views" (with "Views" and "View URLs"), and "User + Group Management" (with "Users" selected). The main content area displays a table of users with columns for "Username", "Type", and "Status". The table shows seven users: "admin" (Local, Active), "hdinsightwatchdog" (Local, Active), "hiveuser1" (LDAP, Active), "hiveuser2" (LDAP, Active), "hiveuser3" (LDAP, Active), and "hiveuser4" (LDAP, Active). A search bar at the top allows filtering by "Any" username, and a "Create Local User" button is available. At the bottom, there are pagination controls showing "7 of 7 users showing" and buttons for "10", "Previous", "1", and "Next".

List the domain groups synchronized from your Active Directory

1. Open the Ambari Management UI. See [Open the Ambari Management UI](#).
2. From the left menu, click **Groups**. You shall see all the groups synced from your Active Directory to the HDInsight cluster.

Group name	Type	Members
Any	All	
hiveusers	LDAP	4 members

1 of 1 groups showing 10 Previous 1 Next

Configure Hive Views permissions

1. Open the Ambari Management UI. See [Open the Ambari Management UI](#).
2. From the left menu, click **Views**.
3. Click **HIVE** to show the details.

View Name	Instances
► CAPACITY-SCHEDULER	1.0.0 (1)
▼ HIVE	1.0.0 (0) , 1.5.0 (1)
Hive View	1.5.0
	This view instance is auto c...
	+ Create Instance
► HUETOAMBARI_MIGRATION	1.0.0 (0)

4. Click the **Hive View** link to configure Hive Views.
5. Scroll down to the **Permissions** section.

Permissions

Permission	Grant permission to these users	Grant permission to these groups
Use	<input type="button" value="Add User"/> <input type="button" value="Edit"/>	<input type="button" value="Add Group"/>

Local Cluster Permissions

Grant **Use** permission for the following **jgshhdi1011** Roles:

Cluster Administrator
 Cluster Operator
 Service Operator
 Service Administrator
 Cluster User

[Check All](#) | [Clear All](#)

6. Click **Add User** or **Add Group**, and then specify the users or groups that can use Hive Views.

Configure users for the roles

To see a list of roles and their permissions, see [Roles of Domain-joined HDInsight clusters](#).

1. Open the Ambari Management UI. See [Open the Ambari Management UI](#).
2. From the left menu, click **Roles**.
3. Click **Add User** or **Add Group** to assign users and groups to different roles.

Next steps

- For configuring a Domain-joined HDInsight cluster, see [Configure Domain-joined HDInsight clusters](#).
- For configuring Hive policies and run Hive queries, see [Configure Hive policies for Domain-joined HDInsight clusters](#).
- For running Hive queries using SSH on Domain-joined HDInsight clusters, see [Use SSH with HDInsight](#).

Add ACLs for Users at the File and Folder Levels

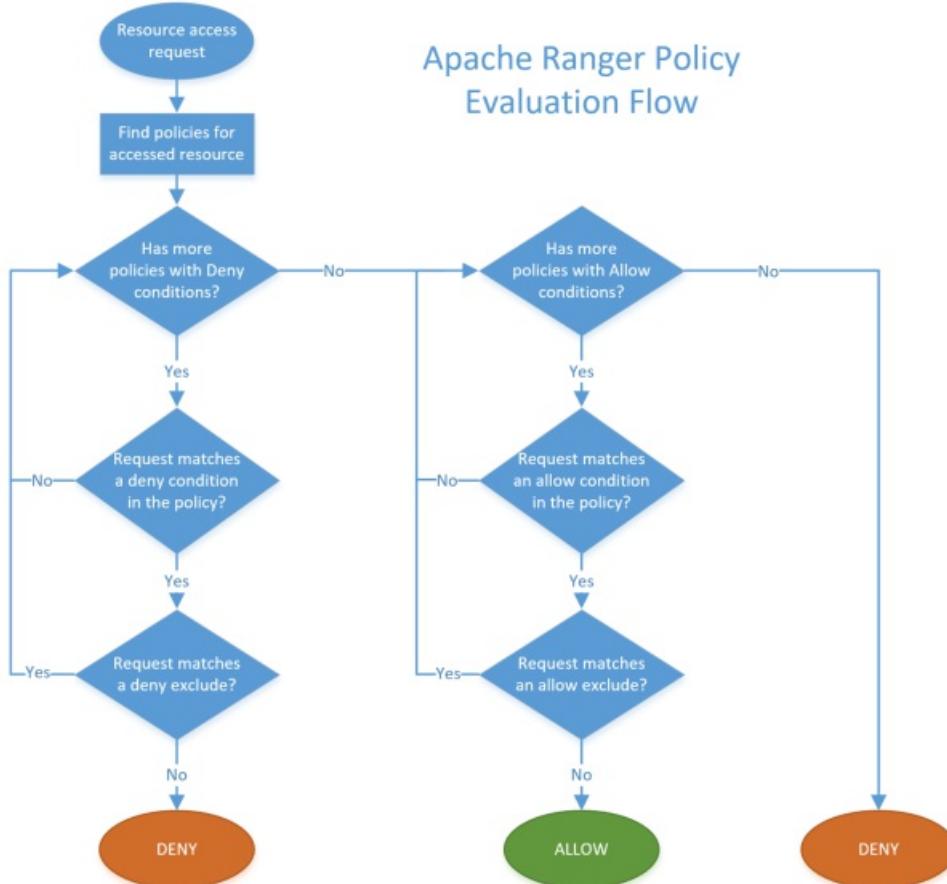
8/16/2017 • 4 min to read • [Edit Online](#)

Domain-joined HDInsight clusters take advantage of strong authentication with Azure Active Directory (Azure AD) users, as well as use role-based access control (RBAC) policies for various services, such as YARN and Hive. If your default data store for your cluster is Azure Storage, or WASB (Windows Azure Storage Blobs), you can enforce file and folder-level permissions as well. Doing so allows you to control access to the cluster's files by assigning your [synchronized Azure AD users and groups](#) through Apache Ranger.

HDInsight domain-joined clusters' Apache Ranger instance comes preconfigured with the **Ranger-WASB** service. This service is a policy management engine that is similar to Ranger-HDFS from a user interface standpoint, but varies in its application and enforcement of Ranger's policy specifications. Namely, if the resource request does not have a matching Ranger policy, the default response is DENY. In other words, Ranger does not hand off permission checking to WASB.

Permission and policy model

Resource access requests are evaluated using the following flow:



DENY rules are evaluated first, followed by ALLOW rules. At the end of matching, a DENY is returned if no policies are matched.

USER variable

When assigning policies for each user to access a corresponding /user/{username} directory, you may use the {USER} variable. For example:

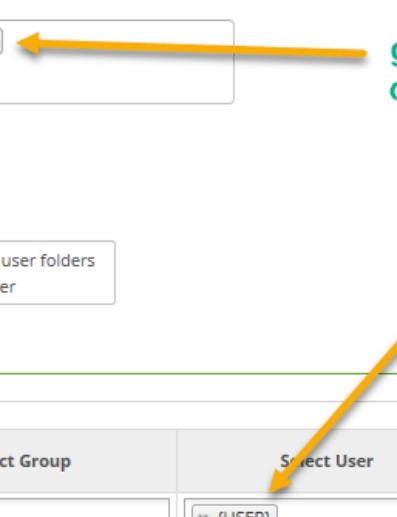
```
resource: path=/app-logs/{USER}, user: {USER}, recursive=true, permissions: all, delegateAdmin=true
```

The above policy grants users access to their own subfolder underneath the `/app-logs/` directory. This is what the policy looks like in the Ranger user interface:

Policy Details :

Policy Type	Access
Policy ID	8
Policy Name *	policy for /app-logs user folders
	enabled
Storage Account *	domainjoined.blob.core.windows.
Storage Account Container *	sol-domain-joined
Relative Path *	/app-logs/{USER}
Audit Logging	YES
Description	policy to enumerate user folders under /app-logs folder

Logged in user is granted access to their own folder named after their username.



Allow Conditions :

Select Group	Select User	Permissions	Delegate Admin	
Select Group	* {USER}	Read Write	<input type="checkbox"/>	x

Policy model examples

This table shows a few examples of how the policy model works, for clarification:

RANGER POLICY	EXISTING FS HIERARCHY	USER-REQUEST	RESULT
/data/finance/, bob, WRITE	/data	bob, Create file /data/finance/mydatafile.txt	ALLOW - Intermediate folder 'finance' is created, because of ancestor check
/data/finance/, bob, WRITE	/data	alice, Create file /data/finance/mydatafile.txt	DENY - No matching policy
/data/finance*, bob, WRITE	/data	bob, Create file /data/finance/mydatafile.txt	ALLOW - Missing <code>/</code> after 'finance' in policy; recursive policy not required, but it will work because of the recursive policy in this case
/data/finance/mydatafile.txt, bob, WRITE	/data	bob, Create file /data/finance/mydatafile.txt	DENY - Ancestor check on '/data' will fail because there is no policy

RANGER POLICY	EXISTING FS HIERARCHY	USER-REQUEST	RESULT
/data/finance/mydatafile.txt, bob, WRITE	/data/finance	bob, Create file /data/finance/mydatafile.txt	DENY - No policy for ancestor check on '/data/finance'

Permissions are required at different levels (at the folder level or at the file level), based on the type of operation. For example, a "read/open" call requires read-access at the file level, whereas a "create" call requires permissions at the ancestor folder level.

Wildcards (*)

When wildcards ("*") are present in the path for a policy, it doesn't only apply to one directory, it also applies to the entire subtree. This is known as "recursion", which some may be familiar with implementing via a `recurse-flag`. In Ranger-WASB, the wildcard implicitly means recursion as well as a partial name match.

Manage file and folder-level permissions with Apache Ranger

If you have not already done so, follow [these instructions](#) to provision a new domain-joined cluster.

Open Ranger by browsing to `https://<YOUR CLUSTER NAME>.azurehdinsight.net/ranger/`, substituting `<YOUR CLUSTER NAME>`. Enter your cluster administrator username and password that you defined when creating your cluster, when prompted.

Once signed in, you should see the Ranger dashboard:

The screenshot shows the Apache Ranger Service Manager interface. At the top, there's a navigation bar with tabs for 'Access Manager', 'Audit', and 'Settings'. On the right, there's a user icon labeled 'admin'. Below the navigation bar, there's a 'Service Manager' header with a 'Import' and 'Export' button. The main area is divided into several sections, each with a '+' button and a delete icon. The sections are: HDFS, HBASE, HIVE, YARN, KNOX, STORM, SOLR, KAFKA, ATLAS, and WASB. The 'WASB' section contains a single policy entry: 'sol-domain-joined_wasd'. This entry is highlighted with a red box around its name.

To view current file and folder permissions for your cluster's associated Azure Storage account, click the **CLUSTERNAME_wasd** link located within the WASB panel.

This screenshot shows the 'WASB' panel within the Ranger interface. It displays a single policy entry: 'sol-domain-joined_wasd'. This entry is highlighted with a red box around its name.

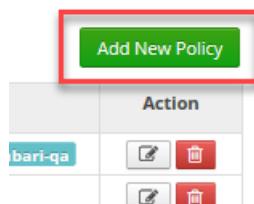
This will take you to your list of policies. As you can see, several policies are added out-of-the-box. Here you can see whether the policy is enabled, if audit logging is configured, what the assigned groups and users are, as well as the policy name and id. In the right-hand Action column are two buttons for each policy: Edit and Delete.

List of Policies : sol-domain-joined_wab

Policy ID	Policy Name	Status	Audit Logging	Groups	Users	Action
1	all - storageaccount, container, relativepath	Enabled	Enabled	--	hiveadmin rangerlookup ambari-qa	
2	policy for /tmp folder	Enabled	Enabled	public	--	
3	recursive policy for /tmp folder	Enabled	Enabled	public	--	
4	policy for accessing /user folder	Enabled	Enabled	public	admin	
5	policy for /user folder	Enabled	Enabled	--	{USER}	
6	recursive policy for /user folder	Enabled	Enabled	--	{USER}	
7	policy for /app-logs folder	Enabled	Enabled	public	--	
8	policy for /app-logs user folders	Enabled	Enabled	--	{USER}	
9	recursive policy for /app-logs user folders	Enabled	Enabled	--	{USER}	
10	policy for /mapreduceestaging folder	Enabled	Enabled	public	admin	
11	policy for /mapreduceestaging user folders	Enabled	Enabled	--	{USER}	
12	recursive policy for /mapreduceestaging u...	Enabled	Enabled	--	{USER}	
--						

Adding a new policy

1. On the top-right section of the WASB policies page, click **Add New Policy**.



2. Enter a descriptive **Policy Name**. Specify the Azure **Storage Account** for your cluster (ACCOUNT_NAME.blob.core.windows.net). Enter the **Storage Account Container** specified when you created your cluster. Type in the **Relative Path** (relative to the cluster) for your folder or file.

Create Policy

Policy Details :

Policy Type	Access
Policy Name *	<input type="text" value="policy for /new-folder folder"/>
Storage Account *	<input type="text" value="ainjoined.blob.core.windows.net"/>
Storage Account Container *	<input type="text" value="sol-domain-joined"/>
Relative Path *	<input type="text" value="x /new-folder"/>
Audit Logging	
Description	<input type="text" value="Policy for /new-folder private access"/>

1. Below the form, specify your **Allow Conditions** for this new resource. You may select groups and/or users, and

their permissions. In this case, we're allowing all users in the `sales` group to have read/write access.

Allow Conditions :

Select Group	Select User	Permissions	Delegate Admin
<input type="text"/> sales	Select User	Read Write	<input type="checkbox"/> x

Exclude from Allow Conditions : show ▾

Allow conditions

1. Click the **Save** button to save your new policy.

Example policy conditions

Given the Apache Ranger policy evaluation flow, as shown in the flowchart at the top of the page, it is possible to use any combination of allow and deny conditions to meet your needs. Here are a few examples:

1. Allow all sales users, but no interns:

Allow Conditions :

Select Group	Select User	Permissions	Delegate Admin
<input type="text"/> sales	Select User	Read Write	<input type="checkbox"/> x

Exclude from Allow Conditions : show ▾

Allow conditions

Deny Conditions :

Select Group	Select User	Permissions	Delegate Admin
<input type="text"/> interns	Select User	Read Write	<input type="checkbox"/> x

1. Allow all sales users, deny all interns, except for an intern whose user name is "hiveuser3", who should have Read access:

Allow Conditions :

Deny Conditions :

Allow conditions

Deny conditions

Exclude deny conditions

Next steps

In this article, we covered the steps necessary to add and edit user and group access policies to Azure Storage (WASB) files and folders. Because domain-joined clusters come preconfigured with many policies out-of-the-box, feel free to look at the details of those policies to learn different ways policies can be expressed in Apache Ranger.

- [Configure Hive policies in Domain-joined HDInsight](#)
- [Manage Domain-joined HDInsight clusters](#)
- [Manage Ambari - Authorize Users to Ambari](#)
- [Synchronized Azure AD users and groups](#)

Sync Other Users from Azure AD to Cluster

8/16/2017 • 4 min to read • [Edit Online](#)

When you [provision a domain-joined HDInsight cluster](#), you are able to take advantage of strong authentication with Azure Active Directory (Azure AD) users, as well as use role-based access control (RBAC) policies. As you add more users and groups to Azure AD, you will need to synchronize those users who you want to have access to your cluster.

Provision a domain-joined HDInsight cluster

If you have not already done so, follow [these instructions](#) to provision a new domain-joined cluster.

Add new Azure AD user(s)

Since each node will need to be updated (as you see fit) with the new unattended upgrade settings, open the Ambari Web UI to view your hosts.

1. From the Azure Portal (portal.azure.com), navigate to the Azure Active Directory directory associated with your domain-joined cluster.
2. Select **All users** from the left-hand menu, then select **New user** from the All users blade.

The screenshot shows the 'Users and groups - All users' blade in the Azure Active Directory portal. The left sidebar has a 'MANAGE' section with links: 'All users' (highlighted with a red box), 'All groups', 'Password reset', 'Company branding', 'User settings', 'Group settings', and 'Device settings'. The main area shows a 'New user' button with a plus sign and 'New guest user' button. Below is a search bar labeled 'Search users'. A list of users is shown with their names and profile icons:

NAME
hiveadmin
hiveuser1
hiveuser2
ike
Sponsored Solliance

1. Complete the new user form. Select groups you created for assigning cluster-based permissions. In our example, we created a group named "HiveUsers", to which we will assign our new users. If you followed the [step-by-step instructions](#) for provisioning your domain-joined cluster, you will have added two groups: "HiveUsers" and "AAD DC Administrators".

The screenshot shows two blades side-by-side. The left blade is titled 'User' and contains fields for 'Name' (hiveuser3) and 'User name' (hiveuser3@). The right blade is titled 'Groups' and shows a list of groups: 'AAD DC Administrators' (blue background), 'HiveUsers' (green background, checked), and 'SSPRSecurityGroupUsers' (orange background).

1. Click **Create**.

Use the Ambari REST API to synchronize users

We will be POSTing to the Ambari REST API, following the instructions found [here](#).

1. Connect to your cluster [with SSH](#). From the overview blade for your cluster in the Azure portal, select the **Secure Shell (SSH)** button.



1. Copy the `ssh` command and paste into your SSH client. Enter the ssh user password when prompted.

2. After authenticating, enter the following command, replacing the `<YOUR PASSWORD>` and `<YOUR CLUSTER NAME>` values:

```
curl -u admin:<YOUR PASSWORD> -sS -H "X-Requested-By: ambari" \
-X POST -d '{"Event": {"specs": [{"principal_type": "groups", "sync_type": "existing"}]} }' \
"https://<YOUR CLUSTER NAME>.azurehdinsight.net/api/v1/ldap_sync_events"
```

You will receive a response similar to:

```
{
  "resources" : [
    {
      "href" : "http://hn0-hadoop.<YOUR DOMAIN>.com:8080/api/v1/ldap_sync_events/1",
      "Event" : {
        "id" : 1
      }
    }
  ]
}
```

To see the status of the synchronization, execute a new `curl` command, using the `href` value returned from the previous command, replacing the `<YOUR PASSWORD>` and `<YOUR DOMAIN>` values:

```
curl -u admin:<YOUR PASSWORD> http://hn0-hadoop.<YOUR DOMAIN>.com:8080/api/v1/ldap_sync_events/1
```

You should see an output with the status similar to:

```
{
  "href" : "http://hn0-hadoop.YOURDOMAIN.com:8080/api/v1/ldap_sync_events/1",
  "Event" : {
    "id" : 1,
    "specs" : [
      {
        "sync_type" : "existing",
        "principal_type" : "groups"
      }
    ],
    "status" : "COMPLETE",
    "status_detail" : "Completed LDAP sync.",
    "summary" : {
      "groups" : {
        "created" : 0,
        "removed" : 0,
        "updated" : 0
      },
      "memberships" : {
        "created" : 1,
        "removed" : 0
      },
      "users" : {
        "created" : 1,
        "removed" : 0,
        "skipped" : 0,
        "updated" : 0
      }
    },
    "sync_time" : {
      "end" : 1497994072182,
      "start" : 1497994071100
    }
  }
}
```

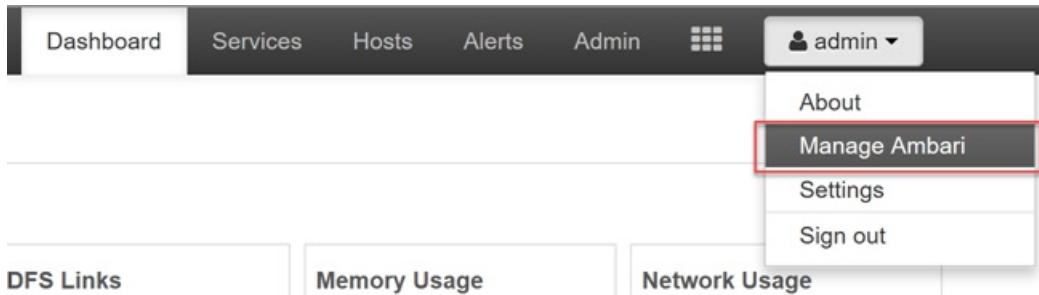
From our result, we can see that the status is **COMPLETE**, and one new user was created, and the user was assigned a membership. This means that the user was assigned to the synchronized LDAP group (HiveUsers in this example) in Ambari, since the user was added to the same group in Azure AD.

Please note, only the Azure AD groups that were specified in the **Access user group** property of the Domain settings during cluster creation will be synchronized using this method. Please see the [Create HDInsight cluster](#) section of the [Configure Domain-joined HDInsight clusters](#) article for reference.

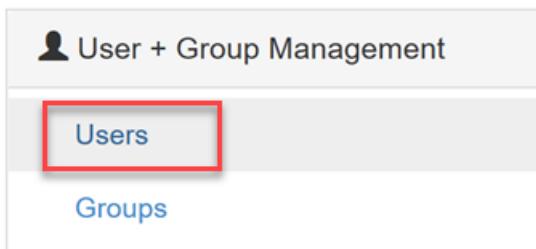
Verify that the new Azure AD user was added

Open the [Ambari Web UI](#) to verify that the new Azure AD user was added. You can access the Ambari Web UI by browsing to `https://<YOUR CLUSTER NAME>.azurehdinsight.net`, substituting `<YOUR CLUSTER NAME>`. Enter your cluster administrator username and password that you defined when creating your cluster, when prompted.

- From the Ambari dashboard, select **Manage Ambari** underneath the **admin** menu.



- Select **Users** underneath the *User + Group Management* menu group on the left-hand side of the page.



- You should see your new user listed within the Users table. Note that the Type is set to LDAP, instead of Local.

A screenshot of the 'Users' table in the Ambari interface. The table has columns for Username, Type, and Status. There are search filters for Username, Type, and Status at the top. The data in the table is as follows:

Username	Type	Status
Any	All	All
admin	Local	Active
hdinsightwatchdog	Local	Active
hiveadmin	LDAP	Active
hiveuser1	LDAP	Active
hiveuser2	LDAP	Active
hiveuser3	LDAP	Active
	LDAP	Active
	LDAP	Active

At the bottom, there is a footer with the text '8 of 8 users showing' and navigation buttons for '10', 'Previous', '1', and 'Next'.

Logging in to Ambari with the new user

When the new user (or any other domain user) logs in to Ambari, they must do so using their Azure AD user name (which is in the form of an email address). Even though Ambari displays users with just the alias, which is the display name of the user in Azure AD, they must log in with the full username. In other words, the same domain

credentials they use to log in to other services.

For example, the new user we added has a user name of *hiveuser3@contoso.com*. In Ambari, this new user shows up simply as **hiveuser3**. However, when we log in to Ambari with this user, we do so with **hiveuser3@contoso.com**.

Next steps

This article focused on the steps required to synchronize users from your Azure AD tenant to your HDInsight cluster. Use the links below to learn more about domain-joined HDInsight and assigning your new users to Hive policies.

- [Configure Hive policies in Domain-joined HDInsight](#)
- [Manage Domain-joined HDInsight clusters](#)
- [Manage Ambari - Authorize Users to Ambari](#)

Create on-demand Hadoop clusters in HDInsight using Azure Data Factory

8/16/2017 • 17 min to read • [Edit Online](#)

Azure Data Factory is a cloud-based data integration service that orchestrates and automates the movement and transformation of data. It can create a HDInsight Hadoop cluster just-in-time to process an input data slice and delete the cluster when the processing is complete. Some of the benefits of using an on-demand HDInsight Hadoop cluster are:

- You only pay for the time a job is running on the HDInsight Hadoop cluster (plus a brief configurable idle time). The billing for HDInsight clusters is pro-rated per minute, whether you are using them or not. When you use an on-demand HDInsight linked service in Data Factory, the clusters are created on-demand, and the clusters are deleted automatically when the jobs are completed. Therefore, you only pay for the job running time and the brief idle time (time-to-live setting).
- You can create a workflow using a Data Factory pipeline. For example, you can have the pipeline to copy data from an on-premises SQL Server to an Azure blob storage, process the data by running a Hive script and a Pig script on an on-demand HDInsight Hadoop cluster. Then, copy the result data to an Azure SQL Data Warehouse for BI applications to consume.
- You can schedule the workflow to run periodically (hourly, daily, weekly, monthly, etc.).

In Azure Data Factory, a data factory can have one or more data pipelines. A data pipeline has one or more activities. There are two types of activities: [Data Movement Activities](#) and [Data Transformation Activities](#). You use data movement activities (currently, only Copy Activity) to move data from a source data store to a destination data store. You use data transformation activities to transform/process data. HDInsight Hive Activity is one of the transformation activities supported by Data Factory. You use the Hive transformation activity in this tutorial.

You can configure a hive activity to use your own HDInsight Hadoop cluster or an on-demand HDInsight Hadoop cluster. In this tutorial, the Hive activity in the data factory pipeline is configured to use an on-demand HDInsight cluster. Therefore, when the activity runs to process a data slice, here is what happens:

1. An HDInsight Hadoop cluster is automatically created for you just-in-time to process the slice.
2. The input data is processed by running a HiveQL script on the cluster.
3. The HDInsight Hadoop cluster is deleted after the processing is complete and the cluster is idle for the configured amount of time (`timeToLive` setting). If the next data slice is available for processing within this `timeToLive` idle time, the same cluster is used to process the slice.

In this tutorial, the HiveQL script associated with the hive activity performs the following actions:

1. Creates an external table that references the raw web log data stored in an Azure Blob Storage container.
2. Partitions the raw data by year and month.
3. Stores the partitioned data in the Azure Blob Storage container.

In this tutorial, the HiveQL script associated with the hive activity creates an external table that references the raw web log data stored in the Azure Blob Storage container. Here are the sample rows for each month in the input file.

```
2014-01-01,02:01:09,SAMPLEWEBSITE,GET,/blogposts/mvc4/step2.png,X-ARR-LOG-ID=2ec4b8ad-3cf0-4442-93ab-837317ece6a1,80,-,1.54.23.196,Mozilla/5.0+(Windows+NT+6.3;+WOW64)+AppleWebKit/537.36+(KHTML,+like+Gecko)+Chrome/31.0.1650.63+Safari/537.36,-,http://weblogs.asp.net/sample/archive/2007/12/09/asp-net-mvc-framework-part-4-handling-form-edit-and-post-scenarios.aspx,\N,200,0,0,53175,871
2014-02-01,02:01:10,SAMPLEWEBSITE,GET,/blogposts/mvc4/step7.png,X-ARR-LOG-ID=d7472a26-431a-4a4d-99eb-c7b4fda2cf4c,80,-,1.54.23.196,Mozilla/5.0+(Windows+NT+6.3;+WOW64)+AppleWebKit/537.36+(KHTML,+like+Gecko)+Chrome/31.0.1650.63+Safari/537.36,-,http://weblogs.asp.net/sample/archive/2007/12/09/asp-net-mvc-framework-part-4-handling-form-edit-and-post-scenarios.aspx,\N,200,0,0,30184,871
2014-03-01,02:01:10,SAMPLEWEBSITE,GET,/blogposts/mvc4/step7.png,X-ARR-LOG-ID=d7472a26-431a-4a4d-99eb-c7b4fda2cf4c,80,-,1.54.23.196,Mozilla/5.0+(Windows+NT+6.3;+WOW64)+AppleWebKit/537.36+(KHTML,+like+Gecko)+Chrome/31.0.1650.63+Safari/537.36,-,http://weblogs.asp.net/sample/archive/2007/12/09/asp-net-mvc-framework-part-4-handling-form-edit-and-post-scenarios.aspx,\N,200,0,0,30184,871
```

The HiveQL script partitions the raw data by year and month. It creates three output folders based on the previous input. Each folder contains a file with entries from each month.

```
adfgetstarted/partitioneddata/year=2014/month=1/000000_0
adfgetstarted/partitioneddata/year=2014/month=2/000000_0
adfgetstarted/partitioneddata/year=2014/month=3/000000_0
```

For a list of Data Factory data transformation activities in addition to Hive activity, see [Transform and analyze using Azure Data Factory](#).

Prerequisites

Before you begin the instructions in this article, you must have the following items:

- [Azure subscription](#)
- Azure PowerShell

IMPORTANT

Azure PowerShell support for managing HDInsight resources using Azure Service Manager is **deprecated**, and was removed on January 1, 2017. The steps in this document use the new HDInsight cmdlets that work with Azure Resource Manager.

Please follow the steps in [Install and configure Azure PowerShell](#) to install the latest version of Azure PowerShell. If you have scripts that need to be modified to use the new cmdlets that work with Azure Resource Manager, see [Migrating to Azure Resource Manager-based development tools for HDInsight clusters](#) for more information.

Prepare storage account

You can use up to three storage accounts in this scenario:

- default storage account for the HDInsight cluster
- storage account for the input data
- storage account for the output data

To simplify the tutorial, you use one storage account to serve the three purposes. The Azure PowerShell sample script found in this section performs the following tasks:

1. Log in to Azure.
2. Create an Azure resource group.
3. Create an Azure Storage account.
4. Create a Blob container in the storage account
5. Copy the following two files to the Blob container:
 - Input data file: <https://hditutorialdata.blob.core.windows.net/adfhiveactivity/inputdata/input.log>

- HiveQL script:

<https://hditutorialdata.blob.core.windows.net/adfhiveactivity/script/partitionweblogs.hql>

Both files are stored in a public Blob container.

To prepare the storage and copy the files using Azure PowerShell:

IMPORTANT

Specify names for the Azure resource group and the Azure storage account that will be created by the script (`$resourceGroupName` and `$storageAccountName` variables). Write down **resource group name, storage account name**, and **storage account key** output by the script. You need them in the next section.

```
$resourceGroupName = "<Azure Resource Group Name>"  
$storageAccountName = "<Azure Storage Account Name>"  
$location = "East US 2"  
  
$sourceStorageAccountName = "hditutorialdata"  
$sourceContainerName = "adfhiveactivity"  
  
$destStorageAccountName = $storageAccountName  
$destContainerName = "adfgtstarted" # don't change this value.  
  
#####  
# Connect to Azure  
#####  
#region - Connect to Azure subscription  
Write-Host "`nConnecting to your Azure subscription ..." -ForegroundColor Green  
Login-AzureRmAccount  
Get-AzureRmContext  
#endregion  
  
#####  
# Create a resource group, storage, and container  
#####  
  
#region - create Azure resources  
Write-Host "`nCreating resource group, storage account and blob container ..." -ForegroundColor Green  
  
New-AzureRmResourceGroup -Name $resourceGroupName -Location $location  
New-AzureRmStorageAccount `  
    -ResourceGroupName $resourceGroupName `  
    -Name $destStorageAccountName `  
    -type Standard_LRS `  
    -Location $location  
  
$destStorageAccountKey = (Get-AzureRmStorageAccountKey `  
    -ResourceGroupName $resourceGroupName `  
    -Name $destStorageAccountName)[0].Value  
  
$sourceContext = New-AzureStorageContext `  
    -StorageAccountName $sourceStorageAccountName `  
    -Anonymous  
$destContext = New-AzureStorageContext `  
    -StorageAccountName $destStorageAccountName `  
    -StorageAccountKey $destStorageAccountKey  
  
New-AzureStorageContainer -Name $destContainerName -Context $destContext  
#endregion  
  
#####  
# Copy files  
#####  
#region - copy files  
Write-Host "`nCopying files ..." -ForegroundColor Green
```

```

$blobs = Get-AzureStorageBlob ` 
    -Context $sourceContext ` 
    -Container $sourceContainerName 

$blobs|Start-AzureStorageBlobCopy ` 
    -DestContext $destContext ` 
    -DestContainer $destContainerName 

Write-Host "`nCopied files ..." -ForegroundColor Green 
Get-AzureStorageBlob -Context $destContext -Container $destContainerName 
#endregion 

Write-host "`nYou will use the following values:" -ForegroundColor Green 
write-host "`nResource group name: $resourceGroupName" 
Write-host "Storage Account Name: $destStorageAccountName" 
write-host "Storage Account Key: $destStorageAccountKey" 

Write-host "`nScript completed" -ForegroundColor Green

```

If you need help with the PowerShell script, see [Using the Azure PowerShell with Azure Storage](#). If you like to use Azure CLI instead, see the [Appendix](#) section for the Azure CLI script.

To examine the storage account and the contents

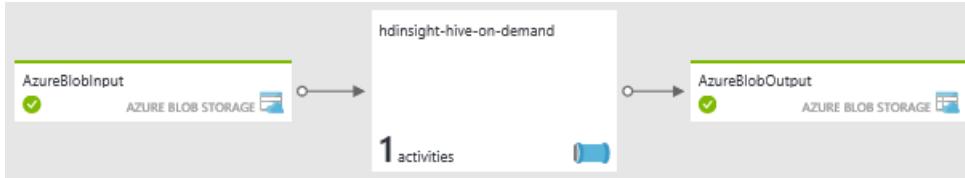
1. Sign on to the [Azure portal](#).
2. Click **Resource groups** on the left pane.
3. Click the resource group name you created in your PowerShell script. Use the filter if you have too many resource groups listed.
4. On the **Resources** tile, you should have one resource listed unless you share the resource group with other projects. That resource is the storage account with the name you specified earlier. Click the storage account name.
5. Click the **Blobs** tiles.
6. Click the **adfgetstarted** container. You see two folders: **inputdata** and **script**.
7. Open the folder and check the files in the folders. The inputdata folder contains the input.log file with input data, and the script folder contains the HiveQL script file.

Create a data factory using a Resource Manager template

With the storage account, the input data, and the HiveQL script prepared, you are ready to create an Azure data factory. There are several methods for creating data factory. In this tutorial, you create a data factory by deploying an Azure Resource Manager (ARM) template using the Azure portal. You can also deploy an ARM template by using [Azure CLI](#) and [Azure PowerShell](#). For other data factory creation methods, see [Tutorial: Build your first data factory](#).

1. Click the following image to sign in to Azure and open the Resource Manager template in the Azure portal. The template is located at <https://hditutorialdata.blob.core.windows.net/adfhiveactivity/data-factory-hdinsight-on-demand.json>. See the [Data Factory entities in the template](#) section for detailed information about entities defined in the template.
- [Deploy to Azure >](https://hditutorialdata.blob.core.windows.net/adfhiveactivity/data-factory-hdinsight-on-demand.json)
2. Select **Use existing** option for the **Resource group** setting, and select the name of the resource group you created in the previous step (using PowerShell script).
 3. Make sure you select the same **location** as your provisioned storage account (from the previous step), if available.
 4. Enter a name for the data factory (**Data Factory Name**). This name must be globally unique.

5. Enter the **storage account name** and **storage account key** you wrote down in the previous step.
6. Select **I agree to the terms and conditions** stated above after reading through **terms and conditions**.
7. Select **Pin to dashboard** option.
8. Click **Purchase**. You will see a tile on the Dashboard called **Deploying Template deployment**. Wait until the **Resource group** blade for your resource group opens. You can also click the tile titled after your resource group name to open the resource group blade.
9. Click the tile to open the resource group if the resource group blade is not already open. Now you should see the data factory resource listed in addition to the storage account resource.
10. Click the name of your data factory (value you specified for the **Data Factory Name** parameter).
11. In the Data Factory blade, click the **Diagram** tile. The diagram shows one activity with an input dataset, and an output dataset:



The names are defined in the Resource Manager template.

12. Double-click **AzureBlobOutput**.
13. On the **Recent updated slices**, you will see one slice. If the status is **In progress**, wait until it is changed to **Ready**. It usually takes about **20 minutes** to create an HDInsight cluster.

Check the data factory output

1. Use the same procedure in the last session to check the containers of the storage account. There are two new containers in addition to **adfgetstarted**:
 - A container with name that follows the pattern: `adf<yourdatafactoryname>-linkedservicename-datetimestamp`. This container is the default container for the HDInsight cluster.
 - **adfjobs**: This is the container for the Azure Data Factory (ADF) job logs.

The data factory output is stored in **adfgetstarted** as you configured in the Resource Manager template.

2. Click **adfgetstarted**.
3. Click **partitioneddata**. You will see a **year=2014** folder because all the web logs are dated in the year 2014.

NAME	MODIFIED	BLOB TYPE	SIZE
year=2014	3/14/2016 4:34 PM	Block blob	0 B

If you drill down the list, you will see three folders for January, February, and March, with a log for each month.

NAME	MODIFIED	BLOB TYPE	SIZE
month=1			-
month=2			-
month=3			-
month=1	3/14/2016 4:35 PM	Block blob	0 B
month=2	3/14/2016 4:35 PM	Block blob	0 B
month=3	3/14/2016 4:34 PM	Block blob	0 B

Data Factory entities in the template

Here is what the top-level Resource Manager template for a data factory looks like:

```
{
  "contentVersion": "1.0.0.0",
  "$schema": "http://schema.management.azure.com/schemas/2015-01-01/deploymentTemplate.json#",
  "parameters": { ... },
  "variables": { ... },
  "resources": [
    {
      "name": "[parameters('dataFactoryName')]",
      "apiVersion": "[variables('apiVersion')]",
      "type": "Microsoft.DataFactory/datafactories",
      "location": "westus",
      "resources": [
        { ... },
        { ... },
        { ... },
        { ... }
      ]
    }
  ]
}
```

Define data factory

You define a data factory in the Resource Manager template as shown in the following sample:

```
"resources": [
{
  "name": "[parameters('dataFactoryName')]",
  "apiVersion": "[variables('apiVersion')]",
  "type": "Microsoft.DataFactory/datafactories",
  "location": "westus",
}
```

The `dataFactoryName` is the name of the data factory you specify when you deploy the template. Data factory is currently only supported in the East US, West US, and North Europe regions.

Defining entities within the data factory

The following Data Factory entities are defined in the JSON template:

- [Azure Storage linked service](#)

- [HDInsight on-demand linked service](#)
- [Azure blob input dataset](#)
- [Azure blob output dataset](#)
- [Data pipeline with a copy activity](#)

Azure Storage linked service

The Azure Storage linked service links your Azure storage account to the data factory. In this tutorial, the same storage account is used as the default HDInsight storage account, input data storage, and output data storage. Therefore, you define only one Azure Storage linked service. In the linked service definition, you specify the name and key of your Azure storage account. See [Azure Storage linked service](#) for details about JSON properties used to define an Azure Storage linked service.

```
{
  "name": "[variables('storageLinkedServiceName')]",
  "type": "linkedservices",
  "dependsOn": [ "[concat('Microsoft.DataFactory/dataFactories/', parameters('dataFactoryName'))]" ],
  "apiVersion": "[variables('apiVersion')]",
  "properties": {
    "type": "AzureStorage",
    "typeProperties": {
      "connectionString": "
[concat('DefaultEndpointsProtocol=https;AccountName=',parameters('storageAccountName'),';AccountKey=',parameters('storageAccountKey'))]"
    }
  }
}
```

The **ConnectionString** uses the storageAccountName and storageAccountKey parameters. You specify values for these parameters while deploying the template.

HDInsight on-demand linked service

In the on-demand HDInsight linked service definition, you specify values for configuration parameters that are used by the Data Factory service to create a HDInsight Hadoop cluster at runtime. See the [Compute linked services](#) article for details about JSON properties used to define an HDInsight on-demand linked service.

```
{
  "type": "linkedservices",
  "name": "[variables('hdInsightOnDemandLinkedServiceName')]",
  "dependsOn": [
    "[concat('Microsoft.DataFactory/dataFactories/', parameters('dataFactoryName'))]",
    "[concat('Microsoft.DataFactory/dataFactories/', parameters('dataFactoryName'), '/linkedservices/',
variables('storageLinkedServiceName'))]"
  ],
  "apiVersion": "[variables('apiVersion')]",
  "properties": {
    "type": "HDInsightOnDemand",
    "typeProperties": {
      "osType": "linux",
      "version": "3.5",
      "clusterSize": 1,
      "sshUserName": "myuser",
      "sshPassword": "MyPassword!",
      "timeToLive": "00:30:00",
      "linkedServiceName": "[variables('storageLinkedServiceName')]"
    }
  }
}
```

Note the following points:

- The Data Factory creates a **Linux-based** HDInsight cluster for you.
- The HDInsight Hadoop cluster is created in the same region as the storage account.
- Notice the *timeToLive* setting. The data factory deletes the cluster automatically after the cluster is idle for 30 minutes.
- The HDInsight cluster creates a **default container** in the blob storage you specified in the JSON (**linkedServiceName**). HDInsight does not delete this container when the cluster is deleted. This behavior is by design. With an on-demand HDInsight linked service, an HDInsight cluster is created every time a slice needs to be processed unless there is an existing live cluster (**timeToLive**) and is deleted when the processing is done.

See [On-demand HDInsight Linked Service](#) for details.

IMPORTANT

As more slices are processed, you see many containers in your Azure blob storage. If you do not need them for troubleshooting the jobs, you may want to delete them to reduce the storage cost. The names of these containers follow a pattern: "adf-linkedservicename-datetimestamp". Use tools such as [Microsoft Storage Explorer](#) to delete containers in your Azure blob storage.

Azure blob input dataset

In the input dataset definition, you specify the names of blob container, folder, and file that contains the input data.

See [Azure Blob dataset properties](#) for details about JSON properties used to define an Azure Blob dataset.

```
{
  "type": "datasets",
  "name": "[variables('blobInputDatasetName')]",
  "dependsOn": [
    "[concat('Microsoft.DataFactory/dataFactories/', parameters('dataFactoryName'))]",
    "[concat('Microsoft.DataFactory/dataFactories/', parameters('dataFactoryName'), '/linkedServices/',
    variables('storageLinkedServiceName'))]"
  ],
  "apiVersion": "[variables('apiVersion')]",
  "properties": {
    "type": "AzureBlob",
    "linkedServiceName": "[variables('storageLinkedServiceName')]",
    "typeProperties": {
      "fileName": "input.log",
      "folderPath": "adfgetstarted/inputdata",
      "format": {
        "type": "TextFormat",
        "columnDelimiter": ","
      }
    },
    "availability": {
      "frequency": "Month",
      "interval": 1
    },
    "external": true,
    "policy": {}
  }
}
```

Notice the following specific settings in the JSON definition:

```
"fileName": "input.log",
"folderPath": "adfgetstarted/inputdata",
```

Azure Blob output dataset

In the output dataset definition, you specify the names of blob container and folder that holds the output data. See [Azure Blob dataset properties](#) for details about JSON properties used to define an Azure Blob dataset.

```
{  
    "type": "datasets",  
    "name": "[variables('blobOutputDatasetName')]",  
    "dependsOn": [  
        "[concat('Microsoft.DataFactory/dataFactories/', parameters('dataFactoryName'))]",  
        "[concat('Microsoft.DataFactory/dataFactories/', parameters('dataFactoryName'), '/linkedServices/',  
variables('storageLinkedServiceName'))]"  
    ],  
    "apiVersion": "[variables('apiVersion')]",  
    "properties": {  
        "type": "AzureBlob",  
        "linkedServiceName": "[variables('storageLinkedServiceName')]",  
        "typeProperties": {  
            "folderPath": "adfgetstarted/partitioneddata",  
            "format": {  
                "type": "TextFormat",  
                "columnDelimiter": ","  
            }  
        },  
        "availability": {  
            "frequency": "Month",  
            "interval": 1,  
            "style": "EndOfInterval"  
        }  
    }  
}
```

The `FolderPath` specifies the path to the folder that holds the output data:

```
"FolderPath": "adfgetstarted/partitioneddata",
```

The [dataset availability](#) setting is as follows:

```
"availability": {  
    "frequency": "Month",  
    "interval": 1,  
    "style": "EndOfInterval"  
},
```

In Azure Data Factory, the output dataset availability drives the pipeline. In this example, the slice is produced monthly on the last day of month (`EndOfInterval`). For more information, see [Data Factory Scheduling and Execution](#).

Data pipeline

You define a pipeline that transforms data by running a Hive script on an on-demand Azure HDInsight cluster. See [Pipeline JSON](#) for descriptions of JSON elements used to define a pipeline in this example.

```
{
    "type": "datapipelines",
    "name": "[parameters('dataFactoryName')]",
    "dependsOn": [
        "[concat('Microsoft.DataFactory/dataFactories/', parameters('dataFactoryName'))]",
        "[concat('Microsoft.DataFactory/dataFactories/', parameters('dataFactoryName'), '/linkedServices/',
variables('storageLinkedServiceName'))]",
        "[concat('Microsoft.DataFactory/dataFactories/', parameters('dataFactoryName'), '/linkedServices/',
variables('hdInsightOnDemandLinkedServiceName'))]",
        "[concat('Microsoft.DataFactory/dataFactories/', parameters('dataFactoryName'), '/datasets/',
variables('blobInputDatasetName'))]",
        "[concat('Microsoft.DataFactory/dataFactories/', parameters('dataFactoryName'), '/datasets/',
variables('blobOutputDatasetName'))]"
    ],
    "apiVersion": "[variables('apiVersion')]",
    "properties": {
        "description": "Azure Data Factory pipeline with an Hadoop Hive activity",
        "activities": [
            {
                "type": "HDInsightHive",
                "typeProperties": {
                    "scriptPath": "adfgetstarted/script/partitionweblogs.hql",
                    "scriptLinkedService": "[variables('storageLinkedServiceName')]",
                    "defines": {
                        "inputtable": "[concat('wasb://adfgetstarted@', parameters('storageAccountName'),
'.blob.core.windows.net/inputdata')]",
                        "partitionedtable": "[concat('wasb://adfgetstarted@',
parameters('storageAccountName'), '.blob.core.windows.net/partitioneddata')]"
                    }
                },
                "inputs": [
                    {
                        "name": "AzureBlobInput"
                    }
                ],
                "outputs": [
                    {
                        "name": "AzureBlobOutput"
                    }
                ],
                "policy": {
                    "concurrency": 1,
                    "retry": 3
                },
                "name": "RunSampleHiveActivity",
                "linkedServiceName": "HDInsightOnDemandLinkedService"
            }
        ],
        "start": "2017-01-01T00:00:00Z",
        "end": "2017-01-31T00:00:00Z",
        "isPaused": false
    }
}
}
```

The pipeline contains one activity, HDInsightHive activity. As both start and end dates are in January 2017, data for only one month (a slice) is processed. Both *start* and *end* of the activity have a past date, so the Data Factory processes data for the month immediately. If the end is a future date, the data factory creates another slice when the time comes. For more information, see [Data Factory Scheduling and Execution](#).

Clean up the tutorial

Delete the blob containers created by the on-demand HDInsight cluster

With the on-demand HDInsight linked service, an HDInsight cluster is created every time a slice needs to be

processed unless there is an existing live cluster (`timeToLive`); and the cluster is deleted when the processing is done. For each cluster, Azure Data Factory creates a blob container in the Azure Blob Storage used as the default storage account for the cluster. Even though the HDInsight cluster is deleted, the default blob storage container and the associated storage account are not deleted. This behavior is by design. As more slices are processed, you see many containers in your Azure blob storage. If you do not need them for troubleshooting the jobs, you may want to delete them to reduce the storage cost. The names of these containers follow a pattern:

```
adfyourdatafactoryname-linkedservicename-datetimestamp .
```

Delete the **adfjobs** and **adfyourdatafactoryname-linkedservicename-datetimestamp** folders. The **adfjobs** container contains job logs from Azure Data Factory.

Delete the resource group

Azure Resource Manager is used to deploy, manage, and monitor your solution as a group. Deleting a resource group deletes all the components inside the group.

1. Sign on to the [Azure portal](#).
2. Click **Resource groups** on the left pane.
3. Click the resource group name you created in your PowerShell script. Use the filter if you have too many resource groups listed. It opens the resource group in a new blade.
4. On the **Resources** tile, you shall have the default storage account and the data factory listed unless you share the resource group with other projects.
5. Click **Delete** on the top of the blade. Doing so deletes the storage account and the data stored in the storage account, as well as the data factory.
6. Enter the resource group name to confirm deletion, and then click **Delete**.

In case you don't want to delete the storage account when you delete the resource group, consider the following architecture by separating the business data from the default storage account. In this case, you have one resource group for the storage account with the business data, and the other resource group for the default storage account for the HDInsight linked service and the data factory. When you delete the second resource group, it does not impact the business data storage account. To do so:

- Add the following to the top-level resource group along with the Microsoft.DataFactory/datafactories resource in your Resource Manager template. It creates a storage account:

```
{
  "name": "[parameters('defaultStorageAccountName')]",
  "type": "Microsoft.Storage/storageAccounts",
  "location": "[parameters('location')]",
  "apiVersion": "[variables('defaultApiVersion')]",
  "dependsOn": [ ],
  "tags": {

  },
  "properties": {
    "accountType": "Standard_LRS"
  }
},
```

- Add a new linked service that points to the new storage account:

```
{
  "dependsOn": [ "[concat('Microsoft.DataFactory/dataFactories/', parameters('dataFactoryName'))]", ],
  "type": "linkedservices",
  "name": "[variables('defaultStorageLinkedServiceName')]",
  "apiVersion": "[variables('apiVersion')]",
  "properties": {
    "type": "AzureStorage",
    "typeProperties": {
      "connectionString": "[concat('DefaultEndpointsProtocol=https;AccountName=',parameters('defaultStorageAccountName'),';AccountKey=',listKeys(resourceId('Microsoft.Storage/storageAccounts', variables('defaultStorageAccountName')), variables('defaultApiVersion')).key1)]"
    }
  }
},
},
```

- Configure the HDInsight on-demand linked service with an additional `dependsOn` and an `additionalLinkedServiceNames`:

```
{
  "dependsOn": [
    "[concat('Microsoft.DataFactory/dataFactories/', parameters('dataFactoryName'))]",
    "[concat('Microsoft.DataFactory/dataFactories/', parameters('dataFactoryName'),
    '/linkedservices/', variables('defaultStorageLinkedServiceName'))]",
    "[concat('Microsoft.DataFactory/dataFactories/', parameters('dataFactoryName'),
    '/linkedservices/', variables('storageLinkedServiceName'))]"
  ],
  "type": "linkedservices",
  "name": "[variables('hdInsightOnDemandLinkedServiceName')]",
  "apiVersion": "[variables('apiVersion')]",
  "properties": {
    "type": "HDInsightOnDemand",
    "typeProperties": {
      "osType": "linux",
      "version": "3.2",
      "clusterSize": 1,
      "sshUserName": "myuser",
      "sshPassword": "MyPassword!",
      "timeToLive": "00:30:00",
      "linkedServiceName": "[variables('storageLinkedServiceName')]",
      "additionalLinkedServiceNames": "[variables('defaultStorageLinkedServiceName')]"
    }
  }
},
```

Next steps

In this article, you have learned how to use Azure Data Factory to create on-demand HDInsight cluster to process Hive jobs. To read more:

- [Hadoop tutorial: Get started using Linux-based Hadoop in HDInsight](#)
- [Create Linux-based Hadoop clusters in HDInsight](#)
- [HDInsight documentation](#)
- [Data factory documentation](#)

Appendix

Azure CLI script

You can use Azure CLI instead of using Azure PowerShell to do the tutorial. To use Azure CLI, first install Azure CLI as per the following instructions:

[! IMPORTANT] Azure CLI support for managing HDInsight resources using Azure Service Manager (ASM) is **deprecated**, and was removed on January 1, 2017. The steps in this document use the new Azure CLI commands that work with Azure Resource Manager.

Please follow the steps in [Install and configure Azure CLI](#) to install the latest version of the Azure CLI. If you have scripts that need to be modified to use the new commands that work with Azure Resource Manager, see [Migrating to Azure Resource Manager-based development tools for HDInsight clusters](#) for more information.

Use Azure CLI to prepare the storage and copy the files

Run these commands one-by-one:

```
azure login

azure config mode arm

azure group create --name "<Azure Resource Group Name>" --location "East US 2"

azure storage account create --resource-group "<Azure Resource Group Name>" --location "East US 2" --sku-name
"LRS" "<Azure Storage Account Name>"

azure storage account keys list --resource-group "<Azure Resource Group Name>" "<Azure Storage Account Name>"
azure storage container create "adfgtgetstarted" --account-name "<Azure Storage AccountName>" --account-key "
<Azure Storage Account Key>"

azure storage blob copy start
"https://hditutorialdata.blob.core.windows.net/adfhiveactivity/inputdata/input.log" --dest-account-name "
<Azure Storage Account Name>" --dest-account-key "<Azure Storage Account Key>" --dest-container
"adfgtgetstarted"
azure storage blob copy start
"https://hditutorialdata.blob.core.windows.net/adfhiveactivity/script/partitionweblogs.hql" --dest-account-
name "<Azure Storage Account Name>" --dest-account-key "<Azure Storage Account Key>" --dest-container
"adfgtgetstarted"
```

The container name is *adfgtgetstarted*. Keep it as it is. Otherwise you need to update the Resource Manager template. If you need help with this CLI script, see [Using the Azure CLI with Azure Storage](#).

Key Scenarios to Monitor

8/16/2017 • 5 min to read • [Edit Online](#)

Overview

Monitoring the health of your HDInsight cluster, from resource utilization, to storage bottlenecks, to whether your jobs are successfully running, is an important process for organizations of any size. Even when your code is well-written and works well under the most stringent testing conditions, issues may surface only when that code is executed under production-scale loads or when working with real world data. Other factors could be introduced, such as human error; someone executed the wrong command or made an incorrect configuration change.

How much effort you put into making cluster monitoring a routine practice depends on how much downtime you can afford, or whether the time to execute tasks is important to you. Outages or degraded performance can be very costly to organizations, in the form of revenue loss or not meeting SLAs, or any number of consequences that can have a negative impact on your business.

This article will go over some of the key scenarios to consider monitoring, linking to more information about steps to conduct the monitoring, where appropriate.

Is the cluster well utilized, or under heavy/light load?

It is important for a Hadoop cluster to be well-balanced. This means ensuring that utilization is even across the nodes of your cluster. This means balancing the usage of RAM, CPU, and disk so that your processing tasks are not constrained by any one of these cluster resources.

To get a high-level look at the nodes of your cluster and their load, log in to the [Ambari Web UI](#), then select the **Hosts** tab. Your hosts are listed by their fully-qualified domain names. The Actions menu gives you options to perform actions on one or more of these hosts. Next to the host names are colored dots that indicates the operating status of each:

COLOR	DESCRIPTION
Red	At least one master component on that host is down. Hover to see a tooltip that lists affected components.
Orange	At least one slave component on that host is down. Hover to see a tooltip that lists affected components.
Yellow	Ambari Server has not received a heartbeat from that host for more than 3 minutes.
Green	Normal running state.

You'll also see columns showing the number of cores and amount of RAM for each host. Plus, there are cores that show disk usage and the load average.

Click on any of the host names for a detailed look at components running on that host, as well as its metrics with a selectable timeline of CPU usage, load, disk usage, memory usage, network usage, and processes.



Are the YARN queues properly configured?

As a distributed platform, Hadoop has various services running across the platform. These services need to be coordinated, cluster resources allocated, and access to a common data set needs to be managed. YARN (Yet Another Resource Negotiator) helps perform these tasks. The fundamental idea of YARN is to split up two major responsibilities of the JobTracker (resource management and job scheduling/monitoring) into separate daemons: a *pure scheduler*. This means that it is strictly limited to arbitrating available resources in the system among the competing applications. While doing this, it makes sure that all resources are in use all the time, optimizing against various constraints like SLAs, capacity guarantees, etc. The ApplicationMaster negotiates resources from the ResourceManager, and works with the NodeManager(s) to execute and monitor the containers and their resource consumption.

[Read Manage HDInsight clusters by using the Ambari Web UI](#) for details on setting alerts and viewing metrics.

When there are multiple tenants who share a large cluster, there can be a lot of competition for that cluster's resources. The CapacityScheduler is a pluggable scheduler which helps facilitate resource sharing through the concept of *queues*. The CapacityScheduler also supports *hierarchical queues* to ensure that resources are shared amongst the sub-queues of an organization before other queues are granted to use free resources. This provides affinity for sharing free resources among the applications of a given organization.

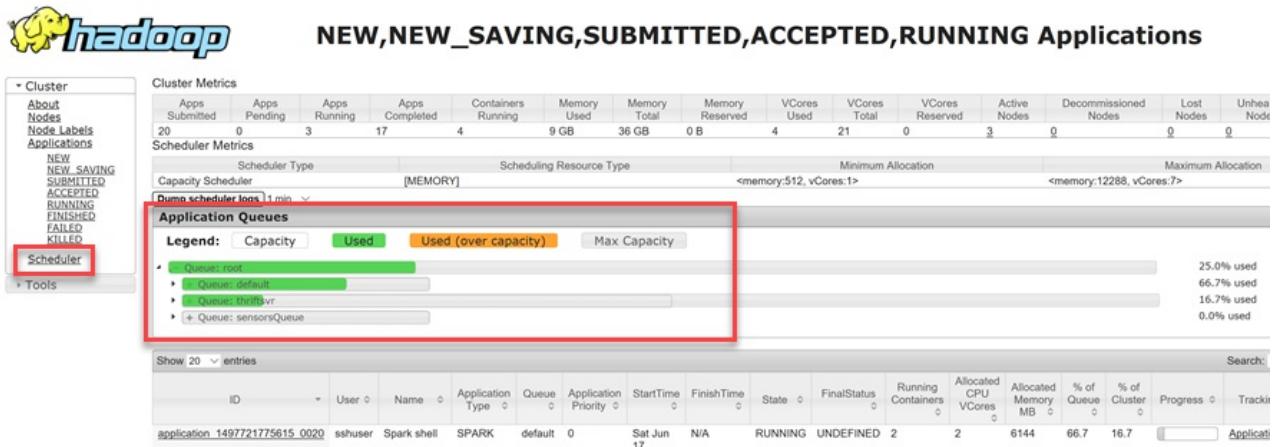
YARN allows us to allocate resources to these queues, and it shows you whether all of your available resources have been assigned. To view information about your queues, login to the Ambari Web UI, then select **YARN Queue Manager** from the top menu.

On the YARN Queue Manager page, you will see a list of your queues on the left, along with the percentage of capacity assigned to each.

For a more detailed look at your queues, from the Ambari dashboard, select the **YARN** service from the list on the left. Then under the **Quick Links** dropdown menu, select **ResourceManager UI** underneath your active node.

On the ResourceManager UI, select **Scheduler** from the left-hand menu. You will see a list of your queues underneath *Application Queues*. Here you can see the capacity used for each of your queues, how well the jobs are

distributed between them, and whether any are resource-constrained.



Is there any storage throttling happening?

There are times when your cluster's performance bottleneck happens at the storage level. These types of bottlenecks are most often due to blocking IOs (Input/Output operations), which happens when your running tasks sends more IO than the storage service can handle. This blocking causes a queue of IO requests waiting to be processed while other IOs are processed. In these cases, the blocks are due to throttling, which is not a physical limit, but a limit imposed by the storage service according to published SLAs. These limits ensure that no single client or tenant can use the service at the expense of others. The SLA in question is the number of IOPS (IO per second), which you can find [here](#) for Azure Storage.

If you are using Azure Storage, detailed information on monitoring storage-related issues, including throttling, can be found here:

- [Monitor, diagnose, and troubleshoot Microsoft Azure Storage](#).

If your cluster's backing store is Azure Data Lake Store (ADLS), your throttling is most likely due to limits of bandwidth provided by ADLS. Throttling in this case could be identified by observing throttling errors in task logs.

For ADLS, view the throttling section for the appropriate service in the articles below:

- [Performance tuning guidance for Hive on HDInsight and Azure Data Lake Store](#)
- [Performance tuning guidance for MapReduce on HDInsight and Azure Data Lake Store](#)
- [Performance tuning guidance for Storm on HDInsight and Azure Data Lake Store](#)

Next steps

This article introduced a few key scenarios to watch out for when monitoring your HDInsight cluster. Visit the links below to find out more about troubleshooting and monitoring your clusters:

- [Analyze HDInsight logs](#)
- [Debug apps with YARN logs](#)
- [Enable heap dumps for Hadoop services on Linux-based HDInsight](#)

Manage HDInsight clusters by using the Ambari Web UI

8/16/2017 • 7 min to read • [Edit Online](#)

Apache Ambari simplifies the management and monitoring of a Hadoop cluster by providing an easy to use web UI and REST API. Ambari is included on Linux-based HDInsight clusters, and is used to monitor the cluster and make configuration changes.

In this document, you learn how to use the Ambari Web UI with an HDInsight cluster.

What is Ambari?

[Apache Ambari](#) simplifies Hadoop management by providing an easy-to-use web UI. You can use Ambari to create, manage, and monitor Hadoop clusters. Developers can integrate these capabilities into their applications by using the [Ambari REST APIs](#).

The Ambari Web UI is provided by default with HDInsight clusters that use the Linux operating system.

IMPORTANT

Linux is the only operating system used on HDInsight version 3.4 or greater. For more information, see [HDInsight retirement on Windows](#).

Connectivity

The Ambari Web UI is available on your HDInsight cluster at [HTTPS://CLUSTERNAME.azurehdidnsight.net](https://CLUSTERNAME.azurehdidnsight.net), where **CLUSTERNAME** is the name of your cluster.

IMPORTANT

Connecting to Ambari on HDInsight requires HTTPS. When prompted for authentication, use the admin account name and password you provided when the cluster was created.

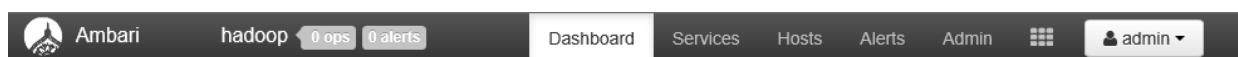
SSH tunnel (proxy)

While Ambari for your cluster is accessible directly over the Internet, some links from the Ambari Web UI (such as to the JobTracker) are not exposed on the internet. To access these services, you must create an SSH tunnel. For more information, see [Use SSH Tunneling with HDInsight](#).

Ambari Web UI

When connecting to the Ambari Web UI, you are prompted to authenticate to the page. Use the cluster admin user (default Admin) and password you used during cluster creation.

When the page opens, note the bar at the top. This bar contains the following information and controls:



- **Ambari logo** - Opens the dashboard, which can be used to monitor the cluster.

- **Cluster name # ops** - Displays the number of ongoing Ambari operations. Selecting the cluster name or **# ops** displays a list of background operations.
- **# alerts** - Displays warnings or critical alerts, if any, for the cluster.
- **Dashboard** - Displays the dashboard.
- **Services** - Information and configuration settings for the services in the cluster.
- **Hosts** - Information and configuration settings for the nodes in the cluster.
- **Alerts** - A log of information, warnings, and critical alerts.
- **Admin** - Software stack/services that are installed on the cluster, service account information, and Kerberos security.
- **Admin button** - Ambari management, user settings, and logout.

Monitoring

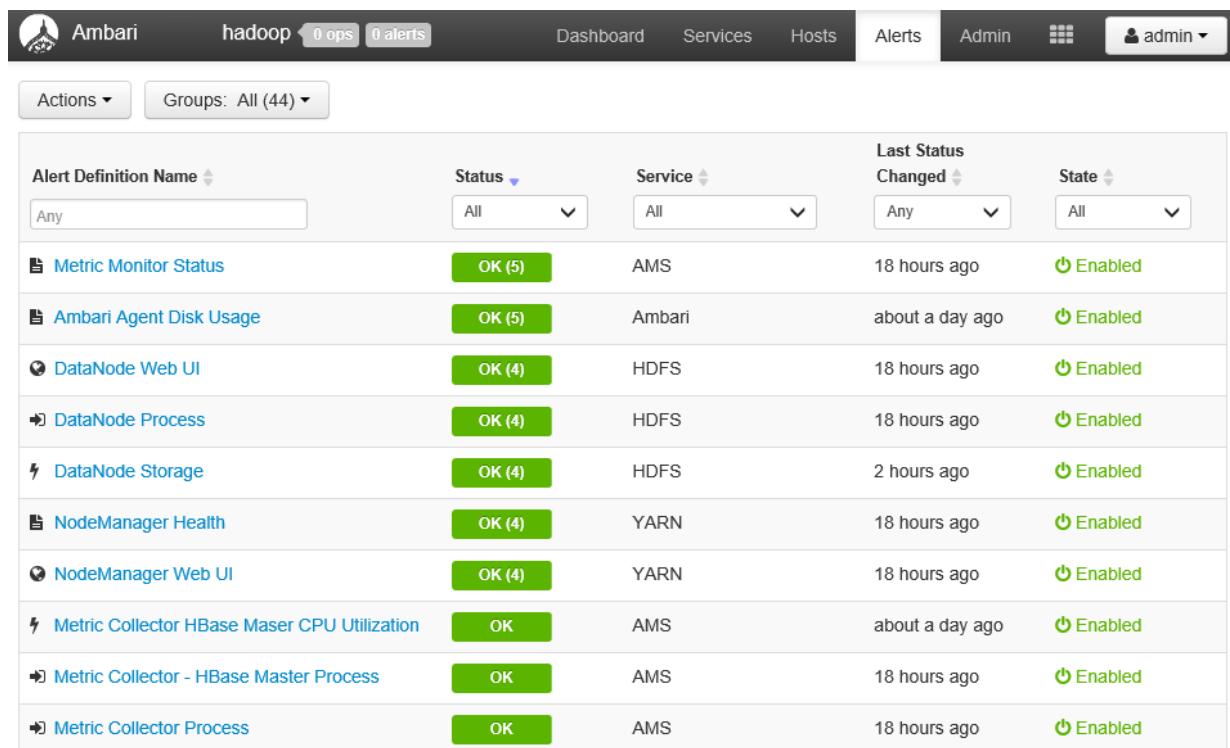
Alerts

The following list contains the common alert statuses used by Ambari:

- **OK**
- **Warning**
- **CRITICAL**
- **UNKNOWN**

Alerts other than **OK** cause the **# alerts** entry at the top of the page to display the number of alerts. Selecting this entry displays the alerts and their status.

Alerts are organized into several default groups, which can be viewed from the **Alerts** page.



The screenshot shows the Ambari UI with the "Alerts" tab selected. At the top, there are navigation links for "Dashboard", "Services", "Hosts", "Alerts" (which is active), and "Admin". On the far right, there's a user dropdown for "admin". Below the header, there are two dropdown menus: "Actions" and "Groups: All (44)". The main content area is a table listing 14 alerts. Each row contains the alert definition name, its status (e.g., OK (5)), the service it monitors (e.g., AMS, HDFS, YARN), the last time its status changed, and its current state (e.g., Enabled). Most alerts are in the OK state.

Alert Definition Name	Status	Service	Last Status Changed	State
Metric Monitor Status	OK (5)	AMS	18 hours ago	Enabled
Ambari Agent Disk Usage	OK (5)	Ambari	about a day ago	Enabled
DataNode Web UI	OK (4)	HDFS	18 hours ago	Enabled
DataNode Process	OK (4)	HDFS	18 hours ago	Enabled
DataNode Storage	OK (4)	HDFS	2 hours ago	Enabled
NodeManager Health	OK (4)	YARN	18 hours ago	Enabled
NodeManager Web UI	OK (4)	YARN	18 hours ago	Enabled
Metric Collector HBase Master CPU Utilization	OK	AMS	about a day ago	Enabled
Metric Collector - HBase Master Process	OK	AMS	18 hours ago	Enabled
Metric Collector Process	OK	AMS	18 hours ago	Enabled

You can manage the groups by using the **Actions** menu and selecting **Manage Alert Groups**.

Manage Alert Groups

You can manage alert groups for each service in this dialog. View the list of alert groups and the alert definitions configured in them. You can also add/remove alert definitions, and pick notification for that alert group.

The screenshot shows the 'Manage Alert Groups' dialog. On the left, a sidebar lists alert groups: 'AMBARI Default (44)', 'AMS Default (44)', 'HDFS Default (44)', 'HIVE Default (44)', 'KAFKA Default (44)', 'MAPREDUCE2 Default (44)', 'OOZIE Default (44)', 'PIG Default (44)', 'SQOOP Default (44)', 'TEZ Default (44)', 'YARN Default (44)', and 'ZOOKEEPER Default (44)'. The 'AMBARI Default' group is highlighted with a blue background. Below this sidebar are three buttons: a plus sign (+), a minus sign (-), and a gear icon with a dropdown arrow. To the right of the sidebar is a large list of alert definitions, which includes: Metric Collector HBase Maser CPU Utilization, Metric Collector - HBase Master Process, Metric Collector Process, Metric Monitor Status, Percent Metric Monitors Available, Metric Collector - ZooKeeper Server Process, History Server Web UI, History Server Process, History Server RPC Latency, History Server CPU Utilization, Secondary NameNode Process, NameNode High Availability Health, DataNode Web UI, NameNode Host CPU Utilization, and NameNode RPC Latency. This list has scroll bars on the right and bottom. At the bottom of the dialog are two buttons: 'Cancel' and 'Save' (in a green box). Between these buttons is a section titled 'Notifications' with a checkbox and an 'Add' button.

Alert Group	Alert Definition
AMBARI Default (44)	Metric Collector HBase Maser CPU Utilization
AMBARI Default (44)	Metric Collector - HBase Master Process
AMBARI Default (44)	Metric Collector Process
AMBARI Default (44)	Metric Monitor Status
AMBARI Default (44)	Percent Metric Monitors Available
AMBARI Default (44)	Metric Collector - ZooKeeper Server Process
AMBARI Default (44)	History Server Web UI
AMBARI Default (44)	History Server Process
AMBARI Default (44)	History Server RPC Latency
AMBARI Default (44)	History Server CPU Utilization
AMBARI Default (44)	Secondary NameNode Process
AMBARI Default (44)	NameNode High Availability Health
AMBARI Default (44)	DataNode Web UI
AMBARI Default (44)	NameNode Host CPU Utilization
AMBARI Default (44)	NameNode RPC Latency

You can also manage alerting methods, and create alert notifications from the **Actions** menu by selecting **Manage Alert Notifications**. Any current notifications are displayed. You can also create notifications from here. Notifications can be sent via **EMAIL** or **SNMP** when specific alert/severity combinations occur. For example, you can send an email message when any of the alerts in the **YARN Default** group is set to **Critical**.

Create Alert Notification

Name Notification name is required

Groups All Custom Select All | Clear All
 AMS Default
 MAPREDUCE2 Default
 PIG Default
 HDFS Default

Severity OK WARNING CRITICAL UNKNOWN Select All | Clear All

Description

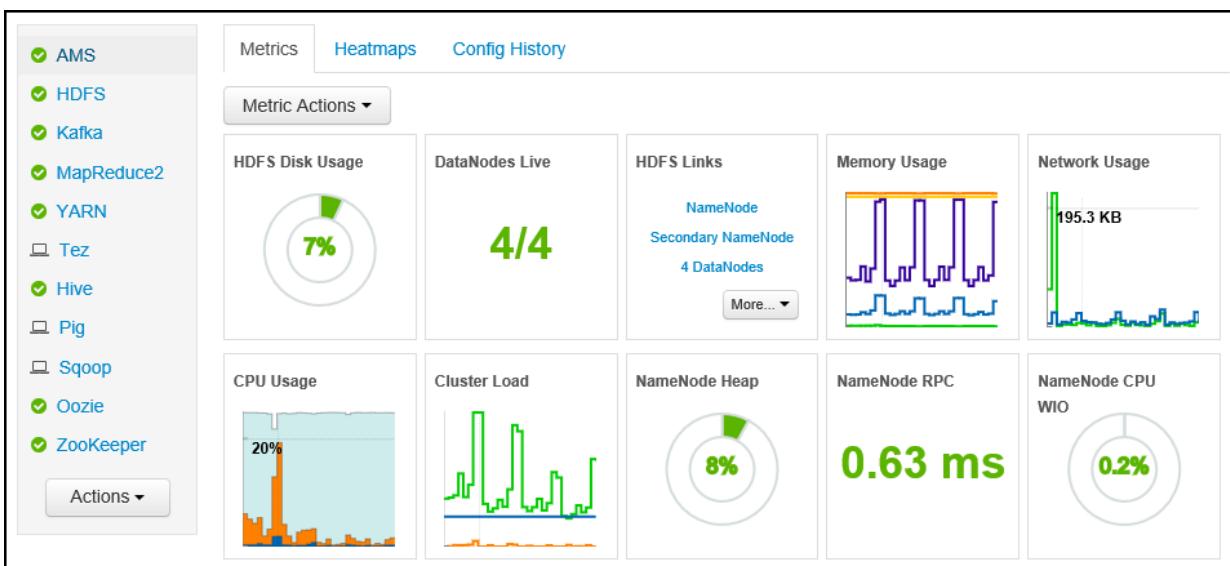
Method

Email To

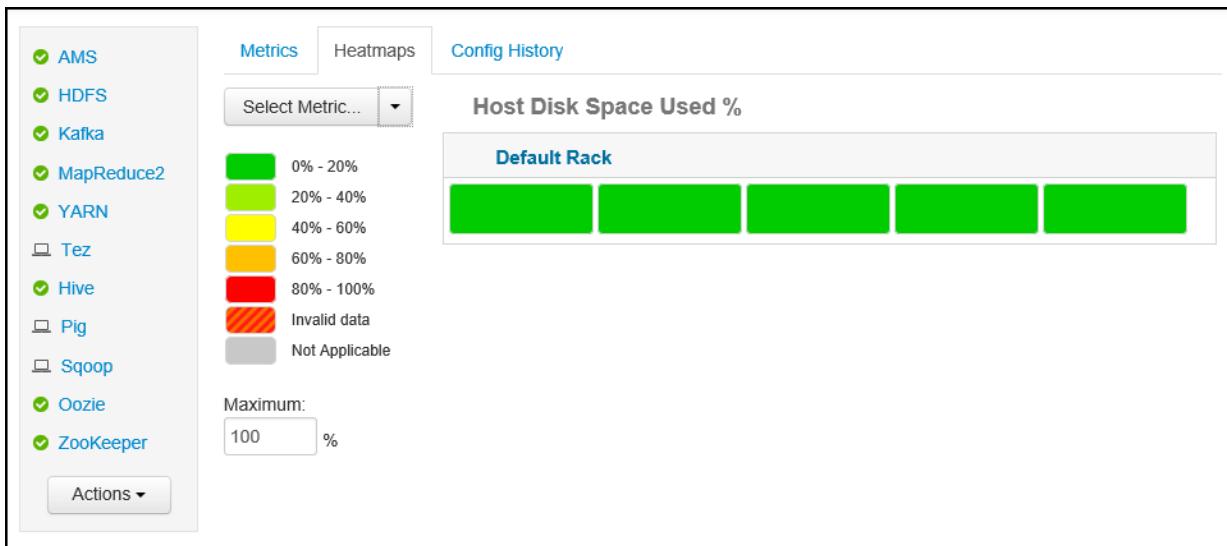
Finally, selecting **Manage Alert Settings** from the **Actions** menu allows you to set the number of times an alert must occur before a notification is sent. This setting can be used to prevent notifications for transient errors.

Cluster

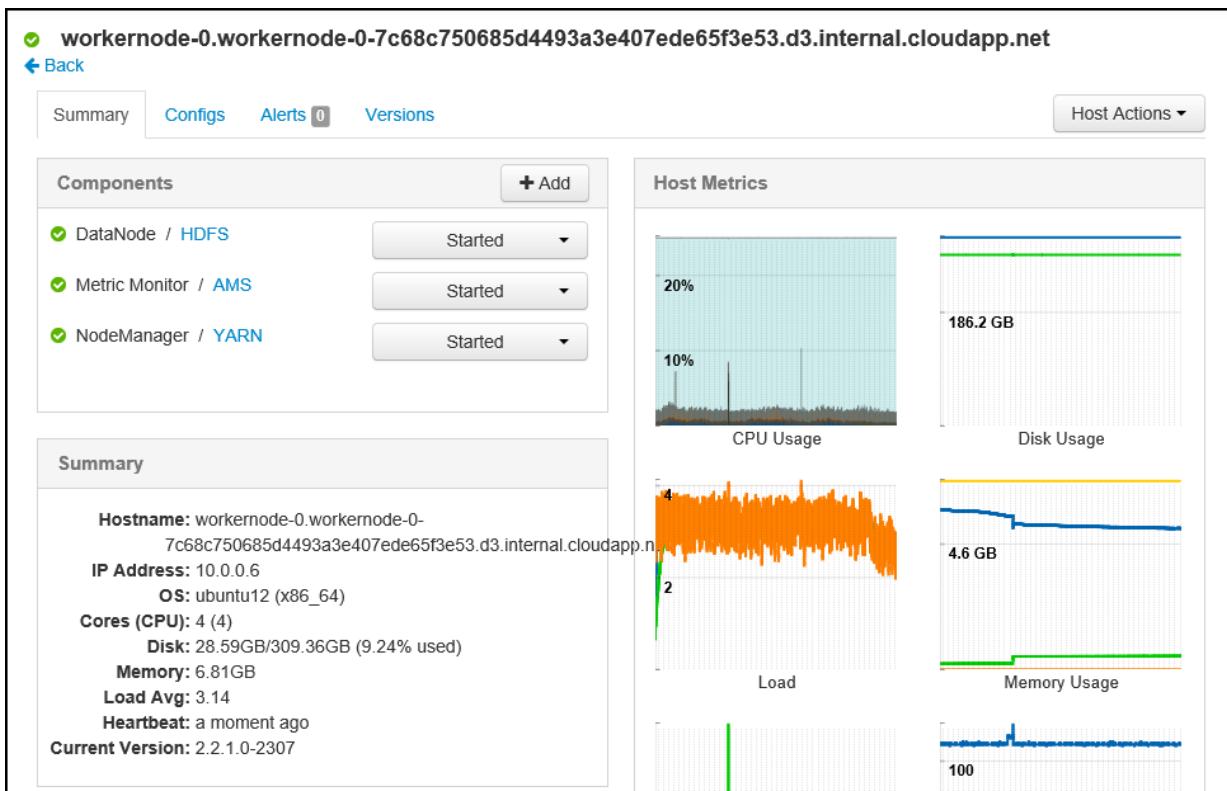
The **Metrics** tab of the dashboard contains a series of widgets that make it easy to monitor the status of your cluster at a glance. Several widgets, such as **CPU Usage**, provide additional information when clicked.



The **Heatmaps** tab displays metrics as colored heatmaps, going from green to red.



For more information on the nodes within the cluster, select **Hosts**. Then select the specific node you are interested in.



Services

The **Services** sidebar on the dashboard provides quick insight into the status of the services running on the cluster. Various icons are used to indicate status or actions that should be taken. For example, a yellow recycle symbol is displayed if a service needs to be recycled.

A sidebar menu listing HDInsight services. Services with green checkmarks are checked: AMS, HDFS, Kafka, MapReduce2, YARN, Hive, Sqoop, Oozie, and ZooKeeper. Services with grey squares are unselected: Tez, Pig, and Swoop. A 'Actions' button is at the bottom.

NOTE

The services displayed differ between HDInsight cluster types and versions. The services displayed here may be different than the services displayed for your cluster.

Selecting a service displays more detailed information on the service.

The screenshot shows the HDFS service details page. The left sidebar has 'HDFS' selected. The main area has tabs for 'Summary' (selected), 'Configs', and 'Quick Links'. The 'Summary' tab shows 'No alerts'. It includes sections for NameNode (Started), SNameNode (Started), DataNodes (4/4 Live), NameNode Uptime (28.72 mins), NameNode Heap (119.2 MB / 1004.0 MB), DataNodes Status (4 live / 0 dead / 0 decommissioning), Disk Usage (DFS Used: 112.0 KB / 1.1 TB), Disk Usage (Non DFS: 81.7 GB / 1.1 TB), Disk Usage (Remaining: 1.0 TB / 1.1 TB), Total Files + Directories (1), Upgrade Status (No pending upgrade), and Safe Mode Status (Not in safe mode). The 'Metrics' tab shows four charts: Total Space Utilization (931.3 GB, 465.6 GB), File Operations, Block Status, and HDFS I/O. The HDFS I/O chart shows two bars: 100 ms and 200 ms.

Quick links

Some services display a **Quick Links** link at the top of the page. This can be used to access service-specific web UIs, such as:

- **Job History** - MapReduce job history.
- **Resource Manager** - YARN ResourceManager UI.
- **NameNode** - Hadoop Distributed File System (HDFS) NameNode UI.
- **Oozie Web UI** - Oozie UI.

Selecting any of these links opens a new tab in your browser, which displays the selected page.

NOTE

Selecting the **Quick Links** entry for a service may return a "server not found" error. If you encounter this error, you must use an SSH tunnel when using the **Quick Links** entry for this service. For information, see [Use SSH Tunneling with HDInsight](#)

Management

Ambari users, groups, and permissions

Working with users, groups, and permissions are supported when using a [domain joined](#) HDInsight cluster. For information on using the Ambari Management UI on a domain-joined cluster, see [Manage domain-joined HDInsight clusters](#).

WARNING

Do not change the password of the Ambari watchdog (`hdinsightwatchdog`) on your Linux-based HDInsight cluster. Changing the password breaks the ability to use script actions or perform scaling operations with your cluster.

Hosts

The **Hosts** page lists all hosts in the cluster. To manage hosts, follow these steps.

Actions ▾ Filter: All (5) ▾							
Name	IP Address	Cores (CPU)	RAM	Disk Usage	Load Avg	Versions	Components
<input type="checkbox"/> headnode-0.larr...	10.0.0.5	4 (4)	6.81GB	<div style="width: 100px; height: 10px; background-color: #ccc; border: 1px solid #ccc;"></div>	12.76	2.2.1.0-2307 (Current)	▶ 23 Components
<input type="checkbox"/> workernode-0.w...	10.0.0.6	4 (4)	6.81GB	<div style="width: 100px; height: 10px; background-color: #ccc; border: 1px solid #ccc;"></div>	2.52	2.2.1.0-2307 (Current)	▶ 3 Components
<input type="checkbox"/> workernode-1.w...	10.0.0.7	4 (4)	6.81GB	<div style="width: 100px; height: 10px; background-color: #ccc; border: 1px solid #ccc;"></div>	3.14	2.2.1.0-2307 (Current)	▶ 3 Components
<input type="checkbox"/> workernode-2.w...	10.0.0.8	4 (4)	6.81GB	<div style="width: 100px; height: 10px; background-color: #ccc; border: 1px solid #ccc;"></div>	2.76	2.2.1.0-2307 (Current)	▶ 3 Components
<input type="checkbox"/> workernode-3.w...	10.0.0.9	4 (4)	6.81GB	<div style="width: 100px; height: 10px; background-color: #ccc; border: 1px solid #ccc;"></div>	2.42	2.2.1.0-2307 (Current)	▶ 3 Components
5 of 5 hosts showing - clear filters				Show:	10	▼	1 - 5 of 5

NOTE

Adding, decommissioning, and recommissioning a host should not be used with HDInsight clusters.

1. Select the host that you wish to manage.
2. Use the **Actions** menu to select the action that you wish to perform:
 - **Start all components** - Start all components on the host.
 - **Stop all components** - Stop all components on the host.
 - **Restart all components** - Stop and start all components on the host.

- **Turn on maintenance mode** - Suppresses alerts for the host. This mode should be enabled if you are performing actions that generate alerts. For example, stopping and starting a service.
- **Turn off maintenance mode** - Returns the host to normal alerting.
- **Stop** - Stops DataNode or NodeManagers on the host.
- **Start** - Starts DataNode or NodeManagers on the host.
- **Restart** - Stops and starts DataNode or NodeManagers on the host.
- **Decommission** - Removes a host from the cluster.

NOTE

Do not use this action on HDInsight clusters.

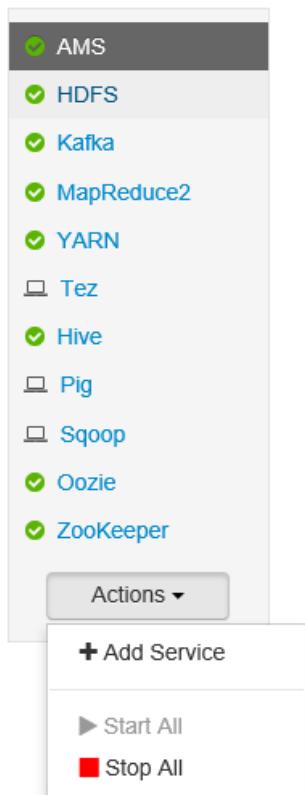
- **Recommission** - Adds a previously decommissioned host to the cluster.

NOTE

Do not use this action on HDInsight clusters.

Services

From the **Dashboard** or **Services** page, use the **Actions** button at the bottom of the list of services to stop and start all services.

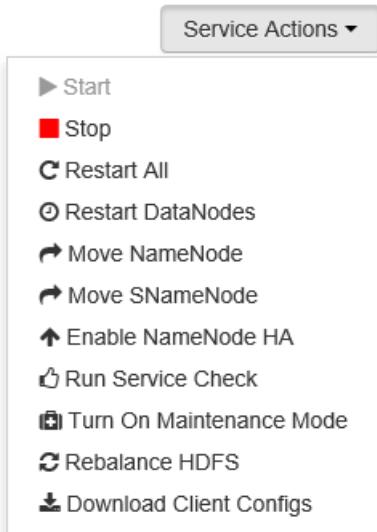


WARNING

While **Add Service** is listed in this menu, it should not be used to add services to the HDInsight cluster. New services should be added using a Script Action during cluster provisioning. For more information on using Script Actions, see [Customize HDInsight clusters using Script Actions](#).

While the **Actions** button can restart all services, often you want to start, stop, or restart a specific service. Use the following steps to perform actions on an individual service:

1. From the **Dashboard** or **Services** page, select a service.
2. From the top of the **Summary** tab, use the **Service Actions** button and select the action to take. This restarts the service on all nodes.



NOTE

Restarting some services while the cluster is running may generate alerts. To avoid alerts, you can use the **Service Actions** button to enable **Maintenance mode** for the service before performing the restart.

3. Once an action has been selected, the **# op** entry at the top of the page increments to show that a background operation is occurring. If configured to display, the list of background operations is displayed.

NOTE

If you enabled **Maintenance mode** for the service, remember to disable it by using the **Service Actions** button once the operation has finished.

To configure a service, use the following steps:

1. From the **Dashboard** or **Services** page, select a service.
2. Select the **Configs** tab. The current configuration is displayed. A list of previous configurations is also displayed.

The screenshot shows the Ambari Web UI for managing cluster configurations. On the left, a sidebar lists various services with checkboxes. The 'Hive' service is selected and highlighted with a dark grey background. The main content area is titled 'Hive Metastore'. It displays configuration details such as 'Hive Metastore hosts' (set to 'headnode-0.larryfrhadoop-ssh.d4.internal.cloudapp.net'), 'Database Type' (set to 'PostgreSQL'), and 'Hive Database' options. The 'Existing PostgreSQL Database' radio button is selected. At the top, there are tabs for 'Summary' and 'Configs', and a 'Service Actions' dropdown. Below the tabs, a section for 'Manage Config Groups' shows two configurations: 'V2' (admin, 2 hours ago) and 'V1' (admin, about a day ago). A 'Current' button is highlighted in green. At the bottom right, there are 'Discard' and 'Save' buttons.

3. Use the fields displayed to modify the configuration, and then select **Save**. Or select a previous configuration and then select **Make current** to roll back to the previous settings.

Ambari views

Ambari Views allow developers to plug UI elements into the Ambari Web UI using the [Ambari Views Framework](#). HDInsight provides the following views with Hadoop cluster types:

- Yarn Queue Manager: The queue manager provides a simple UI for viewing and modifying YARN queues.
- Hive View: The Hive View allows you to run Hive queries directly from your web browser. You can save queries, view results, save results to the cluster storage, or download results to your local system. For more information on using Hive Views, see [Use Hive Views with HDInsight](#).
- Tez View: The Tez View allows you to better understand and optimize jobs. You can view information on how Tez jobs are executed and what resources are used.

Manage HDInsight clusters by using the Ambari REST API

8/16/2017 • 12 min to read • [Edit Online](#)

Learn how to use the Ambari REST API to manage and monitor Hadoop clusters in Azure HDInsight.

Apache Ambari simplifies the management and monitoring of a Hadoop cluster by providing an easy to use web UI and REST API. Ambari is included on HDInsight clusters that use the Linux operating system. You can use Ambari to monitor the cluster and make configuration changes.

What is Ambari

[Apache Ambari](#) provides web UI that can be used to provision, manage, and monitor Hadoop clusters. Developers can integrate these capabilities into their applications by using the [Ambari REST APIs](#).

Ambari is provided by default with Linux-based HDInsight clusters.

How to use the Ambari REST API

IMPORTANT

The information and examples in this document require an HDInsight cluster that uses Linux operating system. For more information, see [Get started with HDInsight](#).

The examples in this document are provided for both the Bourne shell (bash) and PowerShell. The bash examples were tested with GNU bash 4.3.11, but should work with other Unix shells. The PowerShell examples were tested with PowerShell 5.0, but should work with PowerShell 3.0 or higher.

If using the **Bourne shell** (Bash), you must have the following installed:

- [cURL](#): cURL is a utility that can be used to work with REST APIs from the command line. In this document, it is used to communicate with the Ambari REST API.

Whether using Bash or PowerShell, you must also have [jq](#) installed. Jq is a utility for working with JSON documents. It is used in **all** the Bash examples, and **one** of the PowerShell examples.

Base URI for Ambari Rest API

The base URI for the Ambari REST API on HDInsight is

<https://CLUSTERNAME.azurehdinsight.net/api/v1/clusters/CLUSTERNAME>, where **CLUSTERNAME** is the name of your cluster.

IMPORTANT

While the cluster name in the fully qualified domain name (FQDN) part of the URI (CLUSTERNAME.azurehdinsight.net) is case-insensitive, other occurrences in the URI are case-sensitive. For example, if your cluster is named `MyCluster`, the following are valid URLs:

```
https://mycluster.azurehdinsight.net/api/v1/clusters/MyCluster
```

```
https://MyCluster.azurehdinsight.net/api/v1/clusters/MyCluster
```

The following URLs return an error because the second occurrence of the name is not the correct case.

```
https://mycluster.azurehdinsight.net/api/v1/clusters/mycluster
```

```
https://MyCluster.azurehdinsight.net/api/v1/clusters/mycluster
```

Authentication

Connecting to Ambari on HDInsight requires HTTPS. Use the admin account name (the default is **admin**) and password you provided during cluster creation.

Examples: Authentication and parsing JSON

The following examples demonstrate how to make a GET request against the base Ambari REST API:

```
curl -u admin:$PASSWORD -sS -G "https://$CLUSTERNAME.azurehdinsight.net/api/v1/clusters/$CLUSTERNAME"
```

IMPORTANT

The Bash examples in this document make the following assumptions:

- The login name for the cluster is the default value of `admin`.
- `$PASSWORD` contains the password for the HDInsight login command. You can set this value by using `PASSWORD='mypassword'`.
- `$CLUSTERNAME` contains the name of the cluster. You can set this value by using `set CLUSTERNAME='clusternname'`

```
$resp = Invoke-WebRequest -Uri "https://$clusterName.azurehdinsight.net/api/v1/clusters/$clusterName" `  
-Credential $creds  
$resp.Content
```

IMPORTANT

The PowerShell examples in this document make the following assumptions:

- `$creds` is a credential object that contains the admin login and password for the cluster. You can set this value by using `$creds = Get-Credential -UserName "admin" -Message "Enter the HDInsight login"` and providing the credentials when prompted.
- `$clusterName` is a string that contains the name of the cluster. You can set this value by using `$clusterName="clustername"`.

Both examples return a JSON document that begins with information similar to the following example:

```
{
  "href" : "http://10.0.0.10:8080/api/v1/clusters/CLUSTERNAME",
  "Clusters" : {
    "cluster_id" : 2,
    "cluster_name" : "CLUSTERNAME",
    "health_report" : {
      "Host/stale_config" : 0,
      "Host/maintenance_state" : 0,
      "Host/host_state/HEALTHY" : 7,
      "Host/host_state/UNHEALTHY" : 0,
      "Host/host_state/HEARTBEAT_LOST" : 0,
      "Host/host_state/INIT" : 0,
      "Host/host_status/HEALTHY" : 7,
      "Host/host_status/UNHEALTHY" : 0,
      "Host/host_status/UNKNOWN" : 0,
      "Host/host_status/ALERT" : 0
    ...
  }
}
```

Parsing JSON data

The following example uses `jq` to parse the JSON response document and display only the `health_report` information from the results.

```
curl -u admin:$PASSWORD -sS -G "https://$CLUSTERNAME.azurehdinsight.net/api/v1/clusters/$CLUSTERNAME" \
| jq '.Clusters.health_report'
```

PowerShell 3.0 and higher provides the `ConvertFrom-Json` cmdlet, which converts the JSON document into an object that is easier to work with from PowerShell. The following example uses `ConvertFrom-Json` to display only the `health_report` information from the results.

```
$resp = Invoke-WebRequest -Uri "https://$clusterName.azurehdinsight.net/api/v1/clusters/$clusterName" ` 
-Credential $creds
$respObj = ConvertFrom-Json $resp.Content
$respObj.Clusters.health_report
```

NOTE

While most examples in this document use `ConvertFrom-Json` to display elements from the response document, the [Update Ambari configuration](#) example uses `jq`. `Jq` is used in this example to construct a new template from the JSON response document.

For a complete reference of the REST API, see [Ambari API Reference V1](#).

Example: Get the FQDN of cluster nodes

When working with HDInsight, you may need to know the fully qualified domain name (FQDN) of a cluster node. You can easily retrieve the FQDN for the various nodes in the cluster using the following examples:

- **All nodes**

```
curl -u admin:$PASSWORD -sS -G
"https://$CLUSTERNAME.azurehdinsight.net/api/v1/clusters/$CLUSTERNAME/hosts" \
| jq '.items[].Hosts.host_name'
```

```
$resp = Invoke-WebRequest -Uri
"https://$clusterName.azurehdinsight.net/api/v1/clusters/$clusterName/hosts" `

-Credential $creds
$respObj = ConvertFrom-Json $resp.Content
$respObj.items.Hosts.host_name
```

- **Head nodes**

```
curl -u admin:$PASSWORD -sS -G
"https://$CLUSTERNAME.azurehdinsight.net/api/v1/clusters/$CLUSTERNAME/services/HDFS/components/NAMENODE"
" \
| jq '.host_components[].HostRoles.host_name'
```

```
$resp = Invoke-WebRequest -Uri
"https://$clusterName.azurehdinsight.net/api/v1/clusters/$clusterName/services/HDFS/components/NAMENODE"
" `

-Credential $creds
$respObj = ConvertFrom-Json $resp.Content
$respObj.host_components.HostRoles.host_name
```

- **Worker nodes**

```
curl -u admin:PASSWORD -sS -G
"https://$CLUSTERNAME.azurehdinsight.net/api/v1/clusters/$CLUSTERNAME/services/HDFS/components/DATANODE"
" \
| jq '.host_components[].HostRoles.host_name'
```

```
$resp = Invoke-WebRequest -Uri
"https://$clusterName.azurehdinsight.net/api/v1/clusters/$clusterName/services/HDFS/components/DATANODE"
" `

-Credential $creds
$respObj = ConvertFrom-Json $resp.Content
$respObj.host_components.HostRoles.host_name
```

- **Zookeeper nodes**

```
curl -u admin:PASSWORD -sS -G
"https://$CLUSTERNAME.azurehdinsight.net/api/v1/clusters/$CLUSTERNAME/services/ZOOKEEPER/components/ZOOKEEPER_SERVER" \
| jq '.host_components[].HostRoles.host_name'
```

```
$resp = Invoke-WebRequest -Uri
"https://$clusterName.azurehdinsight.net/api/v1/clusters/$clusterName/services/ZOOKEEPER/components/ZOOKEEPER_SERVER" `

-Credential $creds
$respObj = ConvertFrom-Json $resp.Content
$respObj.host_components.HostRoles.host_name
```

Example: Get the internal IP address of cluster nodes

IMPORTANT

The IP addresses returned by the examples in this section are not directly accessible over the internet. They are only accessible within the Azure Virtual Network that contains the HDInsight cluster.

For more information on working with HDInsight and virtual networks, see [Extend HDInsight capabilities by using a custom Azure Virtual Network](#).

To find the IP address, you must know the internal fully qualified domain name (FQDN) of the cluster nodes. Once you have the FQDN, you can then get the IP address of the host. The following examples first query Ambari for the FQDN of all the host nodes, then query Ambari for the IP address of each host.

```
for HOSTNAME in $(curl -u admin:$PASSWORD -sS -G
"https://$CLUSTERNAME.azurehdinsight.net/api/v1/clusters/$CLUSTERNAME/hosts" | jq -r
'.items[].Hosts.host_name')
do
    IP=$(curl -u admin:$PASSWORD -sS -G
"https://$CLUSTERNAME.azurehdinsight.net/api/v1/clusters/$CLUSTERNAME/hosts/$HOSTNAME" | jq -r '.Hosts.ip')
    echo "$HOSTNAME <--> $IP"
done
```

```
$uri = "https://$clusterName.azurehdinsight.net/api/v1/clusters/$clusterName/hosts"
$resp = Invoke-WebRequest -Uri $uri -Credential $creds
$respObj = ConvertFrom-Json $resp.Content
foreach($item in $respObj.items) {
    $hostName = [string]$item.Hosts.host_name
    $hostInfoResp = Invoke-WebRequest -Uri "$uri/$hostName" `

    -Credential $creds
    $hostInfoObj = ConvertFrom-Json $hostInfoResp
    $hostIp = $hostInfoObj.Hosts.ip
    "$hostName <--> $hostIp"
}
```

Example: Get the default storage

When you create an HDInsight cluster, you must use an Azure Storage Account or Data Lake Store as the default storage for the cluster. You can use Ambari to retrieve this information after the cluster has been created. For example, if you want to read/write data to the container outside HDInsight.

The following examples retrieve the default storage configuration from the cluster:

```
curl -u admin:$PASSWORD -sS -G
"https://$CLUSTERNAME.azurehdinsight.net/api/v1/clusters/$CLUSTERNAME/configurations/service_config_versions?
service_name=HDFS&service_config_version=1" \
| jq '.items[].configurations[].properties["fs.defaultFS"] | select(. != null)'
```

```
$resp = Invoke-WebRequest -Uri
"https://$clusterName.azurehdinsight.net/api/v1/clusters/$clusterName/configurations/service_config_versions?
service_name=HDFS&service_config_version=1" `

-Credential $creds
$respObj = ConvertFrom-Json $resp.Content
$respObj.items.configurations.properties.'fs.defaultFS'
```

IMPORTANT

These examples return the first configuration applied to the server (`service_config_version=1`) which contains this information. If you retrieve a value that has been modified after cluster creation, you may need to list the configuration versions and retrieve the latest one.

The return value is similar to one of the following examples:

- `wasb://CONTAINER@ACCOUNTNAME.blob.core.windows.net` - This value indicates that the cluster is using an Azure Storage account for default storage. The `ACCOUNTNAME` value is the name of the storage account. The `CONTAINER` portion is the name of the blob container in the storage account. The container is the root of the HDFS compatible storage for the cluster.
- `adl://home` - This value indicates that the cluster is using an Azure Data Lake Store for default storage.

To find the Data Lake Store account name, use the following examples:

```
curl -u admin:$PASSWORD -sS -G
"https://$CLUSTERNAME.azurehdinsight.net/api/v1/clusters/$CLUSTERNAME/configurations/service_config_versions?service_name=HDFS&service_config_version=1" \
| jq '.items[].configurations[].properties["dfs.adls.home.hostname"] | select(. != null)'
```

```
$resp = Invoke-WebRequest -Uri
"https://$clusterName.azurehdinsight.net/api/v1/clusters/$clusterName/configurations/service_config_versions?service_name=HDFS&service_config_version=1" ` 
-Credential $creds
$respObj = ConvertFrom-Json $resp.Content
$respObj.items.configurations.properties.'dfs.adls.home.hostname'
```

The return value is similar to `ACCOUNTNAME.azuredatalakestore.net`, where `ACCOUNTNAME` is the name of the Data Lake Store account.

To find the directory within Data Lake Store that contains the storage for the cluster, use the following examples:

```
curl -u admin:$PASSWORD -sS -G
"https://$CLUSTERNAME.azurehdinsight.net/api/v1/clusters/$CLUSTERNAME/configurations/service_config_versions?service_name=HDFS&service_config_version=1" \
| jq '.items[].configurations[].properties["dfs.adls.home.mountpoint"] | select(. != null)'
```

```
$resp = Invoke-WebRequest -Uri
"https://$clusterName.azurehdinsight.net/api/v1/clusters/$clusterName/configurations/service_config_versions?service_name=HDFS&service_config_version=1" ` 
-Credential $creds
$respObj = ConvertFrom-Json $resp.Content
$respObj.items.configurations.properties.'dfs.adls.home.mountpoint'
```

The return value is similar to `/clusters/CLUSTERNAME/`. This value is a path within the Data Lake Store account. This path is the root of the HDFS compatible file system for the cluster.

NOTE

The `Get-AzureRmHDInsightCluster` cmdlet provided by [Azure PowerShell](#) also returns the storage information for the cluster.

Example: Get configuration

1. Get the configurations that are available for your cluster.

```
curl -u admin:$PASSWORD -sS -G "https://$CLUSTERNAME.azurehdinsight.net/api/v1/clusters/$CLUSTERNAME?fields=Clusters/desired_configs"
```

```
$respObj = Invoke-WebRequest -Uri  
"https://$clusterName.azurehdinsight.net/api/v1/clusters/$clusterName`?fields=Clusters/desired_configs"  
  
-Credential $creds  
$respObj.Content
```

This example returns a JSON document containing the current configuration (identified by the *tag* value) for the components installed on the cluster. The following example is an excerpt from the data returned from a Spark cluster type.

```
"spark-metrics-properties" : {  
    "tag" : "INITIAL",  
    "user" : "admin",  
    "version" : 1  
},  
"spark-thrift-fairscheduler" : {  
    "tag" : "INITIAL",  
    "user" : "admin",  
    "version" : 1  
},  
"spark-thrift-sparkconf" : {  
    "tag" : "INITIAL",  
    "user" : "admin",  
    "version" : 1  
}
```

2. Get the configuration for the component that you are interested in. In the following example, replace

`INITIAL` with the tag value returned from the previous request.

```
curl -u admin:$PASSWORD -sS -G  
"https://$CLUSTERNAME.azurehdinsight.net/api/v1/clusters/$CLUSTERNAME/configurations?type=core-site&tag=INITIAL"
```

```
$resp = Invoke-WebRequest -Uri  
"https://$clusterName.azurehdinsight.net/api/v1/clusters/$clusterName/configurations?type=core-site&tag=INITIAL`  
-Credential $creds  
$resp.Content
```

This example returns a JSON document containing the current configuration for the `core-site` component.

Example: Update configuration

1. Get the current configuration, which Ambari stores as the "desired configuration":

```
curl -u admin:$PASSWORD -sS -G "https://$CLUSTERNAME.azurehdinsight.net/api/v1/clusters/$CLUSTERNAME?fields=Clusters/desired_configs"
```

```
Invoke-WebRequest -Uri "https://$clusterName.azurehdinsight.net/api/v1/clusters/$clusterName`?
fields=Clusters/desired_configs" `-
-Credential $creds
```

This example returns a JSON document containing the current configuration (identified by the *tag* value) for the components installed on the cluster. The following example is an excerpt from the data returned from a Spark cluster type.

```
"spark-metrics-properties" : {
    "tag" : "INITIAL",
    "user" : "admin",
    "version" : 1
},
"spark-thrift-fairscheduler" : {
    "tag" : "INITIAL",
    "user" : "admin",
    "version" : 1
},
"spark-thrift-sparkconf" : {
    "tag" : "INITIAL",
    "user" : "admin",
    "version" : 1
}
```

From this list, you need to copy the name of the component (for example, **spark_thrift_sparkconf** and the **tag** value.

2. Retrieve the configuration for the component and tag by using the following commands:

```
curl -u admin:$PASSWORD -sS -G
"https://$CLUSTERNAME.azurehdinsight.net/api/v1/clusters/$CLUSTERNAME/configurations?type=spark-thrift-
sparkconf&tag=INITIAL" \
| jq --arg newtag $(echo version$(date +%s%N)) '.items[] | del(.href, .version, .Config) | .tag |=
$newtag | {"Clusters": {"desired_config": .}}' > newconfig.json
```

```
$epoch = Get-Date -Year 1970 -Month 1 -Day 1 -Hour 0 -Minute 0 -Second 0
$now = Get-Date
$unixTimeStamp = [math]::truncate($now.ToUniversalTime().Subtract($epoch).TotalMilliseconds)
$resp = Invoke-WebRequest -Uri
"https://$clusterName.azurehdinsight.net/api/v1/clusters/$clusterName/configurations?type=spark-thrift-
sparkconf&tag=INITIAL" `-
-Credential $creds
$resp.Content | jq --arg newtag "version$unixTimeStamp" '.items[] | del(.href, .version, .Config) | 
.tag |= $newtag | {"Clusters": {"desired_config": .}}' > newconfig.json
```

NOTE

Replace **spark-thrift-sparkconf** and **INITIAL** with the component and tag that you want to retrieve the configuration for.

Jq is used to turn the data retrieved from HDInsight into a new configuration template. Specifically, these examples perform the following actions:

- Creates a unique value containing the string "version" and the date, which is stored in `newtag`.
- Creates a root document for the new desired configuration.

- Gets the contents of the `.items[]` array and adds it under the `desired_config` element.
- Deletes the `href`, `version`, and `Config` elements, as these elements aren't needed to submit a new configuration.
- Adds a `tag` element with a value of `version#####`. The numeric portion is based on the current date. Each configuration must have a unique tag.

Finally, the data is saved to the `newconfig.json` document. The document structure should appear similar to the following example:

```
{
  "Clusters": {
    "desired_config": {
      "tag": "version1459260185774265400",
      "type": "spark-thrift-sparkconf",
      "properties": {
        ...
      },
      "properties_attributes": {
        ...
      }
    }
  }
}
```

3. Open the `newconfig.json` document and modify/add values in the `properties` object. The following example changes the value of `"spark.yarn.am.memory"` from `"1g"` to `"3g"`. It also adds `"spark.kryoserializer.buffer.max"` with a value of `"256m"`.

```
"spark.yarn.am.memory": "3g",
"spark.kryoserializer.buffer.max": "256m",
```

Save the file once you are done making modifications.

4. Use the following commands to submit the updated configuration to Ambari.

```
curl -u admin:$PASSWORD -sS -H "X-Requested-By: ambari" -X PUT -d @newconfig.json
"https://$CLUSTERNAME.azurehdinsight.net/api/v1/clusters/$CLUSTERNAME"
```

```
$newConfig = Get-Content .\newconfig.json
$resp = Invoke-WebRequest -Uri "https://$clusterName.azurehdinsight.net/api/v1/clusters/$clusterName" ` 
-Credential $creds ` 
-Method PUT ` 
-Headers @{"X-Requested-By" = "ambari"} ` 
-Body $newConfig
$resp.Content
```

These commands submit the contents of the `newconfig.json` file to the cluster as the new desired configuration. The request returns a JSON document. The `versionTag` element in this document should match the version you submitted, and the `configs` object contains the configuration changes you requested.

Example: Restart a service component

At this point, if you look at the Ambari web UI, the Spark service indicates that it needs to be restarted before the new configuration can take effect. Use the following steps to restart the service.

1. Use the following to enable maintenance mode for the Spark service:

```
curl -u admin:$PASSWORD -sS -H "X-Requested-By: ambari" \
-X PUT -d '{"RequestInfo": {"context": "turning on maintenance mode for SPARK"}, "Body": {"ServiceInfo": {"maintenance_state": "ON"}}}' \
"https://$CLUSTERNAME.azurehdinsight.net/api/v1/clusters/$CLUSTERNAME/services/SPARK"
```

```
$resp = Invoke-WebRequest -Uri
"https://$clusterName.azurehdinsight.net/api/v1/clusters/$clusterName/services/SPARK" `

-Credential $creds `

-Method PUT `

-Headers @{"X-Requested-By" = "ambari"} `

-Body '{
    "RequestInfo": {
        "context": "turning on maintenance mode for SPARK"
    },
    "Body": {
        "ServiceInfo": {
            "maintenance_state": "ON"
        }
    }
}' `

$resp.Content
```

These commands send a JSON document to the server that turns on maintenance mode. You can verify that the service is now in maintenance mode using the following request:

```
curl -u admin:$PASSWORD -sS -H "X-Requested-By: ambari" \
"https://$CLUSTERNAME.azurehdinsight.net/api/v1/clusters/$CLUSTERNAME/services/SPARK" \
| jq .ServiceInfo.maintenance_state
```

```
$resp = Invoke-WebRequest -Uri
"https://$clusterName.azurehdinsight.net/api/v1/clusters/$clusterName/services/SPARK2" `

-Credential $creds `

$respObj = ConvertFrom-Json $resp.Content `

$respObj.ServiceInfo.maintenance_state
```

The return value is **ON**.

2. Next, use the following to turn off the service:

```
curl -u admin:$PASSWORD -sS -H "X-Requested-By: ambari" \
-X PUT -d '{
    "RequestInfo": {
        "context": "_PARSE_.STOP.SPARK",
        "operation_level": {
            "level": "SERVICE",
            "cluster_name": "CLUSTERNAME",
            "service_name": "SPARK"
        }
    },
    "Body": {
        "ServiceInfo": {
            "state": "INSTALLED"
        }
    }
}' \
"https://$CLUSTERNAME.azurehdinsight.net/api/v1/clusters/$CLUSTERNAME/services/SPARK"
```

```
$resp = Invoke-WebRequest -Uri
"https://$clusterName.azurehdinsight.net/api/v1/clusters/$clusterName/services/SPARK" `

-Credential $creds `

-Method PUT `

-Headers @{"X-Requested-By" = "ambari"} `

-Body '{
    "RequestInfo": {
        "context": "_PARSE_.STOP.SPARK",
        "operation_level": {
            "level": "SERVICE",
            "cluster_name": "CLUSTERNAME",
            "service_name": "SPARK"
        }
    },
    "Body": {
        "ServiceInfo": {
            "state": "INSTALLED"
        }
    }
}' `

$resp.Content
```

The response is similar to the following example:

```
{
    "href" : "http://10.0.0.18:8080/api/v1/clusters/CLUSTERNAME/requests/29",
    "Requests" : {
        "id" : 29,
        "status" : "Accepted"
    }
}
```

IMPORTANT

The `href` value returned by this URI is using the internal IP address of the cluster node. To use it from outside the cluster, replace the '10.0.0.18:8080' portion with the FQDN of the cluster.

The following commands retrieve the status of the request:

```
curl -u admin:$PASSWORD -sS -H "X-Requested-By: ambari" \
"https://$CLUSTERNAME.azurehdinsight.net/api/v1/clusters/$CLUSTERNAME/requests/29" \
| jq .Requests.request_status
```

```
$resp = Invoke-WebRequest -Uri
"https://$clusterName.azurehdinsight.net/api/v1/clusters/$clusterName/requests/29" ` 
-Credential $creds
$respObj = ConvertFrom-Json $resp.Content
$respObj.Requests.request_status
```

A response of `COMPLETED` indicates that the request has finished.

3. Once the previous request completes, use the following to start the service.

```
curl -u admin:$PASSWORD -sS -H "X-Requested-By: ambari" \
-X PUT -d '{"RequestInfo":{"context":"_PARSE_.STOP.SPARK","operation_level": 
{"level":"SERVICE","cluster_name":"CLUSTERNAME","service_name":"SPARK"}}, "Body":{"ServiceInfo": 
 {"state":"STARTED"}}}' \
"https://$CLUSTERNAME.azurehdinsight.net/api/v1/clusters/$CLUSTERNAME/services/SPARK"
```

```
$resp = Invoke-WebRequest -Uri
"https://$clusterName.azurehdinsight.net/api/v1/clusters/$clusterName/services/SPARK" ` 
-Credential $creds ` 
-Method PUT ` 
-Headers @{"X-Requested-By" = "ambari"} ` 
-Body '{
"RequestInfo":{"context":"_PARSE_.STOP.SPARK","operation_level": 
{"level":"SERVICE","cluster_name":"CLUSTERNAME","service_name":"SPARK"}}, "Body":{"ServiceInfo": 
 {"state":"STARTED"}}}'
```

The service is now using the new configuration.

4. Finally, use the following to turn off maintenance mode.

```
curl -u admin:$PASSWORD -sS -H "X-Requested-By: ambari" \
-X PUT -d '{"RequestInfo": {"context": "turning off maintenance mode for SPARK"}, "Body": 
 {"ServiceInfo": {"maintenance_state": "OFF"}}}' \
"https://$CLUSTERNAME.azurehdinsight.net/api/v1/clusters/$CLUSTERNAME/services/SPARK"
```

```
$resp = Invoke-WebRequest -Uri  
"https://$clusterName.azurehdinsight.net/api/v1/clusters/$clusterName/services/SPARK" `  
-Credential $creds `  
-Method PUT `  
-Headers @{"X-Requested-By" = "ambari"} `  
-Body '{"RequestInfo": {"context": "turning off maintenance mode for SPARK"}, "Body":  
{"ServiceInfo": {"maintenance_state": "OFF"}}}'
```

Next steps

For a complete reference of the REST API, see [Ambari API Reference V1](#).

Manage Hadoop clusters in HDInsight by using the Azure portal

8/16/2017 • 12 min to read • [Edit Online](#)

Using the [Azure portal](#), you can manage Hadoop clusters in Azure HDInsight. Use the tab selector for information on managing Hadoop clusters in HDInsight using other tools.

Now let's review your options for managing and administering your HDInsight clusters in the Azure portal.

Prerequisites

Before you begin this article, you must have the following:

- **An Azure subscription.** See [Get Azure free trial](#).

Create clusters

WARNING

Billing for HDInsight clusters is prorated per minute, whether you are using them or not. Be sure to delete your cluster after you have finished using it. For more information, see [How to delete an HDInsight cluster](#).

HDInsight works with a wide range of Hadoop components. For the list of the components that have been verified and supported, see [What version of Hadoop is in Azure HDInsight](#). For the detailed steps to create a cluster using the Azure Portal, see [Create Hadoop clusters in HDInsight](#).

Access control requirements

You must specify an Azure subscription when you create an HDInsight cluster. This cluster can be created in either a new Azure Resource group or an existing Resource group. You can use the following steps to verify your permissions for creating HDInsight clusters:

- To use an existing resource group.
 1. Sign in to the [Azure portal](#).
 2. Select **Resource groups** from the left menu to list the resource groups.
 3. Select the resource group you want to use for creating your HDInsight cluster.
 4. Select **Access control (IAM)**, and verify that you (or a group that you belong to) have at least the Contributor access to the resource group.
- To create a new resource group
 1. Sign in to the [Azure portal](#).
 2. Select **Subscription** from the left menu. It has a yellow key icon. You shall see a list of subscriptions.
 3. Select the subscription that you use to create clusters.
 4. Select **My permissions**. It shows your **role** on the subscription. You need at least Contributor access to create HDInsight cluster.

If you receive the NoRegisteredProviderFound error or the MissingSubscriptionRegistration error, see [Troubleshoot common Azure deployment errors with Azure Resource Manager](#).

List and show clusters

1. Sign in to <https://portal.azure.com>.
2. Select **More Services** from the left menu.
3. Search for HDInsight and select the entry that appears.
4. In the listing, find the cluster by name. If the cluster list is long, you can use filter on the top of the page.
5. Select a cluster from the list to see the overview page:

The screenshot shows the Azure HDInsight Cluster Overview page for a cluster named 'jgaohadoop1215'. The left sidebar contains a navigation menu with sections like Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Settings (Locks, Automation script), Getting Started (Quick Start, Tools for HDInsight), and Configuration (Cluster Login, Subscription Cores Usage). The main content area is titled 'Essentials' and shows cluster details: Resource group (jgaohadoop1215rg), Status (Running), Location (East US 2), Subscription name (jgaohadoop1215), and Subscription ID. Below this is a 'Quick Links' section with buttons for Cluster Dashboard, Ambari Views, and Scale Cluster. The 'Usage' section shows 'Cluster nodes' with a count of 4 nodes. A table provides node details: Head (D3, 8 cores, 2 nodes) and Worker (D3, 8 cores, 2 nodes).

Overview menu:

- **Dashboard:** Opens the cluster dashboard, which is Ambari Web for Linux-based clusters.
- **Secure Shell:** Shows the instructions to connect to the cluster using Secure Shell (SSH) connection.
- **Scale Cluster:** Allows you to change the number of worker nodes for this cluster.
- **Delete:** Deletes the cluster.

Left menu:

- **Activity logs:** Show and query logs of Azure Resource Manager activity.
- **Access control (IAM):** Use role assignments. See [Use role assignments to manage access to your Azure subscription resources](#).
- **Tags:** Allows you to set key/value pairs to define a custom taxonomy of your cloud services. For example, you may create a key named **project**, and then use a common value for all services associated with a specific project.
- **Diagnose and solve problems:** Display troubleshooting information.
- **Locks:** Add lock to prevent the cluster being modified or deleted.
- **Automation script:** Display and export the Azure Resource Manager template for the cluster. Currently, you can only export the dependent Azure storage account. See [Create Linux-based Hadoop clusters in HDInsight using Azure Resource Manager templates](#).
- **Quick Start:** Displays information that will help you get started using HDInsight.
- **Tools for HDInsight:** Help information for HDInsight related tools.
- **Cluster Login:** Display the cluster login information.
- **Subscription Core Usage:** Display the used and available cores for your subscription.

- **Scale Cluster:** Increase and decrease the number of cluster worker nodes. See [Scale clusters](#).
- **Secure Shell:** Shows the instructions to connect to the cluster using Secure Shell (SSH) connection. For more information, see [Use SSH with HDInsight](#).
- **HDInsight Partner:** Add/remove the current HDInsight Partner.
- **External Metastores:** View the Hive and Oozie metastores. The metastores can only be configured during the cluster creation process. See [use Hive/Oozie metastore](#).
- **Script Actions:** Run Bash scripts on the cluster. See [Customize Linux-based HDInsight clusters using Script Action](#).
- **Applications:** Add/remove HDInsight applications. See [Install custom HDInsight applications](#).
- **Properties:** View the cluster properties.
- **Storage accounts:** View the Azure Storage account or Azure Data Lake Store accounts configured during the cluster creation process.
- **Data lake store access:** View the configured Azure Active Directory principal used to represent the cluster when accessing Azure Data Lake Store.
- **New support request:** Allows you to create a support ticket with Microsoft support.

6. Click **Properties**:

The properties are:

- **Hostname:** Cluster name.
- **Cluster URL:** The URL for the Ambari web interface.
- **Secure shell (SSH):** The username and host name to use in accessing the cluster via SSH.
- **Status:** Include Aborted, Accepted, ClusterStorageProvisioned, AzureVMConfiguration, HDInsightConfiguration, Operational, Running, Error, Deleting, Deleted, Timedout, DeleteQueued, DeleteTimedout, DeleteError, PatchQueued, CertRolloverQueued, ResizeQueued, ClusterCustomization
- **Region:** Azure location. For a list of supported Azure locations, see the **Region** dropdown list box on [HDInsight pricing](#).
- **Date created:** The date the cluster was deployed.
- **Operating system:** Either **Windows** or **Linux**.
- **Type:** Hadoop, HBase, Storm, Spark.
- **Version.** See [HDInsight versions](#)
- **Subscription:** Subscription name.
- **Default data source:** The default cluster file system.
- **Worker nodes size:** The selected VM size of the worker nodes.
- **Head node size:** The selected VM size of the head nodes.
- **Virtual network:** The name of the Virtual Network and subnet to which the cluster is deployed, if one was selected at deployment time.

Delete clusters

Delete a cluster will not delete the default storage account or any linked storage accounts. You can re-create the cluster by using the same storage accounts and the same metastores. It is recommended to use a new default Blob container when you re-create the cluster.

1. Sign in to the [Portal](#).
2. Click **HDInsight Clusters** from the left menu. If you don't see **HDInsight Clusters**, click **More services** first.
3. Click the cluster that you want to delete.
4. Click **Delete** from the top menu, and then follow the instructions.

See also [Pause/shut down clusters](#).

Scale clusters

The cluster scaling feature allows you to change the number of worker nodes used by a cluster that is running in Azure HDInsight without having to re-create the cluster.

NOTE

Only clusters with HDInsight version 3.1.3 or higher are supported. If you are unsure of the version of your cluster, you can check the Properties page. See [List and show clusters](#).

The impact of changing the number of data nodes for each type of cluster supported by HDInsight:

- Hadoop

You can seamlessly increase the number of worker nodes in a Hadoop cluster that is running without impacting any pending or running jobs. New jobs can also be submitted while the operation is in progress. Failures in a scaling operation are gracefully handled so that the cluster is always left in a functional state.

When a Hadoop cluster is scaled down by reducing the number of data nodes, some of the services in the cluster are restarted. This causes all running and pending jobs to fail at the completion of the scaling operation. You can, however, resubmit the jobs once the operation is complete.

- HBase

You can seamlessly add or remove nodes to your HBase cluster while it is running. Regional Servers are automatically balanced within a few minutes of completing the scaling operation. However, you can also manually balance the regional servers by logging into the headnode of cluster and running the following commands from a command prompt window:

```
>pushd %HBASE_HOME%\bin  
>hbase shell  
>balancer
```

For more information on using the HBase shell, see [\[\]](#)

- Storm

You can seamlessly add or remove data nodes to your Storm cluster while it is running. But after a successful completion of the scaling operation, you will need to rebalance the topology.

Rebalancing can be accomplished in two ways:

- Storm web UI
- Command-line interface (CLI) tool

Please refer to the [Apache Storm documentation](#) for more details.

The Storm web UI is available on the HDInsight cluster:

Storm UI

Topology summary

Name	Id	Status	Uptime	Num workers
twitterCount	twitterCount-13-1422559680	ACTIVE	9h 19m 43s	1

Topology actions

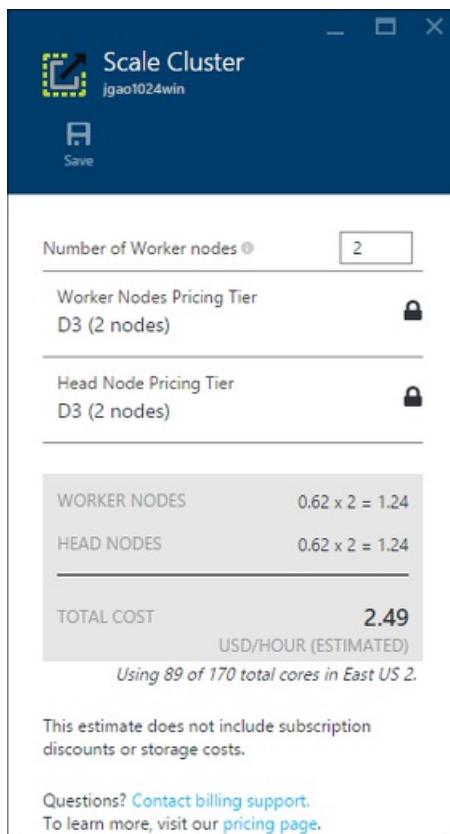
Activate Deactivate Rebalance Kill

Here is an example how to use the CLI command to rebalance the Storm topology:

```
## Reconfigure the topology "mytopology" to use 5 worker processes,
## the spout "blue-spout" to use 3 executors, and
## the bolt "yellow-bolt" to use 10 executors
$ storm rebalance mytopology -n 5 -e blue-spout=3 -e yellow-bolt=10
```

To scale clusters

1. Sign in to the [Portal](#).
2. Click **HDInsight Clusters** from the left menu.
3. Click the cluster you want to scale.
4. Click **Scale Cluster**.
5. Enter **Number of Worker nodes**. The limit on the number of cluster node varies among Azure subscriptions. You can contact billing support to increase the limit. The cost information will reflect the changes you have made to the number of nodes.



Pause/shut down clusters

Most of Hadoop jobs are batch jobs that are only run occasionally. For most Hadoop clusters, there are large periods of time that the cluster is not being used for processing. With HDInsight, your data is stored in Azure Storage, so you can safely delete a cluster when it is not in use. You are also charged for an HDInsight cluster, even when it is not in use. Since the charges for the cluster are many times more than the charges for storage, it makes economic sense to delete clusters when they are not in use.

There are many ways you can program the process:

- User Azure Data Factory. See [Create on-demand Linux-based Hadoop clusters in HDInsight using Azure Data Factory](#) for creating on-demand HDInsight linked services.
- Use Azure PowerShell. See [Analyze flight delay data](#).
- Use Azure CLI. See [Manage HDInsight clusters using Azure CLI](#).
- Use HDInsight .NET SDK. See [Submit Hadoop jobs](#).

For the pricing information, see [HDInsight pricing](#). To delete a cluster from the Portal, see [Delete clusters](#)

Change passwords

An HDInsight cluster can have two user accounts. The HDInsight cluster user account (A.K.A. HTTP user account) and the SSH user account are created during the creation process. You can use the Ambari web UI to change the cluster user account username and password, and script actions to change the SSH user account

Change the cluster user password

You can use the Ambari Web UI to change the Cluster user password. To log into Ambari, you must use the existing cluster username and password.

NOTE

If you change the cluster user (admin) password, this may cause script actions run against this cluster to fail. If you have any persisted script actions that target worker nodes, these may fail when you add nodes to the cluster through resize operations. For more information on script actions, see [Customize HDInsight clusters using script actions](#).

1. Sign in to the Ambari Web UI using the HDInsight cluster user credentials. The default username is **admin**. The URL is <https://<HDInsight Cluster Name>.azurehdinsight.net>.
2. Click **Admin** from the top menu, and then click "Manage Ambari".
3. From the left menu, click **Users**.
4. Click **Admin**.
5. Click **Change Password**.

Ambari then changes the password on all nodes in the cluster.

Change the SSH user password

1. Using a text editor, save the following text as a file named **changepassword.sh**.

IMPORTANT

You must use an editor that uses LF as the line ending. If the editor uses CRLF, then the script will not work.

```
#!/bin/bash
USER=$1
PASS=$2

usermod --password $(echo $PASS | openssl passwd -1 -stdin) $USER
```

2. Upload the file to a storage location that can be accessed from HDInsight using an HTTP or HTTPS address. For example, a public file store such as OneDrive or Azure Blob storage. Save the URI (HTTP or HTTPS address,) to the file, as this is needed in the next step.
3. From the Azure portal, click **HDInsight Clusters**.
4. Click your HDInsight cluster.
5. Click **Script Actions**.
6. From the **Script Actions** blade, select **Submit New**. When the **Submit script action** blade appears, enter the following information.

FIELD	VALUE
Name	Change ssh password

FIELD	VALUE
Bash script URI	The URI to the changepassword.sh file
Nodes (Head, Worker, Nimbus, Supervisor, Zookeeper, etc.)	<input checked="" type="checkbox"/> for all node types listed
Parameters	Enter the SSH user name and then the new password. There should be one space between the user name and the password.
Persist this script action ...	Leave this field unchecked.

- Select **Create** to apply the script. Once the script finishes, you will be able to connect to the cluster using SSH with the new password.

Grant/revoke access

HDInsight clusters have the following HTTP web services (all of these services have RESTful endpoints):

- ODBC
- JDBC
- Ambari
- Oozie
- Templeton

By default, these services are granted for access. You can revoke/grant the access using [Azure CLI](#) and [Azure PowerShell](#).

Find the subscription ID

To find your Azure subscription IDs

- Sign in to the [Portal](#).
- Click **Subscriptions**. Each subscription has a name and an ID.

Each cluster is tied to an Azure subscription. The subscription ID is shown on the cluster **Essential** tile. See [List and show clusters](#).

Find the resource group

In the Azure Resource Manager mode, each HDInsight cluster is created with an Azure Resource Manager group. The Resource Manager group that a cluster belongs to appears in:

- The cluster list has a **Resource Group** column.
- Cluster **Essential** tile.

See [List and show clusters](#).

Find the default storage account

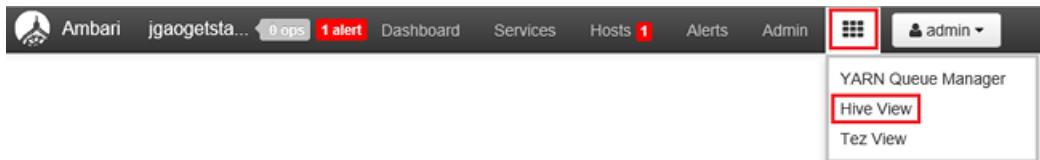
Each HDInsight cluster has a default storage account. The default storage account and its keys for a cluster appears under **Storage Accounts**. See [List and show clusters](#).

Run Hive queries

You cannot run Hive job directly from the Azure portal, but you can use the Hive View on Ambari Web UI.

To run Hive queries using Ambari Hive View

1. Sign in to the Ambari Web UI using the HDInsight cluster user credentials. The default username is **admin**. The URL is <https://<HDInsight Cluster Name>.azurehdinsight.net>.
2. Open Hive View as shown in the following screenshot:



3. Click **Query** from the top menu.
4. Enter a Hive query in **Query Editor**, and then click **Execute**.

Monitor jobs

See [Manage HDInsight clusters by using the Ambari Web UI](#).

Browse files

Using the Azure portal, you can browse the content of the default container.

1. Sign in to <https://portal.azure.com>.
2. Click **HDInsight Clusters** from the left menu to list the existing clusters.
3. Click the cluster name. If the cluster list is long, you can use filter on the top of the page.
4. Click **Storage Accounts** from the cluster left menu.
5. click a storage account.
6. Click the **Blobs** tile.
7. Click the default container name.

Monitor cluster usage

The **Usage** section of the HDInsight cluster blade displays information about the number of cores available to your subscription for use with HDInsight, as well as the number of cores allocated to this cluster and how they are allocated for the nodes within this cluster. See [List and show clusters](#).

IMPORTANT

To monitor the services provided by the HDInsight cluster, you must use Ambari Web or the Ambari REST API. For more information on using Ambari, see [Manage HDInsight clusters using Ambari](#)

Connect to a cluster

- [Use Hive with HDInsight](#)
- [Use SSH with HDInsight](#)

Next steps

In this article, you have learned how to create an HDInsight cluster by using the Portal, and how to open the

Hadoop command-line tool. To learn more, see the following articles:

- [Administer HDInsight Using Azure PowerShell](#)
- [Administer HDInsight Using Azure CLI](#)
- [Create HDInsight clusters](#)
- [Use Hive in HDInsight](#)
- [Use Pig in HDInsight](#)
- [Use Sqoop in HDInsight](#)
- [Get Started with Azure HDInsight](#)
- [What version of Hadoop is in Azure HDInsight?](#)
- [Read more about using the Ambari Web UI](#)
- [Details on using the Ambari REST API](#)

Changing and Optimizing Configs via Ambari

8/16/2017 • 16 min to read • [Edit Online](#)

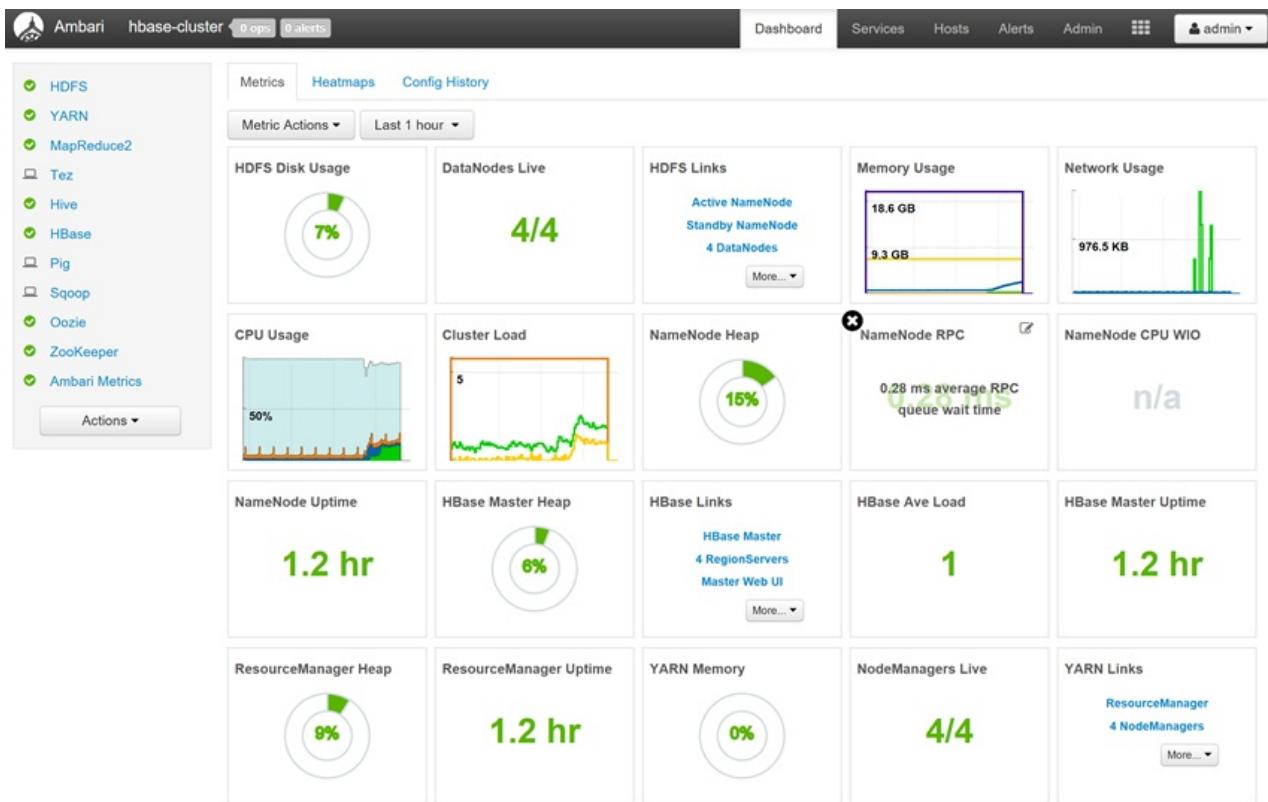
Overview

HDInsight allows for the creation of Apache Hadoop clusters for large-scale data processing applications. Managing and monitoring multinode complex clusters is a tedious job. [Apache Ambari](#) is a web interface to easily manage and monitor HDInsight Linux clusters. The Ambari web interface is only available with Linux clusters. For Windows clusters, the Ambari [REST API](#) can be used.

In this article, you'll learn how to use the Ambari web user interface to manage and optimize configurations of an HDInsight Linux cluster.

For an introduction to using the Ambari Web UI, take a look at [Manage HDInsight clusters by using the Ambari Web UI](#)

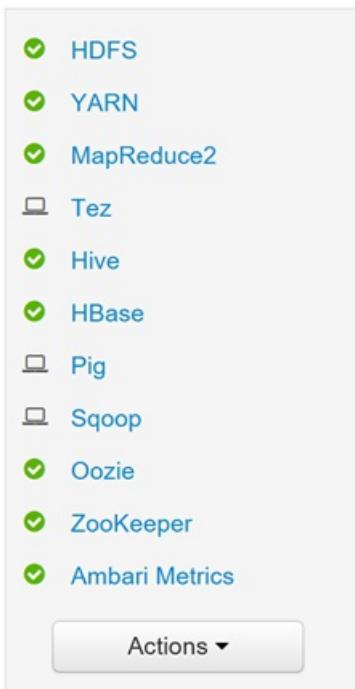
When you log in to its web interface ([HTTPS://CLUSTERNAME.azurehdinsight.net](https://CLUSTERNAME.azurehdinsight.net)), Ambari displays a useful dashboard that gives you a great overview of your cluster at-a-glance.



The Ambari web UI can be used to manage hosts, services, alerts, configurations, and views. It can't be used to create an HDInsight cluster, upgrade services, manage stacks and versions, decommission or recommission hosts, or add services to the cluster.

Manage your cluster's configuration

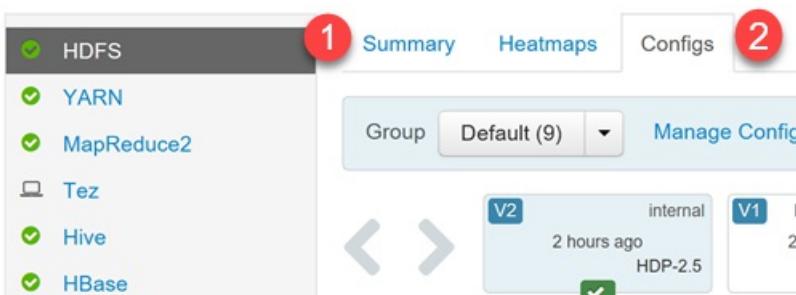
Configuration settings help tune a particular service. To modify the configuration setting of a service, select the service from the **Services** sidebar (on the left), and then navigate to the **Configs** tab in the service detail page.



Modify NameNode Java heap size

To modify the NameNode Java heap size, follow the steps below:

1. Select **HDFS** from the Services sidebar and navigate to the **Configs** tab.



1. Find the setting **NameNode Java heap size**. You can also use the **filter** text box to type and find a particular setting. Click the **pen** icon beside the setting name.



1. Type the new value in the text box, and then press **Enter** to save the change.

NameNode Java heap size

 MB

1. Note that the NameNode Java heap size is changed to 2 GB from 1 GB.



Note: The NameNode Java heap size depends on many factors such as load on the cluster, number of files, and number of blocks. The default size of 1 GB works well with most clusters, although certain workloads may require modification.

1. Save your changes by clicking on the green **Save** button on the top of the configuration screen.



Hive optimization

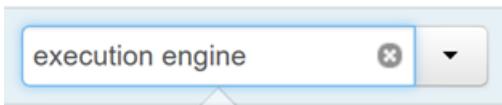
As mentioned earlier, each service has certain configuration parameters that can be easily modified using the Ambari web UI. In this section, we'll learn about important configuration options to optimize overall Hive performance.

1. To modify Hive configuration parameters, select **Hive** from the Services sidebar.
2. Navigate to the **Configs** tab.

Set the Hive execution engine

There are two execution engines: MapReduce and Tez. Tez is faster than MapReduce. HDInsight Linux clusters have Tez as the default execution engine. To change the execution engine, follow these steps.

1. In the Hive **Configs** tab, type **execution engine** in the filter box.

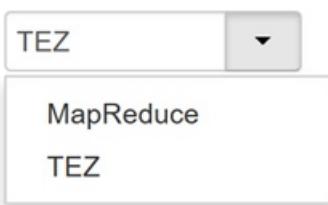


1. In the **Optimization** property, observe that the default value is **Tez**.

Optimization

Tez

Execution Engine



Tune mappers

Hadoop tries to split a single file into multiple files and process the resulting files in parallel. The number of mappers depends on the number of splits. The following two configuration parameters drive the number of splits for the Tez execution engine:

- `tez.grouping.min-size` : Lower limit on the size of a grouped split (default value of 16,777,216 bytes).
- `tez.grouping.max-size` : Upper limit on the size of a grouped split (default value of 1,073,741,824 bytes).

As a performance rule of thumb, decrease both of these parameters to improve latency, increase for more throughput.

For example, to set four mapper tasks for a data size of 128 MB, you would set both parameters to 32 MB each (33,554,432 bytes).

1. Modify the above configuration parameters by navigating to the **Configs** tab of the Tez service. Expand the General panel, and then locate the `tez.grouping.max-size` and `tez.grouping.min-size` parameters.
2. Set both parameters to **33,554,432** bytes (32 MB).

tez.grouping.max-size	33554432	+	o	c
tez.grouping.min-size	33554432	+	o	c

Note: The changes made here will affect all Tez jobs across the server. The parameter values should be carefully modified in order to get the optimal result.

Tune reducers

ORC and Snappy both offer high performance. However, Hive may choose too few reducers by default, causing bottlenecks.

For example, say you have an input data size of 50GB. That data in ORC format with Snappy compression is 1GB. Hive estimates the number of reducers needed as:

(number of bytes input to mappers / `hive.exec.reducers.bytes.per.reducer`)

With the default settings, this means 4 reducers in our scenario.

Thus, the `hive.exec.reducers.bytes.per.reducer` parameter specifies the number of bytes processed per reducer. The default value is 64 MB. Tuning this value down will increase parallelism and may improve performance. Tuning it too low could also cause too many reducers, potentially adversely affecting performance. You will need to adjust this setting based on your particular data requirements, compression settings, and other environmental factors.

1. To modify the parameter, navigate to the Hive **Configs** tab and find the **Data per Reducer** parameter on the Settings page.

Data per Reducer



1. Select **Edit** to modify the value to 128 MB (134217728 bytes), and then press **Enter** to save.

Data per Reducer

134217728	B	↻
-----------	---	---

Given an input size of 1,024 MB, with 128 MB of data per reducer, there will be 1024/128, or 8 reducers.

1. An invalid or wrong value for the Data per Reducer parameter may result in a large number of reducers, adversely affecting query performance. To limit the maximum number of reducers, set `hive.exec.reducers.max` to an appropriate value. The default value is 1009.

Enable parallel execution

A Hive query is executed in one or more stages. If the independent stages can be run in parallel, this will increase query performance.

1. To enable parallel query execution, navigate to the Hive **Config** tab and search the `hive.exec.parallel` property. The default value is false. Change the value to **true**, and then press **Enter** to save the value.
2. To limit the number of jobs to be run in parallel, modify the `hive.exec.parallel.thread.number` property. The default value is 8.

The screenshot shows the Hive Configuration interface. It includes two sections: 'General' and 'Advanced hive-site'. In the 'General' section, there is a configuration entry for `hive.exec.parallel.thread.number` with a value of 8. In the 'Advanced hive-site' section, there is a configuration entry for `hive.exec.parallel` with a value of true.

Section	Configuration	Value
General	<code>hive.exec.parallel.thread.number</code>	8
	<code>hive.exec.parallel</code>	true
Advanced hive-site	<code>hive.exec.parallel</code>	true

Enable vectorization

Hive processes data row by row. Vectorization enables Hive to process data in blocks of 1,024 rows instead of one row at a time.

1. To enable a vectorized query execution, navigate to the Hive **Configs** tab and search for the `hive.vectorized.execution.enabled` parameter. The default value is true for Hive 0.13.0 or later.
2. To enable vectorized execution for the reduce side of the query, set the `hive.vectorized.execution.reduce.enabled` parameter to **true**. The default value is false.

The screenshot shows the Hive Configuration interface with the 'Performance' tab selected. It contains two configuration entries: 'Enable Reduce Vectorization' and 'Enable Vectorization and Map Vectorization', both of which have their values set to true.

Configuration	Value
Enable Reduce Vectorization	true
Enable Vectorization and Map Vectorization	true

Note: Vectorization is only applicable to the ORC file format.

Enable cost-based optimization (CBO)

Hive follows a set of rules to find an optimal query execution plan, which represents an old technique. Cost-based optimization evaluates multiple plans to execute a query and assigns a cost to each plan. It then finds the cheapest plan to execute a query.

1. To enable CBO, navigate to the Hive **Configs** tab and filter the `hive.cbo.enable` parameter. Switch the toggle button to **On** to enable CBO.

CBO

Enable Cost Based Optimizer

On

The following additional configuration parameters increase Hive query performance when CBO is enabled:

`hive.compute.query.using.stats`

When set to **true**, Hive uses stats stored in metastore to answer simple queries like `count(*)`.

▼ **Performance**

Compute simple queries using stats only true

`hive.stats.fetch.column.stats`

Column statistics are created when CBO is enabled. Hive uses column statistics, which are stored in metastore, to optimize queries. Fetching column statistics for each column takes longer when the number of columns is high. When set to **false**, this setting disables fetching column statistics from the metastore.

CBO

Fetch column stats at compiler

On

`hive.stats.fetch.partition.stats`

Basic partition statistics such as number of rows, data size, and file size are stored in metastore. When set to **true**, the partition stats are fetched from metastore. When false, the file size is fetched from the file system, and the number of rows is fetched from row schema.

▼ **Advanced hive-site**

Fetch partition stats at compiler true

Enable intermediate compression

Map tasks create intermediate files that are used by the reducer tasks. Intermediate compression shrinks the intermediate file size.

Hadoop jobs are usually I/O bottlenecked. Compressing data can speed up I/O and overall network transfer.

Here are the various available compression types:

FORMAT	TOOL	ALGORITHM	FILE EXTENSION	SPLITTABLE?
Gzip	Gzip	DEFLATE	.gz	No
Bzip2	Bzip2	Bzip2	.bz2	Yes
LZO	Lzop	LZO	.lzo	Yes, if indexed
Snappy	N/A	Snappy	Snappy	No

As a general rule, having the compression method splittable is important, otherwise very few mappers will be created. If the input data is text, `bzip2` is the best option since it's splittable. For ORC format, Snappy is the fastest compression option.

1. To enable intermediate compression, navigate to the Hive **Configs** tab, and then set the `hive.exec.compress.intermediate` parameter to **true**. The default value is false.

Note: To compress intermediate files, choose a compression codec with lower CPU cost, even if it doesn't have a high compression output.

The screenshot shows a configuration pane for 'Advanced hive-site'. A property 'hive.exec.compress.intermediate' is listed with its value set to 'true'.

1. To set the intermediate compression codec, add the custom property `mapred.map.output.compression.codec` to the `hive-site.xml` or `mapred-site.xml` file.
2. To add a custom setting:
 - a. Navigate to the Hive **Configs** tab and select the **Advanced** tab.
 - b. Under the Advanced tab, find and expand the **Custom hive-site** pane.
 - c. Click the link **Add Property** at the bottom of the Custom hive-site pane.
 - d. In the Add Property window, enter `mapred.map.output.compression.codec` as the key and `org.apache.hadoop.io.compress.SnappyCodec` as the value.
 - e. Click **Add**.

Add Property

Type	hive-site.xml
Key	mapred.map.output.compression.codec
Value	org.apache.hadoop.io.compress.SnappyCodec

This will compress the intermediate file using Snappy compression. Once the property is added, it will appear in the Custom hive-site pane.

Note: This modifies the `$HADOOP_HOME/conf/hive-site.xml` file.

Compress final output

The final Hive output can also be compressed.

1. To compress the final Hive output, navigate to the Hive **Configs** tab, and then set the `hive.exec.compress.output` parameter to **true**. The default value is false.
2. To choose the output compression codec, add the `mapred.output.compression.codec` custom property to the Custom hive-site pane, as explained above.

Add Property

Type	hive-site.xml
Key	mapred.output.compression.codec
Value	org.apache.hadoop.io.compress.SnappyCodec

Enable speculative execution

Speculative execution launches a certain number of duplicate tasks in order to detect and blacklist the slow-running task tracker, while improving the overall job execution by optimizing individual task results.

1. To enable speculative execution, navigate to the Hive **Configs** tab, and then set the `hive.mapred.reduce.tasks.speculative.execution` parameter to **true**. The default value is false.

Advanced hive-site

hive.mapred.reduce.
tasks.speculative.
execution true

Note: Speculative execution shouldn't be turned on for long-running MapReduce tasks with large amounts of input.

Tune dynamic partitions

Hive allows for creating dynamic partitions when inserting records into a table, without predefining each and every

partition. This is a powerful feature, although it may result in the creation of a large number of partitions and an accordingly large number of files for each partition.

1. For Hive to do dynamic partitions, the `hive.exec.dynamic.partition parameter` value should be **true**. The default value is true.
2. Change the dynamic partition mode to strict. In strict mode, at least one partition has to be static. This prevents queries without the partition filter in the WHERE clause. Therefore, it prevents queries that scan all partitions. Navigate to the Hive **Configs** tab, and then set `hive.exec.dynamic.partition.mode` to **strict**. The default value is nonstrict.
3. To limit the number of dynamic partitions to be created, modify the ``hive.exec.max.dynamic.partitions`` parameter. The default value is 5,000.
4. To limit the total number of dynamic partitions per node, modify `hive.exec.max.dynamic.partitions.pernode`. The default value is 2,000.

Enable local mode

Local mode enables Hive to perform all tasks of a job on a single machine, or sometimes in a single process. This improves query performance if the input data is small and the overhead of launching tasks for queries consumes a significant percentage of the overall query execution.

1. To enable local mode, add the `hive.exec.mode.local.auto` parameter to the Custom hive-site panel, as explained earlier.

Add Property

Type	<input type="text" value="hive-site.xml"/>
Key	<input type="text" value="hive.exec.mode.local.auto"/>
Value	<input type="text" value="true"/>

Set single MapReduce MultiGROUP BY

When this property is set to true, a MultiGROUP BY query with common group by keys will generate a single MapReduce job.

1. To enable this, add the `hive.multigrouby.singlereducer` parameter to the Custom hive-site pane, as explained earlier.

Add Property

Type	<input type="text" value="hive-site.xml"/>
Key	<input type="text" value="hive.multigrouby.singlereducer"/>
Value	<input type="text" value="true"/>

Additional Hive optimizations

Now that you understand how to go about finding and updating configurations in Ambari, here is a list of additional Hive-related optimizations you can set:

Join Optimizations

The default join type in Hive is a **shuffle join**. The way joining works in Hive, is that special mappers read the input, emit a join key/value pair to an intermediate file. Hadoop sorts and merges these pairs in a shuffle stage. This shuffle stage is expensive. Thus, selecting the right Join based on your data can significantly improve performance.

JOIN TYPE	WHEN	HOW	HIVE SETTINGS	COMMENTS
Shuffle Join	<ul style="list-style-type: none"> Default choice Always works 	<ul style="list-style-type: none"> Reads from part of one of the tables Buckets and sorts on Join key Sends one bucket to each reduce Join is done on the Reduce side 	No significant Hive setting needed	Works every time
Map Join	<ul style="list-style-type: none"> One table can fit in memory 	<ul style="list-style-type: none"> Reads small table into memory hash table Streams through part of the big file Joins each record from hash table Joins will be performed by the mapper alone 	<code>hive.auto.convert.join</code>	Very fast, but limited
Sort Merge Bucket	If both tables are: <ul style="list-style-type: none"> Sorted the same Bucketed the same Joining on the sorted/bucket ed column 	Each process: <ul style="list-style-type: none"> Reads a bucket from each table Processes the row with the lowest value 	<code>hive.auto.convert.sortmergejoin</code>	Very efficient

Execution Engine Optimizations

Additional recommendations for optimizing the Hive execution engine:

SETTING	RECOMMENDED	HDIE DEFAULT
<code>hive.mapjoin.hybridgrace.hashtable</code>	true = safer, slower; false = faster	false
<code>tez.am.resource.memory.mb</code>	4GB upper bound for most	Auto-Tuned

SETTING	RECOMMENDED	HDI DEFAULT
<code>tez.session.am.dag.submit.timeout.secs</code>	300+	300
<code>tez.am.container.idle.release-timeout-min.millis</code>	20000+	10000
<code>tez.am.container.idle.release-timeout-max.millis</code>	40000+	20000

Pig optimization

Pig properties can be easily modified from the Ambari web UI to tune Pig queries. Modifying Pig properties from Ambari directly modifies the Pig properties in the `/etc/pig/2.4.2.0-258.0/pig.properties` file.

1. To modify Pig properties, navigate to the Pig **Configs** tab, and then expand the **Advanced pig-properties** pane.
2. Find, uncomment, and change the value of the property you wish to modify.
3. Select **Save** on the top right side of the window to save the new value. Some properties may require a service restart.

```
# Licensed to the Apache Software Foundation (ASF) under one
# or more contributor license agreements. See the NOTICE file
# distributed with this work for additional information
# regarding copyright ownership. The ASF licenses this file
# to you under the Apache License, Version 2.0 (the
# "License"); you may not use this file except in compliance
# with the License. You may obtain a copy of the License at
#
# http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing,
```

Note: The session-level settings override property values in the `pig.properties` file.

Tune execution engine

Two execution engines are available to execute Pig scripts: MapReduce and Tez. Tez is an optimized engine and is much faster than MapReduce.

1. To modify the execution engine, in the Advanced pig-properties pane, find the property `exec-type`.
2. The default value is MapReduce. Change it to **Tez**.

Enable local mode

Similar to Hive, local mode is used to speed jobs with relatively less amounts of data.

1. To enable the local mode, set `pig.auto.local.enabled` to **true**. The default value is false.
2. Jobs with an input data size less than the `pig.auto.local.input.maxbytes` property value are considered to be small jobs. The default value is 1 GB.

Copy user jar cache

Pig copies the jar required by UDFs to a distributed cache in order to make them available for task nodes. These jars do not change frequently. If enabled, this setting allows jars to be placed in a cache to reuse them for jobs run by the same user. This results in a minor increase in job performance.

1. To enable, set `pig.user.cache.enabled` to **true**. The default is false.
2. To set the base path of the cached jars, set `pig.user.cache.location` to the base path. The default is `/tmp`.

Optimize performance with memory settings

The following memory settings can help optimize Pig script performance.

1. `pig.cachedbag.memusage` : The amount of memory allocated to a bag. A bag is collection of tuples. A tuple is an ordered set of fields, and a field is a piece of data. If the data in a bag is beyond the allocated memory, it is spilled to disk. The default value is 0.2, which represents 20 percent of available memory. This memory is shared across all bags in an application.
2. `pig.spill.size.threshold` : Bags smaller than the spill size threshold (bytes) are not spilled to disk. The default value is 5 MB.

Compress temporary files

Pig generates temporary files during job execution. Compressing the temporary files results in a performance increase when reading or writing files to disk. The following settings can be used to compress temporary files.

- `pig.tmpfilecompression` : When true, enables temporary file compression. (Default value is false.)
- `pig.tmpfilecompression.codec` : The compression codec to use for compressing the temporary files.

Note: The recommended compression codecs are LZO and Snappy because of lower CPU utilization.

Enable split combining

When enabled, small files are combined for fewer map tasks. This improves the efficiency of jobs with many small files. To enable, set `pig.noSplitCombination` to **true**. The default value is false.

Tune mappers

The number of mappers can be controlled by modifying the property `pig.maxCombinedSplitSize`. This specifies the size of the data to be processed by a single map task. The default value is the filesystems default block size. Increasing this value will result in a decrease of the number of mapper tasks.

Tune reducers

The number of reducers is calculated based on the parameter `pig.exec.reducers.bytes.per.reducer`. The parameter specifies the number of bytes processed per reducer. The default value is 1 GB. To limit the maximum number of reducers, set the `pig.exec.reducers.max` property. The default value is 999.

HBase optimization with the Ambari web UI

HBase configuration can be easily modified from the HBase Configs tab. In this section, we'll look at some of the important configuration settings that affect HBase performance.

Set HBASE_HEAPSIZE

This specifies the maximum amount of heap to be used in megabytes by **region** and **master** servers. The default value is 1,000 MB. This should be tuned as per the cluster workload.

1. To modify, navigate to the **Advanced HBase-env** pane in the HBase **Configs** tab, and then find the `HBASE_HEAPSIZE` setting.

2. Change the default value to 5,000 MB.

Advanced hbase-env

hbase-env template

```
# The maximum amount of heap to use, in MB. Default is 1000.  
# export HBASE_HEAPSIZE=5000  
  
# Extra Java runtime options
```

Optimize read-heavy workloads

The following configurations are important to improve the performance of read-heavy workloads.

Block cache size

The block cache is the read cache. This is controlled by the `hfile.block.cache.size` parameter. The default value is 0.4, which is 40 percent of the total region server memory. The larger the block cache size, the faster the random reads will be.

1. To modify this parameter, navigate to the **Settings** tab in the HBase **Configs** tab, and then locate **% of RegionServer Allocated to Read Buffers**.



1. Click the **Edit** icon to change the value.

Memstore size

All edits are stored in the memory buffer, or Memstore. This increases the total amount of data that can be written to disk in a single operation, and it speeds subsequent access to the recent edits. The Memstore size is defined by the following two parameters:

- `hbase.regionserver.global.memstore.UpperLimit`: Defines the maximum percentage of the region server that Memstore combined can use.
- `hbase.regionserver.global.memstore.LowerLimit`: Defines the minimum percentage of the region server that Memstore combined can use.

To optimize for random reads, you can reduce the Memstore upper and lower limits using these parameters.

Number of rows fetched when scanning from disk

This setting defines the number of rows read from disk when the next method is called on a scanner. This is defined by the parameter `hbase.client.scanner.caching`. The default value is 100. The higher the number, the fewer the remote calls made from the client to the region server, resulting in faster scans. However, this will also increase memory pressure on the client.

Number of Fetched Rows when Scanning from Disk

 rows × lock refresh cancel

Note: Do not set the values such that the time between invocation of the next method on a scanner is greater than the scanner timeout. The scanner timeout is defined by the `hbase.regionserver.lease.period` property.

Optimize write-heavy workloads

The following configurations are important to improve the performance of write-heavy workloads.

Maximum region file size

The property `hbase.hregion.maxfilesize` defines the size of a single HFile for a region. HBase stores the data in an internal file format, or HFile. A region is split into two regions if the sum of all HFiles in a region is greater than this setting.

Disk

Maximum Region File Size



The larger the region file size, the fewer number of splits. Ideally, you can increase the value and settle for the one that gets you the maximum write performance.

Avoid update blocking

The property `hbase.hregion.memstore.flush.size` defines the size at which Memstore will be flushed to disk. The default size is 128 MB.

The Hbase region block multiplier is defined by `hbase.hregion.memstore.block.multiplier`. The default value is 4. The maximum allowed is 8.

HBase blocks update if the Memstore is (`hbase.hregion.memstore.flush.size` * `hbase.hregion.memstore.block.multiplier`) bytes.

Considering the default values, updates are blocked when Memstore is of $128 * 4 = 512$ MB in size. To reduce the update blocking count, increase the value of `hbase.hregion.memstore.block.multiplier`.

Memstore Flush Size



HBase Region Block Multiplier



Define Memstore size

Memstore size is defined by the `hbase.regionserver.global.memstore.UpperLimit` and `hbase.regionserver.global.memstore.LowerLimit` parameters. Setting these values equal to each other reduces pauses during writes (also causing more frequent flushing) and results in increased write performance.

Set Memstore local allocation buffer

Defined by the property `hbase.hregion.memstore.mslab.enabled`. When enabled, this prevents heap fragmentation during heavy write operation. The default value is true.

Advanced hbase-site

hbase.hregion.memstore.
msslab.enabled

Next steps

- Read more about [managing HDInsight clusters by using the Ambari Web UI](#)
- Learn how to work with the Ambari [REST API](#)

Managing Logs for a HDInsight Cluster

8/16/2017 • 13 min to read • [Edit Online](#)

This article provides guidance on management of the myriad log files that a HDInsight cluster produces. Apache Hadoop and related services, such as Apache Spark, produce detailed job execution logs. Properly managing these log files is an important aspect of maintaining a healthy HDInsight cluster. The management of these files can also be dictated by regulatory requirements. Also, due to number and size of logs, you want to optimize log storage and archival for service cost management.

There are a set of general steps to take when managing HDInsight cluster logs. They include retaining information about all aspects of the environment. This includes, but is not limited to, all associated Azure Services, cluster configuration, job execution information and any error states. The most common steps taken in this process are listed below.

Managing HDInsight Cluster Log Steps

- Step 1: Determine log retention policies
- Step 2: Manage cluster service versions configuration logs
- Step 3: Manage cluster Job Execution Log Files
- Step 4: Forecast log volume storage sizes and costs
- Step 5: Determine log archive policies and processes

Step 1: Determine log retention policies

The first step in creating a HDInsight cluster(s) log management strategy is to gather information about business scenarios and job execution (log) history storage requirements. You will use this information to design and implement your cluster log management practices.

Cluster Details

The following cluster details are useful in helping to gather information in your log management strategy. It's important to gather this information from EVERY HDInsight cluster you have provisioned into the Azure account that are you are working with to when designing your log management strategy.

- Name of the cluster
- Region and availability zone the cluster was launched into.
- State of the cluster, including details of the last state change.
- Type and number of HDInsight instances specified for the master, core, and task nodes.

You can quickly get much of this top level information via the Azure portal. Alternatively, you can use the Azure cli to get information about your HDInsight cluster(s) by running the following commands:

```
azure hdinsight cluster list  
azure hdinsight cluster show <Cluster Name>
```

Or, you can use powershell to view this type of information. See [Manage Hadoop clusters in HDInsight by using Azure PowerShell](#) for details.

Understand the Workloads running on your clusters

It's important to understand workload types running on your HDInsight cluster(s) so that you can design appropriate associated logging strategies for each type of workload.

Listed below are some of the questions you can use to capture the needed information for planning.

- Are the workloads experimental (i.e. dev or test) or production-quality?
- How often do the production-quality workloads normally run?
- Are any of the workloads resource-intensive and/or particularly long-running?
- Do any of the workloads utilize a complex set of Hadoop services for which multiple types of logs would be produced?
- Do any of the workloads have associated regulatory execution lineage requirements?

Your expected output after you complete the information-gathering in this step is a written list of cluster(s) information, workload types and log retention requirements.

Example log retention patterns and practices

- Consider maintaining data lineage tracking by adding an identifier to each log entry, or through other techniques. This allows you to trace back the original source of the data and the operation, and follow it through each stage to understand its consistency and validity.
- Consider how you can collect logs from the cluster, or from more than one cluster, and collate them for purposes such as auditing, monitoring, planning, and alerting. You might use a custom solution to access and download the log files on a regular basis, and combine and analyze them to provide a dashboard-like display with additional capabilities for alerting for security or failure detection. Such utilities could be created using PowerShell, the HDInsight SDKs, or code that accesses the Azure Service Management API.
- Consider if a monitoring solution or service would be a useful benefit. A management pack for HDInsight is available for use with Microsoft System Center (see the Microsoft Download Center for more details). In addition, you can use third-party tools such as Chukwa and Ganglia to collect and centralize logs. Many companies offer services to monitor Hadoop-based big data solutions—some examples are Centerity, Compuware APM, Sematext SPM, and Zettaset Orchestrator.

Step 2: Manage cluster service versions and view Script Action logs

A typical HDInsight cluster uses a number of services and open-source software packages (such as Apache HBase, Apache Spark, etc...). For some workloads, such as bioinformatics, you may be required to retain service configuration log history in addition to job execution logs.

Viewing Cluster Configuration Settings with the Ambari UI

Apache Ambari simplifies the management and monitoring of a HDInsight cluster by providing an easy to use web UI and REST API. Ambari is included on Linux-based HDInsight clusters, and is used to monitor the cluster and make configuration changes. Click on the 'Cluster Dashboard' blade on the Azure Portal HDInsight page to open the 'Cluster Dashboards' link page. Next, click on the 'HDInsight cluster dashboard' blade to open the Ambari UI. You'll be prompted for your cluster login credentials.

Also, you can click the blade named 'Ambari Views' on the Azure portal page for HDInsight to open a list of service views. This list will vary, depending on which libraries you've installed. For example, you may see YARN Queue Manager, Hive View and Tez View, if you've installed these services. Click any service link of interest to drill down to see configuration and service information. The Ambari UI 'Stack and Version' page provides information about the cluster services configuration and service version history. In the Ambari UI, click on the 'Admin' menu and then on 'Stacks and Versions' to navigate to this section. Then click on the 'Versions' tab on the page to see service version information. An example is shown below.

Stack and Versions

Service Accounts

Kerberos

Service Auto Start

Service	Version	Status	Description
HDFS	2.7.3	Installed	Apache Hadoop Distributed File System
YARN	2.7.3	Installed	Apache Hadoop NextGen MapReduce (YARN)
MapReduce2	2.7.3	Installed	Apache Hadoop NextGen MapReduce (YARN)
Tez	0.7.0	Installed	Tez is the next generation Hadoop Query Processing framework written on top of YARN.
Hive	1.2.1000	Installed	Data warehouse system for ad-hoc queries & analysis of large datasets and table & storage management service
Pig	0.16.0	Installed	Scripting platform for analyzing large datasets
Sqoop	1.4.6	Installed	Tool for transferring bulk data between Apache Hadoop and structured data stores such as relational databases
Oozie	4.2.0	Installed	System for workflow coordination and execution of Apache Hadoop jobs. This also includes the installation of the optional Oozie Web Console which relies on and will install the ExtJS Library.
ZooKeeper	3.4.6	Installed	Centralized service which provides highly reliable distributed coordination

Additionally, using the Ambari UI, you can capture (by downloading) the configuration for any (or all) services running on a particular host (or node) in the cluster.

Do this by clicking on the 'Hosts' menu, then on link for the host of interest and then in that host's page, on the 'Host Actions' button and then on 'Download Client Configs'. An example screenshot is shown below.

Ambari hdiz-docs... 0 ops 0 alerts

hn0-hdiz-d.wugqkxcz2rbuvn0sm5ejkkyjxg.xx.internal.cloudapp.net

Back

Summary Configs Alerts 0 Versions

Host Actions ▾

- Start All Components
- Stop All Components
- Restart All Components
- Set Rack
- Turn On Maintenance Mode
- Delete Host
- Download Client Configs

HDFS

YARN
MapReduce2
Tez
Hive
HBase
Pig
Sqoop
Oozie
Ambari Metrics

Group Default (8) Change

Settings Advanced

NameNode

NameNode directories: /hadoop/hdfs/namenode

NameNode Java heap size: 1GB

All Clients On Host

- HBase Client
- HCat Client
- HDFS Client
- Hive Client
- MapReduce2 Client
- Oozie Client
- Pig Client
- Sqoop Client
- Tez Client
- YARN Client

Viewing the Script Action Logs

HDInsight [Script Actions](#) run scripts on the cluster manually or when specified. For example, they can be used to install additional software on the cluster or to alter configuration settings from the default values. These logs may provide insight into errors that occurred during set up of the cluster as well as configuration settings changes that could affect cluster performance and availability. You can view the status of a script action by clicking on the 'ops' button on your Ambari UI or by accessing them from the default storage account. The storage logs are available at `/STORAGE_ACCOUNT_NAME/DEFAULT_CONTAINER_NAME/custom-scriptaction-logs/CLUSTER_NAME\DATE`.

Step 3: Manage the Cluster Job execution log files

The next step is to review the job execution log files for the various services you may be using. Services could include Apache HBase, Apache Spark and many others. A Hadoop cluster produces a large number of verbose logs, so determining which log are useful (and which are not) can be time-consuming. Understanding the logging system is important so that you can target management of useful log files. For perspective, an example log is shown below.

Accessing the Hadoop-generated log files

HDInsight stores its log files in both the cluster file system and in Azure storage. You can examine log files in the cluster by opening an SSH connection to the cluster and browsing the file system or by using the Hadoop YARN Status portal on the remote head node server. You can examine the log files in Azure storage using any of the tools that can access and download data from Azure storage. Examples are AZCopy, CloudXplorer, and the Visual Studio Server Explorer. You can also use powershell and the Azure Storage Client libraries, or the Azure .NET SDKs, to access data in Azure blob storage.

Hadoop runs the work of the jobs in task attempts on various nodes in the cluster. HDInsight can initiate speculative task attempts, terminating the other task attempts that do not complete first. This generates significant activity that is logged to the controller, stderr and syslog log files as it happens. In addition, multiple tasks attempts are running simultaneously, but a log file can only display results linearly.

HDIInsight Logs written to Azure Tables

The logs written to Azure Tables provide one level of insight into what is happening with an HDInsight cluster.

When you create an HDInsight cluster, 6 tables are automatically created for Linux-based clusters in the default Table storage:

- hdinsightagentlog
 - syslog
 - daemonlog
 - hadoopservicelog
 - ambariserverlog

- ambariagentlog

HDInsight Logs Written to Azure Blob Storage

HDInsight clusters are configured to write task logs to an Azure Blob Storage account for any job that is submitted using the Azure PowerShell cmdlets or the .NET Job Submission APIs. If you submit jobs through SSH to the cluster then the execution logging information will be found in the Azure Tables discussed in the previous paragraph. In addition to the core log files, generated by HDInsight, services that are installed, such as YARN, will also generate job execution log files. The number and type of log files depends of the services installed. Common services are Apache HBase, Apache Spark and more. You will want to investigate the job log execution files for each service in order to understand the overall logging files available on your cluster. Each service has it's own unique methods of logging and locations for storing log files. As an example, detail about accessing the most common service log files (YARN) is discussed in the next section.

HDInsight Logs generated by YARN

YARN aggregates logs across all containers on a worker node and stores them as one aggregated log file per worker node. The log is stored on the default file system after an application finishes. Your application may use hundreds or thousands of containers, but logs for ALL containers run on a single worker node are always aggregated to a single file. So there is only one log per worker node used by your application. Log Aggregation is enabled by default on HDInsight clusters version 3.0 and above. Aggregated logs are located in default storage for the cluster. The following path is the HDFS path to the logs:

```
/app-logs/<user>/logs/<applicationId>
```

The aggregated logs are not directly readable, as they are written in a TFile, binary format indexed by container. Use the YARN ResourceManager logs or CLI tools to view these logs as plain text for applications or containers of interest.

YARN CLI tools

To use the YARN CLI tools, you must first connect to the HDInsight cluster using SSH. Specify the <applicationId>, <user-who-started-the-application>, <containerId>, and <worker-node-address> information when running these commands. You can view these logs as plain text by running one of the following commands:

```
yarn logs -applicationId <applicationId> -appOwner <user-who-started-the-application>
yarn logs -applicationId <applicationId> -appOwner <user-who-started-the-application> -containerId
<containerId> -nodeAddress <worker-node-address>
```

YARN ResourceManager UI

The YARN ResourceManager UI runs on the cluster headnode. It is accessed through the Ambari web UI. Use the following steps to view the YARN logs:

1. In your web browser, navigate to <https://CLUSTERNAME.azurehdinsight.net>. Replace CLUSTERNAME with the name of your HDInsight cluster.
2. From the list of services on the left, select YARN. Yarn service selected
3. From the Quick Links dropdown, select one of the cluster head nodes and then select ResourceManager logs. You are presented with a list of links to YARN logs.

Step 4: Forecast log volume storage sizes and costs

After you've completed the previous steps you will have gained an understanding of the types and volumes of log files that your HDInsight cluster(s) are producing.

Next, you should analyze the volume of log data in key log storage locations over a period of time. For example, if you have the data, it's common to analyze volume and growth over 30-60-90 day periods. Record this information in a spreadsheet or you can use other tools such as Visual Studio, the Azure Storage Explorer or Power Query for

Excel.

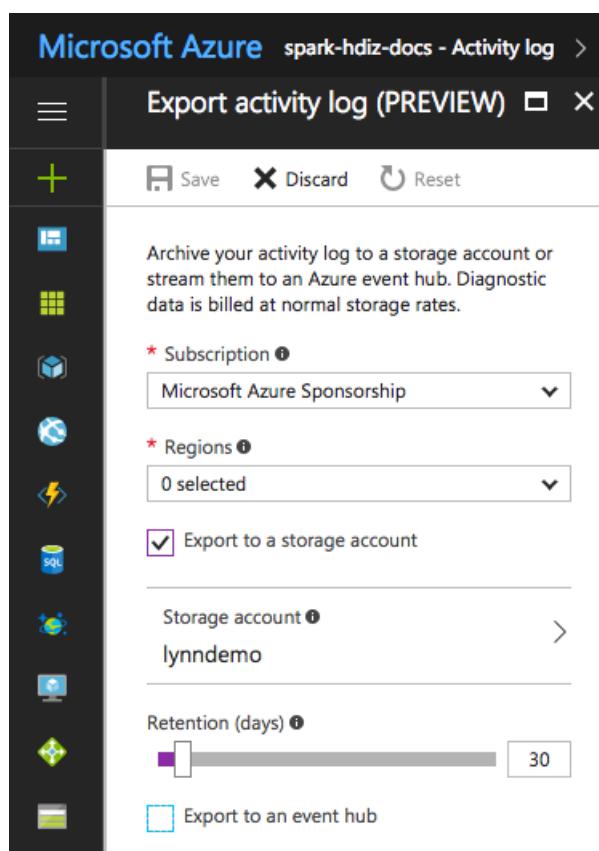
See [Analyze HDInsight Logs](#) for details. Using this information you can now forecast potential future log volume growth and associated Azure service (storage) costs.

You now have enough information to create log management strategy for key logs. To do this, use your spreadsheet(or tool or choice) to forecast both log size growth and also log storage Azure service costs going forward. Next, apply your understanding of log retention requirements to the set of logs that you are examining. Now you can re-forecast future log storage costs, after you've determined which log files can be deleted (if any) and which should be retained and archived to less expensive (i.e. cool) Azure storage.

Step 5: Determine log archive policies and processes

After you determine which log files can be deleted, you can adjust logging parameters on many Hadoop services to automatically delete log files after a specified time period.

For certain log files, you may wish to execute a lower-priced log file archiving approach. For Azure Resource Manager activity logs, you can explore this approach first via the Azure Portal. You can setup archiving of the ARM logs by clicking on the 'Activity Log' link in the Azure portal for your HDInsight instance. On the top of the Activit Log search page click on the 'Export' menu item. This opens the 'Export activity log' blade. You then fill in the subscription, region and whether you'd like to export to a storage account and for how many days you'd like the logs to be retained. On this same blade you can also indicate whether you'd like to export to an event hub. An example screen is shown below.



Alternatively, you can script log archiving by using powershell. A Sample powershell script to [Archive Azure Automation logs to Azure BLOB Storage](#) can be found at the link referenced.

Accessing Azure storage metrics

Azure storage can be configured to log storage operations and access. You can use these logs, which contain a wealth of information, for capacity monitoring and planning, and for auditing requests to storage. The information includes latency details, enabling you to monitor and fine tune performance of your solutions. You can use the .NET SDK for Hadoop to examine the log files generated for the Azure storage that holds the data for an HDInsight

cluster.

Control the Size and Number of backup index of old log files

Do this by using the following properties of the `RollingFileAppender` to efficiently control the Size and the number of log files retained.

```
maxFileSize: This is the critical size of the file above which the file will be rolled. Default value is 10 MB.
```

```
maxBackupIndex: This property denotes the number of backup files to be created. Default value is 1.
```

Other log management techniques

To avoid running out of disk space, you may want to use some OS tools to let them manage handling of log files, e.g. logrotate. You can configure it to run on a daily basis, compressing log files and remove old ones. But this behaviour highly depends on your requirements, e.g. how long do you want to keep the logfiles on local nodes. You can also check if you have DEBUG logging enabled in one of your services where you do not need it, which will increase the size of log output heavily. If you want to collect the logs from all the nodes to one central location for much easier diving into issues, you could create a data flow e.g. ingesting all the log entries into Solr.

Conclusion

There are a number of considerations you need to take into account when creating and implementing your HDInsight cluster log management policies. You should focus on using the best HDInsight log policy for each of your particular workload types running your cluster. Along with that, you'll need to monitor the size of key log files for the execution of long-running job executions to make sure that they don't exceed the expected size. This is not only so that you have predictable costs, but also for performance and service availability. It's also critically important to manage your cluster configuration files over time as part of your log management strategy, so that you can revert to working state should the need arise.

See also

- [Monitoring and Logging Practice for HDInsight](#)
- [Access YARN application log on Linux-based HDInsight](#)
- [How to control size of log files for various Hadoop components](#)

Add additional storage accounts to HDInsight

8/16/2017 • 5 min to read • [Edit Online](#)

Learn how to use script actions to add additional Azure storage accounts to HDInsight. The steps in this document add a storage account to an existing Linux-based HDInsight cluster.

IMPORTANT

The information in this document is about adding additional storage to a cluster after it has been created. For information on adding storage accounts during cluster creation, see [Set up clusters in HDInsight with Hadoop, Spark, Kafka, and more](#).

How it works

This script takes the following parameters:

- **Azure storage account name:** The name of the storage account to add to the HDInsight cluster. After running the script, HDInsight can read and write data stored in this storage account.
- **Azure storage account key:** A key that grants access to the storage account.
- **-p (optional):** If specified, the key is not encrypted and is stored in the core-site.xml file as plain text.

During processing, the script performs the following actions:

- If the storage account already exists in the core-site.xml configuration for the cluster, the script exits and no further actions are performed.
- Verifies that the storage account exists and can be accessed using the key.
- Encrypts the key using the cluster credential.
- Adds the storage account to the core-site.xml file.
- Stops and restarts the Oozie, YARN, MapReduce2, and HDFS services. Stopping and starting these services allows them to use the new storage account.

WARNING

Using a storage account in a different location than the HDInsight cluster is not supported.

The script

Script location: <https://hdiconfigactions.blob.core.windows.net/linuxaddstorageaccountv01/add-storage-account-v01.sh>

Requirements:

- The script must be applied on the **Head nodes**.

To use the script

This script can be used from the Azure portal, Azure PowerShell, or the Azure CLI 1.0. For more information, see the [Customize Linux-based HDInsight clusters using script action](#) document.

IMPORTANT

When using the steps provided in the customization document, use the following information to apply this script:

- Replace any example script action URI with the URI for this script (<https://hdiconfigactions.blob.core.windows.net/linuxaddstorageaccountv01/add-storage-account-v01.sh>).
- Replace any example parameters with the Azure storage account name and key of the storage account to be added to the cluster. If using the Azure portal, these parameters must be separated by a space.
- You do not need to mark this script as **Persisted**, as it directly updates the Ambari configuration for the cluster.

Known issues

Storage accounts not displayed in Azure portal or tools

When viewing the HDInsight cluster in the Azure portal, selecting the **Storage Accounts** entry under **Properties** does not display storage accounts added through this script action. Azure PowerShell and Azure CLI do not display the additional storage account either.

The storage information isn't displayed because the script only modifies the core-site.xml configuration for the cluster. This information is not used when retrieving the cluster information using Azure management APIs.

To view storage account information added to the cluster using this script, use the Ambari REST API. Use the following commands to retrieve this information for your cluster:

```
$creds = Get-Credential -UserName "admin" -Message "Enter the cluster login credentials"
$resp = Invoke-WebRequest -Uri
"https://$clusterName.azurehdinsight.net/api/v1/clusters/$clusterName/configurations/service_config_versions?
service_name=HDFS&service_config_version=1" ` 
-Credential $creds
$respObj = ConvertFrom-Json $resp.Content
$respObj.items.configurations.properties."fs.azure.account.key.$storageAccountName.blob.core.windows.net"
```

NOTE

Set `$clusterName` to the name of the HDInsight cluster. Set `$storageAccountName` to the name of the storage account. When prompted, enter the cluster login (admin) and password.

```
curl -u admin:PASSWORD -G
"https://CLUSTERNAME.azurehdinsight.net/api/v1/clusters/CLUSTERNAME/configurations/service_config_versions?
service_name=HDFS&service_config_version=1" | jq
'.items[].configurations[].properties["fs.azure.account.key.$STORAGEACCOUNTNAME.blob.core.windows.net"] | 
select(. != null)'
```

NOTE

Set `$PASSWORD` to the cluster login (admin) account password. Set `$CLUSTERNAME` to the name of the HDInsight cluster. Set `$STORAGEACCOUNTNAME` to the name of the storage account.

This example uses [curl \(http://curl.haxx.se/\)](http://curl.haxx.se/) and [jq \(https://stedolan.github.io/jq/\)](https://stedolan.github.io/jq/) to retrieve and parse JSON data.

When using this command, replace **CLUSTERNAME** with the name of the HDInsight cluster. Replace **PASSWORD** with the HTTP login password for the cluster. Replace **STORAGEACCOUNT** with the name of the storage account added using script action. Information returned from this command appears similar to the following text:

```
"MIIB+gYJKoZIhvcNAQcDoIIB6zCCAecCAQAxggFaMIIBVgIBADA+MCoxKDAmBgNVBAMTH2RiZW5jcnlwG1vbi5henVyzWhkaW5zaWdodC5uZXCEA6GDZMw1oiESKFHOOEgjcwDQYJKoZIhvcNAQEQQAEggEATIu08MJ45KEQAYBQ1d7WaRkJOWqaCLwFub9zNpscrquA2f3o0emy9r6vu5cD3GTt7PmaAF0pvssbKVMf/Z8yRpHmeezSco2y7e9Qd7xJKRLYtRHm80fsjibHSW9CYkQwxHaOqdR7DBhZyhnj+DHhODsIO2FGM8MxWk4fgBRV06CZ5eTmZ6KVR8wYbFLi8YZXb7GkUEeSn2PsjrKGiQjtpXw1RAyanCagr5vlg8CicZg1HuhCHWF/RYFWM3EBBvz+uFZPR3BqTgbvBhWXRJaISwssvxotppe0ikevnEgaBYrf1B2P+PVrwPTZ7f36HQcn4ifY1WRJQ4qRaUxdYEFzCBgwYJKoZIhvcNAQcBMBQGCCqGSIB3DQMHBAbRdsrgRV3wmYBg3j/T1aEn03wLWRpgZa16MwqmfpQpuansKHjLwbZjTpeirquaQpZVxDK/w4gk1K+t1heNsNo1Wwqu+Y47bsAX1k9Ud7+Ed2oETDI7724IJ213YeGxvu4Ngcf2eHW+FRK"
```

This text is an example of an encrypted key, which is used to access the storage account.

Unable to access storage after changing key

If you change the key for a storage account, HDInsight can no longer access the storage account. HDInsight uses a cached copy of key in the core-site.xml for the cluster. This cached copy must be updated to match the new key.

Running the script action again does **not** update the key, as the script checks to see if an entry for the storage account already exists. If an entry already exists, it does not make any changes.

To work around this problem, you must remove the existing entry for the storage account. Use the following steps to remove the existing entry:

1. In a web browser, open the Ambari Web UI for your HDInsight cluster. The URI is <https://CLUSTERNAME.azurehdinsight.net>. Replace **CLUSTERNAME** with the name of your cluster.
When prompted, enter the HTTP login user and password for your cluster.
2. From the list of services on the left of the page, select **HDFS**. Then select the **Configs** tab in the center of the page.
3. In the **Filter...** field, enter a value of **fs.azure.account**. This returns entries for any additional storage accounts that have been added to the cluster. There are two types of entries: **keyprovider** and **key**. Both contain the name of the storage account as part of the key name.

The following are example entries for a storage account named **mystorage**:

```
fs.azure.account.keyprovider.mystorage.blob.core.windows.net
fs.azure.account.key.mystorage.blob.core.windows.net
```

4. After you have identified the keys for the storage account you need to remove, use the red '-' icon to the right of the entry to delete it. Then use the **Save** button to save your changes.
5. After changes have been saved, use the script action to add the storage account and new key value to the cluster.

Poor performance

If the storage account is in a different region than the HDInsight cluster, you may experience poor performance. Accessing data in a different region sends network traffic outside the regional Azure data center and across the public internet, which can introduce latency.

WARNING

Using a storage account in a different region than the HDInsight cluster is not supported.

Additional charges

If the storage account is in a different region than the HDInsight cluster, you may notice additional egress charges on your Azure billing. An egress charge is applied when data leaves a regional data center. This charge is applied even if the traffic is destined for another Azure data center in a different region.

WARNING

Using a storage account in a different region than the HDInsight cluster is not supported.

Next steps

You have learned how to add additional storage accounts to an existing HDInsight cluster. For more information on script actions, see [Customize Linux-based HDInsight clusters using script action](#)

Customize Linux-based HDInsight clusters using Script Action

8/16/2017 • 25 min to read • [Edit Online](#)

HDInsight provides a configuration option called **Script Action** that invokes custom scripts that customize the cluster. These scripts are used to install additional components and change configuration settings. Script actions can be used during or after cluster creation.

IMPORTANT

The ability to use script actions on an already running cluster is only available for Linux-based HDInsight clusters.

Linux is the only operating system used on HDInsight version 3.4 or greater. For more information, see [HDInsight retirement on Windows](#).

Script actions can also be published to the Azure Marketplace as an HDInsight application. Some of the examples in this document show how you can install an HDInsight application using script action commands from PowerShell and the .NET SDK. For more information on HDInsight applications, see [Publish HDInsight applications into the Azure Marketplace](#).

Permissions

If you are using a domain-joined HDInsight cluster, there are two Ambari permissions that are required when using script actions with the cluster:

- **AMBARI.RUN_CUSTOM_COMMAND**: The Ambari Administrator role has this permission by default.
- **CLUSTER.RUN_CUSTOM_COMMAND**: Both the HDInsight Cluster Administrator and Ambari Administrator have this permission by default.

For more information on working with permissions with domain-joined HDInsight, see [Manage domain-joined HDInsight clusters](#).

Access control

If you are not the administrator/owner of your Azure subscription, your account must have at least **Contributor** access to the resource group that contains the HDInsight cluster.

Additionally, if you are creating an HDInsight cluster, someone with at least **Contributor** access to the Azure subscription must have previously registered the provider for HDInsight. Provider registration happens when a user with Contributor access to the subscription creates a resource for the first time on the subscription. It can also be accomplished without creating a resource by [registering a provider using REST](#).

For more information on working with access management, see the following documents:

- [Get started with access management in the Azure portal](#)
- [Use role assignments to manage access to your Azure subscription resources](#)

Understanding Script Actions

A Script Action is simply a Bash script that you provide a URI to, and parameters for. The script runs on nodes in the HDInsight cluster. The following are characteristics and features of script actions.

- Must be stored on a URI that is accessible from the HDInsight cluster. The following are possible storage locations:
 - An **Azure Data Lake Store** account that is accessible by the HDInsight cluster. For information on using Azure Data Lake Store with HDInsight, see [Create an HDInsight cluster with Data Lake Store](#).

When using a script stored in Data Lake Store, the URI format is

```
adl://DATALAKESTOREACCOUNTNAME.azuredatalakestore.net/path_to_file .
```

NOTE

The service principal HDInsight uses to access Data Lake Store must have read access to the script.

- A blob in an **Azure Storage account** that is either the primary or additional storage account for the HDInsight cluster. HDInsight is granted access to both of these types of storage accounts during cluster creation.
- A public file sharing service such as Azure Blob, GitHub, OneDrive, Dropbox, etc.

For example URIs, see the [Example script action scripts](#) section.

WARNING

HDInsight only supports **General purpose** Azure Storage accounts. It does not currently support the **Blob storage** account type.

- Can be restricted to **run on only certain node types**, for example head nodes or worker nodes.

NOTE

When used with HDInsight Premium, you can specify that the script should be used on the edge node.

- Can be **persisted** or **ad hoc**.

Persisted scripts are applied to worker nodes added to the cluster after the script has ran. For example, when scaling up the cluster.

A persisted script might also apply changes to another node type, such as a head node.

IMPORTANT

Persisted script actions must have a unique name.

Ad hoc scripts are not persisted. They are not applied to worker nodes added to the cluster after the script has ran. You can subsequently promote an ad hoc script to a persisted script, or demote a persisted script to an ad hoc script.

IMPORTANT

Script actions used during cluster creation are automatically persisted.

Scripts that fail are not persisted, even if you specifically indicate that they should be.

- Can accept **parameters** that are used by the script during execution.

- Run with **root level privileges** on the cluster nodes.
- Can be used through the **Azure portal**, **Azure PowerShell**, **Azure CLI**, or **HDInsight .NET SDK**

The cluster keeps a history of all scripts that have been ran. The history is useful when you need to find the ID of a script for promotion or demotion operations.

IMPORTANT

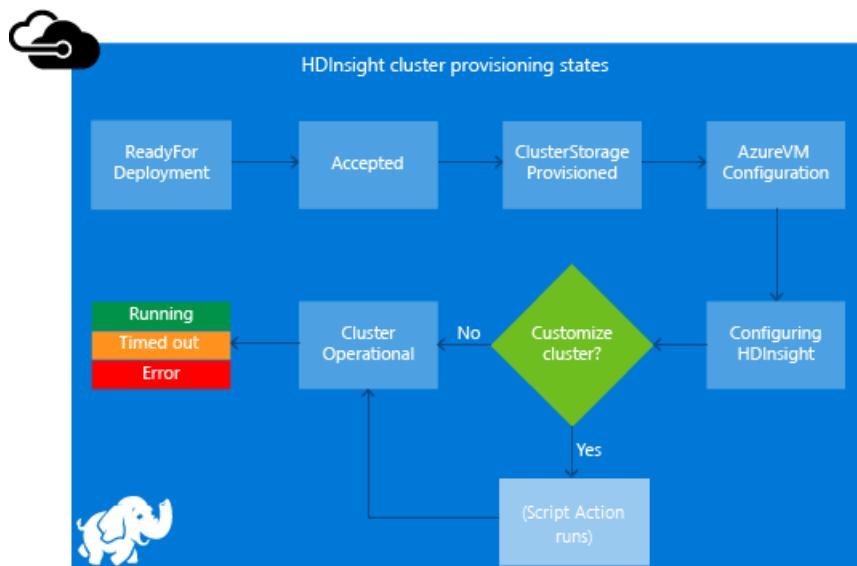
There is no automatic way to undo the changes made by a script action. Either manually reverse the changes or provide a script that reverses them.

Script Action in the cluster creation process

Script Actions used during cluster creation are slightly different from script actions ran on an existing cluster:

- The script is **automatically persisted**.
- A **failure** in the script can cause the cluster creation process to fail.

The following diagram illustrates when Script Action is executed during the creation process:



The script runs while HDInsight is being configured. At this stage, the script runs in parallel on all the specified nodes in the cluster, and runs with root privileges on the nodes.

NOTE

Because the script runs with root level privilege on the cluster nodes, you can perform operations like stopping and starting services, including Hadoop-related services. If you stop services, you must ensure that the Ambari service and other Hadoop-related services are up and running before the script finishes running. These services are required to successfully determine the health and state of the cluster while it is being created.

During cluster creation, you can use multiple script actions at once. These scripts are invoked in the order in which they were specified.

IMPORTANT

Script actions must complete within 60 minutes, or timeout. During cluster provisioning, the script runs concurrently with other setup and configuration processes. Competition for resources such as CPU time or network bandwidth may cause the script to take longer to finish than it does in your development environment.

To minimize the time it takes to run the script, avoid tasks such as downloading and compiling applications from source. Pre-compile applications and store the binary in Azure Storage. This allows the script to quickly download the application to the cluster.

Script action on a running cluster

Unlike script actions used during cluster creation, a failure in a script ran on an already running cluster does not automatically cause the cluster to change to a failed state. Once a script completes, the cluster should return to a "running" state.

IMPORTANT

This does not mean that your running cluster is immune to scripts that do bad things. For example, a script could delete files needed by the cluster.

Scripts actions run with root privileges, so you should make sure that you understand what a script does before applying it to your cluster.

When applying a script to a cluster, the cluster state changes to from **Running** to **Accepted**, then **HDInsight configuration**, and finally back to **Running** for successful scripts. The script status is logged in the script action history, and you can use this information to determine whether the script succeeded or failed. For example, the `Get-AzureRmHDInsightScriptActionHistory` PowerShell cmdlet can be used to view the status of a script. It returns information similar to the following text:

```
ScriptExecutionId : 635918532516474303
StartTime        : 2/23/2016 7:40:55 PM
EndTime          : 2/23/2016 7:41:05 PM
Status           : Succeeded
```

NOTE

If you have changed the cluster user (admin) password after the cluster was created, script actions ran against this cluster may fail. If you have any persisted script actions that target worker nodes, these scripts may fail when you scale the cluster.

Example Script Action scripts

Script Action scripts can be used through the following utilities:

- Azure portal
- Azure PowerShell
- Azure CLI
- HDInsight .NET SDK

HDInsight provides scripts to install the following components on HDInsight clusters:

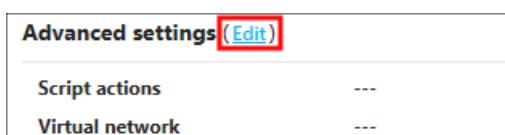
NAME	SCRIPT
Add an Azure Storage account	https://hdiconfigactions.blob.core.windows.net/linuxaddstorageaccountv01/add-storage-account-v01.sh . See Add additional storage to an HDInsight cluster.
Install Hue	https://hdiconfigactions.blob.core.windows.net/linuxhueconfigurationv02/install-hue-uber-v02.sh . See Install and use Hue on HDInsight clusters.
Install Presto	https://raw.githubusercontent.com/hdinsight/presto-hdinsight/master/installpresto.sh . See Install and use Presto on HDInsight clusters.
Install Solr	https://hdiconfigactions.blob.core.windows.net/linuxsolrconfigurationv01/solr-installer-v01.sh . See Install and use Solr on HDInsight clusters.
Install Giraph	https://hdiconfigactions.blob.core.windows.net/linuxgiraphconfigurationv01/giraph-installer-v01.sh . See Install and use Giraph on HDInsight clusters.
Pre-load Hive libraries	https://hdiconfigactions.blob.core.windows.net/linuxsetupcustomhivelibs01/setup-customhivelibs-v01.sh . See Add Hive libraries on HDInsight clusters.
Install or update Mono	https://hdiconfigactions.blob.core.windows.net/installmono/install-mono.bash . See Install or update Mono on HDInsight.

Use a Script Action during cluster creation

This section provides examples on the different ways you can use script actions when creating an HDInsight cluster.

Use a Script Action during cluster creation from the Azure portal

1. Start creating a cluster as described at [Create Hadoop clusters in HDInsight](#). Stop when you reach the **Cluster summary** blade.
2. From the **Cluster summary** blade, select the **edit** link for **Advanced settings**.



3. From the **Advanced settings** blade, select **Script actions**. From the **Script actions** blade, select **+ Submit new**

Advanced settings

Script Actions (optional)

Script actions

Optional

Virtual Network Settings (optional)

Filtered to location and subscription of cluster.

Virtual network

Select or type to filter virtual networks

Subnet

Please select a valid virtual network

NAME BASH SCRIPT URI HEAD WORKER ZOOKEEPER PARAMETERS

No script actions found

Next Select

4. Use the **Select a script** entry to select a pre-made script. To use a custom script, select **Custom** and then provide the **Name** and **Bash script URI** for your script.

Select premade script (learn more)

Custom

* Name

MyScript

* Bash script URI

https://hdiconfigactions.blob.core.windows...

Head

Worker

Zookeeper

Parameters

Persist this script action to rerun when new nodes are added to the cluster.

Create

The following table describes the elements on the form:

PROPERTY	VALUE
Select a script	Select one of the pre-made script, or Custom to use your own script.
Name	Specify a name for the script action.
Bash script URI	Specify the URI to the script that is invoked to customize the cluster.

PROPERTY	VALUE
Head/Worker/Zookeeper	Specify the nodes (Head , Worker , or ZooKeeper) on which the customization script is run.
Parameters	Specify the parameters, if required by the script.

Use the **Persist this script action** entry to ensure that the script is applied to worker nodes when you scale your cluster.

5. Select **Create** to save the script. You can then use + **Submit new** to add another script.

NAME	BASH SCRIPT URI	HEAD	WORKER	ZOOKEEPER	PARAMETERS
MyScript	https://hdiconfigactions.blob...	true	false	false	...
Install Giraph	https://hdiconfigactions.blob...	true	false	false	...

When you are done adding scripts, use the **Select** button, and then the **Next** button to return to the **Cluster summary** blade.

6. To create the cluster, select **Create** from the **Cluster summary** blade.

Use a Script Action from Azure Resource Manager templates

The examples in this section demonstrate how to use script actions with Azure Resource Manager templates when creating an HDInsight cluster.

Before you begin

- For information about configuring a workstation to run HDInsight Powershell cmdlets, see [Install and configure Azure PowerShell](#)
- For instructions on how to create templates, see [Authoring Azure Resource Manager templates](#).
- If you have not previously used Azure PowerShell with Resource Manager, see [Using Azure PowerShell with Azure Resource Manager](#).

Create clusters using Script Action

1. Copy the following template to a location on your computer. This template installs Giraph on the headnodes and worker nodes in the cluster. You can also verify if the JSON template is valid. Paste your template content into [JSONLint](#), an online JSON validation tool.

```
{
"$schema": "http://schema.management.azure.com/schemas/2015-01-01/deploymentTemplate.json#",
"contentVersion": "1.0.0.0",
"parameters": {
    "clusterLocation": {
        "type": "string",
        "defaultValue": "West US",
        "allowedValues": [ "West US" ]
    }
}
```

```

},
"clusterName": {
    "type": "string"
},
"clusterUserName": {
    "type": "string",
    "defaultValue": "admin"
},
"clusterUserPassword": {
    "type": "securestring"
},
"sshUserName": {
    "type": "string",
    "defaultValue": "username"
},
"sshPassword": {
    "type": "securestring"
},
"clusterStorageAccountName": {
    "type": "string"
},
"clusterStorageAccountResourceGroup": {
    "type": "string"
},
"clusterStorageType": {
    "type": "string",
    "defaultValue": "Standard_LRS",
    "allowedValues": [
        "Standard_LRS",
        "Standard_GRS",
        "Standard_ZRS"
    ]
},
"clusterStorageAccountContainer": {
    "type": "string"
},
"clusterHeadNodeCount": {
    "type": "int",
    "defaultValue": 1
},
"clusterWorkerNodeCount": {
    "type": "int",
    "defaultValue": 2
}
},
"variables": {
},
"resources": [
{
    "name": "[parameters('clusterStorageAccountName')]",
    "type": "Microsoft.Storage/storageAccounts",
    "location": "[parameters('clusterLocation')]",
    "apiVersion": "2015-05-01-preview",
    "dependsOn": [ ],
    "tags": { },
    "properties": {
        "accountType": "[parameters('clusterStorageType')]"
    }
},
{
    "name": "[parameters('clusterName')]",
    "type": "Microsoft.HDInsight/clusters",
    "location": "[parameters('clusterLocation')]",
    "apiVersion": "2015-03-01-preview",
    "dependsOn": [
        "[concat('Microsoft.Storage/storageAccounts/',
parameters('clusterStorageAccountName'))]"
    ],
    "tags": { }
}
]

```

```

"properties": {
    "clusterVersion": "3.2",
    "osType": "Linux",
    "clusterDefinition": {
        "kind": "hadoop",
        "configurations": {
            "gateway": {
                "restAuthCredential.isEnabled": true,
                "restAuthCredential.username": "[parameters('clusterUserName')]",
                "restAuthCredential.password": "[parameters('clusterUserPassword')]"
            }
        }
    },
    "storageProfile": {
        "storageaccounts": [
            {
                "name": "
[concat(parameters('clusterStorageAccountName'),'.blob.core.windows.net')]",
                "isDefault": true,
                "container": "[parameters('clusterStorageAccountContainer')]",
                "key": "[listKeys(resourceId('Microsoft.Storage/storageAccounts',
parameters('clusterStorageAccountName')), '2015-05-01-preview').key1]"
            }
        ]
    },
    "computeProfile": [
        "roles": [
            {
                "name": "headnode",
                "targetInstanceCount": "[parameters('clusterHeadNodeCount')]",
                "hardwareProfile": {
                    "vmSize": "Large"
                },
                "osProfile": {
                    "linuxOperatingSystemProfile": {
                        "username": "[parameters('sshUserName')]",
                        "password": "[parameters('sshPassword')]"
                    }
                },
                "scriptActions": [
                    {
                        "name": "installGiraph",
                        "uri": "
"https://hdiconfigactions.blob.core.windows.net/linuxgiraphconfigactionv01/giraph-installer-v01.sh",
                        "parameters": ""
                    }
                ]
            },
            {
                "name": "workernode",
                "targetInstanceCount": "[parameters('clusterWorkerNodeCount')]",
                "hardwareProfile": {
                    "vmSize": "Large"
                },
                "osProfile": {
                    "linuxOperatingSystemProfile": {
                        "username": "[parameters('sshUserName')]",
                        "password": "[parameters('sshPassword')]"
                    }
                },
                "scriptActions": [
                    {
                        "name": "installR",
                        "uri": "
"https://hdiconfigactions.blob.core.windows.net/linuxrconfigactionv01/r-installer-v01.sh",
                        "parameters": ""
                    }
                ]
            }
        ]
    }
}

```

```

        ]
    }
}
],
"outputs": {
    "cluster": {
        "type" : "object",
        "value" : "
[reference(resourceId('Microsoft.HDInsight/clusters',parameters('clusterName')))]"
    }
}
}

```

- Start Azure PowerShell and Log in to your Azure account. After providing your credentials, the command returns information about your account.

```
Add-AzureRmAccount

Id          Type      ...
--          ----
someone@example.com   User      ...
```

- If you have multiple subscriptions, provide the subscription id you wish to use for deployment.

```
Select-AzureRmSubscription -SubscriptionID <YourSubscriptionId>
```

NOTE

You can use `Get-AzureRmSubscription` to get a list of all subscriptions associated with your account, which includes the subscription Id for each one.

- If you do not have an existing resource group, create a resource group. Provide the name of the resource group and location that you need for your solution. A summary of the new resource group is returned.

```
New-AzureRmResourceGroup -Name myresourcegroup -Location "West US"

ResourceGroupName : myresourcegroup
Location         : westus
ProvisioningState : Succeeded
Tags             :
Permissions       :
                  Actions  NotActions
                  =====  =====
                  *
ResourceId       : /subscriptions/#####/resourceGroups/ExampleResourceGroup
```

- To create a deployment for your resource group, run the **New-AzureRmResourceGroupDeployment** command and provide the necessary parameters. The parameters include a name for your deployment, the name of your resource group, and the path or URL to the template you created. If your template requires any parameters, you must pass those parameters as well. In this case, the script action to install R on the cluster does not require any parameters.

```
New-AzureRmResourceGroupDeployment -Name mydeployment -ResourceGroupName myresourcegroup -
TemplateFile <PathOrLinkToTemplate>
```

You are prompted to provide values for the parameters defined in the template.

- When the resource group has been deployed, a summary of the deployment is displayed.

```
DeploymentName      : mydeployment
ResourceGroupName  : myresourcegroup
ProvisioningState  : Succeeded
Timestamp          : 8/17/2015 7:00:27 PM
Mode               : Incremental
...
...
```

- If your deployment fails, you can use the following cmdlets to get information about the failures.

```
Get-AzureRmResourceGroupDeployment -ResourceGroupName myresourcegroup -ProvisioningState Failed
```

Use a Script Action during cluster creation from Azure PowerShell

In this section, we use the [Add-AzureRmHDInsightScriptAction](#) cmdlet to invoke scripts by using Script Action to customize a cluster. Before proceeding, make sure you have installed and configured Azure PowerShell. For information about configuring a workstation to run HDInsight PowerShell cmdlets, see [Install and configure Azure PowerShell](#).

The following script demonstrates how to apply a script action when creating a cluster using PowerShell:

```
# Login to your Azure subscription
# Is there an active Azure subscription?
$sub = Get-AzureRmSubscription -ErrorAction SilentlyContinue
if(-not($sub))
{
    Add-AzureRmAccount
}

# If you have multiple subscriptions, set the one to use
# $subscriptionID = "<subscription ID to use>"
# Select-AzureRmSubscription -SubscriptionId $subscriptionID

# Get user input/default values
$resourceGroupName = Read-Host -Prompt "Enter the resource group name"
$location = Read-Host -Prompt "Enter the Azure region to create resources in"

# Create the resource group
New-AzureRmResourceGroup -Name $resourceGroupName -Location $location

$defaultStorageAccountName = Read-Host -Prompt "Enter the name of the storage account"

# Create an Azure storage account and container
New-AzureRmStorageAccount ` 
    -ResourceGroupName $resourceGroupName ` 
    -Name $defaultStorageAccountName ` 
    -Type Standard_LRS ` 
    -Location $location
$defaultStorageAccountKey = (Get-AzureRmStorageAccountKey ` 
    -ResourceGroupName $resourceGroupName ` 
    -Name $defaultStorageAccountName)[0].Value

$defaultStorageContext = New-AzureStorageContext ` 
    -StorageAccountName $defaultStorageAccountName ` 
    -StorageAccountKey $defaultStorageAccountKey

# Get information for the HDInsight cluster
$clusterName = Read-Host -Prompt "Enter the name of the HDInsight cluster"
# Cluster login is used to secure HTTPS services hosted on the cluster
$httpCredential = Get-Credential -Message "Enter Cluster login credentials" -UserName "admin"
# SSH user is used to remotely connect to the cluster using SSH clients
# $sshCredential = Get-Credential -Message "Enter SSH user credentials"
```

```

$sshCredential = Get-Credential -Message "Enter SSH user credentials"

# Default cluster size (# of worker nodes), version, type, and OS
$clusterSizeInNodes = "4"
$clusterVersion = "3.5"
$clusterType = "Hadoop"
$clusterOS = "Linux"
# Set the storage container name to the cluster name
$defaultBlobContainerName = $clusterName

# Create a blob container. This holds the default data store for the cluster.
New-AzureStorageContainer `

    -Name $clusterName -Context $defaultStorageContext

# Create an HDInsight configuration object
$config = New-AzureRmHDInsightClusterConfig
# Add the script action
$scriptActionUri="https://hdiconfigactions.blob.core.windows.net/linuxgiraphconfigactionv01/giraph-
installer-v01.sh"
# Add for the head nodes
$config = Add-AzureRmHDInsightScriptAction `

    -Config $config `

    -Name "Install Giraph" `

    -NodeType HeadNode `

    -Uri $scriptActionUri
# Continue adding the script action for any other node types
# that it must run on.
$config = Add-AzureRmHDInsightScriptAction `

    -Config $config `

    -Name "Install Giraph" `

    -NodeType WorkerNode `

    -Uri $scriptActionUri

# Create the cluster using the configuration object
New-AzureRmHDInsightCluster `

    -Config $config `

    -ResourceGroupName $resourceGroupName `

    -ClusterName $clusterName `

    -Location $location `

    -ClusterSizeInNodes $clusterSizeInNodes `

    -ClusterType $clusterType `

    -OSType $clusterOS `

    -Version $clusterVersion `

    -HttpCredential $httpCredential `

    -DefaultStorageAccountName "$defaultStorageAccountName.blob.core.windows.net" `

    -DefaultStorageAccountKey $defaultStorageAccountKey `

    -DefaultStorageContainer $containerName `

    -SshCredential $sshCredential

```

It can take several minutes before the cluster is created.

Use a Script Action during cluster creation from the HDInsight .NET SDK

The HDInsight .NET SDK provides client libraries that makes it easier to work with HDInsight from a .NET application. For a code sample, see [Create Linux-based clusters in HDInsight using the .NET SDK](#).

Apply a Script Action to a running cluster

In this section, learn how to apply script actions to a running cluster.

Apply a Script Action to a running cluster from the Azure portal

- From the [Azure portal](#), select your HDInsight cluster.
- From the HDInsight cluster blade, select the **Script Actions** tile.

The screenshot shows the Cloudera Manager main dashboard. At the top, there are 'Quick Links' for 'Cluster Dashboard', 'Ambari Views', and 'Scale Cluster'. Below that is a section titled 'Usage' with a pie chart showing cores in North Central US for subscription. The chart indicates 170 cores in total, with 16 in 'THIS CLUSTER' and 92 in 'OTHER CLUSTERS'. To the right of the chart is a 'Script Actions' tile, which is highlighted with a red box.

NOTE

You can also select **All settings** and then select **Script Actions** from the Settings blade.

3. From the top of the Script Actions blade, select **Submit new**.

The screenshot shows the 'Script actions' blade. The title bar says 'Script actions' and includes a note: 'You can use Script Actions to run Bash scripts on your cluster.' Below the title is a button labeled '+ Submit new', which is highlighted with a red box. The main area is divided into two sections: 'PERSISTED SCRIPT ACTIONS' and 'SCRIPT ACTION HISTORY'. The 'PERSISTED SCRIPT ACTIONS' section shows a single entry: 'No script actions found'. The 'SCRIPT ACTION HISTORY' section shows a single entry with a green checkmark: 'Add an Azure Storage account' submitted by user on 5/22/2017.

4. Use the **Select a script** entry to select a pre-made script. To use a custom script, select **Custom** and then provide the **Name** and **Bash script URI** for your script.

The screenshot shows the 'Submit script action' dialog box. At the top, it says 'Select premade script (learn more)'. Below that is a dropdown menu set to 'Custom'. The next section is 'Name', with the value 'MyScript' entered. Under 'Bash script URI', the URL 'https://hdiconfigactions.blob.core.windows.net/...' is listed. There are three checkboxes for 'Head', 'Worker', and 'Zookeeper'; 'Head' is checked, while 'Worker' and 'Zookeeper' are unchecked. A large text area labeled 'Parameters' is empty. At the bottom left is a checked checkbox labeled 'Persist this script action to rerun when new nodes are added to the cluster.' At the bottom right is a blue 'Create' button.

The following table describes the elements on the form:

PROPERTY	VALUE
Select a script	Select one of the pre-made script, or Custom to use your own script.
Name	Specify a name for the script action.
Bash script URI	Specify the URI to the script that is invoked to customize the cluster.
Head/Worker/Zookeeper	Specify the nodes (Head , Worker , or ZooKeeper) on which the customization script is run.
Parameters	Specify the parameters, if required by the script.

Use the **Persist this script action** entry to ensure that the script is applied to worker nodes when you scale your cluster.

- Finally, use the **Create** button to apply the script to the cluster.

Apply a Script Action to a running cluster from Azure PowerShell

Before proceeding, make sure you have installed and configured Azure PowerShell. For information about configuring a workstation to run HDInsight PowerShell cmdlets, see [Install and configure Azure PowerShell](#).

The following example demonstrates how to apply a script action to a running cluster:

```

# Get information for the HDInsight cluster
$clusterName = Read-Host -Prompt "Enter the name of the HDInsight cluster"
$scriptActionName = Read-Host -Prompt "Enter the name of the script action"
$scriptActionUri = Read-Host -Prompt "Enter the URI of the script action"
# The node types that the script action is applied to
$nodeTypes = "headnode", "workernode"

# Apply the script and mark as persistent
Submit-AzureRmHDInsightScriptAction -ClusterName $clusterName ` 
    -Name $scriptActionName ` 
    -Uri $scriptActionUri ` 
    -NodeTypes $nodeTypes ` 
    -PersistOnSuccess

```

Once the operation completes, you receive information similar to the following:

```

OperationState  : Succeeded
ErrorMessage    :
Name          : Giraph
Uri           : https://hdiconfigactions.blob.core.windows.net/linuxgiraphconfigactionv01/giraph-
installer-v01.sh
Parameters     :
NodeTypes      : {HeadNode, WorkerNode}

```

Apply a Script Action to a running cluster from the Azure CLI

Before proceeding, make sure you have installed and configured the Azure CLI. For more information, see [Install the Azure CLI](#).

[! IMPORTANT] Azure CLI support for managing HDInsight resources using Azure Service Manager (ASM) is **deprecated**, and was removed on January 1, 2017. The steps in this document use the new Azure CLI commands that work with Azure Resource Manager.

Please follow the steps in [Install and configure Azure CLI](#) to install the latest version of the Azure CLI. If you have scripts that need to be modified to use the new commands that work with Azure Resource Manager, see [Migrating to Azure Resource Manager-based development tools for HDInsight clusters](#) for more information.

1. Open a shell session, terminal, command-prompt or other command line for your system and use the following command to switch to Azure Resource Manager mode.

```
azure config mode arm
```

2. Use the following to authenticate to your Azure subscription.

```
azure login
```

3. Use the following command to apply a script action to a running cluster

```
azure hdinsight script-action create <clustername> -g <resourcegroupname> -n <scriptname> -u
<scriptURI> -t <nodetypes>
```

If you omit parameters for this command, you are prompted for them. If the script you specify with `-u` accepts parameters, you can specify them using the `-p` parameter.

Valid node types are `headnode`, `workernode`, and `zookeeper`. If the script should be applied to multiple node types, specify the types separated by a `:`. For example, `-n headnode;workernode`.

To persist the script, add the `--persistOnSuccess`. You can also persist the script later by using `azure hdinsight script-action persisted set`.

Once the job completes, you receive output similar to the following text:

```
info: Executing command hdinsight script-action create
+ Executing Script Action on HDInsight cluster
data: Operation Info
data: -----
data: Operation status:
data: Operation ID: b707b10e-e633-45c0-baa9-8aed3d348c13
info: hdinsight script-action create command OK
```

Apply a Script Action to a running cluster using REST API

See [Run Script Actions on a running cluster](#).

Apply a Script Action to a running cluster from the HDInsight .NET SDK

For an example of using the .NET SDK to apply scripts to a cluster, see <https://github.com/Azure-Samples/hdinsight-dotnet-script-action>.

View history, promote, and demote Script Actions

Using the Azure portal

1. From the [Azure portal](#), select your HDInsight cluster.
2. From the HDInsight cluster blade, select the **Script Actions** tile.

The screenshot shows the Azure HDInsight cluster blade. At the top, there are 'Quick Links' for Cluster Dashboard, Ambari Views, and Scale Cluster. Below that is a 'Usage' section showing cores in North Central US for subscription. A donut chart indicates 170 cores in total, with 16 assigned to this cluster and 92 to other clusters. To the right of the usage section is an 'Applications' tile and a 'Script Actions' tile, which is highlighted with a red box.

NOTE

You can also select **All settings** and then select **Script Actions** from the Settings blade.

3. A history of scripts for this cluster is displayed on the Script Actions blade. This information includes a list of persisted scripts. In the screenshot below, you can see that the Solr script has been ran on this cluster. The screenshot does not show any persisted scripts.

The screenshot shows the 'Script actions' blade. At the top, it says 'PERSISTED SCRIPT ACTIONS' and 'No script actions found'. Below that is 'SCRIPT ACTION HISTORY' with a single entry: 'Add an Azure Storage account' submitted by user on 5/22/2017.

4. Selecting a script from the history displays the Properties blade for this script. From the top of the blade, you can rerun the script or promote it.

The 'Properties' blade shows details for the script 'Add an Azure Storage account'. It includes fields for NAME, URI, PARAMETERS, ROLES, and STATUS. The 'Promote' button at the top is highlighted with a red box.

5. You can also use the ... to the right of entries on the Script Actions blade to perform actions.

The 'PERSISTED SCRIPT ACTIONS' table has a row for 'Giraph' selected, highlighted with a blue background. To the right of this row is a context menu with options: 'Pin to dashboard' (with a pin icon) and 'Delete' (with a red box around it).

Using Azure PowerShell

USE THE FOLLOWING...	TO ...
Get-AzureRmHDInsightPersistedScriptAction	Retrieve information on persisted script actions
Get-AzureRmHDInsightScriptActionHistory	Retrieve a history of script actions applied to the cluster, or details for a specific script
Set-AzureRmHDInsightPersistedScriptAction	Promotes an ad hoc script action to a persisted script action
Remove-AzureRmHDInsightPersistedScriptAction	Demotes a persisted script action to an ad hoc action

IMPORTANT

Using `Remove-AzureRmHDInsightPersistedScriptAction` does not undo the actions performed by a script. This cmdlet only removes the persisted flag.

The following example script demonstrates using the cmdlets to promote, then demote a script.

```
# Get a history of scripts
Get-AzureRmHDInsightScriptActionHistory -ClusterName mycluster

# From the list, we want to get information on a specific script
Get-AzureRmHDInsightScriptActionHistory -ClusterName mycluster ` 
    -ScriptExecutionId 635920937765978529

# Promote this to a persisted script
# Note: the script must have a unique name to be promoted
# if the name is not unique, you receive an error
Set-AzureRmHDInsightPersistedScriptAction -ClusterName mycluster ` 
    -ScriptExecutionId 635920937765978529

# Demote the script back to ad hoc
# Note that demotion uses the unique script name instead of
# execution ID.
Remove-AzureRmHDInsightPersistedScriptAction -ClusterName mycluster ` 
    -Name "Install Giraph"
```

Using the Azure CLI

USE THE FOLLOWING...	TO ...
<code>azure hdinsight script-action persisted list <clustername></code>	Retrieve a list of persisted script actions
<code>azure hdinsight script-action persisted show <clustername> <scriptname></code>	Retrieve information on a specific persisted script action
<code>azure hdinsight script-action history list <clustername></code>	Retrieve a history of script actions applied to the cluster
<code>azure hdinsight script-action history show <clustername> <scriptname></code>	Retrieve information on a specific script action
<code>azure hdinsight script action persisted set <clustername> <scriptexecutionid></code>	Promotes an ad hoc script action to a persisted script action

USE THE FOLLOWING...	TO ...
<pre>azure hdinsight script-action persisted delete <clustername> <scriptname></pre>	Demotes a persisted script action to an ad hoc action

IMPORTANT

Using `azure hdinsight script-action persisted delete` does not undo the actions performed by a script. This cmdlet only removes the persisted flag.

Using the HDInsight .NET SDK

For an example of using the .NET SDK to retrieve script history from a cluster, promote or demote scripts, see <https://github.com/Azure-Samples/hdinsight-dotnet-script-action>.

NOTE

This example also demonstrates how to install an HDInsight application using the .NET SDK.

Support for open-source software used on HDInsight clusters

The Microsoft Azure HDInsight service uses an ecosystem of open-source technologies formed around Hadoop. Microsoft Azure provides a general level of support for open-source technologies. For more information, see the **Support Scope** section of the [Azure Support FAQ website](#). The HDInsight service provides an additional level of support for some of the components, as described below.

There are two types of open-source components that are available in the HDInsight service:

- **Built-in components** - These components are pre-installed on HDInsight clusters and provide core functionality of the cluster. For example, YARN ResourceManager, the Hive query language (HiveQL), and the Mahout library belong to this category. A full list of cluster components is available in [What's new in the Hadoop cluster versions provided by HDInsight](#).
- **Custom components** - You, as a user of the cluster, can install or use in your workload any component available in the community or created by you.

WARNING

Components provided with the HDInsight cluster are fully supported. Microsoft Support helps to isolate and resolve issues related to these components.

Custom components receive commercially reasonable support to help you to further troubleshoot the issue. This might result in resolving the issue OR asking you to engage available channels for the open source technologies where deep expertise for that technology is found. For example, there are many community sites that can be used, like: [MSDN forum for HDInsight](#), <http://stackoverflow.com>. Also Apache projects have project sites on <http://apache.org>, for example: [Hadoop](#).

The HDInsight service provides several ways to use custom components. The same level of support applies, regardless of how a component is used or installed on the cluster. The following list describes the most common ways that custom components can be used on HDInsight clusters:

1. Job submission - Hadoop or other types of jobs that execute or use custom components can be submitted to the cluster.
2. Cluster customization - During cluster creation, you can specify additional settings and custom

components that are installed on the cluster nodes.

3. Samples - For popular custom components, Microsoft and others may provide samples of how these components can be used on the HDInsight clusters. These samples are provided without support.

Troubleshooting

You can use Ambari web UI to view information logged by script actions. If the script fails during cluster creation, the logs are also available in the default storage account associated with the cluster. This section provides information on how to retrieve the logs using both these options.

Using the Ambari Web UI

1. In your browser, navigate to <https://CLUSTERNAME.azurehdinsight.net>. Replace CLUSTERNAME with the name of your HDInsight cluster.

When prompted, enter the admin account name (admin) and password for the cluster. You may have to reenter the admin credentials in a web form.

2. From the bar at the top of the page, select the **ops** entry. A list of current and previous operations performed on the cluster through Ambari is displayed.



3. Find the entries that have **run_customscriptaction** in the **Operations** column. These entries are created when the Script Actions run.

0 Background Operations Running			
Operations	Start Time	Duration	
! run_customscriptaction	Fri Aug 14 2015 15:25	15.71 secs	<div style="width: 100%;">100%</div>
! run_customscriptaction	Fri Aug 14 2015 15:24	10.98 secs	<div style="width: 100%;">100%</div>
! run_customscriptaction	Fri Aug 14 2015 15:22	13.83 secs	<div style="width: 100%;">100%</div>
! run_customscriptaction	Fri Aug 14 2015 15:19	24.63 secs	<div style="width: 100%;">100%</div>
! run_customscriptaction	Fri Aug 14 2015 15:15	25.53 secs	<div style="width: 100%;">100%</div>
! run_customscriptaction	Fri Aug 14 2015 15:13	17.43 secs	<div style="width: 100%;">100%</div>
! run_customscriptaction	Fri Aug 14 2015 15:11	28.33 secs	<div style="width: 100%;">100%</div>
! run_customscriptaction	Fri Aug 14 2015 15:04	9.84 secs	<div style="width: 100%;">100%</div>

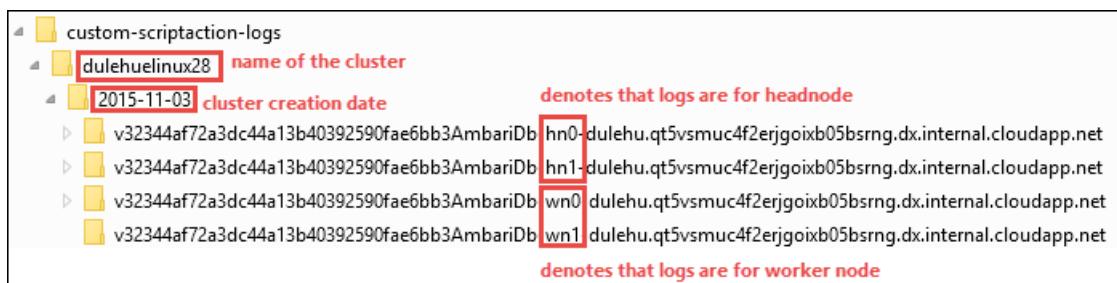
To view the STDOUT and STDERR output, select the run\customscriptaction entry and drill down through the links. This output is generated when the script runs, and may contain useful information.

Access logs from the default storage account

If the cluster creation fails due to a script action error, the logs can be accessed from the default storage account.

- The storage logs are available at

```
\STORAGE_ACCOUNT_NAME\DEFAULT_CONTAINER_NAME\custom-scriptaction-logs\CLUSTER_NAME\DATE .
```



Under this directory, the logs are organized separately for headnode, workernode, and zookeeper nodes.

Some examples are:

- **Headnode** - <uniqueidentifier>AmbariDb-hn0-<generated_value>.cloudapp.net
 - **Worker node** - <uniqueidentifier>AmbariDb-wn0-<generated_value>.cloudapp.net
 - **Zookeeper node** - <uniqueidentifier>AmbariDb-zk0-<generated_value>.cloudapp.net
- All stdout and stderr of the corresponding host is uploaded to the storage account. There is one **output-* .txt** and **errors-* .txt** for each script action. The output-* .txt file contains information about the URI of the script that got run on the host. For example

```
'Start downloading script locally: ',  
u'https://hdiconfigactions.blob.core.windows.net/linuxrconfigactionv01/r-installer-v01.sh'
```

- It's possible that you repeatedly create a script action cluster with the same name. In such case, you can distinguish the relevant logs based on the DATE folder name. For example, the folder structure for a cluster (mycluster) created on different dates appears similar to the following log entries:

```
\STORAGE_ACCOUNT_NAME\DEFAULT_CONTAINER_NAME\custom-scriptaction-logs\mycluster\2015-10-04  
\STORAGE_ACCOUNT_NAME\DEFAULT_CONTAINER_NAME\custom-scriptaction-logs\mycluster\2015-10-05
```

- If you create a script action cluster with the same name on the same day, you can use the unique prefix to identify the relevant log files.
- If you create a cluster at the end of the day, it's possible that the log files span across two days. In such cases, you see two different date folders for the same cluster.
- Uploading log files to the default container can take up to 5 mins, especially for large clusters. So, if you want to access the logs, you should not immediately delete the cluster if a script action fails.

Ambri watchdog

WARNING

Do not change the password for the Ambari Watchdog (hdinsightwatchdog) on your Linux-based HDInsight cluster. Changing the password for this account breaks the ability to run new script actions on the HDInsight cluster.

Cannot import name BlobService

Symptoms: The script action fails, and an error similar to the following example is displayed when you view the operation in Ambari:

```
Traceback (most recent call list):  
  File "/var/lib/ambari-agent/cache/custom_actions/scripts/run_customscriptaction.py", line 21, in <module>  
    from azure.storage.blob import BlobService  
ImportError: cannot import name BlobService
```

Cause: This error occurs if you upgrade the Python Azure Storage client that is included with the HDInsight cluster. HDInsight expects Azure Storage client 0.20.0.

Resolution: To resolve this error, manually connect to each cluster node using `ssh` and use the following command to reinstall the correct storage client version:

```
sudo pip install azure-storage==0.20.0
```

For information on connecting to the cluster with SSH, see [Use SSH with HDInsight](#).

History doesn't show scripts used during cluster creation

If your cluster was created before March 15th, 2016, you may not see an entry in Script Action history. If you resize the cluster after March 15th, 2016, the scripts using during cluster creation appear in history as they are applied to new nodes in the cluster as part of the resize operation.

There are two exceptions:

- If your cluster was created before September 1st, 2015. This date is when Script Actions were introduced. Any cluster created before this date could not have used Script Actions for cluster creation.
- If you used multiple Script Actions during cluster creation, and used the same name for multiple scripts, or the same name, same URI, but different parameters for multiple scripts. In these cases, you receive the following error:

No new script actions can be executed on this cluster due to conflicting script names in existing scripts. Script names provided at cluster create must be all unique. Existing scripts will still be executed on resize.

Next steps

- [Develop Script Action scripts for HDInsight](#)
- [Install and use Solr on HDInsight clusters](#)
- [Install and use Giraph on HDInsight clusters](#)
- [Add additional storage to an HDInsight cluster](#)

Script action development with HDInsight

8/16/2017 • 13 min to read • [Edit Online](#)

Learn how to customize your HDInsight cluster using Bash scripts. Script actions are a way to customize HDInsight during or after cluster creation.

IMPORTANT

The steps in this document require an HDInsight cluster that uses Linux. Linux is the only operating system used on HDInsight version 3.4 or greater. For more information, see [HDInsight retirement on Windows](#).

What are script actions

Script actions are Bash scripts that Azure runs on the cluster nodes to make configuration changes or install software. A script action is executed as root, and provides full access rights to the cluster nodes.

Script actions can be applied through the following methods:

USE THIS METHOD TO APPLY A SCRIPT...	DURING CLUSTER CREATION...	ON A RUNNING CLUSTER...
Azure portal	✓	✓
Azure PowerShell	✓	✓
Azure CLI		✓
HDInsight .NET SDK	✓	✓
Azure Resource Manager Template	✓	

For more information on using these methods to apply script actions, see [Customize HDInsight clusters using script actions](#).

Best practices for script development

When you develop a custom script for an HDInsight cluster, there are several best practices to keep in mind:

- [Target the Hadoop version](#)
- [Target the OS Version](#)
- [Provide stable links to script resources](#)
- [Use pre-compiled resources](#)
- [Ensure that the cluster customization script is idempotent](#)
- [Ensure high availability of the cluster architecture](#)
- [Configure the custom components to use Azure Blob storage](#)
- [Write information to STDOUT and STDERR](#)
- [Save files as ASCII with LF line endings](#)
- [Use retry logic to recover from transient errors](#)

IMPORTANT

Script actions must complete within 60 minutes or the process fails. During node provisioning, the script runs concurrently with other setup and configuration processes. Competition for resources such as CPU time or network bandwidth may cause the script to take longer to finish than it does in your development environment.

Target the Hadoop version

Different versions of HDInsight have different versions of Hadoop services and components installed. If your script expects a specific version of a service or component, you should only use the script with the version of HDInsight that includes the required components. You can find information on component versions included with HDInsight using the [HDInsight component versioning](#) document.

Target the OS version

Linux-based HDInsight is based on the Ubuntu Linux distribution. Different versions of HDInsight rely on different versions of Ubuntu, which may change how your script behaves. For example, HDInsight 3.4 and earlier are based on Ubuntu versions that use Upstart. Version 3.5 is based on Ubuntu 16.04, which uses Systemd. Systemd and Upstart rely on different commands, so your script should be written to work with both.

Another important difference between HDInsight 3.4 and 3.5 is that `JAVA_HOME` now points to Java 8.

You can check the OS version by using `lsb_release`. The following code demonstrates how to determine if the script is running on Ubuntu 14 or 16:

```
OS_VERSION=$(lsb_release -sr)
if [[ $OS_VERSION == 14* ]]; then
    echo "OS verion is $OS_VERSION. Using hue-binaries-14-04."
    HUE_TARFILE=hue-binaries-14-04.tgz
elif [[ $OS_VERSION == 16* ]]; then
    echo "OS verion is $OS_VERSION. Using hue-binaries-16-04."
    HUE_TARFILE=hue-binaries-16-04.tgz
fi
...
if [[ $OS_VERSION == 16* ]]; then
    echo "Using systemd configuration"
    systemctl daemon-reload
    systemctl stop webwasb.service
    systemctl start webwasb.service
else
    echo "Using upstart configuration"
    initctl reload-configuration
    stop webwasb
    start webwasb
fi
...
if [[ $OS_VERSION == 14* ]]; then
    export JAVA_HOME=/usr/lib/jvm/java-7-openjdk-amd64
elif [[ $OS_VERSION == 16* ]]; then
    export JAVA_HOME=/usr/lib/jvm/java-8-openjdk-amd64
fi
```

You can find the full script that contains these snippets at

<https://hdiconfigactions.blob.core.windows.net/linuxhueconfigactionv02/install-hue-uber-v02.sh>.

For the version of Ubuntu that is used by HDInsight, see the [HDInsight component version](#) document.

To understand the differences between Systemd and Upstart, see [Systemd for Upstart users](#).

Provide stable links to script resources

The script and associated resources must remain available throughout the lifetime of the cluster. These resources

are required if new nodes are added to the cluster during scaling operations.

The best practice is to download and archive everything in an Azure Storage account on your subscription.

IMPORTANT

The storage account used must be the default storage account for the cluster or a public, read-only container on any other storage account.

For example, the samples provided by Microsoft are stored in the <https://hdiconfigactions.blob.core.windows.net/> storage account. This is a public, read-only container maintained by the HDInsight team.

Use pre-compiled resources

To reduce the time it takes to run the script, avoid operations that compile resources from source code. For example, pre-compile resources and store them in an Azure Storage account blob in the same data center as HDInsight.

Ensure that the cluster customization script is idempotent

Scripts must be idempotent. If the script runs multiple times, it should return the cluster to the same state every time.

For example, a script that modifies configuration files should not add duplicate entries if ran multiple times.

Ensure high availability of the cluster architecture

Linux-based HDInsight clusters provide two head nodes that are active within the cluster, and script actions run on both nodes. If the components you install expect only one head node, do not install the components on both head nodes.

IMPORTANT

Services provided as part of HDInsight are designed to fail over between the two head nodes as needed. This functionality is not extended to custom components installed through script actions. If you need high availability for custom components, you must implement your own failover mechanism.

Configure the custom components to use Azure Blob storage

Components that you install on the cluster might have a default configuration that uses Hadoop Distributed File System (HDFS) storage. HDInsight uses either Azure Storage or Data Lake Store as the default storage. Both provide an HDFS compatible file system that persists data even if the cluster is deleted. You may need to configure components you install to use WASB or ADL instead of HDFS.

For most operations, you do not need to specify the file system. For example, the following copies the giraph-examples.jar file from the local file system to cluster storage:

```
hdfs dfs -put /usr/hdp/current/giraph/giraph-examples.jar /example/jars/
```

In this example, the `hdfs` command transparently uses the default cluster storage. For some operations, you may need to specify the URI. For example, `adl:///example/jars` for Data Lake Store or `wasb:///example/jars` for Azure Storage.

Write information to STDOUT and STDERR

HDInsight logs script output that is written to STDOUT and STDERR. You can view this information using the Ambari web UI.

NOTE

Ambari is only available if the cluster is successfully created. If you use a script action during cluster creation, and creation fails, see the troubleshooting section [Customize HDInsight clusters using script action](#) for other ways of accessing logged information.

Most utilities and installation packages already write information to STDOUT and STDERR, however you may want to add additional logging. To send text to STDOUT, use `echo`. For example:

```
echo "Getting ready to install Foo"
```

By default, `echo` sends the string to STDOUT. To direct it to STDERR, add `>&2` before `echo`. For example:

```
>&2 echo "An error occurred installing Foo"
```

This redirects information written to STDOUT to STDERR (2) instead. For more information on IO redirection, see <http://www.tldp.org/LDP/abs/html/io-redirection.html>.

For more information on viewing information logged by script actions, see [Customize HDInsight clusters using script action](#)

Save files as ASCII with LF line endings

Bash scripts should be stored as ASCII format, with lines terminated by LF. Files that are stored as UTF-8, or use CRLF as the line ending may fail with the following error:

```
$'\r': command not found
line 1: #!/usr/bin/env: No such file or directory
```

Use retry logic to recover from transient errors

When downloading files, installing packages using apt-get, or other actions that transmit data over the internet, the action may fail due to transient networking errors. For example, the remote resource you are communicating with may be in the process of failing over to a backup node.

To make your script resilient to transient errors, you can implement retry logic. The following function demonstrates how to implement retry logic. It retries the operation three times before failing.

```

#retry
MAXATTEMPTS=3

retry() {
    local -r CMD="$@"
    local -i ATTEMPTNUM=1
    local -i RETRYINTERVAL=2

    until $CMD
    do
        if (( ATTEMPTNUM == MAXATTEMPTS ))
        then
            echo "Attempt $ATTEMPTNUM failed. no more attempts left."
            return 1
        else
            echo "Attempt $ATTEMPTNUM failed! Retrying in $RETRYINTERVAL seconds..."
            sleep $(( RETRYINTERVAL ))
            ATTEMPTNUM=$ATTEMPTNUM+1
        fi
    done
}

```

The following examples demonstrate how to use this function.

```

retry ls -ltr foo

retry wget -O ./tmpfile.sh https://hdiconfigactions.blob.core.windows.net/linuxhueconfigactionv02/install-hue-
uber-v02.sh

```

Helper methods for custom scripts

Script action helper methods are utilities that you can use while writing custom scripts. These methods are contained in the <https://hdiconfigactions.blob.core.windows.net/linuxconfigactionmodulev01/HDInsightUtilities-v01.sh> script. Use the following to download and use them as part of your script:

```

# Import the helper method module.
wget -O /tmp/HDInsightUtilities-v01.sh -q
https://hdiconfigactions.blob.core.windows.net/linuxconfigactionmodulev01/HDInsightUtilities-v01.sh && source
/tmp/HDInsightUtilities-v01.sh && rm -f /tmp/HDInsightUtilities-v01.sh

```

The following helpers available for use in your script:

HELPER USAGE	DESCRIPTION
<code>download_file SOURCEURL DESTFILEPATH [OVERWRITE]</code>	Downloads a file from the source URI to the specified file path. By default, it does not overwrite an existing file.
<code>untar_file TARFILE DESTDIR</code>	Extracts a tar file (using <code>-xf</code>) to the destination directory.
<code>test_is_headnode</code>	If ran on a cluster head node, return 1; otherwise, 0.
<code>test_is_datanode</code>	If the current node is a data (worker) node, return a 1; otherwise, 0.
<code>test_is_first_datanode</code>	If the current node is the first data (worker) node (named workernode0) return a 1; otherwise, 0.

Helper Usage	Description
<code>get_headnodes</code>	Return the fully qualified domain name of the headnodes in the cluster. Names are comma delimited. An empty string is returned on error.
<code>get_primary_headnode</code>	Gets the fully qualified domain name of the primary headnode. An empty string is returned on error.
<code>get_secondary_headnode</code>	Gets the fully qualified domain name of the secondary headnode. An empty string is returned on error.
<code>get_primary_headnode_number</code>	Gets the numeric suffix of the primary headnode. An empty string is returned on error.
<code>get_secondary_headnode_number</code>	Gets the numeric suffix of the secondary headnode. An empty string is returned on error.

Common usage patterns

This section provides guidance on implementing some of the common usage patterns that you might run into while writing your own custom script.

Passing parameters to a script

In some cases, your script may require parameters. For example, you may need the admin password for the cluster when using the Ambari REST API.

Parameters passed to the script are known as *positional parameters*, and are assigned to `$1` for the first parameter, `$2` for the second, and so-on. `$0` contains the name of the script itself.

Values passed to the script as parameters should be enclosed by single quotes (''). Doing so ensures that the passed value is treated as a literal.

Setting environment variables

Setting an environment variable is performed by the following statement:

```
VARIABLENAME=value
```

Where VARIABLENAME is the name of the variable. To access the variable, use `$VARIABLENAME`. For example, to assign a value provided by a positional parameter as an environment variable named PASSWORD, you would use the following statement:

```
PASSWORD=$1
```

Subsequent access to the information could then use `$PASSWORD`.

Environment variables set within the script only exist within the scope of the script. In some cases, you may need to add system-wide environment variables that will persist after the script has finished. To add system-wide environment variables, add the variable to `/etc/environment`. For example, the following statement adds

```
HADOOP_CONF_DIR :
```

```
echo "HADOOP_CONF_DIR=/etc/hadoop/conf" | sudo tee -a /etc/environment
```

Access to locations where the custom scripts are stored

Scripts used to customize a cluster needs to be stored in one of the following locations:

- An **Azure Storage account** that is associated with the cluster.
- An **additional storage account** associated with the cluster.
- A **publicly readable URI**. For example, a URL to data stored on OneDrive, Dropbox, or other file hosting service.
- An **Azure Data Lake Store account** that is associated with the HDInsight cluster. For more information on using Azure Data Lake Store with HDInsight, see [Create an HDInsight cluster with Data Lake Store](#).

NOTE

The service principal HDInsight uses to access Data Lake Store must have read access to the script.

Resources used by the script must also be publicly available.

Storing the files in an Azure Storage account or Azure Data Lake Store provides fast access, as both within the Azure network.

NOTE

The URI format used to reference the script differs depending on the service being used. For storage accounts associated with the HDInsight cluster, use `wasb://` or `wasbs://`. For publicly readable URIs, use `http://` or `https://`. For Data Lake Store, use `adl://`.

Checking the operating system version

Different versions of HDInsight rely on specific versions of Ubuntu. There may be differences between OS versions that you must check for in your script. For example, you may need to install a binary that is tied to the version of Ubuntu.

To check the OS version, use `lsb_release`. For example, the following script demonstrates how to reference a specific tar file depending on the OS version:

```
OS_VERSION=$(lsb_release -sr)
if [[ $OS_VERSION == 14* ]]; then
    echo "OS verion is $OS_VERSION. Using hue-binaries-14-04."
    HUE_TARFILE=hue-binaries-14-04.tgz
elif [[ $OS_VERSION == 16* ]]; then
    echo "OS verion is $OS_VERSION. Using hue-binaries-16-04."
    HUE_TARFILE=hue-binaries-16-04.tgz
fi
```

Checklist for deploying a script action

Here are the steps we took when preparing to deploy these scripts:

- Put the files that contain the custom scripts in a place that is accessible by the cluster nodes during deployment. For example, the default storage for the cluster. Files can also be stored in publicly readable hosting services.
- Verify that the script is impotent. Doing so allows the script to be executed multiple times on the same node.
- Use a temporary file directory /tmp to keep the downloaded files used by the scripts and then clean them up after scripts have executed.
- If OS-level settings or Hadoop service configuration files are changed, you may want to restart HDInsight

services.

How to run a script action

You can use script actions to customize HDInsight clusters using the following methods:

- Azure portal
- Azure PowerShell
- Azure Resource Manager templates
- The HDInsight .NET SDK.

For more information on using each method, see [How to use script action](#).

Custom script samples

Microsoft provides sample scripts to install components on an HDInsight cluster. See the following links for more example script actions.

- [Install and use Hue on HDInsight clusters](#)
- [Install and use Solr on HDInsight clusters](#)
- [Install and use Giraph on HDInsight clusters](#)
- [Install or upgrade Mono on HDInsight clusters](#)

Troubleshooting

The following are errors you may encounter when using scripts you have developed:

Error: `$'\r'`: command not found. Sometimes followed by `syntax error: unexpected end of file`.

Cause: This error is caused when the lines in a script end with CRLF. Unix systems expect only LF as the line ending.

This problem most often occurs when the script is authored on a Windows environment, as CRLF is a common line ending for many text editors on Windows.

Resolution: If it is an option in your text editor, select Unix format or LF for the line ending. You may also use the following commands on a Unix system to change the CRLF to an LF:

NOTE

The following commands are roughly equivalent in that they should change the CRLF line endings to LF. Select one based on the utilities available on your system.

COMMAND	NOTES
<code>unix2dos -b INFILe</code>	The original file is backed up with a .BAK extension
<code>tr -d '\r' < INFILe > OUTFILE</code>	OUTFILE contains a version with only LF endings
<code>perl -pi -e 's/\r\n/\n/g' INFILe</code>	Modifies the file directly
<code>sed 's/\$"/`echo \\\r`/" INFILe > OUTFILE</code>	OUTFILE contains a version with only LF endings.

Error: `line 1: #!/usr/bin/env: No such file or directory`.

Cause: This error occurs when the script was saved as UTF-8 with a Byte Order Mark (BOM).

Resolution: Save the file either as ASCII, or as UTF-8 without a BOM. You may also use the following command on a Linux or Unix system to create a file without the BOM:

```
awk 'NR==1{sub(/^\xEF\xBB\xBF/, "")}{print}' INFILE > OUTFILE
```

Replace `INFILE` with the file containing the BOM. `OUTFILE` should be a new file name, which contains the script without the BOM.

Next steps

- Learn how to [Customize HDInsight clusters using script action](#)
- Use the [HDInsight .NET SDK reference](#) to learn more about creating .NET applications that manage HDInsight
- Use the [HDInsight REST API](#) to learn how to use REST to perform management actions on HDInsight clusters.

OS patching for HDInsight

8/15/2017 • 3 min to read • [Edit Online](#)

As a managed Hadoop service, HDInsight takes care of patching the OS of the underlying VMs used by HDInsight clusters. As of August 1, 2016, we have changed the guest OS patching policy for Linux-based HDInsight clusters (version 3.4 or greater). The goal of the new policy is to significantly reduce the number of reboots due to patching. The new policy will continue to patch virtual machines (VMs) on Linux clusters every Monday or Thursday starting at 12AM UTC in a staggered fashion across nodes in any given cluster. However, any given VM will only reboot at most once every 30 days due to guest OS patching. In addition, the first reboot for a newly created cluster will not happen sooner than 30 days from the cluster creation date. Patches will be effective once the VMs are rebooted.

How to configure the OS patching schedule for Linux-based HDInsight clusters

The virtual machines in an HDInsight cluster need to be rebooted occasionally so that important security patches can be installed. As of August 1, 2016, new Linux-based HDInsight clusters (version 3.4 or greater,) are rebooted using the following schedule:

1. A virtual machine in the cluster can only reboot for patches at most, once within a 30-day period.
2. The reboot occurs starting at 12AM UTC.
3. The reboot process is staggered across virtual machines in the cluster, so the cluster is still available during the reboot process.
4. The first reboot for a newly created cluster will not happen sooner than 30 days after the cluster creation date.

Using the script action described in this article, you can modify the OS patching schedule as follows:

1. Enable or disable automatic reboots
2. Set the frequency of reboots (days between reboots)
3. Set the day of the week when a reboot occurs

NOTE

This script action will only work with Linux-based HDInsight clusters created after August 1st, 2016. Patches will be effective only when VMs are rebooted.

How to use the script

When using this script requires the following information:

1. The script location: <https://hdiconfigactions.blob.core.windows.net/linuxospatchingrebootconfigv01/os-patching-reboot-config.sh>. HDInsight uses this URI to find and run the script on all the virtual machines in the cluster.
2. The cluster node types that the script is applied to: headnode, workernode, zookeeper. This script must be applied to all node types in the cluster. If it is not applied to a node type, then the virtual machines for that node type will continue to use the previous patching schedule.
3. Parameter: This script accepts three numeric parameters:

PARAMETER	DEFINITION
Enable/disable automatic reboots	0 or 1. A value of 0 disables automatic reboots while 1 enables automatic reboots.
Frequency	7 to 90 (inclusive). The number of days to wait before rebooting the virtual machines for patches that require a reboot.
Day of week	1 to 7 (inclusive). A value of 1 indicates the reboot should occur on a Monday, and 7 indicates a Sunday. For example, using parameters of 1 60 2 results in automatic reboots every 60 days (at most) on Tuesday.
Persistence	When applying a script action to an existing cluster, you can mark the script as persisted. Persisted scripts are applied when new workernodes are added to the cluster through scaling operations.

NOTE

You must mark this script as persisted when applying to an existing cluster. Otherwise, any new nodes created through scaling operations will use the default patching schedule. If you apply the script as part of the cluster creation process, it is persisted automatically.

Next steps

For specific steps on using the script action, see the following sections in the [Customize Linux-based HDInsight clusters using script action](#):

- [Use a script action during cluster creation](#)
- [Apply a script action to a running cluster](#)

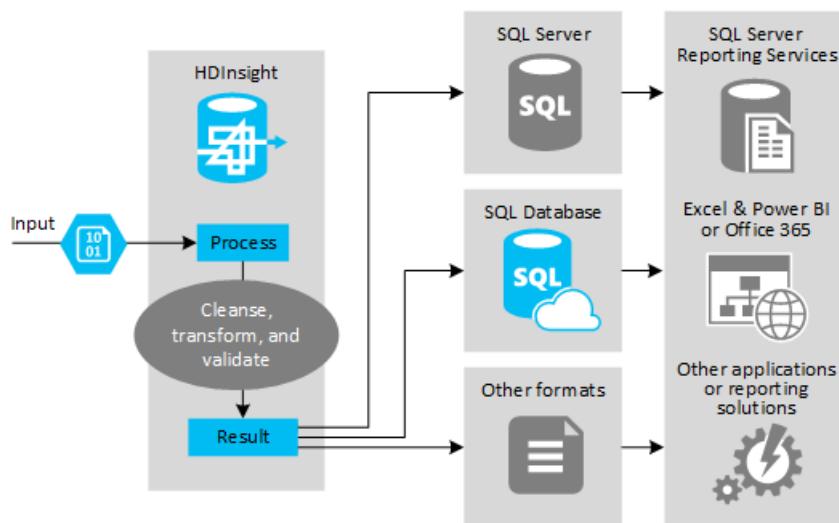
Using Apache Hive as an ETL Tool

8/15/2017 • 5 min to read • [Edit Online](#)

You will typically need to cleanse and transform data before loading it into a destination suitable for analytics. Extract, Transform, and Load (ETL) operations are used to prepare data and load them into a data destination. One of the more popular uses of Hive on HDInsight is to take unstructured data and use it to process and then load data into a relational data warehouse to support decision support systems. In this approach, data is extracted from the source and stored in scalable storage (such as Azure Storage blobs or Azure Data Lake Store). The data is then transformed using a sequence of Hive queries and is ultimately staged within Hive in preparation for bulk loading into the destination data store. This is the typical ETL process with Hive.

Use case and model overview

The figure below shows an overview of the use case and model for ETL automation. Input data is transformed to generate the appropriate output. During that transformation the data can change shape, data type, and even language. ETL processes can convert Imperial to Metric, change time zones, improve precision to properly align with existing data in the destination. ETL processes can also combine new data with existing data to either keep reporting up to date or provide further insight into existing data. Applications such as reporting tools and services can then consume this data in an appropriate format, and use it for a variety of purposes.



Hadoop is typically used in ETL processes that import either a massive amount of text files (like CSVs) or a smaller, but frequently changing amount of text files, or both massive and frequently changing. Hive is a great tool to use to prepare the data before loading it into the data destination. Hive allows you to create a schema over the CSV and use a SQL-like language to generate MapReduce programs that interact with the data. This is a compelling benefit of Hive, since SQL is an accessible language that most developers have already mastered and they can quickly get to productive without having to learn to implement MapReduce programs in Java.

The typical steps to using Hive to perform ETL are as follows:

- 1) Load data into Azure Data Lake Store or Azure Blob Storage.
- 2) Create an HDInsight cluster and connect the data store with HDInsight. Also, create a Metadata Store database (using Azure SQL Database) for use by Hive in storing your schemas.
- 3) Define the schema to apply at read-time over data in the data store:

```

```
DROP TABLE IF EXISTS hvac;

--create the hvac table on comma-separated sensor data stored in Azure Storage blobs

CREATE EXTERNAL TABLE hvac(`date` STRING, time STRING, targettemp BIGINT,
 actualtemp BIGINT,
 system BIGINT,
 systemage BIGINT,
 buildingid BIGINT)
ROW FORMAT DELIMITED FIELDS TERMINATED BY ','
STORED AS TEXTFILE LOCATION
'wasb://{container}@{storageaccount}.blob.core.windows.net/HdiSamples/SensorSampleData/hvac/';
```

```

5) Transform the data and load it into the destination. There are a few ways to use Hive during the transformation and loading:

- 1) Query and prep data using Hive and save it as a CSV in Azure Data Lake Store or Blob storage. Then use a tool like SQL Server Integration Services (SSIS) to acquire those CSVs and load the data into the destination relational database (like SQL Server).
- 2) Query the data directly from Excel or C# using the Hive ODBC driver.
- 3) Use [Apache Sqoop](https://docs.microsoft.com/en-us/azure/hdinsight/hdinsight-use-sqoop-mac-linux) to read the prepared flat CSV files and load them into the destination relational database.

Data sources

Data sources are typically external data that can be matched to existing data in your data store. Some examples are:

- Social media data, log files, sensors, and applications that generate data files.
- Datasets obtained from data providers, like weather statistics, or vendor sales numbers.
- Streaming data captured, filtered, and processed through a suitable tool or framework (see Collecting and loading data into HDInsight).

Output targets

We can use hive to output data to a variety of targets including:

- A relational database such as SQL Server or Azure SQL Database.
- A data warehouse, like Azure SQL Data Warehouse.
- Excel
- Azure table and blob storage.
- Applications or services that require data to be processed into specific formats, or as files that contain specific types of information structure.
- A JSON Document Store like [CosmosDB](#).

Considerations

There are some important points to consider when choosing to perform ETL:

- This model is typically used when you want to:
 - Load stream data or large volumes of semi-structured or unstructured data from external sources into an existing database or information system.
 - Cleanse, transform, and validate the data before loading it; perhaps by using more than one

transformation pass through the cluster.

- Generate reports and visualizations that are regularly updated. This would be great if the report takes too long to generate during the day, so instead you schedule the report to run at night. You can use Azure Scheduler and PowerShell to automatically run a Hive query.
- If the target for the data is not a database, you can generate a file in the appropriate format within the query, like a CSV. This can easily be imported into Excel or Power BI.
- If you need to execute several operations on the data as part of the ETL process you should consider how you manage these. If they are controlled by an external program, rather than as a workflow within the solution, you will need to decide whether some can be executed in parallel, and you must be able to detect when each job has completed. Using a workflow mechanism such as Oozie within Hadoop may be easier than trying to orchestrate several operations using external scripts or custom programs. See [Workflow and job orchestration](#) for more information about Oozie.

See Next

- [ETL at scale](#): Learn more about performing ETL at scale.
- [Operationalize Data Pipelines with Oozie](#): Learn how to build a data pipeline that uses Hive to summarize CSV flight delay data, stage the prepared data in Azure Storage blobs and then use Sqoop to load the summarized data into Azure SQL Database.
- [ETL Deep Dive](#): Walk thru an end-to-end ETL pipeline.

Query Hive through the JDBC driver in HDInsight

8/16/2017 • 4 min to read • [Edit Online](#)

Learn how to use the JDBC driver from a Java application to submit Hive queries to Hadoop in Azure HDInsight. The information in this document demonstrates how to connect programmatically and from the SQuirreL SQL client.

For more information on the Hive JDBC Interface, see [HiveJDBCInterface](#).

Prerequisites

- A Hadoop on HDInsight cluster. Either Linux-based or Windows-based clusters work.

IMPORTANT

Linux is the only operating system used on HDInsight version 3.4 or greater. For more information, see [HDInsight 3.3 retirement](#).

- [SQuirreL SQL](#). SQuirreL is a JDBC client application.
- The [Java Developer Kit \(JDK\) version 7](#) or higher.
- [Apache Maven](#). Maven is a project build system for Java projects that is used by the project associated with this article.

JDBC connection string

JDBC connections to an HDInsight cluster on Azure are made over 443, and the traffic is secured using SSL. The public gateway that the clusters sit behind redirects the traffic to the port that HiveServer2 is actually listening on. The following is an example connection string:

```
jdbc:hive2://CLUSTERNAME.azurehdinsight.net:443/default;transportMode=http;ssl=true;httpPath=/hive2
```

Replace `CLUSTERNAME` with the name of your HDInsight cluster.

Authentication

When establishing the connection, you must use the HDInsight cluster admin name and password to authenticate to the cluster gateway. When connecting from JDBC clients such as SQuirreL SQL, you must enter the admin name and password in client settings.

From a Java application, you must use the name and password when establishing a connection. For example, the following Java code opens a new connection using the connection string, admin name, and password:

```
DriverManager.getConnection(connectionString,clusterAdmin,clusterPassword);
```

Connect with SQuirreL SQL client

SQuirreL SQL is a JDBC client that can be used to remotely run Hive queries with your HDInsight cluster. The following steps assume that you have already installed SQuirreL SQL.

1. Copy the Hive JDBC drivers from your HDInsight cluster.

- For **Linux-based HDInsight**, use the following steps to download the required jar files.

a. Create a directory that contains the files. For example, `mkdir hivedriver`.

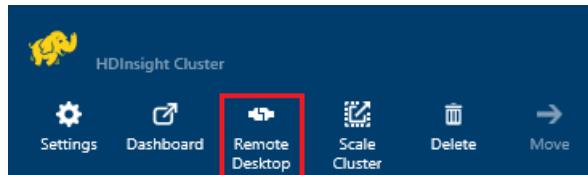
b. From a command line, use the following commands to copy the files from the HDInsight cluster:

```
scp USERNAME@CLUSTERNAME:/usr/hdp/current/hive-client/lib/hive-jdbc*standalone.jar .
scp USERNAME@CLUSTERNAME:/usr/hdp/current/hadoop-client/hadoop-common.jar .
scp USERNAME@CLUSTERNAME:/usr/hdp/current/hadoop-client/hadoop-auth.jar .
```

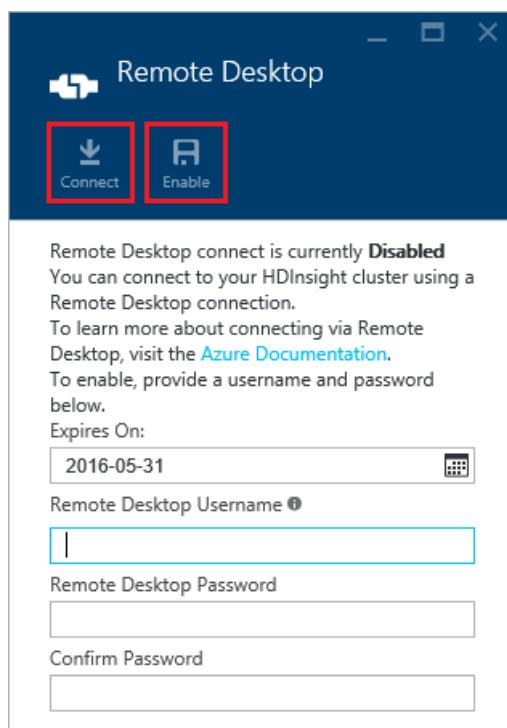
Replace `USERNAME` with the SSH user account name for the cluster. Replace `CLUSTERNAME` with the HDInsight cluster name.

- For **Windows-based HDInsight**, use the following steps to download the jar files.

a. From the Azure portal, select your HDInsight cluster, and then select the **Remote Desktop** icon.



b. On the Remote Desktop blade, use the **Connect** button to connect to the cluster. If the Remote Desktop is not enabled, use the form to provide a user name and password, then select **Enable** to enable Remote Desktop for the cluster.



After selecting **Connect**, a .rdp file is downloaded. Use this file to launch the Remote Desktop client. When prompted, use the user name and password you entered for Remote Desktop access.

c. Once connected, copy the following files from the Remote Desktop session to your local machine. Put them in a local directory named `hivedriver`.

- C:\apps\dist\hive-0.14.0.2.2.9.1-7\lib\hive-jdbc-0.14.0.2.2.9.1-7-standalone.jar
- C:\apps\dist\hadoop-2.6.0.2.2.9.1-7\share\hadoop\common\hadoop-common-2.6.0.2.2.9.1-7.jar
- C:\apps\dist\hadoop-2.6.0.2.2.9.1-7\share\hadoop\common\lib\hadoop-auth-2.6.0.2.2.9.1-7.jar

NOTE

The version numbers included in the paths and file names may be different for your cluster.

- d. Disconnect the Remote Desktop session once you have finished copying the files.

2. Start the SQuirreL SQL application. From the left of the window, select **Drivers**.

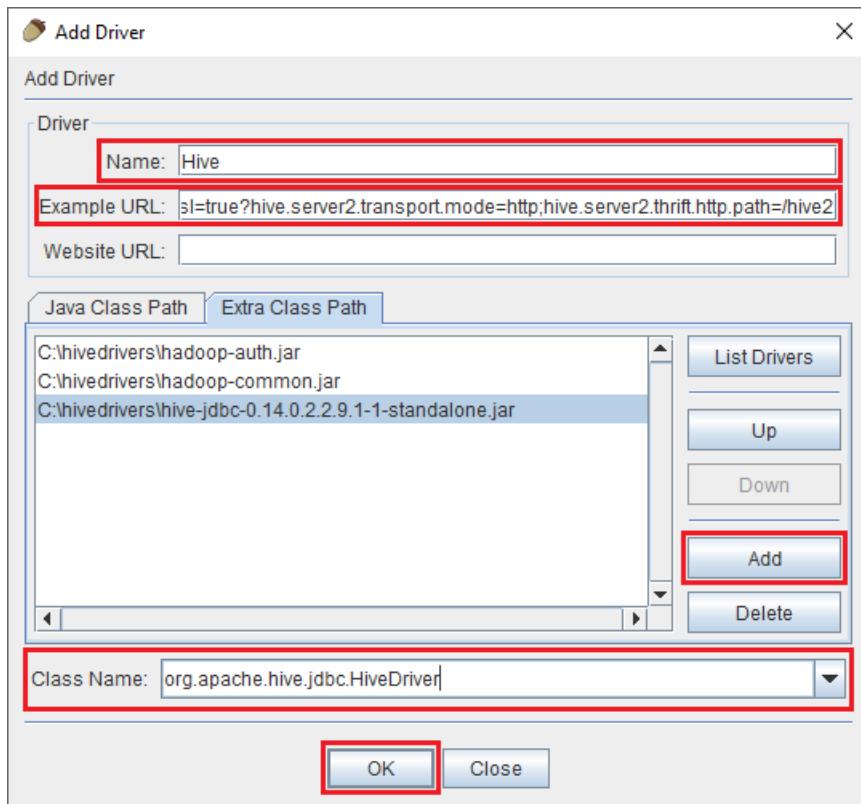


3. From the icons at the top of the **Drivers** dialog, select the + icon to create a driver.



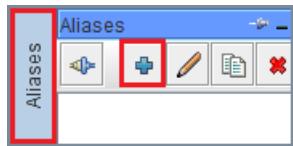
4. In the Add Driver dialog, add the following information:

- **Name:** Hive
- **Example URL:** `jdbc:hive2://localhost:443/default;transportMode=http;ssl=true;httpPath=/hive2`
- **Extra Class Path:** Use the Add button to add the jar files downloaded earlier
- **Class Name:** org.apache.hive.jdbc.HiveDriver



Click **OK** to save these settings.

- On the left of the SQuirreL SQL window, select **Aliases**. Then click the + icon to create a connection alias.

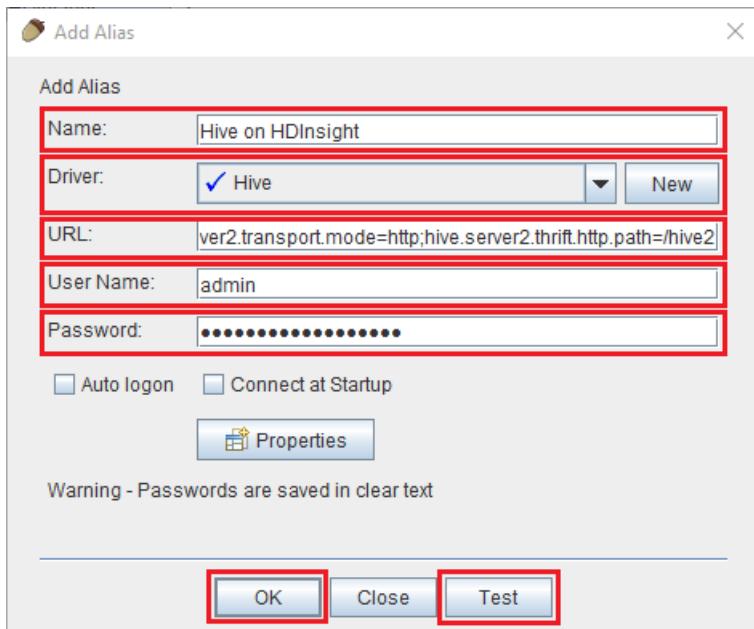


- Use the following values for the **Add Alias** dialog.

- Name:** Hive on HDInsight
- Driver:** Use the dropdown to select the **Hive** driver
- URL:**
jdbc:hive2://CLUSTERNAME.azurehdinsight.net:443/default;transportMode=http;ssl=true;httpPath=/hive2

Replace **CLUSTERNAME** with the name of your HDInsight cluster.

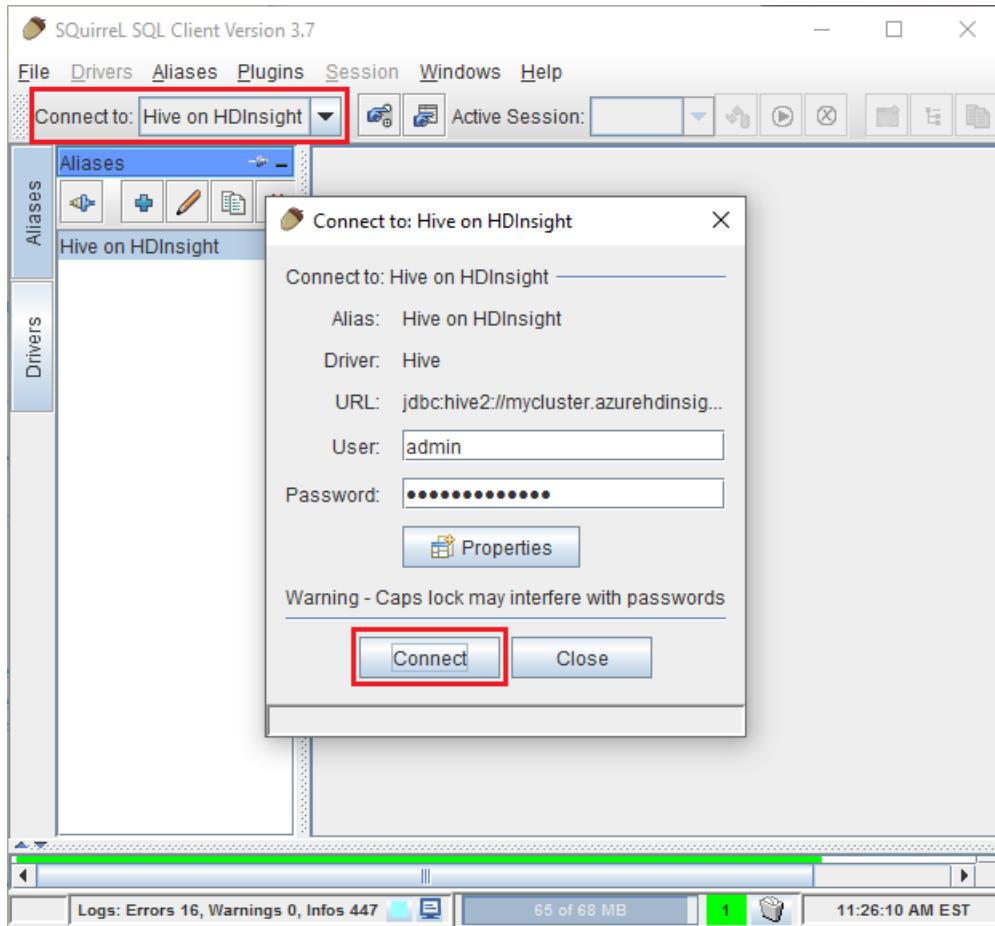
- User Name:** The cluster login account name for your HDInsight cluster. The default is `admin`.
- Password:** The password for the cluster login account.



Use the **Test** button to verify that the connection works. When **Connect to: Hive on HDInsight** dialog appears, select **Connect** to perform the test. If the test succeeds, you see a **Connection successful** dialog.

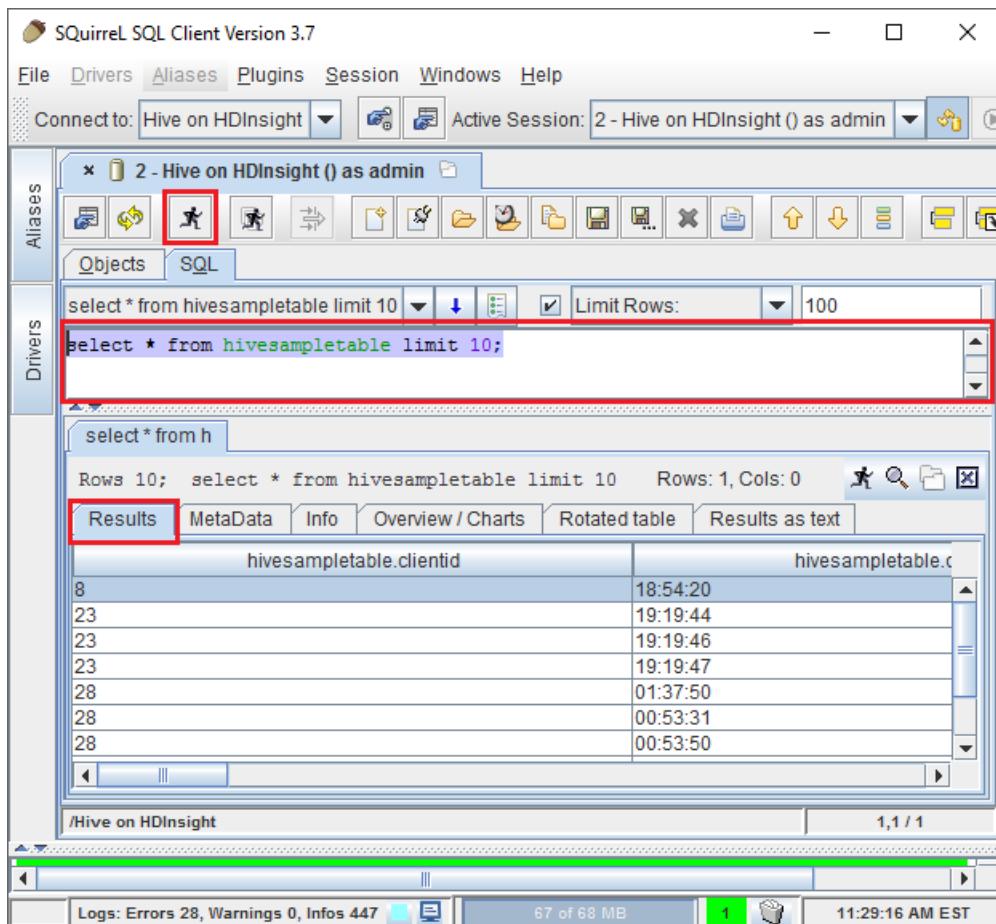
To save the connection alias, use the **Ok** button at the bottom of the **Add Alias** dialog.

7. From the **Connect to** dropdown at the top of SQuirreL SQL, select **Hive on HDInsight**. When prompted, select **Connect**.



8. Once connected, enter the following query into the SQL query dialog, and then select the **Run** icon. The results area should show the results of the query.

```
select * from hivesamplable limit 10;
```



Connect from an example Java application

An example of using a Java client to query Hive on HDInsight is available at <https://github.com/Azure-Samples/hdinsight-java-hive-jdbc>. Follow the instructions in the repository to build and run the sample.

Troubleshooting

Unexpected Error occurred attempting to open an SQL connection

Symptoms: When connecting to an HDInsight cluster that is version 3.3 or 3.4, you may receive an error that an unexpected error occurred. The stack trace for this error begins with the following lines:

```
java.util.concurrent.ExecutionException: java.lang.RuntimeException: java.lang.NoSuchMethodError:  
org.apache.commons.codec.binary.Base64.<init>(I)V  
at java.util.concurrent.FutureTask...{FutureTask.java:122}  
at java.util.concurrent.FutureTask.get(FutureTask.java:206)
```

Cause: This error is caused by a mismatch in the version of the commons-codecjar file used by SQuirreL and the one required by the Hive JDBC components.

Resolution: To fix this error, use the following steps:

1. Download the commons-codec jar file from your HDInsight cluster.

```
scp USERNAME@CLUSTERNAME:/usr/hdp/current/hive-client/lib/commons-codec*.jar ./commons-codec.jar
```

2. Exit SQuirreL, and then go to the directory where SQuirreL is installed on your system. In the SQuirreL

directory, under the `lib` directory, replace the existing commons-codec.jar with the one downloaded from the HDInsight cluster.

3. Restart SQuirreL. The error should no longer occur when connecting to Hive on HDInsight.

Next steps

Now that you have learned how to use JDBC to work with Hive, use the following links to explore other ways to work with Azure HDInsight.

- [Upload data to HDInsight](#)
- [Use Hive with HDInsight](#)
- [Use Pig with HDInsight](#)
- [Use MapReduce jobs with HDInsight](#)

Use a Java UDF with Hive in HDInsight

8/16/2017 • 4 min to read • [Edit Online](#)

Learn how to create a Java-based user-defined function (UDF) that works with Hive. The Java UDF in this example converts a table of text strings to all-lowercase characters.

Requirements

- An HDInsight cluster

IMPORTANT

Linux is the only operating system used on HDInsight version 3.4 or greater. For more information, see [HDInsight retirement on Windows](#).

Most steps in this document work on both Windows- and Linux-based clusters. However, the steps used to upload the compiled UDF to the cluster and run it are specific to Linux-based clusters. Links are provided to information that can be used with Windows-based clusters.

- [Java JDK](#) 8 or later (or an equivalent, such as OpenJDK)
- [Apache Maven](#)
- A text editor or Java IDE

IMPORTANT

If you create the Python files on a Windows client, you must use an editor that uses LF as a line ending. If you are not sure whether your editor uses LF or CRLF, see the [Troubleshooting](#) section for steps on removing the CR character.

Create an example Java UDF

1. From a command line, use the following to create a new Maven project:

```
mvn archetype:generate -DgroupId=com.microsoft.examples -DartifactId=ExampleUDF -  
DarchetypeArtifactId=maven-archetype-quickstart -DinteractiveMode=false
```

NOTE

If you are using PowerShell, you must put quotes around the parameters. For example,

```
mvn archetype:generate "-DgroupId=com.microsoft.examples" "-DartifactId=ExampleUDF" "-  
DarchetypeArtifactId=maven-archetype-quickstart" "-DinteractiveMode=false"
```

This command creates a directory named **exampleudf**, which contains the Maven project.

2. Once the project has been created, delete the **exampleudf/src/test** directory that was created as part of the project.
3. Open the **exampleudf/pom.xml**, and replace the existing `<dependencies>` entry with the following XML:

```
<dependencies>
  <dependency>
    <groupId>org.apache.hadoop</groupId>
    <artifactId>hadoop-client</artifactId>
    <version>2.7.3</version>
    <scope>provided</scope>
  </dependency>
  <dependency>
    <groupId>org.apache.hive</groupId>
    <artifactId>hive-exec</artifactId>
    <version>1.2.1</version>
    <scope>provided</scope>
  </dependency>
</dependencies>
```

These entries specify the version of Hadoop and Hive included with HDInsight 3.5. You can find information on the versions of Hadoop and Hive provided with HDInsight from the [HDInsight component versioning](#) document.

Add a `<build>` section before the `</project>` line at the end of the file. This section should contain the following XML:

```

<build>
  <plugins>
    <!-- build for Java 1.8. This is required by HDInsight 3.5 -->
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>3.3</version>
      <configuration>
        <source>1.8</source>
        <target>1.8</target>
      </configuration>
    </plugin>
    <!-- build an uber jar -->
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-shade-plugin</artifactId>
      <version>2.3</version>
      <configuration>
        <!-- Keep us from getting a can't overwrite file error -->
        <transformers>
          <transformer
            implementation="org.apache.maven.plugins.shade.resource.ApacheLicenseResourceTransformer">
            </transformer>
            <transformer
              implementation="org.apache.maven.plugins.shade.resource.ServicesResourceTransformer">
              </transformer>
            </transformers>
          <!-- Keep us from getting a bad signature error -->
          <filters>
            <filter>
              <artifact>*:*</artifact>
              <excludes>
                <exclude>META-INF/*.SF</exclude>
                <exclude>META-INF/*.DSA</exclude>
                <exclude>META-INF/*.RSA</exclude>
              </excludes>
            </filter>
          </filters>
        </configuration>
        <executions>
          <execution>
            <phase>package</phase>
            <goals>
              <goal>shade</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>

```

These entries define how to build the project. Specifically, the version of Java that the project uses and how to build an uberjar for deployment to the cluster.

Save the file once the changes have been made.

4. Rename **exampleudf/src/main/java/com/microsoft/examples/App.java** to **ExampleUDF.java**, and then open the file in your editor.
5. Replace the contents of the **ExampleUDF.java** file with the following, then save the file.

```

package com.microsoft.examples;

import org.apache.hadoop.hive.ql.exec.Description;
import org.apache.hadoop.hive.ql.exec.UDF;
import org.apache.hadoop.io.*;

// Description of the UDF
@Description(
    name="ExampleUDF",
    value="returns a lower case version of the input string.",
    extended="select ExampleUDF(deviceplatform) from hivesamplatable limit 10;"
)
public class ExampleUDF extends UDF {
    // Accept a string input
    public String evaluate(String input) {
        // If the value is null, return a null
        if(input == null)
            return null;
        // Lowercase the input string and return it
        return input.toLowerCase();
    }
}

```

This code implements a UDF that accepts a string value, and returns a lowercase version of the string.

Build and install the UDF

1. Use the following command to compile and package the UDF:

```
mvn compile package
```

This command builds and packages the UDF into the `exampleudf/target/ExampleUDF-1.0-SNAPSHOT.jar` file.

2. Use the `scp` command to copy the file to the HDInsight cluster.

```
scp ./target/ExampleUDF-1.0-SNAPSHOT.jar myuser@mycluster-ssh.azurehdinsight
```

Replace `myuser` with the SSH user account for your cluster. Replace `mycluster` with the cluster name. If you used a password to secure the SSH account, you are prompted to enter the password. If you used a certificate, you may need to use the `-i` parameter to specify the private key file.

3. Connect to the cluster using SSH.

```
ssh myuser@mycluster-ssh.azurehdinsight.net
```

For more information, see [Use SSH with HDInsight](#).

4. From the SSH session, copy the jar file to HDInsight storage.

```
hdfs dfs -put ExampleUDF-1.0-SNAPSHOT.jar /example/jars
```

Use the UDF from Hive

1. Use the following to start the Beeline client from the SSH session.

```
beeline -u 'jdbc:hive2://localhost:10001/;transportMode=http' -n admin
```

This command assumes that you used the default of **admin** for the login account for your cluster.

- Once you arrive at the `jdbc:hive2://localhost:10001/` prompt, enter the following to add the UDF to Hive and expose it as a function.

```
ADD JAR wasb:///example/jars/ExampleUDF-1.0-SNAPSHOT.jar;
CREATE TEMPORARY FUNCTION tolower as 'com.microsoft.examples.ExampleUDF';
```

NOTE

This example assumes that Azure Storage is default storage for the cluster. If your cluster uses Data Lake Store instead, change the `wasb://` value to `adl://`.

- Use the UDF to convert values retrieved from a table to lower case strings.

```
SELECT tolower(deviceplatform) FROM hivesampletable LIMIT 10;
```

This query selects the device platform (Android, Windows, iOS, etc.) from the table, convert the string to lower case, and then display them. The output appears similar to the following text:

```
+-----+
| _c0   |
+-----+
| android |
+-----+
```

Next steps

For other ways to work with Hive, see [Use Hive with HDInsight](#).

For more information on Hive User-Defined Functions, see [Hive Operators and User-Defined Functions](#) section of the Hive wiki at apache.org.

Use Python User Defined Functions (UDF) with Hive and Pig in HDInsight

8/16/2017 • 13 min to read • [Edit Online](#)

Learn how to use Python user-defined functions (UDF) with Apache Hive and Pig in Hadoop on Azure HDInsight.

Python on HDInsight

Python2.7 is installed by default on HDInsight 3.0 and later. Apache Hive can be used with this version of Python for stream processing. Stream processing uses STDOUT and STDIN to pass data between Hive and the UDF.

HDInsight also includes Jython, which is a Python implementation written in Java. Jython runs directly on the Java Virtual Machine and does not use streaming. Jython is the recommended Python interpreter when using Python with Pig.

WARNING

The steps in this document make the following assumptions:

- You create the Python scripts on your local development environment.
- You upload the scripts to HDInsight using either the `scp` command from a local Bash session or the provided PowerShell script.

If you want to use the [Azure Cloud Shell \(bash\)](#) preview to work with HDInsight, then you must:

- Create the scripts inside the cloud shell environment.
- Use `scp` to upload the files from the cloud shell to HDInsight.
- Use `ssh` from the cloud shell to connect to HDInsight and run the examples.

Hive UDF

Python can be used as a UDF from Hive through the HiveQL `TRANSFORM` statement. For example, the following HiveQL invokes the `hiveudf.py` file stored in the default Azure Storage account for the cluster.

Linux-based HDInsight

```
add file wasb:///hiveudf.py;

SELECT TRANSFORM (clientid, devicemake, devicemode)
  USING 'python hiveudf.py' AS
  (clientid string, phoneLabel string, phoneHash string)
FROM hivesampletable
ORDER BY clientid LIMIT 50;
```

Windows-based HDInsight

```
add file wasb:///hiveudf.py;

SELECT TRANSFORM (clientid, devicemake, devicemodel)
    USING 'D:\Python27\python.exe hiveudf.py' AS
        (clientid string, phoneLabel string, phoneHash string)
FROM hivesampletable
ORDER BY clientid LIMIT 50;
```

NOTE

On Windows-based HDInsight clusters, the `USING` clause must specify the full path to python.exe.

Here's what this example does:

1. The `add file` statement at the beginning of the file adds the `hiveudf.py` file to the distributed cache, so it's accessible by all nodes in the cluster.
2. The `SELECT TRANSFORM ... USING` statement selects data from the `hivesampletable`. It also passes the `clientid`, `devicemake`, and `devicemode` values to the `hiveudf.py` script.
3. The `AS` clause describes the fields returned from `hiveudf.py`.

Create the `hiveudf.py` file

On your development environment, create a text file named `hiveudf.py`. Use the following code as the contents of the file:

```
#!/usr/bin/env python
import sys
import string
import hashlib

while True:
    line = sys.stdin.readline()
    if not line:
        break

    line = string.strip(line, "\n ")
    clientid, devicemake, devicemode = string.split(line, "\t")
    phone_label = devicemake + ' ' + devicemode
    print "\t".join([clientid, phone_label, hashlib.md5(phone_label).hexdigest()])
```

This script performs the following actions:

1. Read a line of data from STDIN.
2. The trailing newline character is removed using `string.strip(line, "\n ")`.
3. When doing stream processing, a single line contains all the values with a tab character between each value. So `string.split(line, "\t")` can be used to split the input at each tab, returning just the fields.
4. When processing is complete, the output must be written to STDOUT as a single line, with a tab between each field. For example, `print "\t".join([clientid, phone_label, hashlib.md5(phone_label).hexdigest()])`.
5. The `while` loop repeats until no `line` is read.

The script output is a concatenation of the input values for `devicemake` and `devicemode`, and a hash of the concatenated value.

See [Running the examples](#) for how to run this example on your HDInsight cluster.

Pig UDF

A Python script can be used as a UDF from Pig through the `GENERATE` statement. You can run the script using either Jython or C Python.

- Jython runs on the JVM, and can natively be called from Pig.
- C Python is an external process, so the data from Pig on the JVM is sent out to the script running in a Python process. The output of the Python script is sent back into Pig.

To specify the Python interpreter, use `register` when referencing the Python script. The following examples register scripts with Pig as `myfuncs`:

- **To use Jython:** `register '/path/to/pigudf.py' using jython as myfuncs;`
- **To use C Python:** `register '/path/to/pigudf.py' using streaming_python as myfuncs;`

IMPORTANT

When using Jython, the path to the `pig_jython` file can be either a local path or a `WASB://` path. However, when using C Python, you must reference a file on the local file system of the node that you are using to submit the Pig job.

Once past registration, the Pig Latin for this example is the same for both:

```
LOGS = LOAD 'wasb:///example/data/sample.log' as (LINE:chararray);
LOG = FILTER LOGS by LINE is not null;
DETAILS = FOREACH LOG GENERATE myfuncs.create_structure(LINE);
DUMP DETAILS;
```

Here's what this example does:

1. The first line loads the sample data file, `sample.log` into `LOGS`. It also defines each record as a `chararray`.
2. The next line filters out any null values, storing the result of the operation into `LOG`.
3. Next, it iterates over the records in `LOG` and uses `GENERATE` to invoke the `create_structure` method contained in the Python/Jython script loaded as `myfuncs`. `LINE` is used to pass the current record to the function.
4. Finally, the outputs are dumped to STDOUT using the `DUMP` command. This command displays the results after the operation completes.

Create the pigudf.py file

On your development environment, create a text file named `pigudf.py`. Use the following code as the contents of the file:

```
# Uncomment the following if using C Python
#from pig_util import outputSchema

@outputSchema("log: {{date:chararray, time:chararray, classname:chararray, level:chararray,
detail:chararray}}")
def create_structure(input):
    if (input.startswith('java.lang.Exception')):
        input = input[21:len(input)] + ' - java.lang.Exception'
        date, time, classname, level, detail = input.split(' ', 4)
        return date, time, classname, level, detail
```

In the Pig Latin example, we defined the `LINE` input as a `chararray` because there is no consistent schema for the input. The Python script transforms the data into a consistent schema for output.

1. The `@outputSchema` statement defines the format of the data that is returned to Pig. In this case, it's a **data bag**, which is a Pig data type. The bag contains the following fields, all of which are `chararray` (strings):
 - date - the date the log entry was created

- time - the time the log entry was created
 - classname - the class name the entry was created for
 - level - the log level
 - detail - verbose details for the log entry
2. Next, the `def create_structure(input)` defines the function that Pig passes line items to.

3. The example data, `sample.log`, mostly conforms to the date, time, classname, level, and detail schema we want to return. However, it contains a few lines that begin with `*java.lang.Exception*`. These lines must be modified to match the schema. The `if` statement checks for those, then massages the input data to move the `*java.lang.Exception*` string to the end, bringing the data in-line with our expected output schema.
4. Next, the `split` command is used to split the data at the first four space characters. The output is assigned into `date`, `time`, `classname`, `level`, and `detail`.
5. Finally, the values are returned to Pig.

When the data is returned to Pig, it has a consistent schema as defined in the `@outputSchema` statement.

Upload and run the examples

IMPORTANT

The **SSH** steps only work with a Linux-based HDInsight cluster. The **PowerShell** steps work with either a Linux or Windows-based HDInsight cluster, but require a Windows client.

SSH

For more information on using SSH, see [Use SSH with HDInsight](#).

1. Use `scp` to copy the files to your HDInsight cluster. For example, the following command copies the files to a cluster named **mycluster**.

```
scp hiveudf.py pigudf.py myuser@mycluster-ssh.azurehdinsight.net:
```

2. Use SSH to connect to the cluster.

```
ssh myuser@mycluster-ssh.azurehdinsight.net
```

3. From the SSH session, add the python files uploaded previously to the WASB storage for the cluster.

```
hdfs dfs -put hiveudf.py /hiveudf.py
hdfs dfs -put pigudf.py /pigudf.py
```

After uploading the files, use the following steps to run the Hive and Pig jobs.

Use the Hive UDF

1. Use the `hive` command to start the hive shell. You should see a `hive>` prompt once the shell has loaded.
2. Enter the following query at the `hive>` prompt:

```
add file wasb:///hiveudf.py;
SELECT TRANSFORM (clientid, devicemode, devicemode)
    USING 'python hiveudf.py' AS
    (clientid string, phoneLabel string, phoneHash string)
FROM hivesampletable
ORDER BY clientid LIMIT 50;
```

- After entering the last line, the job should start. Once the job completes, it returns output similar to the following example:

```
100041 RIM 9650 d476f3687700442549a83fac4560c51c
100041 RIM 9650 d476f3687700442549a83fac4560c51c
100042 Apple iPhone 4.2.x 375ad9a0ddc4351536804f1d5d0ea9b9
100042 Apple iPhone 4.2.x 375ad9a0ddc4351536804f1d5d0ea9b9
100042 Apple iPhone 4.2.x 375ad9a0ddc4351536804f1d5d0ea9b9
```

Use the Pig UDF

- Use the `pig` command to start the shell. You see a `grunt>` prompt once the shell has loaded.
- Enter the following statements at the `grunt>` prompt:

```
Register wasb:///pigudf.py using jython as myfuncs;
LOGS = LOAD 'wasb:///example/data/sample.log' as (LINE:chararray);
LOG = FILTER LOGS by LINE is not null;
DETAILS = foreach LOG generate myfuncs.create_structure(LINE);
DUMP DETAILS;
```

- After entering the following line, the job should start. Once the job completes, it returns output similar to the following data:

```
((2012-02-03,20:11:56,SampleClass5,[TRACE],verbose detail for id 990982084))
((2012-02-03,20:11:56,SampleClass7,[TRACE],verbose detail for id 1560323914))
((2012-02-03,20:11:56,SampleClass8,[DEBUG],detail for id 2083681507))
((2012-02-03,20:11:56,SampleClass3,[TRACE],verbose detail for id 1718828806))
((2012-02-03,20:11:56,SampleClass3,[INFO],everything normal for id 530537821))
```

- Use `quit` to exit the Grunt shell, and then use the following to edit the `pigudf.py` file on the local file system:

```
nano pigudf.py
```

- Once in the editor, uncomment the following line by removing the `#` character from the beginning of the line:

```
#from pig_util import outputSchema
```

Once the change has been made, use `Ctrl+X` to exit the editor. Select `Y`, and then enter to save the changes.

- Use the `pig` command to start the shell again. Once you are at the `grunt>` prompt, use the following to run the Python script using the C Python interpreter.

```
Register 'pigudf.py' using streaming_python as myfuncs;
LOGS = LOAD 'wasb:///example/data/sample.log' as (LINE:chararray);
LOG = FILTER LOGS by LINE is not null;
DETAILS = foreach LOG generate myfuncs.create_structure(LINE);
DUMP DETAILS;
```

Once this job completes, you should see the same output as when you previously ran the script using Jython.

PowerShell: Upload the files

You can use PowerShell to upload the files to the HDInsight server. Use the following script to upload the Python files:

IMPORTANT

The steps in this section use Azure PowerShell. For more information on using Azure PowerShell, see [How to install and configure Azure PowerShell](#).

```
# Login to your Azure subscription
# Is there an active Azure subscription?
$sub = Get-AzureRmSubscription -ErrorAction SilentlyContinue
if(-not($sub))
{
    Add-AzureRmAccount
}

# Get cluster info
$clusterName = Read-Host -Prompt "Enter the HDInsight cluster name"
# Change the path to match the file location on your system
$pathToStreamingFile = "C:\path\to\hiveudf.py"
$pathToJythonFile = "C:\path\to\pigudf.py"

$clusterInfo = Get-AzureRmHDInsightCluster -ClusterName $clusterName
$resourceGroup = $clusterInfo.ResourceGroup
$storageAccountName=$clusterInfo.DefaultStorageAccount.split('.')[0]
$container=$clusterInfo.DefaultStorageContainer
$storageAccountKey=(Get-AzureRmStorageAccountKey `

    -Name $storageAccountName `

    -ResourceGroupName $resourceGroup)[0].Value

#Create a storage content and upload the file

```

IMPORTANT

Change the `C:\path\to` value to the path to the files on your development environment.

This script retrieves information for your HDInsight cluster, then extracts the account and key for the default storage account, and uploads the files to the root of the container.

NOTE

For more information on uploading files, see the [Upload data for Hadoop jobs in HDInsight](#) document.

PowerShell: Use the Hive UDF

PowerShell can also be used to remotely run Hive queries. Use the following PowerShell script to run a Hive query that uses **hiveudf.py** script:

IMPORTANT

Before running, the script prompts you for the HTTPS/Admin account information for your HDInsight cluster.

```

# Login to your Azure subscription
# Is there an active Azure subscription?
$sub = Get-AzureRmSubscription -ErrorAction SilentlyContinue
if(-not($sub))
{
    Add-AzureRmAccount
}

# Get cluster info
$clusterName = Read-Host -Prompt "Enter the HDInsight cluster name"
$creds=Get-Credential -Message "Enter the login for the cluster"

# If using a Windows-based HDInsight cluster, change the USING statement to:
# "USING 'D:\Python27\python.exe hiveudf.py' AS " +
$HiveQuery = "add file wasb:///hiveudf.py;" +
    "SELECT TRANSFORM (clientid, devicemode, devicemode) " +
    "USING 'python hiveudf.py' AS " +
    "(clientid string, phoneLabel string, phoneHash string) " +
    "FROM hivesampletable " +
    "ORDER BY clientid LIMIT 50;"

$jobDefinition = New-AzureRmHDInsightHiveJobDefinition `

-Query $HiveQuery

$job = Start-AzureRmHDInsightJob `

-ClusterName $clusterName `

-JobDefinition $jobDefinition `

-HttpCredential $creds

Write-Host "Wait for the Hive job to complete ..." -ForegroundColor Green
Wait-AzureRmHDInsightJob `

-JobId $job.JobId `

-ClusterName $clusterName `

-HttpCredential $creds

# Uncomment the following to see stderr output
# Get-AzureRmHDInsightJobOutput `

# -Clustername $clusterName `

# -JobId $job.JobId `

# -HttpCredential $creds `

# -DisplayOutputType StandardError

Write-Host "Display the standard output ..." -ForegroundColor Green
Get-AzureRmHDInsightJobOutput `

-Clustername $clusterName `

-JobId $job.JobId `

-HttpCredential $creds

```

The output for the **Hive** job should appear similar to the following example:

100041	RIM 9650	d476f3687700442549a83fac4560c51c
100041	RIM 9650	d476f3687700442549a83fac4560c51c
100042	Apple iPhone 4.2.x	375ad9a0ddc4351536804f1d5d0ea9b9
100042	Apple iPhone 4.2.x	375ad9a0ddc4351536804f1d5d0ea9b9
100042	Apple iPhone 4.2.x	375ad9a0ddc4351536804f1d5d0ea9b9

Pig (Python)

PowerShell can also be used to run Pig Latin jobs. To run a Pig Latin job that uses the **pigudf.py** script, use the following PowerShell script:

NOTE

When remotely submitting a job using PowerShell, it is not possible to use C Python as the interpreter.

```

# Login to your Azure subscription
# Is there an active Azure subscription?
$sub = Get-AzureRmSubscription -ErrorAction SilentlyContinue
if(-not($sub))
{
    Add-AzureRmAccount
}

# Get cluster info
$clusterName = Read-Host -Prompt "Enter the HDInsight cluster name"
$creds=Get-Credential -Message "Enter the login for the cluster"

$PigQuery = "Register wasb:///pigudf.py using jython as myfuncs;" +
    "LOGS = LOAD 'wasb:///example/data/sample.log' as (LINE:chararray);" +
    "LOG = FILTER LOGS by LINE is not null;" +
    "DETAILS = foreach LOG generate myfuncs.create_structure(LINE);" +
    "DUMP DETAILS;"

$jobDefinition = New-AzureRmHDInsightPigJobDefinition -Query $PigQuery

$job = Start-AzureRmHDInsightJob ` 
    -ClusterName $clusterName ` 
    -JobDefinition $jobDefinition ` 
    -HttpCredential $creds

Write-Host "Wait for the Pig job to complete ..." -ForegroundColor Green
Wait-AzureRmHDInsightJob ` 
    -Job $job.JobId ` 
    -ClusterName $clusterName ` 
    -HttpCredential $creds
# Uncomment the following to see stderr output
# Get-AzureRmHDInsightJobOutput ` 
#     -Clustername $clusterName ` 
#     -JobId $job.JobId ` 
#     -HttpCredential $creds ` 
#     -DisplayOutputType StandardError
Write-Host "Display the standard output ..." -ForegroundColor Green
Get-AzureRmHDInsightJobOutput ` 
    -Clustername $clusterName ` 
    -JobId $job.JobId ` 
    -HttpCredential $creds

```

The output for the **Pig** job should appear similar to the following data:

```

((2012-02-03,20:11:56,SampleClass5,[TRACE],verbose detail for id 990982084))
((2012-02-03,20:11:56,SampleClass7,[TRACE],verbose detail for id 1560323914))
((2012-02-03,20:11:56,SampleClass8,[DEBUG],detail for id 2083681507))
((2012-02-03,20:11:56,SampleClass3,[TRACE],verbose detail for id 1718828806))
((2012-02-03,20:11:56,SampleClass3,[INFO],everything normal for id 530537821))

```

Troubleshooting

Errors when running jobs

When running the hive job, you may encounter an error similar to the following text:

```

Caused by: org.apache.hadoop.hive.ql.metadata.HiveException: [Error 20001]: An error occurred while reading or
writing to your custom script. It may have crashed with an error.

```

This problem may be caused by the line endings in the Python file. Many Windows editors default to using CRLF as the line ending, but Linux applications usually expect LF.

You can use the following PowerShell statements to remove the CR characters before uploading the file to HDInsight:

```
$original_file = 'c:\path\to\hiveudf.py'  
$text = [IO.File]::ReadAllText($original_file) -replace "`r`n", "`n"  
[IO.File]::WriteAllText($original_file, $text)
```

PowerShell scripts

Both of the example PowerShell scripts used to run the examples contain a commented line that displays error output for the job. If you are not seeing the expected output for the job, uncomment the following line and see if the error information indicates a problem.

```
# Get-AzureRmHDInsightJobOutput `  
-Clustername $clusterName `  
-JobId $job.JobId `  
-HttpCredential $creds `  
-DisplayOutputType StandardError
```

The error information (STDERR) and the result of the job (STDOUT) are also logged to your HDInsight storage.

FOR THIS JOB...	LOOK AT THESE FILES IN THE BLOB CONTAINER
Hive	/HivePython/stderr /HivePython/stdout
Pig	/PigPython/stderr /PigPython/stdout

Next steps

If you need to load Python modules that aren't provided by default, see [How to deploy a module to Azure HDInsight](#).

For other ways to use Pig, Hive, and to learn about using MapReduce, see the following documents:

- [Use Hive with HDInsight](#)
- [Use Pig with HDInsight](#)
- [Use MapReduce with HDInsight](#)

Use C# user-defined functions with Hive and Pig streaming on Hadoop in HDInsight

8/16/2017 • 7 min to read • [Edit Online](#)

Learn how to use C# user defined functions (UDF) with Apache Hive and Pig on HDInsight.

IMPORTANT

The steps in this document work with both Linux-based and Windows-based HDInsight clusters. Linux is the only operating system used on HDInsight version 3.4 or greater. For more information, see [HDInsight component versioning](#).

Both Hive and Pig can pass data to external applications for processing. This process is known as *streaming*. When using a .NET application, the data is passed to the application on STDIN, and the application returns the results on STDOUT. To read and write from STDIN and STDOUT, you can use `Console.ReadLine()` and `Console.WriteLine()` from a console application.

Prerequisites

- A familiarity with writing and building C# code that targets .NET Framework 4.5.
 - Use whatever IDE you want. We recommend [Visual Studio](#) 2015, 2017, or [Visual Studio Code](#). The steps in this document use Visual Studio 2017.
- A way to upload .exe files to the cluster and run Pig and Hive jobs. We recommend the Data Lake Tools for Visual Studio, Azure PowerShell and Azure CLI. The steps in this document use the Data Lake Tools for Visual Studio to upload the files and run the example Hive query.

For information on other ways to run Hive queries and Pig jobs, see the following documents:

- [Use Apache Hive with HDInsight](#)
- [Use Apache Pig with HDInsight](#)
- A Hadoop on HDInsight cluster.

.NET on HDInsight

- **Linux-based HDInsight** clusters using [Mono](#) (<https://mono-project.com>) to run .NET applications. Mono version 4.2.1 is included with HDInsight version 3.5.

For more information on Mono compatibility with .NET Framework versions, see [Mono compatibility](#).

To use a specific version of Mono, see the [Install or update Mono](#) document.

- **Windows-based HDInsight** clusters use the Microsoft .NET CLR to run .NET applications.

For more information on the version of the .NET framework and Mono included with HDInsight versions, see [HDInsight component versions](#).

Create the C# projects

Hive UDF

1. Open Visual Studio and create a solution. For the project type, select **Console App (.NET Framework)**, and

name the new project **HiveCSharp**.

IMPORTANT

Select **.NET Framework 4.5** if you are using a Linux-based HDInsight cluster. For more information on Mono compatibility with .NET Framework versions, see [Mono compatibility](#).

2. Replace the contents of **Program.cs** with the following:

```
using System;
using System.Security.Cryptography;
using System.Text;
using System.Threading.Tasks;

namespace HiveCSharp
{
    class Program
    {
        static void Main(string[] args)
        {
            string line;
            // Read stdin in a loop
            while ((line = Console.ReadLine()) != null)
            {
                // Parse the string, trimming line feeds
                // and splitting fields at tabs
                line = line.TrimEnd('\n');
                string[] field = line.Split('\t');
                string phoneLabel = field[1] + ' ' + field[2];
                // Emit new data to stdout, delimited by tabs
                Console.WriteLine("{0}\t{1}\t{2}", field[0], phoneLabel, GetMD5Hash(phoneLabel));
            }
        }
        /// <summary>
        /// Returns an MD5 hash for the given string
        /// </summary>
        /// <param name="input">string value</param>
        /// <returns>an MD5 hash</returns>
        static string GetMD5Hash(string input)
        {
            // Step 1, calculate MD5 hash from input
            MD5 md5 = System.Security.Cryptography.MD5.Create();
            byte[] inputBytes = System.Text.Encoding.ASCII.GetBytes(input);
            byte[] hash = md5.ComputeHash(inputBytes);

            // Step 2, convert byte array to hex string
            StringBuilder sb = new StringBuilder();
            for (int i = 0; i < hash.Length; i++)
            {
                sb.Append(hash[i].ToString("x2"));
            }
            return sb.ToString();
        }
    }
}
```

3. Build the project.

Pig UDF

1. Open Visual Studio and create a solution. For the project type, select **Console Application**, and name the new project **PigUDF**.
2. Replace the contents of the **Program.cs** file with the following code:

```

using System;

namespace PigUDF
{
    class Program
    {
        static void Main(string[] args)
        {
            string line;
            // Read stdin in a loop
            while ((line = Console.ReadLine()) != null)
            {
                // Fix formatting on lines that begin with an exception
                if(line.StartsWith("java.lang.Exception"))
                {
                    // Trim the error info off the beginning and add a note to the end of the line
                    line = line.Remove(0, 21) + " - java.lang.Exception";
                }
                // Split the fields apart at tab characters
                string[] field = line.Split('\t');
                // Put fields back together for writing
                Console.WriteLine(String.Join("\t",field));
            }
        }
    }
}

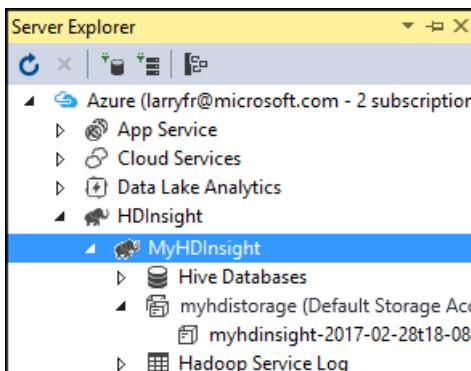
```

This application parses the lines sent from Pig, and reformat lines that begin with `java.lang.Exception`.

- Save **Program.cs**, and then build the project.

Upload to storage

- In Visual Studio, open **Server Explorer**.
- Expand **Azure**, and then expand **HDInsight**.
- If prompted, enter your Azure subscription credentials, and then click **Sign In**.
- Expand the HDInsight cluster that you wish to deploy this application to. An entry with the text **(Default Storage Account)** is listed.



- If this entry can be expanded, you are using an **Azure Storage Account** as default storage for the cluster. To view the files on the default storage for the cluster, expand the entry and then double-click the **(Default Container)**.
- If this entry cannot be expanded, you are using **Azure Data Lake Store** as the default storage for the cluster. To view the files on the default storage for the cluster, double-click the **(Default Storage Account)** entry.

5. To upload the .exe files, use one of the following methods:

- If using an **Azure Storage Account**, click the upload icon, and then browse to the **bin\debug** folder for the **HiveCSharp** project. Finally, select the **HiveCSharp.exe** file and click **Ok**.



- If using **Azure Data Lake Store**, right-click an empty area in the file listing, and then select **Upload**. Finally, select the **HiveCSharp.exe** file and click **Open**.

Once the **HiveCSharp.exe** upload has finished, repeat the upload process for the **PigUDF.exe** file.

Run a Hive query

1. In Visual Studio, open **Server Explorer**.
2. Expand **Azure**, and then expand **HDInsight**.
3. Right-click the cluster that you deployed the **HiveCSharp** application to, and then select **Write a Hive Query**.
4. Use the following text for the Hive query:

```
-- Uncomment the following if you are using Azure Storage
-- add file wasb:///HiveCSharp.exe;
-- Uncomment the following if you are using Azure Data Lake Store
-- add file adl:///HiveCSharp.exe;

SELECT TRANSFORM (clientid, devicemake, devicemode)
USING 'HiveCSharp.exe' AS
(clientid string, phoneLabel string, phoneHash string)
FROM hivesampletable
ORDER BY clientid LIMIT 50;
```

IMPORTANT

Uncomment the `add file` statement that matches the type of default storage used for your cluster.

This query selects the `clientid`, `devicemake`, and `devicemode` fields from `hivesampletable`, and passes the fields to the `HiveCSharp.exe` application. The query expects the application to return three fields, which are stored as `clientid`, `phoneLabel`, and `phoneHash`. The query also expects to find `HiveCSharp.exe` in the root of the default storage container.

5. Click **Submit** to submit the job to the HDInsight cluster. The **Hive Job Summary** window opens.
6. Click **Refresh** to refresh the summary until **Job Status** changes to **Completed**. To view the job output, click **Job Output**.

Run a Pig job

1. Use one of the following methods to connect to your HDInsight cluster:
 - If you are using a **Linux-based** HDInsight cluster, use SSH. For example,
`ssh sshuser@mycluster-ssh.azurehdinsight.net`. For more information, see [Use SSH with HDInsight](#)
 - If you are using a **Windows-based** HDInsight cluster, [Connect to the cluster using Remote Desktop](#)
2. Use one the following command to start the Pig command line:

```
pig
```

IMPORTANT

If you are using a Windows-based cluster, use the following commands instead:

```
cd %PIG_HOME%
bin\pig
```

A `grunt>` prompt is displayed.

3. Enter the following to run a Pig job that uses the .NET Framework application:

```
DEFINE streamer `PigUDF.exe` CACHE('/PigUDF.exe');
LOGS = LOAD '/example/data/sample.log' as (LINE:chararray);
LOG = FILTER LOGS by LINE is not null;
DETAILS = STREAM LOG through streamer as (col1, col2, col3, col4, col5);
DUMP DETAILS;
```

The `DEFINE` statement creates an alias of `streamer` for the pigudf.exe applications, and `CACHE` loads it from default storage for the cluster. Later, `streamer` is used with the `STREAM` operator to process the single lines contained in `LOG` and return the data as a series of columns.

NOTE

The application name that is used for streaming must be surrounded by the ` (backtick) character when aliased, and ' (single quote) when used with `SHIP`.

4. After entering the last line, the job should start. It returns output similar to the following text:

```
(2012-02-03 20:11:56 SampleClass5 [WARN] problem finding id 1358451042 - java.lang.Exception)
(2012-02-03 20:11:56 SampleClass5 [DEBUG] detail for id 1976092771)
(2012-02-03 20:11:56 SampleClass5 [TRACE] verbose detail for id 1317358561)
(2012-02-03 20:11:56 SampleClass5 [TRACE] verbose detail for id 1737534798)
(2012-02-03 20:11:56 SampleClass7 [DEBUG] detail for id 1475865947)
```

Next steps

In this document, you have learned how to use a .NET Framework application from Hive and Pig on HDInsight. If you would like to learn how to use Python with Hive and Pig, see [Use Python with Hive and Pig in HDInsight](#).

For other ways to use Pig and Hive, and to learn about using MapReduce, see the following documents:

- [Use Hive with HDInsight](#)
- [Use Pig with HDInsight](#)
- [Use MapReduce with HDInsight](#)

Process and analyze JSON documents using Hive in HDInsight

8/16/2017 • 6 min to read • [Edit Online](#)

Learn how to process and analyze JSON files using Hive in HDInsight. The following JSON document is used in the tutorial:

```
{  
    "StudentId": "trgfg-5454-fdfdg-4346",  
    "Grade": 7,  
    "StudentDetails": [  
        {  
            "FirstName": "Peggy",  
            "LastName": "Williams",  
            "YearJoined": 2012  
        }  
    ],  
    "StudentClassCollection": [  
        {  
            "ClassId": "89084343",  
            "ClassParticipation": "Satisfied",  
            "ClassParticipationRank": "High",  
            "Score": 93,  
            "PerformedActivity": false  
        },  
        {  
            "ClassId": "78547522",  
            "ClassParticipation": "NotSatisfied",  
            "ClassParticipationRank": "None",  
            "Score": 74,  
            "PerformedActivity": false  
        },  
        {  
            "ClassId": "78675563",  
            "ClassParticipation": "Satisfied",  
            "ClassParticipationRank": "Low",  
            "Score": 83,  
            "PerformedActivity": true  
        }  
    ]  
}
```

The file can be found at `wab://processjson@hditutorialdata.blob.core.windows.net/`. For more information on using Azure Blob storage with HDInsight, see [Use HDFS-compatible Azure Blob storage with Hadoop in HDInsight](#). You can copy the file to the default container of your cluster.

In this tutorial, you use the Hive console. For instructions of opening the Hive console, see [Use Hive with Hadoop on HDInsight with Remote Desktop](#).

Flatten JSON documents

The methods listed in the next section require the JSON document in a single row. So you must flatten the JSON document to a string. If your JSON document is already flattened, you can skip this step and go straight to the next section on Analyzing JSON data.

```

DROP TABLE IF EXISTS StudentsRaw;
CREATE EXTERNAL TABLE StudentsRaw (textcol string) STORED AS TEXTFILE LOCATION
"wasb://processjson@hditutorialdata.blob.core.windows.net/";

DROP TABLE IF EXISTS StudentsOneLine;
CREATE EXTERNAL TABLE StudentsOneLine
(
    json_body string
)
STORED AS TEXTFILE LOCATION '/json/students';

INSERT OVERWRITE TABLE StudentsOneLine
SELECT CONCAT_WS(' ',COLLECT_LIST(textcol)) AS singlelineJSON
    FROM (SELECT INPUT__FILE__NAME,BLOCK__OFFSET__INSIDE__FILE, textcol FROM StudentsRaw DISTRIBUTE BY
INPUT__FILE__NAME SORT BY BLOCK__OFFSET__INSIDE__FILE) x
    GROUP BY INPUT__FILE__NAME;

SELECT * FROM StudentsOneLine

```

The raw JSON file is located at **wasb://processjson@hditutorialdata.blob.core.windows.net/**. The *StudentsRaw* Hive table points to the raw unflattened JSON document.

The *StudentsOneLine* Hive table stores the data in the HDInsight default file system under the */json/students/* path.

The INSERT statement populates the *StudentOneLine* table with the flattened JSON data.

The SELECT statement shall only return one row.

Here is the output of the SELECT statement:

The screenshot shows a terminal window titled "Hadoop Command Line - hive". The command entered is "hive> SELECT * FROM StudentsOneLine;". The output displays a single row of flattened JSON data. The data includes fields like StudentId, FirstName, LastName, YearJoined, ClassId, ClassParticipation, Score, and PerformedActivity. The output is as follows:

```

hive> SELECT * FROM StudentsOneLine;
OK
{"StudentId": "trgfg-5454-fdfdg-4346", "Grade": 7, "StudentDetails": {"FirstName": "Peggy", "LastName": "Willian"}, "YearJoined": 2012, "StudentClassCollection": [{"ClassId": "89084343", "ClassParticipation": "Satisfied", "ClassParticipationRank": "High", "Score": 93, "PerformedActivity": false}, {"ClassId": "78547522", "ClassParticipation": "NotSatisfied", "ClassParticipationRank": "None", "Score": 74, "PerformedActivity": false}, {"ClassId": "78675563", "ClassParticipation": "Satisfied", "ClassParticipationRank": "Low", "Score": 83, "PerformedActivity": true}], "Time taken: 0.203 seconds, Fetched: 1 row(s)}
hive>

```

Analyze JSON documents in Hive

Hive provides three different mechanisms to run queries on JSON documents:

- use the GET_JSON_OBJECT UDF (User-defined function)
- use the JSON_TUPLE UDF
- use custom SerDe
- write your own UDF using Python or other languages. See [this article](#) on running your own Python code with Hive.

Use the GET_JSON_OBJECT UDF

Hive provides a built-in UDF called [get json object](#), which can perform JSON querying during run time. This method takes two arguments – the table name and method name, which has the flattened JSON document and the JSON field that needs to be parsed. Let's look at an example to see how this UDF works.

Get the first name and last name for each student

```

SELECT
  GET_JSON_OBJECT(StudentsOneLine.json_body,'$.StudentDetails.FirstName'),
  GET_JSON_OBJECT(StudentsOneLine.json_body,'$.StudentDetails.LastName')
FROM StudentsOneLine;

```

Here is the output when running this query in console window.

```

hive> SELECT
>   GET_JSON_OBJECT(StudentsOneLine.json_body,'$.StudentDetails.FirstName'),
>   GET_JSON_OBJECT(StudentsOneLine.json_body,'$.StudentDetails.LastName')
> FROM StudentsOneLine;
Query ID = rupuser_20150622162727_4ffaf570e5-4003-93ea-zbca51671766
Total jobs = 1
Launching Job 1 out of 1
Number of reduce tasks is set to 0 since there's no reduce operator
Starting Job = job_1434391043751_0006, Tracking URL = http://headnodehost:9014/p
roxy/application_1434391043751_0006/
Kill Command = C:\apps\dist\hadoop-2.4.0.2.1.12.1-0003\bin\hadoop.cmd job -kill
job_1434391043751_0006
Hadoop job information for Stage-1: number of mappers: 1; number of reducers: 0
2015-06-22 16:29:24,075 Stage-1 map = 0%, reduce = 0%
2015-06-22 16:29:36,362 Stage-1 map = 100%, reduce = 0%, Cumulative CPU 3.984 s
ec
MapReduce Total cumulative CPU time: 3 seconds 984 msec
Ended Job = job_1434391043751_0006
MapReduce Jobs Launched:
Job 0: Map: 1 Cumulative CPU: 3.984 sec HDFS Read: 0 HDFS Write: 23 SUCCESS
Total MapReduce CPU Time Spent: 3 seconds 984 msec
OK
["Peggy"]      ["Williams"]
Time taken: 26.667 seconds, Fetched: 1 row(s)
hive> -

```

There are a few limitations of the get_json_object UDF.

- Because each field in the query requires reparsing the query, it affects the performance.
- GET_JSON_OBJECT() returns the string representation of an array. To convert this array to a Hive array, you have to use regular expressions to replace the square brackets '[' and ']' and then also call split to get the array.

This is why the Hive wiki recommends using json_tuple.

Use the JSON_TUPLE UDF

Another UDF provided by Hive is called [json_tuple](#), which performs better than [get_json_object](#). This method takes a set of keys and a JSON string, and returns a tuple of values using one function. The following query returns the student id and the grade from the JSON document:

```

SELECT q1.StudentId, q1.Grade
  FROM StudentsOneLine jt
 LATERAL VIEW JSON_TUPLE(jt.json_body, 'StudentId', 'Grade') q1
    AS StudentId, Grade;

```

The output of this script in the Hive console:

```

> select q1.StudentId, q1.Grade
> from one_line_json jt
> LATERAL VIEW json_tuple(jt.json_body, 'StudentId', 'Grade') q1
> as StudentId, Grade;
Query ID = rashimg_20150401062525_bfa41181-956d-4843-8881-4bb962362d88
Total jobs = 1
Launching Job 1 out of 1

Status: Running (application id: application_1427392057137_0211)

Map 1: -/
Map 1: 0/1
Map 1: 0/1
Map 1: 0/1
Map 1: 0/1
Map 1: 1/1
Status: Finished successfully
OK
trgfg-5454-fdfdg-4346 7
Time taken: 30.172 seconds, Fetched: 1 row(s)
hive> -

```

JSON_TUPLE uses the [lateral view](#) syntax in Hive, which allows json_tuple to create a virtual table by applying the UDT function to each row of the original table. Complex JSONs become too unwieldy because of the repeated use of LATERAL VIEW. Furthermore, JSON_TUPLE cannot handle nested JSONs.

Use custom SerDe

SerDe is the best choice for parsing nested JSON documents, it allows you to define the JSON schema, and use the schema to parse the documents. In this tutorial, you use one of the more popular SerDe that has been developed by [Roberto Congiu](#).

To use the custom SerDe:

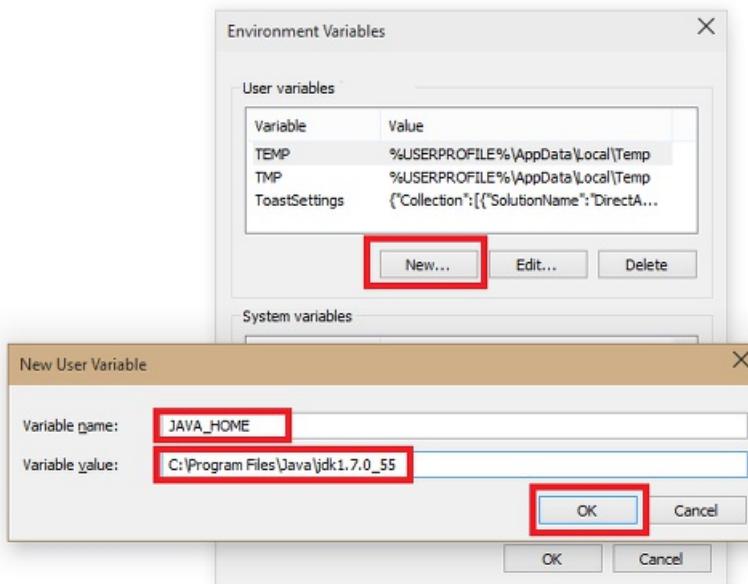
1. Install [Java SE Development Kit 7u55 JDK 1.7.0_55](#). Choose the Windows X64 version of the JDK if you are going to be using the Windows deployment of HDInsight

WARNING

JDK 1.8 doesn't work with this SerDe.

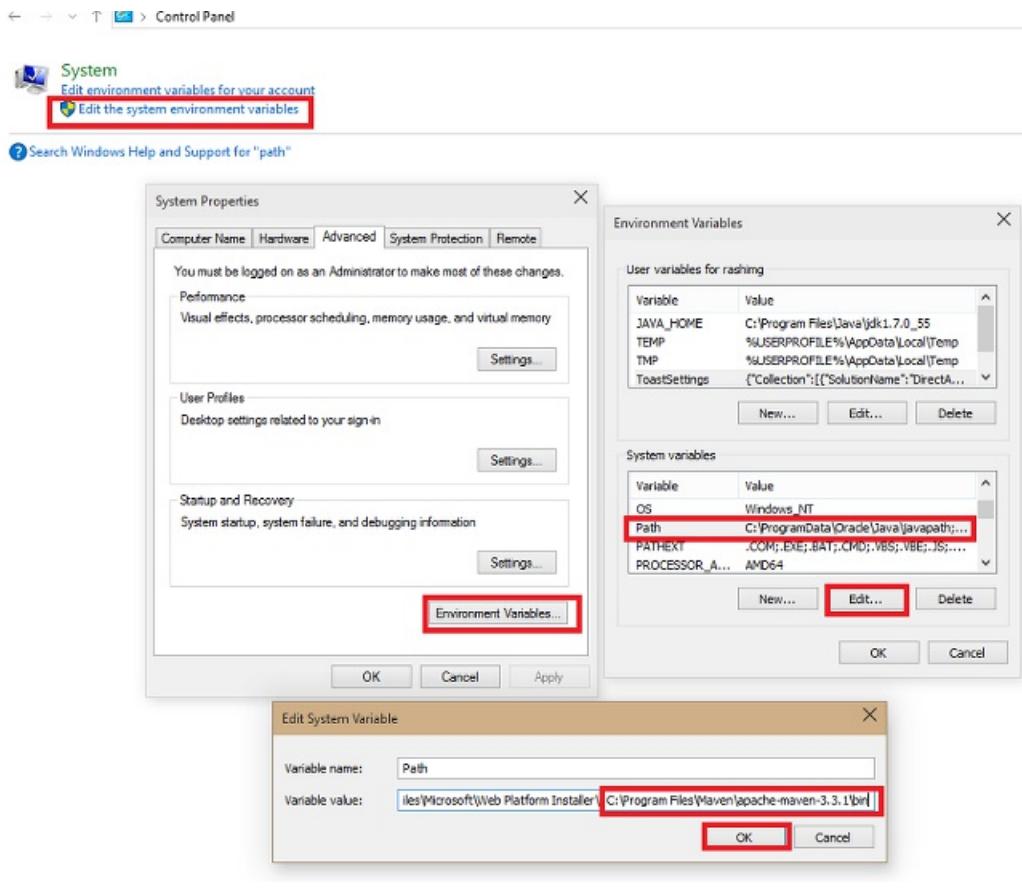
After the installation is completed, add a new user environment variable:

- a. Open **View advanced system settings** from the Windows screen.
- b. Click **Environment Variables**.
- c. Add a new **JAVA_HOME** environment variable is pointing to **C:\Program Files\Java\jdk1.7.0_55** or wherever your JDK is installed.



2. Install [Maven 3.3.1](#)

Add the bin folder to your path by going to Control Panel-->Edit the System Variables for your account Environment variables. The following screenshot shows you how to do this.



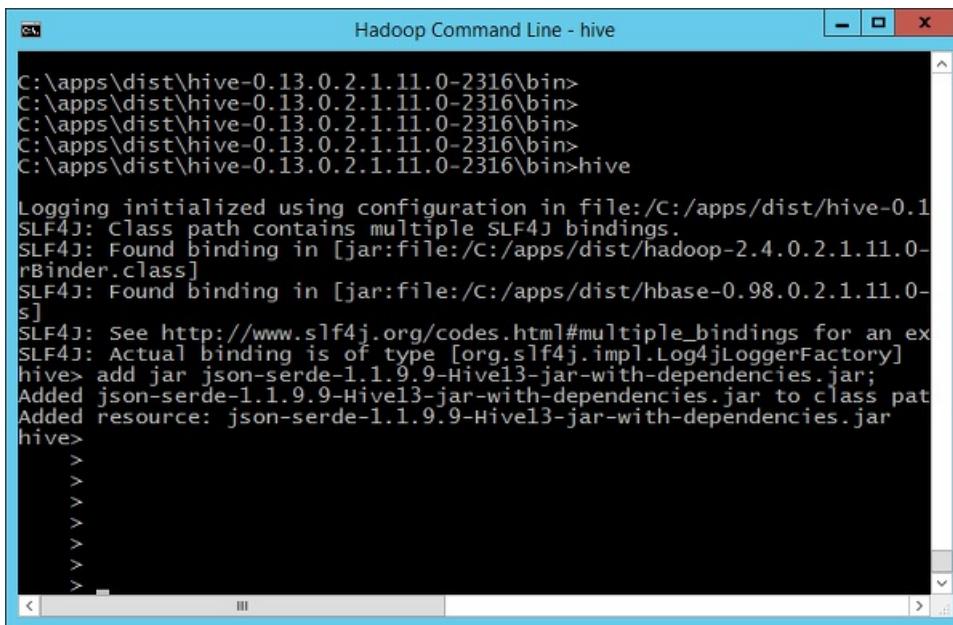
3. Clone the project from [Hive-JSON-SerDe](#) github site. You can do this by clicking on the "Download Zip" button as shown in the following screenshot.

4: Go to the folder where you have downloaded this package and then type "mvn package". This should create the necessary jar files that you can then copy over to the cluster.

5: Go to the target folder under the root folder where you downloaded the package. Upload the json-serde-1.1.9.9-Hive13-jar-with-dependencies.jar file to head-node of your cluster. I usually put it under the hive binary folder: C:\apps\dist\hive-0.13.0.2.1.11.0-2316\bin or something similar.

6: In the hive prompt, type "add jar /path/to/json-serde-1.1.9.9-Hive13-jar-with-dependencies.jar". Since in my case, the jar is in the C:\apps\dist\hive-0.13.x\bin folder, I can directly add the jar with the name as shown:

```
add jar json-serde-1.1.9.9-Hive13-jar-with-dependencies.jar;
```



The screenshot shows a terminal window titled "Hadoop Command Line - hive". The command entered was "add jar json-serde-1.1.9.9-Hive13-jar-with-dependencies.jar;". The output shows the path to the jar file being added and some SLF4J initialization logs.

```
C:\apps\dist\hive-0.13.0.2.1.11.0-2316\bin>
C:\apps\dist\hive-0.13.0.2.1.11.0-2316\bin>
C:\apps\dist\hive-0.13.0.2.1.11.0-2316\bin>
C:\apps\dist\hive-0.13.0.2.1.11.0-2316\bin>
C:\apps\dist\hive-0.13.0.2.1.11.0-2316\bin>hive
Logging initialized using configuration in file:/C:/apps/dist/hive-0.1
SLF4J: Class path contains multiple SLF4J bindings.
SLF4J: Found binding in [jar:file:/C:/apps/dist/hadoop-2.4.0.2.1.11.0-
rBinder.class]
SLF4J: Found binding in [jar:file:/C:/apps/dist/hbase-0.98.0.2.1.11.0-
s]
SLF4J: See http://www.slf4j.org/codes.html#multiple_bindings for an ex
SLF4J: Actual binding is of type [org.slf4j.impl.Log4jLoggerFactory]
hive> add jar json-serde-1.1.9.9-Hive13-jar-with-dependencies.jar;
Added json-serde-1.1.9.9-Hive13-jar-with-dependencies.jar to class pat
Added resource: json-serde-1.1.9.9-Hive13-jar-with-dependencies.jar
hive>
>
>
>
>
>
>
```

Now, you are ready to use the SerDe to run queries against the JSON document.

The following statement creates a table with a defined schema:

```
DROP TABLE json_table;
CREATE EXTERNAL TABLE json_table (
    StudentId string,
    Grade int,
    StudentDetails array<struct<
        FirstName:string,
        LastName:string,
        YearJoined:int
    >
),
    StudentClassCollection array<struct<
        ClassId:string,
        ClassParticipation:string,
        ClassParticipationRank:string,
        Score:int,
        PerformedActivity:boolean
    >
)
) ROW FORMAT SERDE 'org.openx.data.jsonserde.JsonSerDe'
LOCATION '/json/students';
```

To list the first name and last name of the student

```
SELECT StudentDetails.FirstName, StudentDetails.LastName FROM json_table;
```

Here is the result from the Hive console.

```
Hadoop Command Line - hive
>
> select StudentDetails.FirstName, StudentDetails.LastName from js
Query ID = rashimg_20150401071919_c8f259ad-76c6-4a1b-b0d7-e0f7d2bb27d1
Total jobs = 1
Launching Job 1 out of 1
Tez session was closed. Reopening...
Session re-established.

Status: Running (application id: application_1427392057137_0213)

Map 1: -/
Map 1: 0/1
Map 1: 0/1
Map 1: 0/1
Map 1: 0/1
Map 1: 1/1
Status: Finished successfully
OK
["Peggy"]      ["williams"]
Time taken: 40.12 seconds, Fetched: 1 row(s)
hive>
>
>
>
>
```

To calculate the sum of scores of the JSON document

```
SELECT SUM(scores)
FROM json_table jt
    lateral view explode(jt.StudentClassCollection.Score) collection as scores;
```

The preceding query uses [lateral view explode](#) UDF to expand the array of scores so that they can be summed.

Here is the output from the Hive console.

```
Hadoop Command Line - hive
hive> select
>     sum(scores)
>   from json_table jt
>   lateral view explode(jt.StudentClassCollection.Score) collection as scores
Query ID = rashimg_20150401071111_d25040ee-3f85-45e8-884d-8138ae94578c
Total jobs = 1
Launching Job 1 out of 1
Tez session was closed. Reopening...
Session re-established.

Status: Running (application id: application_1427392057137_0212)

Map 1: -/
Map 1: 0/1      Reducer 2: 0/1
Map 1: 1/1      Reducer 2: 0/1
Map 1: 1/1      Reducer 2: 1/1
Status: Finished successfully
OK
250
Time taken: 43.706 seconds, Fetched: 1 row(s)
hive>
>
```

To find which subjects a given student has scored more than 80 points:

```
SELECT
  jt.StudentClassCollection.ClassId
FROM json_table jt
  lateral view explode(jt.StudentClassCollection.Score) collection as score where score > 80;
```

The preceding query returns a Hive array unlike `get_json_object`, which returns a string.

The screenshot shows a Windows command-line interface window titled "Hadoop Command Line - hive". The command entered is:

```
hive> select
>   jt.StudentClassCollection.ClassId
>   from json_table jt
>   lateral view explode(jt.StudentClassCollection.Score) collect
Query ID = rashimg_20150401072626_2d590447-6a29-4cf0-89d9-38848dfe46ac
Total jobs = 1
Launching Job 1 out of 1
Tez session was closed. Reopening...
Session re-established.

Status: Running (application id: application_1427392057137_0214)

Map 1: -/
Map 1: 0/1
Map 1: 0/1
Map 1: 0/1
Map 1: 0/1
Map 1: 1/1
Status: Finished successfully
OK
[{"89084343","78547522","78675563"}]
[{"89084343","78547522","78675563"}]
Time taken: 22.392 seconds, Fetched: 2 row(s)
hive>
```

If you want to skip malformed JSON, then as explained in the [wiki page](#) of this SerDe you can achieve that by typing the following code:

```
ALTER TABLE json_table SET SERDEPROPERTIES ( "ignore.malformed.json" = "true");
```

Summary

In conclusion, the type of JSON operator in Hive that you choose depends on your scenario. If you have a simple JSON document and you only have one field to look up on – you can choose to use the Hive UDF get_json_object. If you have more than one key to look up on, then you can use json_tuple. If you have a nested document, then you should use the JSON SerDe.

Next steps

For other related articles, see

- [Use Hive and HiveQL with Hadoop in HDInsight to analyze a sample Apache log4j file](#)
- [Analyze flight delay data by using Hive in HDInsight](#)
- [Analyze Twitter data using Hive in HDInsight](#)
- [Run a Hadoop job using Azure Cosmos DB and HDInsight](#)

Connect Excel to Hadoop in Azure HDInsight with the Microsoft Hive ODBC driver

8/16/2017 • 4 min to read • [Edit Online](#)

Microsoft's Big Data solution integrates Microsoft Business Intelligence (BI) components with Apache Hadoop clusters that have been deployed by the Azure HDInsight. An example of this integration is the ability to connect Excel to the Hive data warehouse of a Hadoop cluster in HDInsight using the Microsoft Hive Open Database Connectivity (ODBC) Driver.

It is also possible to connect the data associated with an HDInsight cluster and other data sources, including other (non-HDInsight) Hadoop clusters, from Excel using the Microsoft Power Query add-in for Excel. For information on installing and using Power Query, see [Connect Excel to HDInsight with Power Query](#).

NOTE

While the steps in this article can be used with either a Linux or Windows-based HDInsight cluster, Windows is required for the client workstation.

Prerequisites:

Before you begin this article, you must have the following items:

- **An HDInsight cluster.** To create one, see [Get started with Azure HDInsight](#).
- **A workstation** with Office 2013 Professional Plus, Office 365 Pro Plus, Excel 2013 Standalone, or Office 2010 Professional Plus.

Install Microsoft Hive ODBC driver

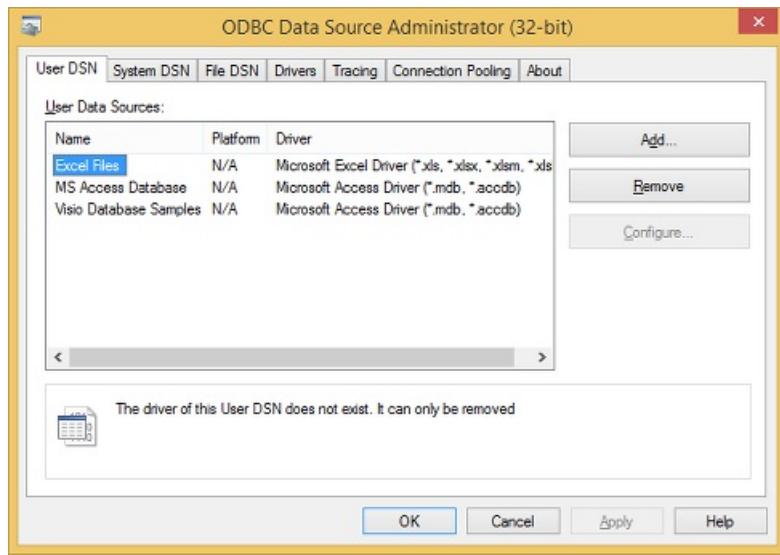
Download and install Microsoft Hive ODBC Driver from the [Download Center](#).

This driver can be installed on 32-bit or 64-bit versions of Windows 7, Windows 8, Windows 10, Windows Server 2008 R2, and Windows Server 2012. The driver allows connection to Azure HDInsight (version 1.6 and later) and Azure HDInsight Emulator (v.1.0.0.0 and later). You shall install the version that matches the version of the application where you use the ODBC driver. For this tutorial, the driver is used from Office Excel.

Create Hive ODBC data source

The following steps show you how to create a Hive ODBC Data Source.

1. From Windows 8 or Windows 10, press the Windows key to open the Start screen, and then type **data sources**.
2. Click **Set up ODBC Data sources (32-bit)** or **Set up ODBC Data Sources (64-bit)** depending on your Office version. If you are using Windows 7, choose **ODBC Data Sources (32 bit)** or **ODBC Data Sources (64 bit)** from **Administrative Tools**. You shall see the **ODBC Data Source Administrator** dialog.



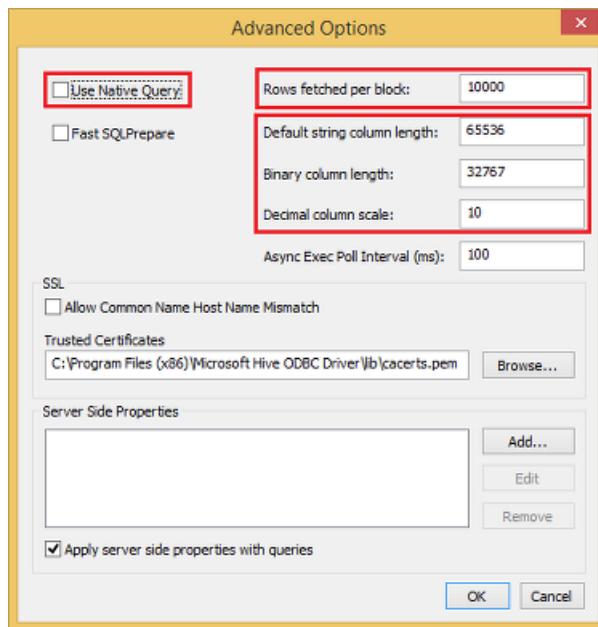
3. From User DNS, click **Add** to open the **Create New Data Source** wizard.
4. Select **Microsoft Hive ODBC Driver**, and then click **Finish**. You shall see the **Microsoft Hive ODBC Driver DNS Setup** dialog.
5. Type or select the following values:

PROPERTY	DESCRIPTION
Data Source Name	Give a name to your data source
Host	Enter <HDInsightClusterName>.azurehdinsight.net. For example, myHDIcluster.azurehdinsight.net
Port	Use 443 . (This port has been changed from 563 to 443.)
Database	Use Default .
Hive Server Type	Select Hive Server 2
Mechanism	Select Azure HDInsight Service
HTTP Path	Leave it blank.
User Name	Enter HDInsight cluster HTTP user username. The default username is admin .
Password	Enter HDInsight cluster user password.

There are some important parameters to be aware of when you click **Advanced Options**:

PARAMETER	DESCRIPTION
Use Native Query	When it is selected, the ODBC driver will NOT try to convert TSQL into HiveQL. You shall use it only if you are 100% sure you are submitting pure HiveQL statements. When connecting to SQL Server or Azure SQL Database, you should leave it unchecked.

PARAMETER	DESCRIPTION
Rows fetched per block	When fetching a large number of records, tuning this parameter may be required to ensure optimal performances.
Default string column length, Binary column length, Decimal column scale	The data type lengths and precisions may affect how data is returned. They cause incorrect information to be returned due to loss of precision and/or truncation.

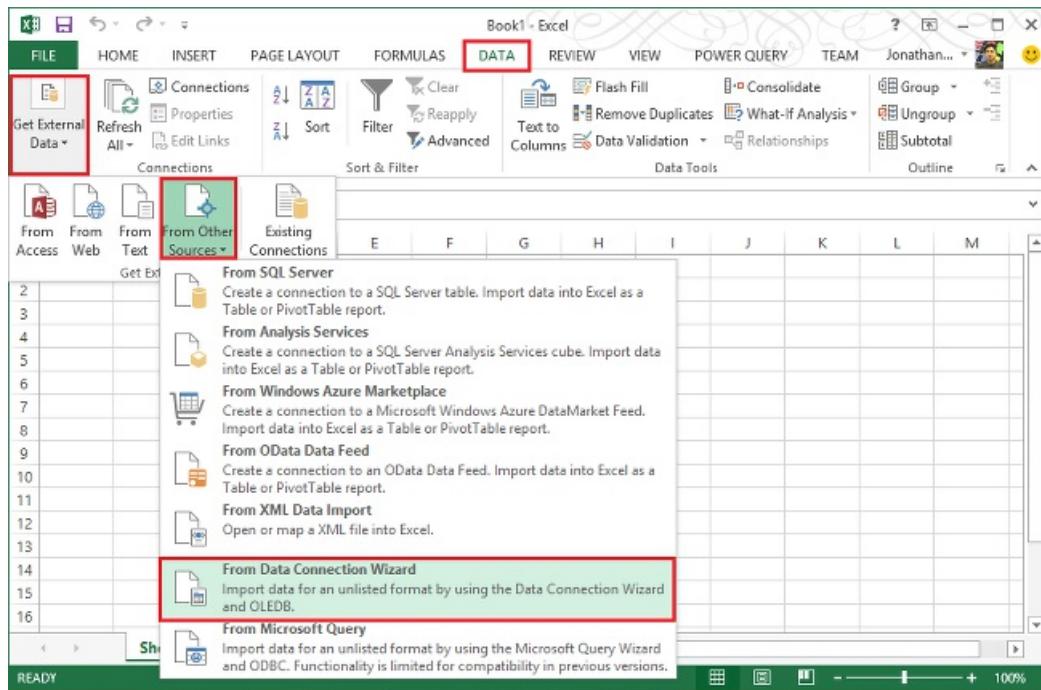


6. Click **Test** to test the data source. When the data source is configured correctly, it shows *TESTS COMPLETED SUCCESSFULLY!*.
7. Click **OK** to close the Test dialog. The new data source should now be listed on the **ODBC Data Source Administrator**.
8. Click **OK** to exit the wizard.

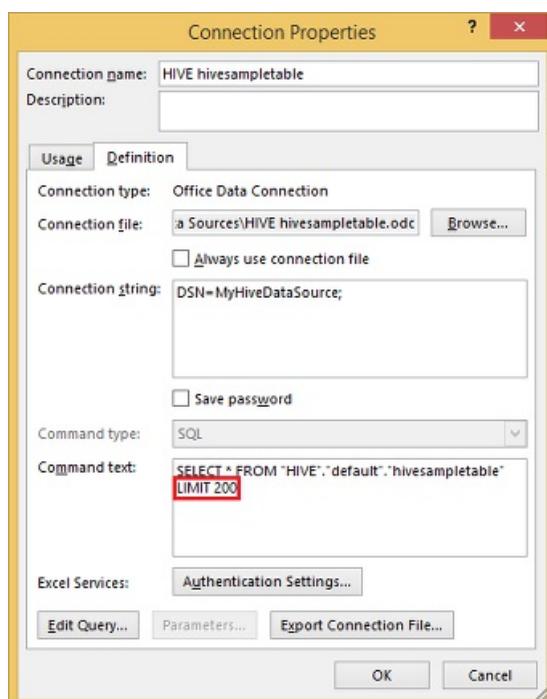
Import data into Excel from HDInsight

The following steps describe the way to import data from a hive table into an Excel workbook using the ODBC data source that you created in the steps above.

1. Open a new or existing workbook in Excel.
2. From the **Data** tab, click **From Other Data Sources**, and then click **From Data Connection Wizard** to launch the **Data Connection Wizard**.



3. Select **ODBC DSN** as the data source, and then click **Next**.
4. From ODBC data sources, select the data source name that you created in the previous step, and then click **Next**.
5. Re-enter the password for the cluster in the wizard, and then click **Test** to verify the configuration once again.
6. Click **OK** to close the test dialog.
7. Click **OK**. Wait for the **Select Database and Table** dialog to open. This step can take a few seconds.
8. Select the table that you want to import, and then click **Next**. The *hivesampletable* is a sample hive table that comes with HDInsight clusters. You can choose it if you haven't created one. For more information on run Hive queries and create Hive tables, see [Use Hive with HDInsight](#).
9. Click **Finish**.
10. In the **Import Data** dialog, you can change or specify the query. To do so, click **Properties**. This step can take a few seconds.
11. Click the **Definition** tab, and then append **LIMIT 200** to the Hive select statement in the **Command text** textbox. The modification limits the returned record set to 200.



12. Click **OK** to close the Connection Properties dialog.
13. Click **OK** to close the **Import Data** dialog.
14. Re-enter the password, and then click **OK**. It takes a few seconds before data gets imported to Excel.

Next steps

In this article, you learned how to use the Microsoft Hive ODBC driver to retrieve data from the HDInsight Service into Excel. Similarly, you can retrieve data from the HDInsight Service into SQL Database. It is also possible to upload data into an HDInsight Service. To learn more, see:

- [Analyze flight delay data using HDInsight](#)
- [Upload Data to HDInsight](#)
- [Use Sqoop with HDInsight](#)

Using Apache Hive and the Ambari Hive view to Analyze Sensor Data

8/16/2017 • 5 min to read • [Edit Online](#)

In this article you will learn how to analyze sensor data by using the Hive Query Console with HDInsight (Hadoop), then visualize the data in Microsoft Excel by using Power View.

In this example, you'll use Hive to process historical data produced by heating, ventilation, and air conditioning (HVAC) systems to identify systems that are not able to reliably maintain a set temperature. You will learn how to:

- Create HIVE tables to query data stored in comma separated value (CSV) files.
- Create HIVE queries to analyze the data.
- Use Microsoft Excel to connect to HDInsight (using open database connectivity (ODBC) to retrieve the analyzed data.
- Use Power View to visualize the data.

Prerequisites

- An HDInsight (Hadoop) cluster: See [Provision Hadoop clusters in HDInsight](#) for information about creating a cluster.
- Microsoft Excel 2016

[AZURE.NOTE] Microsoft Excel is used for data visualization with [Power View](#).

- [Microsoft Hive ODBC Driver](#)

To run the sample

1. From your web browser, navigate to the following URL. Replace <clustername> with the name of your HDInsight cluster.

`https://<clustername>.azurehdinsight.net`

When prompted, authenticate by using the administrator user name and password you used when provisioning this cluster. On the right, click on "Hive". Look for Hive View 2.0 towards the bottom of the center page, then click on "Go to View." Stay on this page as we begin the tutorial.

The screenshot shows the HDInsight cluster summary page. On the left, a sidebar lists services: HDFS, YARN, MapReduce2, Tez, Hive (selected), Pig, Sqoop, Oozie, ZooKeeper, Ambari Metrics, and Slider. Below the sidebar is an 'Actions' button. The main pane has tabs for 'Summary' and 'Configs', with 'Summary' selected. The summary section displays various service status cards:

- Hive Metastore: Started, No alerts
- Hive Metastore: Started, No alerts
- HiveServer2: Started, No alerts
- HiveServer2: Started, No alerts
- WebHCat Server: Started, No alerts
- WebHCat Server: Started, No alerts
- HCat Client: 1 HCat Client Installed
- Hive Clients: 6 Hive Clients Installed
- HiveServer2 JDBC URL: jdbc:hive2://zk0-hdihiw.wyy2c
- Hive View 2.0: Go To View
- Debug Hive Query: Go To View

1. Install the Hive ODBC Driver on Windows.

2. Install [Excel 2016 on Windows](#).

Introduction

Many personal and commercial devices now contain sensors, which collect information from the physical world. For example, most phones have a GPS, fitness devices track how many steps you've taken, and thermostats can monitor the temperature of a building.

In this tutorial, you'll learn how HDInsight can be used to process historical data produced by heating, ventilation, and air conditioning (HVAC) systems to identify systems that are not able to reliably maintain a set temperature. You will learn how to:

- Refine and enrich temperature data from buildings in several countries
- Analyze the data to determine which buildings have problems maintaining comfortable temperatures (actual recorded temperature vs. temperature the thermostat was set to)
- Infer reliability of HVAC systems used in the buildings
- Visualize the data in Microsoft Excel

Load Sensors Data Into Azure Storage Blobs

Let's load the sensor data into your default storage account.

1. Login to the Azure dashboard and click on your HDInsight cluster.

2. On the right pane, click on **Storage Accounts**.

PROPERTIES



Properties



Storage accounts



Data Lake Store access

SUPPORT + TROUBLESHOOTING



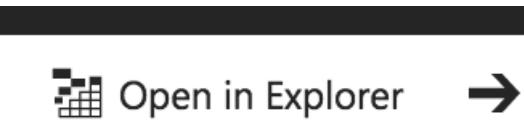
Resource health



New support request

1. Click on your storage account.

2. Click on "Open in Explorer"



1. If you haven't already installed Azure Storage Explorer, this will prompt you to.

2. Connect your Azure Account to Azure Storage Explorer. Look at [Getting Started with Azure Storage Explorer](#) for more information.

3. Expand your Storage Account.

4. Expand **Blob Containers**.

5. Create a new blob container called "sensordata".

6. Under the "sensordata" blob container, create a new folder called "hvac".

7. Upload the `hvac.csv` file to the new hvac folder.

Creating Hive Tables to Query the Sensor Data in the Azure Storage blobs

The following Hive statement creates an external table that allows Hive to query data stored in Azure Blob Storage. External tables preserve the data in the original file format while allowing Hive to perform queries against the data within the file. In this case, the data is stored in the file as comma separated values (CSV).

The Hive statements below create a new table, named hvac, by describing the fields within the files, the delimiter (comma) between fields, and the location of the file in Azure Blob Storage. This will allow you to create Hive queries over your data. Remember to replace the path with your individual HDInsight cluster name.

1. In your Ambari console that you logged into earlier, you should be on the Hive screen.

2. Copy and paste the query below and put it in the white textbox in the middle of the screen.

```
DROP TABLE IF EXISTS hvac;

--create the hvac table on comma-separated sensor data
CREATE EXTERNAL TABLE hvac(`date` STRING, time STRING, targettemp BIGINT,
    actualtemp BIGINT,
    system BIGINT,
    systemage BIGINT,
    buildingid BIGINT)
ROW FORMAT DELIMITED FIELDS TERMINATED BY ','
STORED AS TEXTFILE LOCATION 'wasbs://sensordata@hivesensordata.blob.core.windows.net/hvac/';
```

1. Click the green "Execute" button.
2. You can always execute a "SELECT * FROM hvac;" to see if you successfully loaded data.

Creating Hive Queries over Sensor Data

The following Hive query creates a new table selecting temperatures from the HVAC data, looking for temperature variations (see the query below). Specifically, the difference between the target temperature the thermostat was set to and the recorded temperature. If the difference is greater than 5, the temp_diff column will be set to 'HOT', or 'COLD' and extremetemp will be set to 1; otherwise, temp_diff will be set to 'NORMAL' and extremetemp will be set to 0.

The query will write the results into a new tables: hvac_temperatures (see the CREATE TABLE statements below).

1. In your Ambari console that you logged into earlier, you should be on the Hive screen.
2. Copy and paste the query below and put it in the white textbox in the middle of the screen.

```
DROP TABLE IF EXISTS hvac_temperatures;

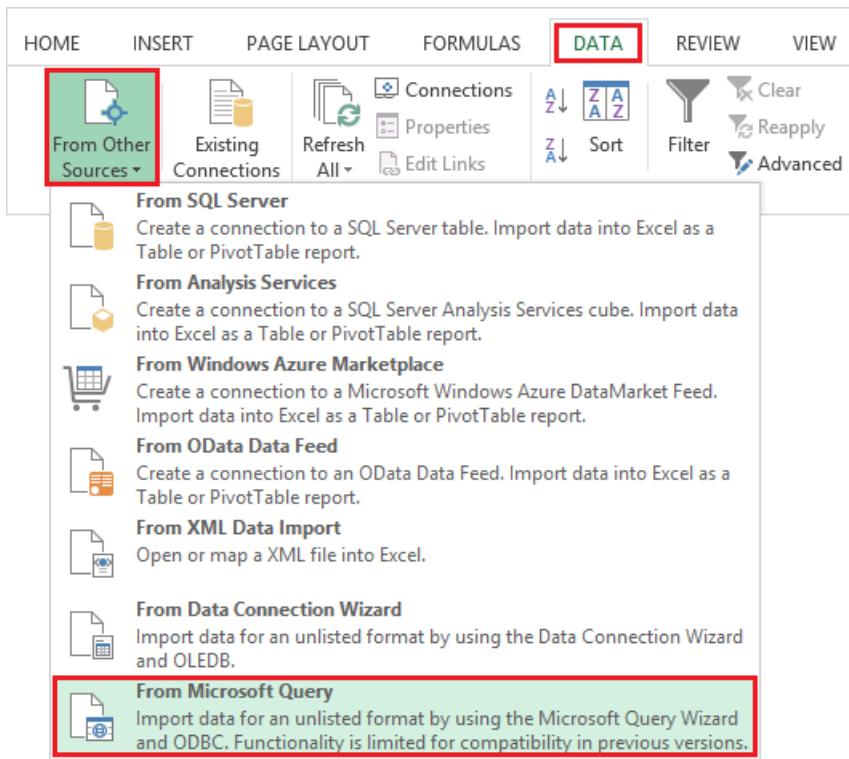
--create the hvac_temperatures table by selecting from the hvac table
CREATE TABLE hvac_temperatures AS
SELECT *, targettemp - actualtemp AS temp_diff,
    IF((targettemp - actualtemp) > 5, 'COLD',
    IF((targettemp - actualtemp) < -5, 'HOT', 'NORMAL')) AS temprange,
    IF((targettemp - actualtemp) > 5, '1', IF((targettemp - actualtemp) < -5, '1', 0)) AS extremetemp
FROM hvac;
```

1. Click the green "Execute" button.
2. Run SELECT COUNT(*) FROM hvac_temperatures; to see if there are any records in the table.

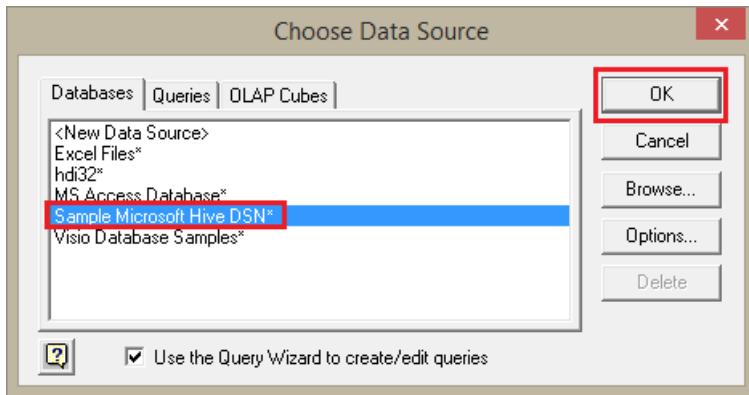
Loading Data into Excel

Once the job has successfully completed, you can use the Microsoft Hive ODBC Driver to import data from Hive into Excel 2016. Once you have installed the driver, use the following steps to connect to the table.

1. Open Excel and create a blank workbook.
2. From the Data tab, select From Other Sources, and then select From Microsoft Query.

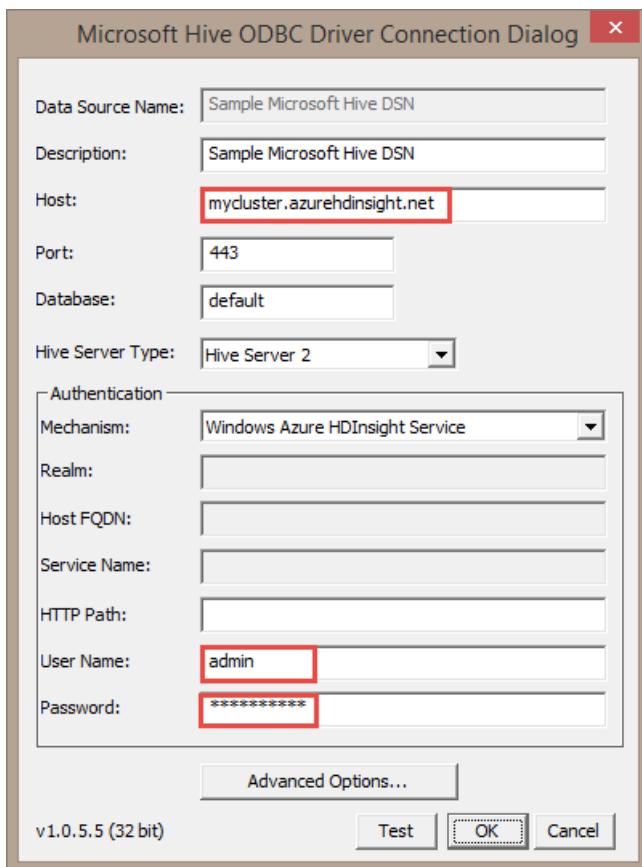


1. Choose the hive ODBC driver.

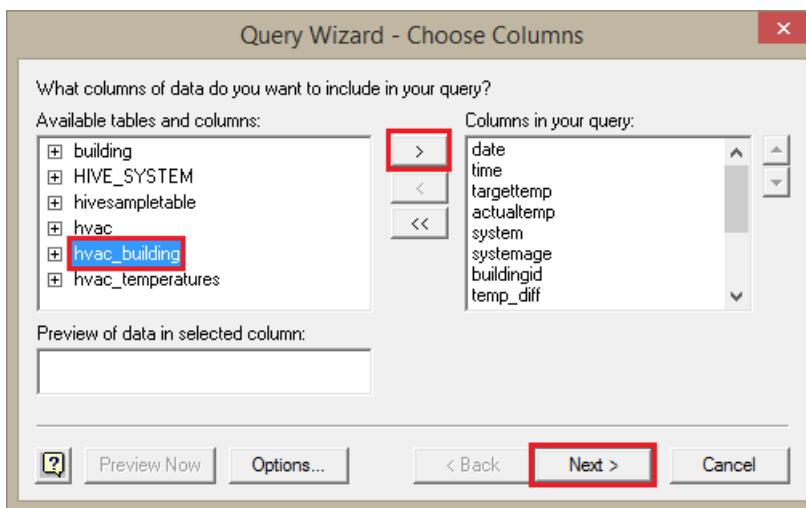


1. In the Microsoft Hive ODBC Driver Connection dialog, enter the following values, and then click OK.

- Host - The host name of your HDInsight cluster. For example, mycluster.azurehdinsight.net
- User Name - The administrator name for your HDInsight cluster (usually admin)
- Password - The administrator password
- All other fields can be left as the default values.



1. In the Query Wizard, select the hvac_temperatures table, and then select the > button.



1. Click Next to continue through the wizard, until you reach a dialog with a Finish button. Click Finish.
2. When the Import Data dialog appears, click OK to accept the defaults. After the query completes, the data will be displayed in Excel.
3. Click **Pivot Table Report** and then **OK**

Import Data

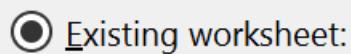
?

X

Select how you want to view this data in your workbook.



Where do you want to put the data?

 =H\$5

Add this data to the Data Model

[Properties...](#)

[OK](#)

[Cancel](#)

1. In the pivot table, drag buildingid to the rows area, date to the columns area, and actualtemp to the values area.

PivotTable F..

▼



Choose fields to add to report:



Search



actualtemp

buildingid

date

extremetemp

system



systemage

targettemp

temp_diff

temprange

time

Drag fields between areas below:

Filters	Columns
	<input type="text" value="date"/> ▼
Rows	Σ Values
<input type="text" value="buildingid"/> ▼	<input type="text" value="Average o..."/> ▼

Defer Layout Update **Update**

...

1. Click on the dropdown arrow next to actualtemp in the values section and click **Value Field Settings**.

Value Field Settings

?

X

Source Name: actualtemp

Custom Name: Count of actualtemp

Summarize Values By

Show Values As

Summarize value field by

Choose the type of calculation that you want to use to summarize data from the selected field

Sum

Count

Average

Max

Min

Product

Number Format

OK

Cancel

1. Change the count to an average and click **OK**.

2. You should be able to see interesting trends by date for each buildings temperature. Your results should look like the following:

A	B	C	D	E	F	G	H	I	J	
Average of actualtemp	Column Labels	6/1/13	6/10/13	6/11/13	6/12/13	6/13/13	6/14/13	6/15/13	6/16/13	6/17/13
Row Labels		66.82352941	70.11111111	66.52941176	71.625	70	67.09090909	65.90909091	66.28571429	66.52941176
1		70.52941176	73.29411765	67.4	70.69230769	67.7	73.83333333	72.63636364	68.61538462	68.88888889
2		67.14285714	64	64.21428571	67.625	68.08333333	71.8	68.10526316	69.88888889	69.11111111
3		64.5	69.25	66.44444444	71	65.35714286	66.33333333	67	68.11111111	68.88888889
4		67.5	66.21428571	67.21428571	66.5	68.66666667	64.72727273	68.92307692	70.15384615	69.11111111
5		65.83333333	72	65.45454545	70.36363636	67.27777778	67.89473684	66.13333333	69.68421053	69.11111111
6		69.41666667	67.44444444	67.86666667	68.35714286	67.4	69.11111111	67.75	67	69.11111111
7		66.44444444	67.66666667	68.8	67.5	71.875	67.33333333	67.54545455	68.33333333	69.11111111
8		67.09090909	68.78571429	69.42857143	66.9	67.76923077	69.625	70.11764706	69.6	69.11111111
9		66.35714286	66.8125	67.73684211	67.28571429	64.75	68.22222222	65.26666667	73	69.11111111
10		66.83333333	70.41666667	66.25	69.21428571	67.16666667	67.875	66.2	67.36363636	69.11111111
11		66.5	65.70588235	64.5	66.91666667	65.86666667	64.88235294	66.31578947	63.23076923	69.11111111
12		66.77777778	64	66.07692308	70.33333333	66.73333333	64.6	62.08333333	73.28571429	69.11111111
13		72.36363636	69.125	66	69.78571429	69.69230769	61.7	65.09090909	67.55555556	69.11111111
14		70.61538462	67.58823529	68.69230769	73.30769231	70.46153846	68.625	69.54545455	67.46153846	69.11111111
15		68.5	67	65.54545455	65.72727273	70	65.38461538	67.30769231	69.11111111	69.11111111
16		64.77777778	67.9375	67.77777778	64.2	67.54545455	68.125	65.9375	65.55555556	69.11111111
17		69.66666667	67.73333333	65.92307692	65.82352941	65.09090909	67.83333333	70	68.27272727	69.11111111
18		68.16666667	70	69.71428571	70.875	71.57142857	71.75	64.75	66.85714286	69.11111111
19		66.41666667	64.2	67.61538462	66.41666667	65.08333333	66	69.25	68.53846154	69.11111111
20		(blank)								
	Grand Total	67.60299625	67.89513109	66.99250936	68.37078652	67.81273408	67.69662921	67.3670412	68.3670412	69.11111111

See also

- [Connect to Hive with JDBC or ODBC](#)
- [Use Hive with HDInsight](#)

Analyze Twitter data using Hive and Hadoop on HDInsight

8/16/2017 • 5 min to read • [Edit Online](#)

Learn how to use Apache Hive to process Twitter data. The result is a list of Twitter users who sent the most tweets that contain a certain word.

IMPORTANT

The steps in this document were tested on HDInsight 3.6.

Linux is the only operating system used on HDInsight version 3.4 or greater. For more information, see [HDInsight retirement on Windows](#).

Get the data

Twitter allows you to retrieve the [data for each tweet](#) as a JavaScript Object Notation (JSON) document through a REST API. [OAuth](#) is required for authentication to the API.

Create a Twitter application

1. From a web browser, sign in to <https://apps.twitter.com/>. Click the **Sign-up now** link if you don't have a Twitter account.
2. Click **Create New App**.
3. Enter **Name, Description, Website**. You can make up a URL for the **Website** field. The following table shows some sample values to use:

FIELD	VALUE
Name	MyHDInsightApp
Description	MyHDInsightApp
Website	http://www.myhdinsightapp.com

4. Check **Yes, I agree**, and then click **Create your Twitter application**.
5. Click the **Permissions** tab. The default permission is **Read only**.
6. Click the **Keys and Access Tokens** tab.
7. Click **Create my access token**.
8. Click **Test OAuth** in the upper-right corner of the page.
9. Write down **consumer key**, **Consumer secret**, **Access token**, and **Access token secret**.

Download tweets

The following Python code downloads 10,000 tweets from Twitter and save them to a file named **tweets.txt**.

NOTE

The following steps are performed on the HDInsight cluster, since Python is already installed.

1. Connect to the HDInsight cluster using SSH:

```
ssh USERNAME@CLUSTERNAME-ssh.azurehdinsight.net
```

For more information, see [Use SSH with HDInsight](#).

2. Use the following commands to install [Tweepy](#), [Progressbar](#), and other required packages:

```
sudo apt install python-dev libffi-dev libssl-dev
sudo apt remove python-openssl
pip install virtualenv
mkdir gettweets
cd gettweets
virtualenv gettweets
source gettweets/bin/activate
pip install tweepy progressbar pyOpenSSL requests[security]
```

3. Use the following command to create a file named **gettweets.py**:

```
nano gettweets.py
```

4. Use the following text as the contents of the **gettweets.py** file:

```

#!/usr/bin/python

from tweepy import Stream, OAuthHandler
from tweepy.streaming import StreamListener
from progressbar import ProgressBar, Percentage, Bar
import json
import sys

#Twitter app information
consumer_secret='Your consumer secret'
consumer_key='Your consumer key'
access_token='Your access token'
access_token_secret='Your access token secret'

#The number of tweets we want to get
max_tweets=10000

#Create the listener class that receives and saves tweets
class listener(StreamListener):
    #On init, set the counter to zero and create a progress bar
    def __init__(self, api=None):
        self.num_tweets = 0
        self.pbar = ProgressBar(widgets=[Percentage(), Bar()], maxval=max_tweets).start()

    #When data is received, do this
    def on_data(self, data):
        #Append the tweet to the 'tweets.txt' file
        with open('tweets.txt', 'a') as tweet_file:
            tweet_file.write(data)
        #Increment the number of tweets
        self.num_tweets += 1
        #Check to see if we have hit max_tweets and exit if so
        if self.num_tweets >= max_tweets:
            self.pbar.finish()
            sys.exit(0)
        else:
            #increment the progress bar
            self.pbar.update(self.num_tweets)
    return True

    #Handle any errors that may occur
    def on_error(self, status):
        print status

#Get the OAuth token
auth = OAuthHandler(consumer_key, consumer_secret)
auth.set_access_token(access_token, access_token_secret)
#Use the listener class for stream processing
twitterStream = Stream(auth, listener())
#Filter for these topics
twitterStream.filter(track=["azure","cloud","hdinsight"])

```

IMPORTANT

Replace the placeholder text for the following items with the information from your twitter application:

- consumer_secret
- consumer_key
- access_token
- access_token_secret

5. Use **Ctrl + X**, then **Y** to save the file.

6. Use the following command to run the file and download tweets:

```
python gettweets.py
```

A progress indicator appears. It counts up to 100% as the tweets are downloaded.

NOTE

If it is taking a long time for the progress bar to advance, you should change the filter to track trending topics. When there are many tweets about the topic in your filter, you can quickly get the 10000 tweets needed.

Upload the data

To upload the data to HDInsight storage, use the following commands:

```
hdfs dfs -mkdir -p /tutorials/twitter/data  
hdfs dfs -put tweets.txt /tutorials/twitter/data/tweets.txt
```

These commands store the data in a location that all nodes in the cluster can access.

Run the HiveQL job

1. Use the following command to create a file containing HiveQL statements:

```
nano twitter.hql
```

Use the following text as the contents of the file:

```
set hive.exec.dynamic.partition = true;  
set hive.exec.dynamic.partition.mode = nonstrict;  
-- Drop table, if it exists  
DROP TABLE tweets_raw;  
-- Create it, pointing toward the tweets logged from Twitter  
CREATE EXTERNAL TABLE tweets_raw (  
    json_response STRING  
)  
STORED AS TEXTFILE LOCATION '/tutorials/twitter/data';  
-- Drop and recreate the destination table  
DROP TABLE tweets;  
CREATE TABLE tweets  
(  
    id BIGINT,  
    created_at STRING,  
    created_at_date STRING,  
    created_at_year STRING,  
    created_at_month STRING,  
    created_at_day STRING,  
    created_at_time STRING,  
    in_reply_to_user_id_str STRING,  
    text STRING,  
    contributors STRING,  
    retweeted STRING,  
    truncated STRING,  
    coordinates STRING,  
    source STRING,  
    retweet_count INT,  
    url STRING,  
    hashtags array<STRING>,  
    user_mentions array<STRING>,  
    first_hashtag STRING
```

```

    first_hashtag STRING,
    first_user_mention STRING,
    screen_name STRING,
    name STRING,
    followers_count INT,
    listed_count INT,
    friends_count INT,
    lang STRING,
    user_location STRING,
    time_zone STRING,
    profile_image_url STRING,
    json_response STRING
);
-- Select tweets from the imported data, parse the JSON,
-- and insert into the tweets table
FROM tweets_raw
INSERT OVERWRITE TABLE tweets
SELECT
    cast(get_json_object(json_response, '$.id_str') as BIGINT),
    get_json_object(json_response, '$.created_at'),
    concat(substr (get_json_object(json_response, '$.created_at'),1,10), ' ',
    substr (get_json_object(json_response, '$.created_at'),27,4)),
    substr (get_json_object(json_response, '$.created_at'),27,4),
    case substr (get_json_object(json_response, '$.created_at'),5,3)
        when "Jan" then "01"
        when "Feb" then "02"
        when "Mar" then "03"
        when "Apr" then "04"
        when "May" then "05"
        when "Jun" then "06"
        when "Jul" then "07"
        when "Aug" then "08"
        when "Sep" then "09"
        when "Oct" then "10"
        when "Nov" then "11"
        when "Dec" then "12" end,
    substr (get_json_object(json_response, '$.created_at'),9,2),
    substr (get_json_object(json_response, '$.created_at'),12,8),
    get_json_object(json_response, '$.in_reply_to_user_id_str'),
    get_json_object(json_response, '$.text'),
    get_json_object(json_response, '$.contributors'),
    get_json_object(json_response, '$.retweeted'),
    get_json_object(json_response, '$.truncated'),
    get_json_object(json_response, '$.coordinates'),
    get_json_object(json_response, '$.source'),
    cast (get_json_object(json_response, '$.retweet_count') as INT),
    get_json_object(json_response, '$.entities.display_url'),
    array(
        trim(lower(get_json_object(json_response, '$.entities.hashtags[0].text'))),
        trim(lower(get_json_object(json_response, '$.entities.hashtags[1].text'))),
        trim(lower(get_json_object(json_response, '$.entities.hashtags[2].text'))),
        trim(lower(get_json_object(json_response, '$.entities.hashtags[3].text'))),
        trim(lower(get_json_object(json_response, '$.entities.hashtags[4].text'))),
    array(
        trim(lower(get_json_object(json_response, '$.entities.user_mentions[0].screen_name'))),
        trim(lower(get_json_object(json_response, '$.entities.user_mentions[1].screen_name'))),
        trim(lower(get_json_object(json_response, '$.entities.user_mentions[2].screen_name'))),
        trim(lower(get_json_object(json_response, '$.entities.user_mentions[3].screen_name'))),
        trim(lower(get_json_object(json_response, '$.entities.user_mentions[4].screen_name'))),
        trim(lower(get_json_object(json_response, '$.entities.hashtags[0].text'))),
        trim(lower(get_json_object(json_response, '$.entities.user_mentions[0].screen_name'))),
        get_json_object(json_response, '$.user.screen_name'),
        get_json_object(json_response, '$.user.name'),
        cast (get_json_object(json_response, '$.user.followers_count') as INT),
        cast (get_json_object(json_response, '$.user.listed_count') as INT),
        cast (get_json_object(json_response, '$.user.friends_count') as INT),
        get_json_object(json_response, '$.user.lang'),
        get_json_object(json_response, '$.user.location'),
        get_json_object(json_response, '$.user.time_zone'),
        ...
    )
)

```

```
get_json_object(json_response, '$.user.profile_image_url'),  
json_response  
WHERE (length(json_response) > 500);
```

2. Press **Ctrl + X**, then press **Y** to save the file.
3. Use the following command to run the HiveQL contained in the file:

```
beeline -u 'jdbc:hive2://headnodehost:10001/;transportMode=http' -i twitter.hql
```

This command runs the the **twitter.hql** file. Once the query completes, you see a `jdbc:hive2//localhost:10001/>` prompt.

4. From the beeline prompt, use the following query to verify that data was imported:

```
SELECT name, screen_name, count(1) as cc  
FROM tweets  
WHERE text like "%Azure%"  
GROUP BY name,screen_name  
ORDER BY cc DESC LIMIT 10;
```

This query returns a maximum of 10 tweets that contain the word **Azure** in the message text.

Next steps

You have learned how to transform an unstructured JSON dataset into a structured Hive table. To learn more about Hive on HDInsight, see the following documents:

- [Get started with HDInsight](#)
- [Analyze flight delay data using HDInsight](#)

Analyze flight delays with Hive and export to SQL Database using Sqoop

8/16/2017 • 6 min to read • [Edit Online](#)

Learn how to analyze flight delay data using Hive on Linux-based HDInsight then export the data to Azure SQL Database using Sqoop.

IMPORTANT

The steps in this document require an HDInsight cluster that uses Linux. Linux is the only operating system used on HDInsight version 3.4 or greater. For more information, see [HDInsight retirement on Windows](#).

Prerequisites

- **An HDInsight cluster.** See [Get started using Hadoop with Hive in HDInsight on Linux](#) for steps on creating a new Linux-based HDInsight cluster.
- **Azure SQL Database.** You use an Azure SQL database as a destination data store. If you do not have a SQL Database already, see [SQL Database tutorial: Create a SQL database in minutes](#).
- **Azure CLI.** If you have not installed the Azure CLI, see [Install and Configure the Azure CLI](#) for more steps.

Download the flight data

1. Browse to [Research and Innovative Technology Administration, Bureau of Transportation Statistics](#).
2. On the page, select the following values:

NAME	VALUE
Filter Year	2013
Filter Period	January
Fields	Year, FlightDate, UniqueCarrier, Carrier, FlightNum, OriginAirportID, Origin, OriginCityName, OriginState, DestAirportID, Dest, DestCityName, DestState, DepDelayMinutes, ArrDelay, ArrDelayMinutes, CarrierDelay, WeatherDelay, NASDelay, SecurityDelay, LateAircraftDelay. Clear all other fields

3. Click **Download**.

Upload the data

1. Use the following command to upload the zip file to the HDInsight cluster head node:

```
scp FILENAME.zip USERNAME@CLUSTERNAME-ssh.azurehdinsight.net:
```

Replace **FILENAME** with the name of the zip file. Replace **USERNAME** with the SSH login for the HDInsight cluster. Replace CLUSTERNAME with the name of the HDInsight cluster.

NOTE

If you use a password to authenticate your SSH login, you are prompted for the password. If you used a public key, you may need to use the `-i` parameter and specify the path to the matching private key. For example,

```
scp -i ~/.ssh/id_rsa FILENAME.zip USERNAME@CLUSTERNAME-ssh.azurehdinsight.net: .
```

- Once the upload has completed, connect to the cluster using SSH:

```
ssh USERNAME@CLUSTERNAME-ssh.azurehdinsight.net
```

For more information, see [Use SSH with HDInsight](#).

- Once connected, use the following to unzip the .zip file:

```
unzip FILENAME.zip
```

This command extracts a .csv file that is roughly 60 MB.

- Use the following command to create a directory on HDInsight storage, and then copy the file to the directory:

```
hdfs dfs -mkdir -p /tutorials/flightdelays/data  
hdfs dfs -put FILENAME.csv /tutorials/flightdelays/data/
```

Create and run the HiveQL

Use the following steps to import data from the CSV file into a Hive table named **Delays**.

- Use the following command to create and edit a new file named **flightdelays.hql**:

```
nano flightdelays.hql
```

Use the following text as the contents of this file:

```

DROP TABLE delays_raw;
-- Creates an external table over the csv file
CREATE EXTERNAL TABLE delays_raw (
    YEAR string,
    FL_DATE string,
    UNIQUE_CARRIER string,
    CARRIER string,
    FL_NUM string,
    ORIGIN_AIRPORT_ID string,
    ORIGIN string,
    ORIGIN_CITY_NAME string,
    ORIGIN_CITY_NAME_TEMP string,
    ORIGIN_STATE_ABR string,
    DEST_AIRPORT_ID string,
    DEST string,
    DEST_CITY_NAME string,
    DEST_CITY_NAME_TEMP string,
    DEST_STATE_ABR string,
    DEP_DELAY_NEW float,
    ARR_DELAY_NEW float,
    CARRIER_DELAY float,
    WEATHER_DELAY float,
    NAS_DELAY float,
    SECURITY_DELAY float,
    LATE_AIRCRAFT_DELAY float)
-- The following lines describe the format and location of the file
ROW FORMAT DELIMITED FIELDS TERMINATED BY ','
LINES TERMINATED BY '\n'
STORED AS TEXTFILE
LOCATION '/tutorials/flightdelays/data';

-- Drop the delays table if it exists
DROP TABLE delays;
-- Create the delays table and populate it with data
-- pulled in from the CSV file (via the external table defined previously)
CREATE TABLE delays AS
SELECT YEAR AS year,
    FL_DATE AS flight_date,
    substring(UNIQUE_CARRIER, 2, length(UNIQUE_CARRIER) -1) AS unique_carrier,
    substring(CARRIER, 2, length(CARRIER) -1) AS carrier,
    substring(FL_NUM, 2, length(FL_NUM) -1) AS flight_num,
    ORIGIN_AIRPORT_ID AS origin_airport_id,
    substring(ORIGIN, 2, length(ORIGIN) -1) AS origin_airport_code,
    substring(ORIGIN_CITY_NAME, 2) AS origin_city_name,
    substring(ORIGIN_STATE_ABR, 2, length(ORIGIN_STATE_ABR) -1) AS origin_state_abr,
    DEST_AIRPORT_ID AS dest_airport_id,
    substring(DEST, 2, length(DEST) -1) AS dest_airport_code,
    substring(DEST_CITY_NAME, 2) AS dest_city_name,
    substring(DEST_STATE_ABR, 2, length(DEST_STATE_ABR) -1) AS dest_state_abr,
    DEP_DELAY_NEW AS dep_delay_new,
    ARR_DELAY_NEW AS arr_delay_new,
    CARRIER_DELAY AS carrier_delay,
    WEATHER_DELAY AS weather_delay,
    NAS_DELAY AS nas_delay,
    SECURITY_DELAY AS security_delay,
    LATE_AIRCRAFT_DELAY AS late_aircraft_delay
FROM delays_raw;

```

2. To save the file, use **Ctrl + X**, then **Y**.
3. To start Hive and run the **flightdelays.hql** file, use the following command:

```
beeline -u 'jdbc:hive2://localhost:10001;transportMode=http' -n admin -f flightdelays.hql
```

NOTE

In this example, `localhost` is used since you are connected to the head node of the HDInsight cluster, which is where HiveServer2 is running.

- Once the **flightdelays.hql** script finishes running, use the following command to open an interactive Beeline session:

```
beeline -u 'jdbc:hive2://localhost:10001/;transportMode=http' -n admin
```

- When you receive the `jdbc:hive2://localhost:10001/` prompt, use the following query to retrieve data from the imported flight delay data.

```
INSERT OVERWRITE DIRECTORY '/tutorials/flightdelays/output'
ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t'
SELECT regexp_replace(origin_city_name, ' ', ''),
       avg(weather_delay)
FROM delays
WHERE weather_delay IS NOT NULL
GROUP BY origin_city_name;
```

This query retrieves a list of cities that experienced weather delays, along with the average delay time, and save it to `/tutorials/flightdelays/output`. Later, Sqoop reads the data from this location and export it to Azure SQL Database.

- To exit Beeline, enter `!quit` at the prompt.

Create a SQL Database

If you already have a SQL Database, you must get the server name. You can find the server name in the [Azure portal](#) by selecting **SQL Databases**, and then filtering on the name of the database you wish to use. The server name is listed in the **SERVER** column.

If you do not already have a SQL Database, use the information in [SQL Database tutorial: Create a SQL database in minutes](#) to create one. Save the server name used for the database.

Create a SQL Database table

NOTE

There are many ways to connect to SQL Database and create a table. The following steps use [FreeTDS](#) from the HDInsight cluster.

- Use SSH to connect to the Linux-based HDInsight cluster, and run the following steps from the SSH session.

- Use the following command to install FreeTDS:

```
sudo apt-get --assume-yes install freetds-dev freetds-bin
```

- Once the install completes, use the following command to connect to the SQL Database server. Replace **serverName** with the SQL Database server name. Replace **adminLogin** and **adminPassword** with the login for SQL Database. Replace **databaseName** with the database name.

```
TDSVER=8.0 tsql -H <serverName>.database.windows.net -U <adminLogin> -P <adminPassword> -p 1433 -D <databaseName>
```

You receive output similar to the following text:

```
locale is "en_US.UTF-8"
locale charset is "UTF-8"
using default charset "UTF-8"
Default database being set to sqooptest
1>
```

4. At the **1>** prompt, enter the following lines:

```
CREATE TABLE [dbo].[delays](
[origin_city_name] [nvarchar](50) NOT NULL,
[weather_delay] float,
CONSTRAINT [PK_delays] PRIMARY KEY CLUSTERED
([origin_city_name] ASC))
GO
```

When the **GO** statement is entered, the previous statements are evaluated. This query creates a table named **delays**, with a clustered index.

Use the following query to verify that the table has been created:

```
SELECT * FROM information_schema.tables
GO
```

The output is similar to the following text:

TABLE_CATALOG	TABLE_SCHEMA	TABLE_NAME	TABLE_TYPE
databaseName	dbo	delays	BASE TABLE

5. Enter **exit** at the **1>** prompt to exit the tsql utility.

Export data with Sqoop

1. Use the following command to verify that Sqoop can see your SQL Database:

```
sqoop list-databases --connect jdbc:sqlserver://<serverName>.database.windows.net:1433 --username <adminLogin> --password <adminPassword>
```

This command returns a list of databases, including the database that you created the delays table in earlier.

2. Use the following command to export data from hivesamptable to the delays table:

```
sqoop export --connect 'jdbc:sqlserver://<serverName>.database.windows.net:1433;database=<databaseName>' --username <adminLogin> --password <adminPassword> --table 'delays' --export-dir '/tutorials/flightdelays/output' --fields-terminated-by '\t' -m 1
```

Sqoop connects to the database containing the delays table, and exports data from the **/tutorials/flightdelays/output** directory to the delays table.

3. After the command completes, use the following to connect to the database using TSQL:

```
TDSVER=8.0 tsql -H <serverName>.database.windows.net -U <adminLogin> -P <adminPassword> -p 1433 -D  
<databaseName>
```

Once connected, use the following statements to verify that the data was exported to the delays table:

```
SELECT * FROM delays  
GO
```

You should see a listing of data in the table. Type `exit` to exit the tsql utility.

Next steps

To learn more ways to work with data in HDInsight, see the following documents:

- [Use Hive with HDInsight](#)
- [Use Oozie with HDInsight](#)
- [Use Sqoop with HDInsight](#)
- [Use Pig with HDInsight](#)
- [Develop Java MapReduce programs for HDInsight](#)
- [Develop Python Hadoop streaming programs for HDInsight](#)

Use Hive with Windows-based HDInsight to analyze logs from websites

8/16/2017 • 1 min to read • [Edit Online](#)

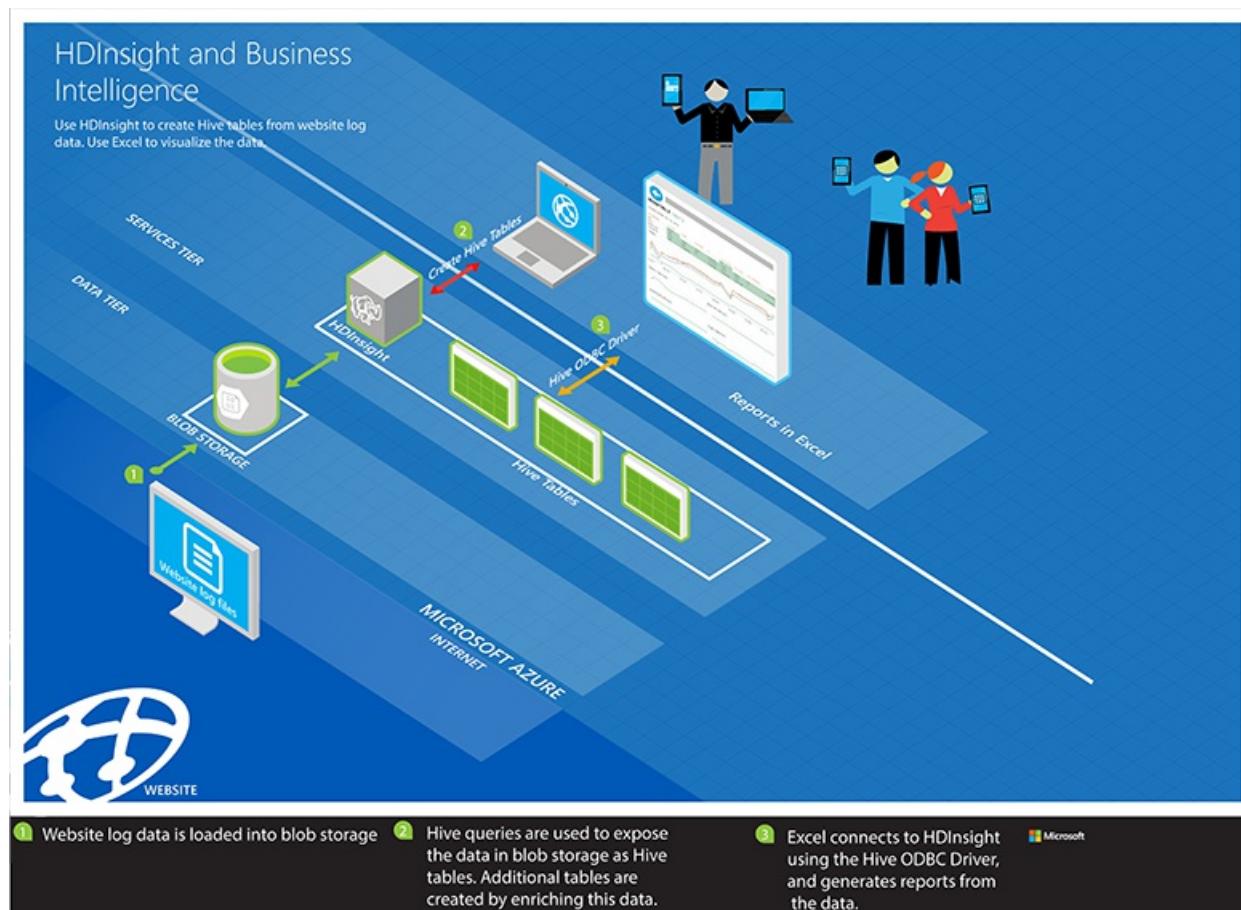
Learn how to use HiveQL with HDInsight to analyze logs from a website. Website log analysis can be used to segment your audience based on similar activities, categorize site visitors by demographics, and to find out the content they view, the websites they come from, and so on.

IMPORTANT

The steps in this document only work with Windows-based HDInsight clusters. HDInsight is only available on Windows for versions lower than HDInsight 3.4. Linux is the only operating system used on HDInsight version 3.4 or greater. For more information, see [HDInsight retirement on Windows](#).

In this sample, you will use an HDInsight cluster to analyze website log files to get insight into the frequency of visits to the website from external websites in a day. You'll also generate a summary of website errors that the users experience. You will learn how to:

- Connect to a Azure Blob storage, which contains website log files.
- Create HIVE tables to query those logs.
- Create HIVE queries to analyze the data.
- Use Microsoft Excel to connect to HDInsight (by using open database connectivity (ODBC) to retrieve the analyzed data.



Prerequisites

- You must have provisioned a Hadoop cluster on Azure HDInsight. For instructions, see [Provision HDInsight Clusters](#).
- You must have Microsoft Excel 2013 or Excel 2010 installed.
- You must have [Microsoft Hive ODBC Driver](#) to import data from Hive into Excel.

To run the sample

1. From the [Azure Portal](#), from the Startboard (if you pinned the cluster there), click the cluster tile on which you want to run the sample.
2. From the cluster blade, under **Quick Links**, click **Cluster Dashboard**, and then from the **Cluster Dashboard** blade, click **HDInsight Cluster Dashboard**. Alternatively, you can directly open the dashboard by using the following URL:

```
https://<clustername>.azurehdinsight.net
```

When prompted, authenticate by using the administrator user name and password you used when provisioning the cluster.

3. From the web page that opens, click the **Getting Started Gallery** tab, and then under the **Solutions with Sample Data** category, click the **Website Log Analysis** sample.
4. Follow the instructions provided on the web page to finish the sample.

Next steps

Try the following sample: [Analyzing sensor data using Hive with HDInsight](#).

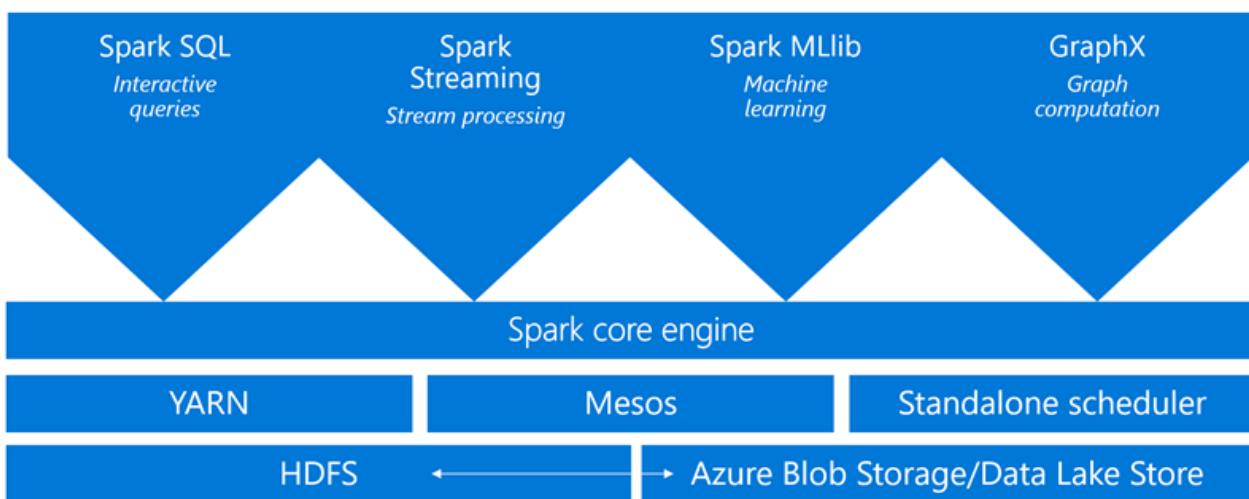
Spark Scenarios - Build high-speed scalable data pipelines

8/16/2017 • 7 min to read • [Edit Online](#)

What is Apache Spark?

Apache Spark is an open-source processing framework that runs large-scale data analytics applications. Spark is built on an in-memory compute engine, which enables high-performance querying on big data. It takes advantage of a parallel data-processing framework that persists data in-memory and disk if needed. This allows Spark to deliver 100-times faster speed and a common execution model for tasks such as extract, transform, load (ETL), batch, interactive queries and others on data in an Apache Hadoop Distributed File System (HDFS). Azure makes Apache Spark easy and cost effective to deploy with no hardware to buy, no software to configure, a full notebook experience for authoring compelling narratives and integration with partner business intelligence tools.

Spark includes an integrated set of APIs which allows for many different types of interaction, processing and query of associated data. These include SparkSQL, SparkML and others. Shown below is the Spark architecture for reference.



Example data pipeline use cases for Spark on HDInsight

Spark clusters in HDInsight enable a large number of high-volume data pipeline scenarios. Below are just a few of the common example scenarios.

1. Interactive data analysis and BI

Apache Spark in HDInsight stores data in Azure Storage or Azure Data Lake Store. Business experts and key decision makers can analyze and build reports over that data and use Microsoft Power BI to build interactive reports from the analyzed data. Analysts can start from unstructured/semi structured data in cluster storage, define a schema for the data using notebooks, and then build data models using Microsoft Power BI. Spark clusters in HDInsight also support a number of third party BI tools such as Tableau making it an ideal platform for data analysts, business experts, and key decision makers. [Look at a tutorial for this scenario.](#)

2. Spark streaming and real-time data analysis

Spark clusters in HDInsight offer a rich support for building real-time analytics solutions. While Spark already has connectors to ingest data from many sources like Kafka, Flume, Twitter, ZeroMQ, or TCP sockets, Spark in HDInsight adds first-class support for ingesting data from Azure Event Hubs. Event Hubs are the most widely used

queuing service on Azure. Having an out-of-the-box support for Event Hubs makes Spark clusters in HDInsight an ideal platform for building real time analytics pipeline. [Look at a tutorial for this scenario.](#)

3. Spark Machine Learning

Apache Spark comes with MLlib, a machine learning library built on top of Spark that you can use from a Spark cluster in HDInsight. Spark cluster on HDInsight also includes Anaconda, a Python distribution with a variety of packages for machine learning. Couple this with a built-in support for Jupyter and Zeppelin notebooks, and you have a top-of-the-line environment for creating machine learning applications. Look at a tutorial for this scenario - [Predict building temperatures using HVAC data.](#)

Additionally, some of the common applications of machine learning scenarios with Spark on Azure are shown in the chart below.

Vertical	Sales and Marketing	Finance and Risk	Customer and Channel	Operations and Workforce
Retail	Demand forecasting	Fraud detection	Personalization	Store location demographics
	Loyalty programs	Pricing strategy	Lifetime customer value	Supply chain management
	Cross-sell and upsell		Product segmentation	
	Customer acquisition			Inventory management
Financial Services	Customer churn	Fraud detection	Personalization	Call center optimization
	Loyalty programs	Risk and compliance	Lifetime customer value	Pay for performance
	Cross-sell and upsell	Loan defaults		
	Customer acquisition			
Healthcare	Marketing mix optimization	Fraud detection	Population health	Operational efficiency
	Patient acquisition	Bill collection	Patient demographics	Pay for performance
Manufacturing	Demand forecasting	Pricing strategy	Supply chain optimization	Remote monitoring
	Marketing mix optimization	Perf risk management	Personalization	Predictive maintenance
				Asset management

For even more scenarios, see [Eight scenarios with Apache Spark on Azure](#)

Building with HDInsight and Spark by example - Analyze weblogs

This example uses a Jupyter notebook and demonstrates how to analyze website log data using a custom library with Spark on HDInsight. The custom library we use is a Python library called `iislogparser.py`.

Create and use a Jupyter Notebook

Create a HDInsight cluster of type Apache Spark Click the `Cluster Dashboards` blade in the Azure portal for your HDInsight instance to open that page On the `cluster Dashboards` page, click the link to `Jupyter Notebook` to open that page

On the Jupyter Notebook page, click the New->PySpark link to create a new PySpark Jupyter notebook

The screenshot shows the Jupyter Notebook interface. At the top, there's a navigation bar with tabs for 'Files', 'Running', and 'Clusters'. Below the navigation bar is a sidebar titled 'Select items to perform actions on them.' containing a file tree with 'PySpark' and 'Scala' folders. To the right of the file tree is a context menu with options like 'Text File', 'Folder', 'Terminal', 'Notebooks', 'PySpark' (which is highlighted), 'PySpark3', and 'Spark'. The main workspace is currently empty.

At the top of the new notebook, click to rename the notebook from Untitled to Log Analysis

The screenshot shows the newly renamed Jupyter Notebook titled 'LogAnalysis.ipynb'. The top navigation bar includes 'File', 'Edit', 'View', 'Insert', 'Cell', 'Kernel', 'Widgets', and 'Help'. A toolbar below the navigation bar contains icons for code, cell, and cell toolbar. The main workspace shows a single code cell labeled 'In []:'.

Load Data into Spark using PySpark

You'll use this notebook to run jobs that process your raw sample data and save it as a Hive table. The sample data is a .csv file (hvac.csv) available on all clusters by default. Once your data is saved as a Hive table, in the next section we will connect to the Hive table using BI tools such as Power BI and Tableau. Because you created a notebook using the PySpark kernel, you do not need to create any contexts explicitly. The Spark and Hive contexts will be automatically created for you when you run the first code cell. You can start by importing the types that are required for this scenario. Paste the following snippet in an empty cell, and then press SHIFT + ENTER.

In the first cell of the notebook, enter the python code shown below and press SHIFT + ENTER to execute the code.

```
from pyspark.sql import Row
from pyspark.sql.types import *
```

You should see output showing that the SparkContext and HiveContext objects are not available, as shown in the example below.

The screenshot shows the Jupyter Notebook interface with the notebook titled 'LogAnalysis'. The top navigation bar and toolbar are identical to the previous screenshot. The main workspace shows the code cell 'In [1]:' with the imported modules. Below the code cell, the output shows the message 'Starting Spark application' followed by a table of cluster information. The table has columns for ID, YARN Application ID, Kind, State, Spark UI, Driver log, and Current session?. There is one row with ID 0, YARN Application ID 'application_1498784307215_0004', Kind 'pyspark', State 'idle', and both Spark UI and Driver log links pointing to 'Link'. The 'Current session?' column shows a checked checkbox. The final output message indicates that 'SparkContext available as 'sc''. and 'HiveContext available as 'sqlContext''. A new code cell 'In []:' is visible at the bottom.

Create an RDD using the sample log data already available on the cluster. You can access the data in the default storage account associated with the cluster at `\HdiSamples\HdiSamples\WebsiteLogSampleData\SampleLog\909f2b.log`.

```
logs = sc.textFile('wasbs:///HdiSamples/HdiSamples/WebsiteLogSampleData/SampleLog/909f2b.log')
```

Retrieve a sample log set to verify that the previous step completed successfully. You should see output similar to the following example shown below.

```
logs.take(5)
```

```
In [2]: logs = sc.textFile('wasbs:///HdiSamples/HdiSamples/WebsiteLogSampleData/SampleLog/909f2b.log')
```

```
In [3]: logs.take(5)
```

```
[u'#Software: Microsoft Internet Information Services 8.0', u'#Fields: date time s-sitename c s-method cs-uri-stem cs-uri-query s-port cs-username c-ip cs(User-Agent) cs(Cookie) cs(Refere r) cs-host sc-status sc-substatus sc-win32-status sc-bytes cs-bytes time-taken', u'2014-01-01 02:01:09 SAMPLEWEBSITE GET /blogposts/mvc4/step2.png X-ARR-LOG-ID=2ec4b8ad-3cf0-4442-93ab-837 317ece6a1 80 - 1.54.23.196 Mozilla/5.0+(Windows+NT+6.3;+WOW64)+AppleWebKit/537.36+(KHTML,+lik e+Gecko)+Chrome/31.0.1650.63+Safari/537.36 - http://weblogs.asp.net/sample/archive/2007/12/0 9/asp-net-mvc-framework-part-4-handling-form-edit-and-post-scenarios.aspx www.sample.com 200 0 0 53175 871 46', u'2014-01-01 02:01:09 SAMPLEWEBSITE GET /blogposts/mvc4/step3.png X-ARR-L OG-ID=9eace870-2f49-4efd-b204-0d170da46b4a 80 - 1.54.23.196 Mozilla/5.0+(Windows+NT+6.3;+WOW6 4)+AppleWebKit/537.36+(KHTML,+like+Gecko)+Chrome/31.0.1650.63+Safari/537.36 - http://weblogs. asp.net/sample/archive/2007/12/09/asp-net-mvc-framework-part-4-handling-form-edit-and-post-sc enarios.aspx www.sample.com 200 0 0 51237 871 32', u'2014-01-01 02:01:09 SAMPLEWEBSITE GET /b logposts/mvc4/step4.png X-ARR-LOG-ID=4bea5b3d-8ac9-46c9-9b8c-ec3e9500cbea 80 - 1.54.23.196 Mo zilla/5.0+(Windows+NT+6.3;+WOW64)+AppleWebKit/537.36+(KHTML,+like+Gecko)+Chrome/31.0.1650.63+ Safari/537.36 - http://weblogs.asp.net/sample/archive/2007/12/09/asp-net-mvc-framework-part-4 -handling-form-edit-and-post-scenarios.aspx www.sample.com 200 0 0 72177 871 47']
```

Analyze Data using a custom Python library

In the output above, the first couple lines include the header information and each remaining line matches the schema described in that header. Parsing such logs could be complicated. So, we use a custom Python library `iislogparser.py` that makes parsing such logs much easier. By default, this library is included with your Spark cluster on HDInsight at `/HdiSamples/HdiSamples/WebsiteLogSampleData/iislogparser.py`. Because this library is not in the `PYTHONPATH` we must distribute it to all the worker nodes by running the `addPyFile` command on the `SparkContext`. Use the command shown below to do this.

```
sc.addPyFile('wasbs:///HdiSamples/HdiSamples/WebsiteLogSampleData/iislogparser.py')
```

The `iislogparser` provides a function `parse_log_line` that returns `None` if a log line is a header row, and returns an instance of the `LogLine` class if it encounters a log line. Use the `LogLine` class to extract only the log lines from the RDD. Use the command shown below to do this.

```
def parse_line(l):
    import iislogparser
    return iislogparser.parse_log_line(l)
logLines = logs.map(parse_line).filter(lambda p: p is not None).cache()
```

Verify successful completion using the command shown below. Your processed output should look similar to the image shown after the command.

```
logLines.take(2)
```

```

def parse_line(l):
    import islogparser
    return islogparser.parse_log_line(l)
logLines = logs.map(parse_line).filter(lambda p: p is not None).cache()

logLines.take(2)

[2014-01-01 02:01:09 SAMPLEWEBSITE GET /blogposts/mvc4/step2.png X-ARR-LOG-ID=2ec4b8ad-3cf0-4
442-93ab-837317ece6a1 80 - 1.54.23.196 Mozilla/5.0+(Windows+NT+6.3;+WOW64)+AppleWebKit/537.36
+(KHTML,+like+Gecko)+Chrome/31.0.1650.63+Safari/537.36 - http://weblogs.asp.net/sample/archiv
e/2007/12/09/asp-net-mvc-framework-part-4-handling-form-edit-and-post-scenarios.aspx www.samp
le.com 200 0 0 53175 871 46, 2014-01-01 02:01:09 SAMPLEWEBSITE GET /blogposts/mvc4/step3.png
X-ARR-LOG-ID=9eace870-2f49-4efd-b204-0d170da46b4a 80 - 1.54.23.196 Mozilla/5.0+(Windows+NT+
6.3;+WOW64)+AppleWebKit/537.36+(KHTML,+like+Gecko)+Chrome/31.0.1650.63+Safari/537.36 - ht
tp://weblogs.asp.net/sample/archive/2007/12/09/asp-net-mvc-framework-part-4-handling-form-edit-
-and-post-scenarios.aspx www.sample.com 200 0 0 51237 871 32]

```

The `LogLine` class, in turn, has some useful methods, like `is_error()`, which returns whether a log entry has an error code. Use this to compute the number of errors in the extracted log lines, and then log all the errors to a different file.

```

errors = logLines.filter(lambda p: p.is_error())
numLines = logLines.count()
numErrors = errors.count()
print 'There are', numErrors, 'errors and', numLines, 'log entries'
errors.map(lambda p:
str(p)).saveAsTextFile('wasbs:///HdiSamples/HdiSamples/WebsiteLogSampleData/SampleLog/909f2b-2.log')

```

You can also use `Matplotlib` to construct a visualization of the data. For example, if you want to isolate the cause of requests that run for a long time, you might want to find the files that take the most time to serve on average. The snippet below retrieves the top 25 resources that took most time to serve a request.

```

def avgTimeTakenByKey(rdd):
    return rdd.combineByKey(lambda line: (line.time_taken, 1),
                           lambda x, line: (x[0] + line.time_taken, x[1] + 1),
                           lambda x, y: (x[0] + y[0], x[1] + y[1]))\
    .map(lambda x: (x[0], float(x[1][0]) / float(x[1][1])))

avgTimeTakenByKey(logLines.map(lambda p: (p.cs_uri_stem, p))).top(25, lambda x: x[1])

```

You can also present this information in the form of plot. As a first step to create a plot, let us first create a temporary table `AverageTime`. The table groups the logs by time to see if there were any unusual latency spikes at any particular time. You can then use the `%%sql` command to be able to run the following SparkSQL query to get all the records in the `AverageTime` table and to persist the output locally as a Pandas dataframe by adding the `-o averagetime` parameter to the SQL query. Results will be as similiar to those shown in the screenshot below.

```

avgTimeTakenByMinute = avgTimeTakenByKey(logLines.map(lambda p: (p.datetime.minute, p))).sortByKey()
schema = StructType([StructField('Minutes', IntegerType(), True),StructField('Time', FloatType(), True)])
avgTimeTakenByMinuteDF = sqlContext.createDataFrame(avgTimeTakenByMinute, schema)
avgTimeTakenByMinuteDF.registerTempTable('AverageTime')

```

```

%%sql -o averagetime
SELECT * FROM AverageTime

```

```

avgTimeTakenByMinute = avgTimeTakenByKey(logLines.map(lambda p: (p.datetime.minute, p))).sortByKey()
schema = StructType([StructField('Minutes', IntegerType(), True),
                     StructField('Time', FloatType(), True)])
avgTimeTakenByMinuteDF = sqlContext.createDataFrame(avgTimeTakenByMinute, schema)
avgTimeTakenByMinuteDF.registerTempTable('AverageTime')

```

```

%%sql -o averagetime
SELECT * FROM AverageTime

```

Type: [Table](#) [Pie](#) [Scatter](#) [Line](#) [Area](#) [Bar](#)

Minutes	Time
1	52.333332
2	66.125000
3	60.000000
4	45.062500
5	43.000000
6	35.750000
7	48.500000
8	57.153847
9	37.500000
13	15.000000
14	46.593750
16	59.200000
17	67.000000

You can now use `Matplotlib`, a library used to construct visualization of data, to create a plot. Because the plot must be created from the locally persisted `averagetime` dataframe, the code snippet must begin with the `%%local` magic. This ensures that the code is run locally on the Jupyter server. You'll see output as shown in the screenshot below.

```

%%local
%matplotlib inline
import matplotlib.pyplot as plt

plt.plot(averagetime['Minutes'], averagetime['Time'], marker='o', linestyle='--')
plt.xlabel('Time (min)')
plt.ylabel('Average time taken for request (ms)')

```

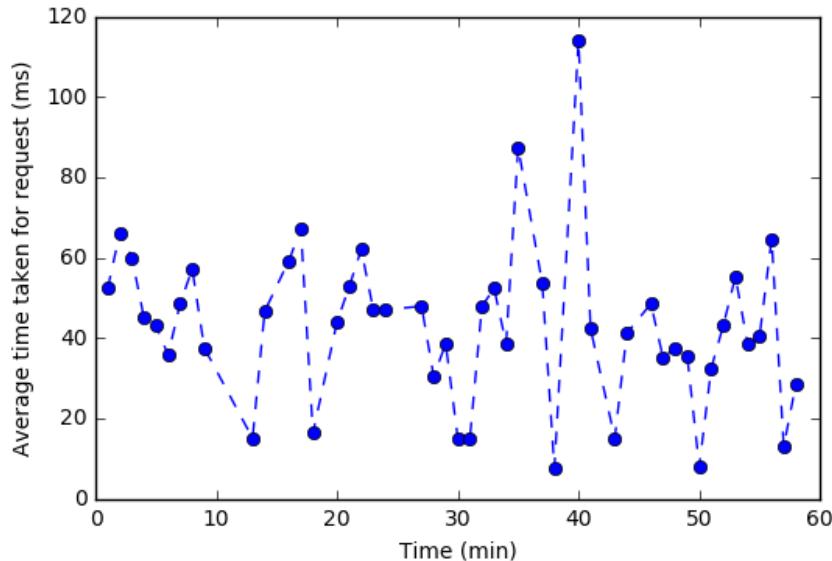
```

%%local
%matplotlib inline
import matplotlib.pyplot as plt

plt.plot(averagetime['Minutes'], averagetime['Time'], marker='o', linestyle='--')
plt.xlabel('Time (min)')
plt.ylabel('Average time taken for request (ms)')

<matplotlib.text.Text at 0x7f1f484bb150>

```



After you have finished running the application, you should shutdown the notebook to release the resources. To do so, from the File menu on the notebook, click Close and Halt. This will shutdown and close the notebook.

NOTE: The complete `LogAnalysis.ipynb` notebook is also available as a Jupyter notebook that you can upload and run on your HDInsight-Spark cluster.

Next steps

In this article, you learned about the large number of scalable data pipeline scenario that are possible using Apache Spark on HDInsight. You also saw a complete example running in a Jupyter PySpark notebook. The flexibility of both HDInsight and the Spark ecosystem allow for building a wide variety of data pipelines that can address many business needs.

For more information, see:

- [Introductions to Spark on HDInsight](#)
- [Build Apache Spare machine learning applications on Azure HDInsight](#) *Use Spark MLlib to build a machine learning application and analyze a dataset

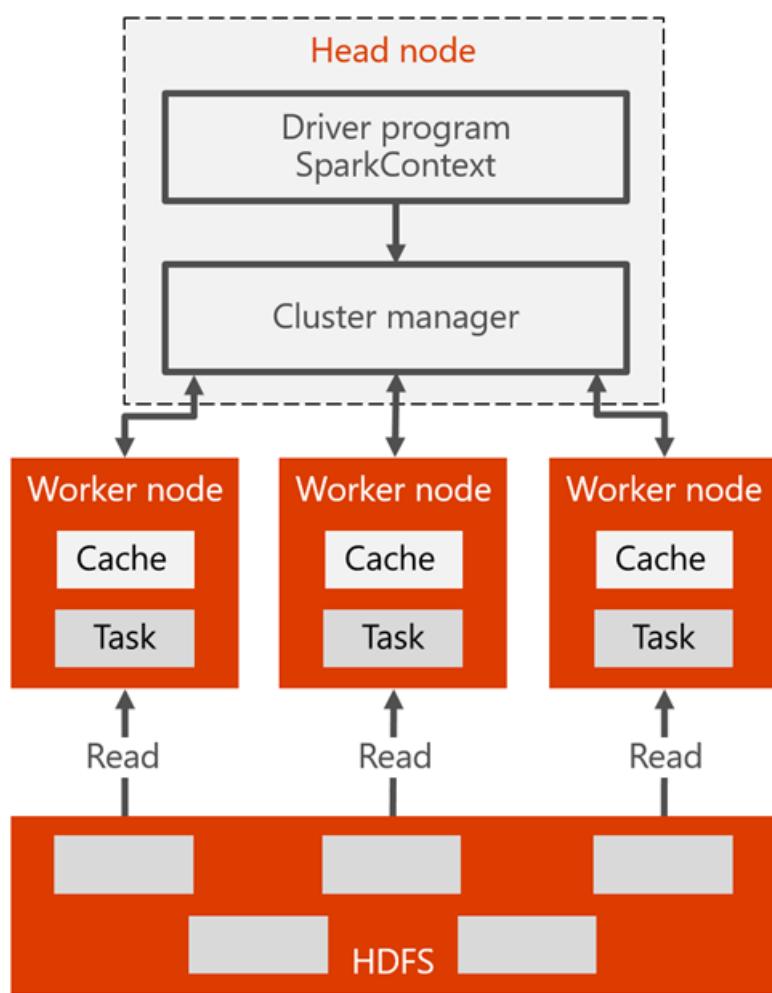
Use Spark with HDInsight

8/16/2017 • 10 min to read • [Edit Online](#)

[Spark on HDInsight](#) provides us with a unified framework for running large-scale data analytics applications that capitalizes on an in-memory compute engine at its core, for high performance querying on big data. It leverages a parallel data processing framework that persists data in-memory and disk if needed. This allows Spark to deliver both 100x faster speed and a common execution model to various tasks like extract, transform, load (otherwise known as ETL), batch, and interactive queries on data in Hadoop distributed file system (or, HDFS). One of the advantages Spark's unified framework gives us, is the ability to use the same code for both batch processing and realtime stream processing.

Spark cluster architecture

Here is the Spark cluster architecture and how it works:



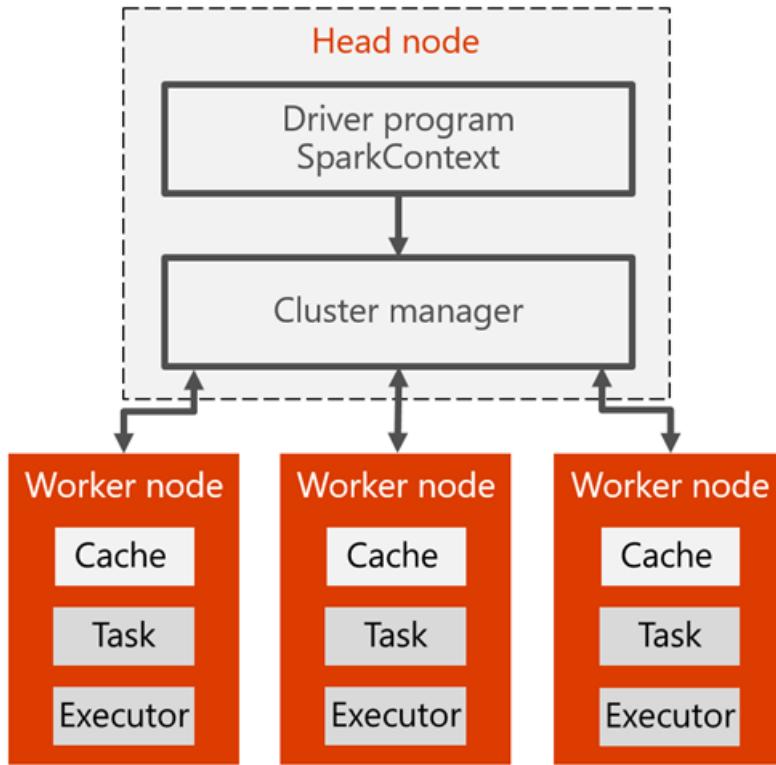
In the head node, we have the Spark master that manages the number of applications, the apps are mapped to the Spark driver. Every app is managed by Spark master in various ways. Spark can be deployed on top of Mesos, YARN, or the Spark cluster manager, which allocates worker node resources to an application. In HDInsight, Spark runs using the YARN cluster manager. The resources in the cluster are managed by Spark master in HDInsight. That means the Spark master has knowledge of which resources, like memory, are occupied or available on the worker node.

The driver runs the user's main function and executes the various parallel operations on the worker nodes. Then,

the driver collects the results of the operations. The worker nodes read and write data from and to the Hadoop distributed file system (HDFS). The worker nodes also cache transformed data in-memory as Resilient Distributed Datasets (RDDs).

The Spark cluster architecture driver

The driver performs the following:



Once the app is created in the Spark master, the resources are allocated to the apps by Spark master, creating an execution called the Spark driver. The Spark driver basically creates the `SparkContext`. When it creates the `SparkContext`, it starts creating the RDDs. The metadata of the RDDs are stored on the Spark driver.

The Spark driver connects to the Spark master and is responsible for converting an application to a directed graph (DAG) of individual tasks that get executed within an executor process on the worker nodes. Each application gets its own executor processes, which stay up for the duration of the whole application and run tasks in multiple threads.

Batch processing in Spark

When working with big data, you have two high-level options with which you can process that data: [stream processing](#) and batch processing. If your needs dictate real-time (subsecond) processing, you will opt to process your data using a stream processing component, like Spark Streaming. On the other hand, batch processing is for queries or programs that take much longer, such as tens of minutes, hours, or days to complete.

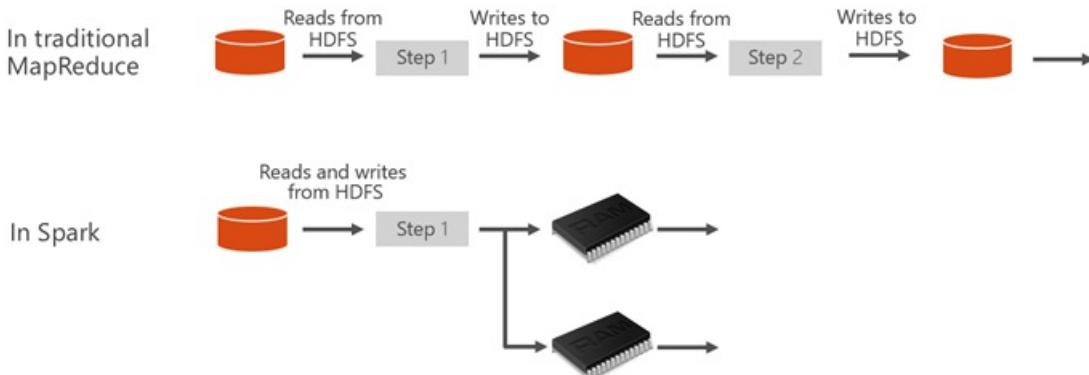
Some example batch processing scenarios include ETL (extract-transform-load) pipelines, working with extremely large, pre-existing datasets, or in situations where computation or transformation against the data takes significant time.

Whether working with large datasets through batch processing, or stream processing, a common way to work with the data more efficiently is through a concept called *schema on read*. As the name implies, you apply the data's schema as you are loading it from disk, or memory. This offers the flexibility of working with data from various sources and formats that do not already have the schema applied. You can take the data in whatever format it arrives and overlay a schema you've created to make it easier to work with that data.

Spark offers a very fast processing engine for running batch processing against very large data sets, while making the core processing engine available to stream processing as well.

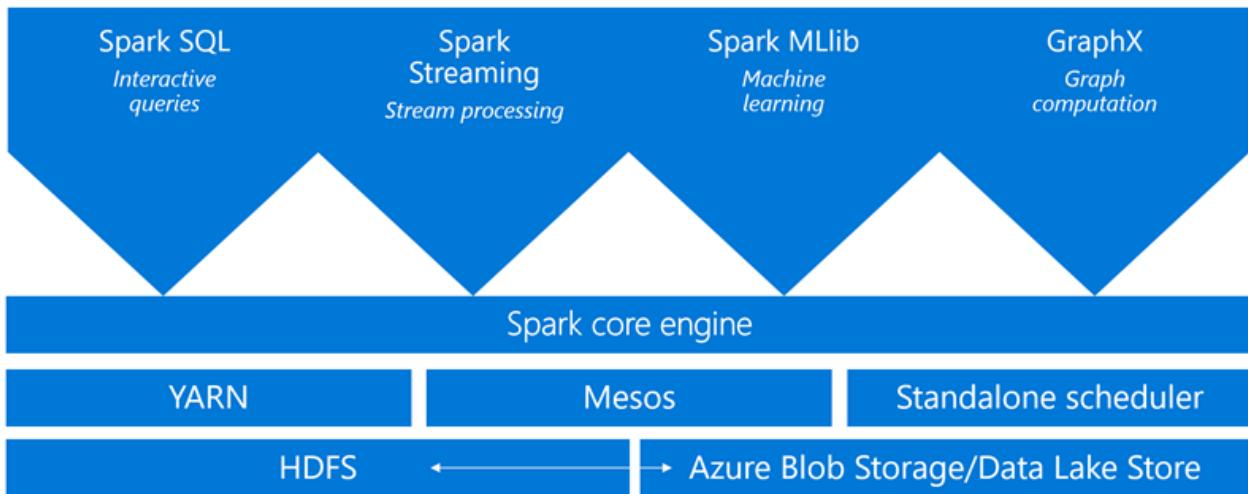
Spark vs. traditional MapReduce

What makes Spark fast? How is the architecture of Apache Spark different than traditional MapReduce, allowing it to offer better performance for data sharing?



Spark provides primitives for in-memory cluster computing. A Spark job can load and cache data into memory and query it repeatedly, much more quickly than disk-based systems. Spark also integrates into the Scala programming language to let you manipulate distributed data sets like local collections. There's no need to structure everything as map and reduce operations.

Data sharing between operations is faster, since data is in-memory. Hadoop shares data through HDFS, an expensive option. It also maintains three replicas.



At its base, [Spark Core](#) is the engine that drives the distributed, large-scale parallel processing, memory management/fault recovery, the scheduling, distribution, and monitoring of jobs on a cluster, and interaction with the underlying storage system.

On top of Spark Core runs a compliment of higher-level libraries that can be seamlessly used in the same application: Spark SQL, Spark Streaming, MLlib, and GraphX. This means that much of the work you perform to execute batch processing on Spark can be reused for streaming data and other activities.

Working with data

[Resilient Distributed Datasets \(RDDs\)](#) are the primary abstraction in Spark, a fault-tolerant collection of elements stored in-memory or on-disk that can be operated on in parallel. An RDD can hold any type of object, and is created by loading an external dataset or distributing a collection from the driver program.

There are two types of RDDs:

1. **Parallelized collection** which applies a parallel transformation to an existing Scala collection; users must specify the number of partitions.
2. **A Hadoop data set** to run functions on each record of a file in HDFS or any other storage system supported by Hadoop.

An RDD can be persisted in-memory across operations. When an RDD is persisted, each node stores any partitions of it that it computes in-memory and then reuses them in other actions on the data set. You can mark an RDD as persistent just by calling the `persist()` or `cache()` method. You can also specify the storage level: on-disk or in-memory as a serialized Java object. Cached, or persistent, RDDs are fault-tolerant without replication.

Each RDD maintains its lineage (for example, the sequence of transformations that resulted in the RDD). If an RDD is lost because a node crashed, it can be reconstructed by replaying the sequence of operations.

There are two types of operations that RDDs support:

1. **Transformations** create a new data set from an existing data set. They're considered *lazy*, meaning they do not compute their results right away. They are only computed when an action requires a result to be returned to the driver program. This does not apply to persistent RDDs. Examples include: map, filter, sample, union, and more.
2. **Actions** return a value to the driver program after running a computation on the data set. Examples include: reduce, collect, count, first, foreach, etc.

Transformations and actions code sample

The following code sample demonstrates searching through error messages in a log file that is stored in HDFS:

```
val file = spark.textFile("hdfs://...")  
val errors = file.filter(line => line.contains("ERROR"))  
// Cache errors  
errors.cache()  
// Count all the errors  
errors.count()  
// Count errors mentioning MySQL  
errors.filter(line => line.contains("Web")).count()  
// Fetch the MySQL errors as an array of strings  
errors.filter(line => line.contains("Error")).collect()
```

In this sample, we're using `hdfs()` and `filter()` **transformations**, and `count()` and `collect()` **actions**.

Notice that in the block of sample code, there are 4 comments:

- **Cache errors** – Implementing the `cache()` method will collect all of the errors present.
- **Count all errors** – Calling the `count()` action counts all the errors in the referenced data.
- **Count errors mentioning MySQL** – When implementing this code, MySQL errors are counted with the count action.
- **Fetch the MySQL errors as an array of strings** – When implementing this code, MySQL errors are extracted as an array of strings by way of the collect action.

RDD-supported transformations

TRANSFORMATION	DESCRIPTION
<code>map(func)</code>	Returns a new distributed data set formed by passing each element of the source through a function func.

TRANSFORMATION	DESCRIPTION
<code>filter(func)</code>	Returns a new data set formed by selecting those elements of the source on which func returns true.
<code>flatmap(func)</code>	Similar to map, but allows each input item to be mapped to zero or more output items (func should return a Seq rather than a single item).
<code>sample(withReplacement, fraction, seed)</code>	Samples a fraction of the data, with or without replacement, using a given random number generator seed.
<code>union(otherDataset)</code>	Returns a new data set that contains the union of the elements in the source data set and in the argument.
<code>distinct([numTasks])</code>	Returns a new data set that contains the distinct elements of the source data set.
<code>groupByKey([numTasks])</code>	When called on a data set of (K, V) pairs, returns a data set of (K, Seq[V]) pairs.
<code>reduceByKey(func, [numTasks])</code>	When called on a data set of (K, V) pairs, returns a data set of (K, V) pairs where the values for each key are aggregated using the given reduce function.
<code>sortByKey([ascending], [numTasks])</code>	When called on a data set of (K, V) pairs where K implements Ordering, returns a data set of (K, V) pairs sorted by keys in ascending or descending order, as specified in the Boolean ascending argument.
<code>join(otherDataset,[numTasks])</code>	When called on data sets of type (K, V) and (K, W), returns a data set of (K, (V, W)) pairs with all pairs of elements for each key.
<code>cogroup(otherDataset, [numTasks])</code>	When called on data sets of type (K, V) and (K, W), returns a data set of (K, Seq[V], Seq[W]) tuples, also called groupWith.
<code>cartesian(otherDataset)</code>	When called on data sets of types T and U, returns a data set of (T, U) pairs (all pairs of elements).

RDD-supported actions

ACTION	DESCRIPTION
<code>saveAsTextFile(path)</code>	Writes the elements of the data set as a text file (or a set of text files) in a given directory in either the local filesystem, HDFS, or other Hadoop-supported file systems. Spark will call <code>ToString</code> on each element to convert it to a line of text in the file.
<code>saveAsSequenceFile(path)</code>	Writes the elements of the data set as a Hadoop SequenceFile in a given path in the local filesystem, HDFS, or any other Hadoop-supported file system. Only available on RDDs of key-value pairs that either implement the <code>Writable</code> interface of Hadoop, or are implicitly convertible to <code>Writable</code> (Spark includes conversions for basic types like <code>Int</code> and <code>Double</code> , <code>String</code>).

ACTION	DESCRIPTION
<code>countByKey()</code>	Returns a "Map" of (K, Int) pairs with the count of each key. Only available on RDDs of type (K, V).
<code>foreach(func)</code>	Runs a function func on each element of the data set. Usually done for side effects, such as updating an accumulator.
<code>reduce(func)</code>	Aggregates elements of the data set using a function func (which takes two arguments and returns one), and should be commutative and associative in order to be computed correctly in parallel.
<code>collect()</code>	Returns all the elements of the data set as an array at the driver program. Usually useful after a filter or other operation returns a sufficiently small subset of the data.
<code>first()</code>	Returns the first element of the data set—similar to <code>take(n)</code> .
<code>count()</code>	Returns the number of elements in the data set.
<code>take(n)</code>	Returns an array with the first n elements of the data set. Currently not executed in parallel, instead the driver program computes all the elements.
<code>takeSample (withReplacement, fraction, seed)</code>	Returns an array with a random sample of num elements of the data set, with or without replacement, using the given random number generator seed.

RDD-supported persistence options

OPTION	DESCRIPTION
<code>MEMORY_ONLY</code>	Stores RDD as deserialized Java objects in the JVM. If the RDD does not fit in-memory, some partitions will not be cached and will be recomputed on the fly each time they are needed. This is the default level.
<code>MEMORY_AND_DISK</code>	Stores RDD as deserialized Java objects in the JVM. If the RDD does not fit in-memory, store the partitions that do not fit on-disk, and read them from there when they are needed.
<code>MEMORY_ONLY_SER</code>	Stores RDD as serialized Java objects (one byte array per partition). This is generally more space-efficient than deserialized objects, especially when using a fast serializer, but it is more CPU-intensive to read.
<code>MEMORY_AND_DISK_SER</code>	Similar to <code>MEMORY_ONLY_SER</code> , but spills partitions that do not fit in-memory to disk, instead of recomputing them on the fly each time they are needed.
<code>DISK_ONLY</code>	Stores the RDD partitions only on-disk.
<code>MEMORY_ONLY_2, MEMORY_AND_DISK_2, and more</code>	Same as the levels above, but replicates each partition on two cluster nodes.

Accumulators



Accumulators are variables that can only be added to through an associative operation. They are used to implement counters and sums efficiently in parallel. Spark natively supports accumulators of numeric value types and standard mutable collections. It is possible for programmers to extend for new types. One thing of note, only the driver program can read the value of an accumulator; the tasks cannot. Tasks can only write to the accumulator.

See also

- [Spark SQL with HDInsight](#)
- [Spark Scenarios - Build high-speed scalable data pipelines](#)
- [Optimizing and configuring Spark Jobs for performance](#)
- [Configuring Spark settings](#)
- Submit [remote batch jobs](#) to an HDInsight Spark cluster.

Use Spark SQL with HDInsight

8/16/2017 • 5 min to read • [Edit Online](#)

SQL (Structured Query Language) is the most common and widely used language for querying and defining data. Having been developed since the 1970s, and officially ANSI-standardized in 1986, SQL has had its foothold in the industry long enough for data analysts to turn to it as a natural way think about breaking down complex problems and define data relationships. The founders of Spark sought to harness this knowledge, opening up the well-known data querying language to a wider audience of analysts who wish to work with data that lives on Hadoop Distributed File System (HDFS).

Spark SQL is that offering. It functions as an extension to Apache [Spark](#) for processing structured data, using the familiar SQL syntax. It has been part of the core distribution since Spark 1.0 (April 2014), and is a distributed SQL query engine. It also functions as a general purpose distributed data processing API. It can be used in conjunction with the Spark core API within a single application.



To use Spark SQL, first create an Azure storage account, which HDInsight uses to store data within a blob container. Alternately, you can use an Azure Data Lake Store account. Next, HDInsight will make Apache Spark available as a service in the cloud. Using this service, we can run Spark SQL statements against our stored data by using notebooks.

Spark SQL data sources

Spark SQL supports both SQL and HiveQL as query languages. Its capabilities include binding in Python, Scala, and Java. With it, you can query data stored in many locations, such as external databases, structured data files (example: JSON), and Hive tables.

There are many built-in data sources that come prepackaged with the Spark distribution. The Data Sources API provides an integration point for external developers to add support for custom data sources. Visit [spark-packages.org](#) to find custom data sources contributed by the community.

Built-in



External



SparkSession

The entry point into all functionality in Spark SQL is the `sparkSession` class. Use the `SparkSession.builder()` method to create a basic `sparkSession` object:

```
import org.apache.spark.sql.SparkSession

val spark = SparkSession
  .builder()
  .appName("Spark SQL basic example")
  .config("spark.some.config.option", "some-value")
  .getOrCreate()

// For implicit conversions like converting RDDs to DataFrames
import spark.implicits._
```

`SparkSession` is new to Spark 2.0. In earlier versions, you have the choice between a `SQLContext` and a `HiveContext`. These two older contexts are kept for backward compatibility.

`SparkSession` also allows you to:

- Write using the more complete HiveQL parser
- Gain access to Hive UDFs
- Read data from Hive Tables

You do not need to have an existing Hive setup to write HiveQL queries with `SparkSession`.

Working with DataFrames

A DataFrame is a distributed collection of data organized into named columns. It is conceptually equivalent to a table in a relational database or a data frame in R/Python, but with richer optimizations under the hood.

DataFrames can be constructed from a wide array of sources such as: structured data files, tables in Hive, external databases, or existing RDDs. The DataFrame API is available in Scala, Java, and Python.

RDDs are a collection of opaque objects (such as internal structures unknown to Spark).

User	User	User
User	User	User

DataFrames is a collection of objects with schema that are known to Spark SQL..

Name	Age	Title
Name	Age	Title
Name	Age	Title
Name	Age	Title

Once built, DataFrames provide a domain-specific language for distributed data manipulation. You can also incorporate Spark SQL while working with DataFrames. Data scientists are employing increasingly sophisticated techniques that go beyond joins and aggregations. To support this, DataFrames can be used directly in the MLlib machine learning pipeline API. In addition, programs can run arbitrarily complex user functions on DataFrames.

Creating DataFrames from data sources

A Spark data source can read in data to create DataFrames, which has a schema that Spark understands. Examples include: JSON files, JDBC source, Parquet, and Hive tables.

```
>>> val df = spark.read.json("somejsonfile.json")           //from JSON file
>>> val df = spark.read.parquet("someparquetsource")      //from a parquet file
>>> val df = spark.read
       .format("jdbc")
       .option("url", "UrlToConnect")
       .option("dbtable", "schema.tablename")
       .option("user", "username")
       .option("password", "pass")
       .load()                                                 //from JDBC source
>>> val spark = SparkSession
       .builder()
       .appName("Spark Hive Application")
       .config("spark.sql.warehouse.dir", "spark-warehouse")
       .enableHiveSupport()
       .getOrCreate()
>>> import spark.implicits._
>>> import spark.sql
>>> val sqlDF = sql("SELECT key, value FROM src WHERE key < 10 ORDER BY key") //from a Hive Table
```

The DataFrame interface makes it possible to operate on a variety of data sources. A DataFrame can be operated on as a normal RDD and/or registered as a temporary table. Registering a DataFrame as a table allows you to run SQL queries over its data. Below is a list of the general methods for loading and saving data using the Spark data sources, with some specific options that are available for the built-in data sources.

Read the [Spark documentation](#) for the latest information on the various supported data sources and how to use them.

Creating DataFrames from RDDs

You can create DataFrames from existing RDDs in two ways:

1. **Use reflection:** Infer the schema of an RDD that contains specific types of objects. This approach leads to more concise code, and enables you to construct DataFrames when the columns and their types are not known until runtime. One thing to watch out for, is that the data type for each field is inferred by evaluating the first record. Because of this, it's important to make sure the first record has no missing values and is a good representation of the dataset.
2. **Specify the schema programmatically:** Enables you to construct a schema then apply it to an existing RDD. Because this method is more verbose, it works well when you already know the schema while writing your Spark application. Despite the extra level of effort, explicitly defining a schema for your DataFrame in code is the preferred method. To create a schema, create a `StructType` object containing `StructField` objects with the field and data type information. This is supplied when creating the DataFrame.

DataFrame operations

DataFrames provide a domain-specific language for structured data manipulation in Scala, Java, and Python.

Below is a sample of various operations you can use:

```
>>> val df = spark.read.json("somejsonfile.json")
>>> df.show()                                // Show the contents of the DataFrame
>>> df.printSchema()                          // Print the schema in a tree format
>>> df.select("name").show()                  // Select and show the name columns
>>> df.select(df("name"),df ("age") +1).show() // Select all but increment the age by 1
>>> df.filter(df("age") > 21).show()          // Select people older than 21
>>> df.groupBy("age").count().show()           // Count people by age
```

Through certain DataFrame operations, you can use `SQLContext` to understand the construction and demographics of your data.

OPTION	DESCRIPTION
<code>df.show()</code>	This operation shows the contents of the data you have selected
<code>df.printSchema()</code>	This operation prints your schema (table) in a tree format
<code>df.select("name").show()</code>	This operation will select and display the name you have selected for the columns you've specified
<code>df.select(dr("name"),df("age") + 1).show()</code>	This operation example will select all your data as well as add one to each of the ages shown
<code>df.filter(df("age") > a21.show()</code>	This operation example will select all data that shows that the data is greater than 21

OPTION	DESCRIPTION
<pre>df.groupBy("age").count().show()</pre>	This operation example will count the data you've selected by age and show you only the results that meet the criteria you've presented

See also

- [Spark with HDInsight](#)
- [Spark Scenarios - Build high-speed scalable data pipelines](#)
- [Optimizing and configuring Spark Jobs for performance](#)
- [Configuring Spark settings](#)

Run Spark from the Shell

8/15/2017 • 2 min to read • [Edit Online](#)

To run Spark interactively, you can use the Spark Shell. This is useful for development and debugging. There are shells for each language supported by Spark, but they all provide the same capability- a REPL (read, execute, print loop) environment for running Spark commands one at a time and viewing the results.

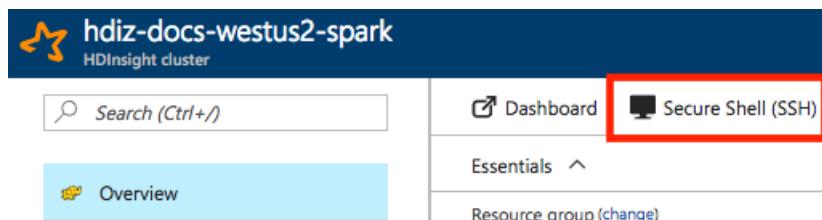
Get to a Shell via SSH

You access the Spark Shell on HDInsight by connecting to the primary head node of the cluster via SSH. The standard syntax is as follows:

```
ssh <sshusername>@<clustername>-ssh.azurehdinsight.net
```

You can get easily retrieve the complete command for your cluster, from the Azure Portal by following these steps:

1. Log into the [Azure Portal](#).
2. Navigate to the blade for your HDInsight Spark cluster.
3. Select Secure Shell (SSH).



4. Copy the provided SSH command and run it in the terminal of your choice.



Host name

hdiz-docs-westus2-spark-ssh.azurehdinsight.net



Linux, Unix, and OS X users

For instructions on how to use the terminal to connect to your HDInsight cluster, visit that [Azure documentation](#).

To connect to the <head> node of your cluster, open a new terminal session and enter the following:

ssh sshuser@hdiz-docs-westus2-spark-ssh.azurehdinsight.net



For details on using SSH to connect to HDInsight, see [Use SSH with HDInsight](#)

Run the Shell

Spark provides shells for Scala (spark-shell), Python (pyspark) and R (sparkR) . Within your SSH session to the head node of your HDInsight cluster, you run either

```
./bin/spark-shell
```

to launch the Scala Spark Shell or

```
./bin/pyspark
```

to launch the Python Spark Shell or

```
./bin/sparkR
```

to launch the R Spark Shell.

Within each shell you can enter Spark commands in the expected language for the shell.

SparkSession and SparkContext

By default when you run the Spark Shell, instances of SparkSession and SparkContext are automatically instantiated for you.

to access the SparkSession instance, use:

```
spark
```

to access the SparkContext instance, use:

```
sc
```

Important Shell Parameters

The Spark Shell also supports numerous command line parameters.

If you run the Spark Shell with the switch `--help` you will get the full list of commands available (note that some of these switches do not apply to instances of Spark Shell, as they may only apply to `spark-submit` which the Spark Shell wraps).

SWITCH	DESCRIPTION	EXAMPLE
<code>--master MASTER_URL</code>	Used to specify the master URL. In HDInsight this should always use the value <code>yarn</code> .	<code>--master yarn</code>
<code>--jars JAR_LIST</code>	Comma-separated list of local jars to include on the driver and executor classpaths. In HDInsight, these are paths to the default filesystem in Azure Storage or Data Lake Store.	<code>--jars /path/to/examples.jar</code>

SWITCH	DESCRIPTION	EXAMPLE
--packages MAVEN_COORDS	Comma-separated list of maven coordinates of jars to include on the driver and executor classpaths. Will search the local maven repo, then maven central and any additional remote repositories given by --repositories. The format for the coordinates should be groupId:artifactId:version.	--packages "com.microsoft.azure:azure-eventhubs:0.14.0"
--py-files LIST	Comma-separated list of .zip, .egg, or .py files to place on the PYTHONPATH for Python apps.	--pyfiles "samples.py"

Next Steps

This article covered how to run the various Spark Shells available to each language support by Spark.

- See [Use Spark with HDInsight](#) for an overview of using Spark with HDInsight.
- Read [Use Spark SQL with HDInsight](#) to understand how to write applications that use DataFrames from SparkSQL.
- Review [What is Spark Structured Streaming?](#) to learn how to write application to process streaming data with Spark.

Use Zeppelin notebooks with Apache Spark cluster on Azure HDInsight

8/16/2017 • 6 min to read • [Edit Online](#)

HDInsight Spark clusters include Zeppelin notebooks that you can use to run Spark jobs. In this article, you learn how to use the Zeppelin notebook on an HDInsight cluster.

NOTE

Zeppelin notebooks are available only for Spark 1.6.3 on HDInsight 3.5 and Spark 2.1.0 on HDInsight 3.6.

Prerequisites:

- An Azure subscription. See [Get Azure free trial](#).
- An Apache Spark cluster on HDInsight. For instructions, see [Create Apache Spark clusters in Azure HDInsight](#).

Launch a Zeppelin notebook

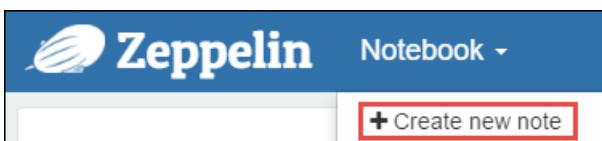
1. From the Spark cluster blade, click **Cluster Dashboard**, and then click **Zeppelin Notebook**. If prompted, enter the admin credentials for the cluster.

NOTE

You may also reach the Zeppelin Notebook for your cluster by opening the following URL in your browser. Replace **CLUSTERNAME** with the name of your cluster:

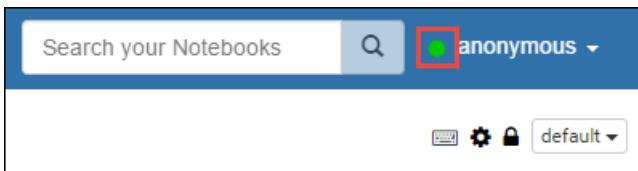
```
https://CLUSTERNAME.azurehdinsight.net/zeppelin
```

2. Create a new notebook. From the header pane, click **Notebook**, and then click **Create New Note**.



Enter a name for the notebook, and then click **Create Note**.

3. Also, make sure the notebook header shows a connected status. It is denoted by a green dot in the top-right corner.



4. Load sample data into a temporary table. When you create a Spark cluster in HDInsight, the sample data file, **hvac.csv**, is copied to the associated storage account under **\HdiSamples\SensorSampleData\hvac**.

In the empty paragraph that is created by default in the new notebook, paste the following snippet.

```
%livy.spark
//The above magic instructs Zeppelin to use the Livy Scala interpreter

// Create an RDD using the default Spark context, sc
val hvacText = sc.textFile("wasb:///HdiSamples/HdiSamples/SensorSampleData/hvac/HVAC.csv")

// Define a schema
case class Hvac(date: String, time: String, targettemp: Integer, actualtemp: Integer, buildingID: String)

// Map the values in the .csv file to the schema
val hvac = hvacText.map(s => s.split(",")).filter(s => s(0) != "Date").map(
    s => Hvac(s(0),
              s(1),
              s(2).toInt,
              s(3).toInt,
              s(6)
            )
  ).toDF()

// Register as a temporary table called "hvac"
hvac.registerTempTable("hvac")
```

Press **SHIFT + ENTER** or click the **Play** button for the paragraph to run the snippet. The status on the right-corner of the paragraph should progress from READY, PENDING, RUNNING to FINISHED. The output shows up at the bottom of the same paragraph. The screenshot looks like the following:

```
%livy.spark
//The above magic instructs Zeppelin to use the Livy interpreter

// Create an RDD using the default Spark context, sc
val hvacText = sc.textFile("wasbs:///HdiSamples/HdiSamples/SensorSampleData/hvac/HVAC.csv")

// Define a schema
case class Hvac(date: String, time: String, targettemp: Integer, actualtemp: Integer, buildingID: String)

// Map the values in the .csv file to the schema
val hvac = hvacText.map(s => s.split(",")).filter(s => s(0) != "Date").map(
    s => Hvac(s(0),
              s(1),
              s(2).toInt,
              s(3).toInt,
              s(6)
            )
  ).toDF()

// Register as a temporary table called "hvac"
hvac.registerTempTable("hvac")

hvacText: org.apache.spark.rdd.RDD[String] = wasbs:///HdiSamples/HdiSamples/SensorSampleData/hvac/HVAC.csv MapPartitionsRDD[15] at textFile at <console>:20
defined class Hvac
hvac: org.apache.spark.sql.DataFrame = [date: string, time: string, targettemp: int, actualtemp: int, buildingID: string]
```

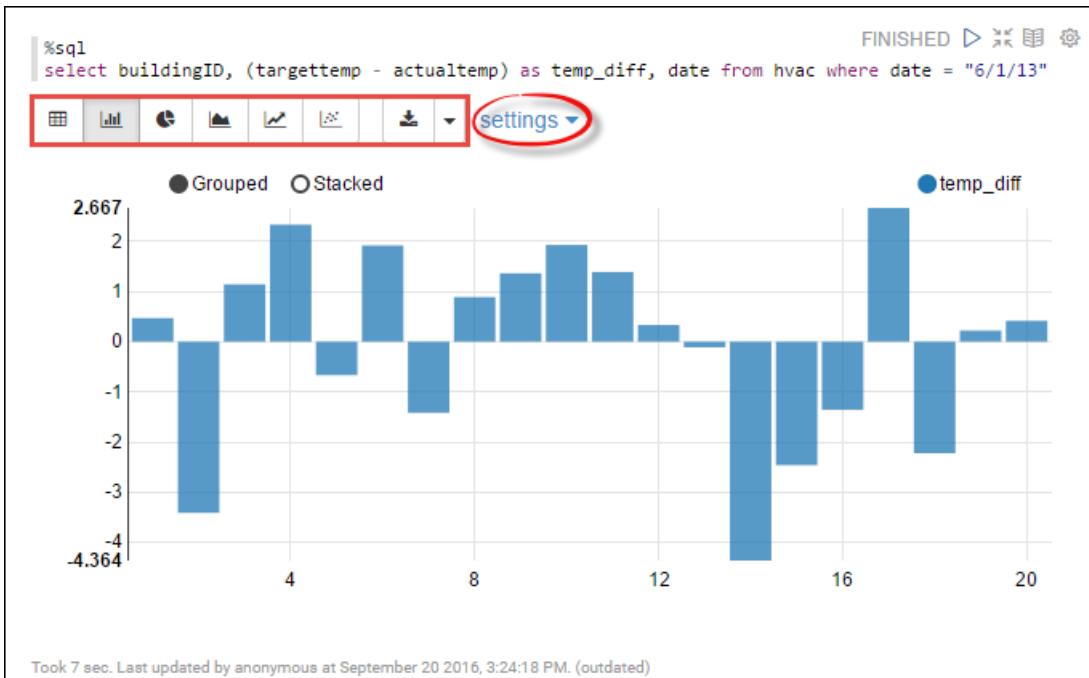
You can also provide a title to each paragraph. From the right-hand corner, click the **Settings** icon, and then click **Show title**.

5. You can now run Spark SQL statements on the **hvac** table. Paste the following query in a new paragraph. The query retrieves the building ID and the difference between the target and actual temperatures for each building on a given date. Press **SHIFT + ENTER**.

```
%sql
select buildingID, (targettemp - actualtemp) as temp_diff, date from hvac where date = "6/1/13"
```

The **%sql** statement at the beginning tells the notebook to use the Livy Scala interpreter.

The following screenshot shows the output.

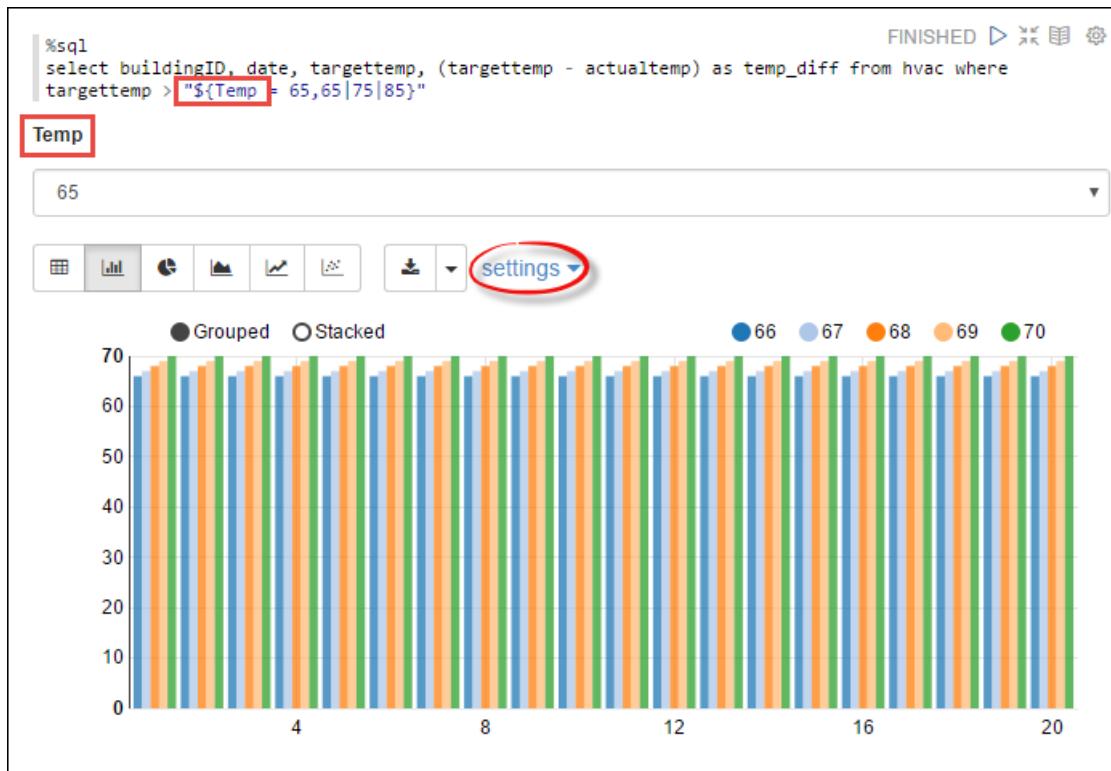


Click the display options (highlighted in rectangle) to switch between different representations for the same output. Click **Settings** to choose what constitutes the key and values in the output. The screen capture above uses **buildingID** as the key and the average of **temp_diff** as the value.

6. You can also run Spark SQL statements using variables in the query. The next snippet shows how to define a variable, **Temp**, in the query with the possible values you want to query with. When you first run the query, a drop-down is automatically populated with the values you specified for the variable.

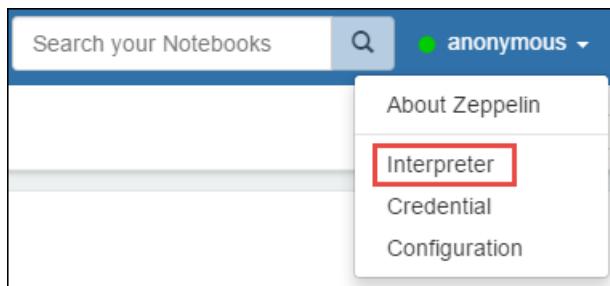
```
%sql  
select buildingID, date, targettemp, (targettemp - actualtemp) as temp_diff from hvac where targettemp  
> "${Temp = 65,65|75|85}"
```

Paste this snippet in a new paragraph and press **SHIFT + ENTER**. The following screenshot shows the output.



For subsequent queries, you can select a new value from the drop-down and run the query again. Click **Settings** to choose what constitutes the key and values in the output. The screen capture above uses **buildingID** as the key, the average of **temp_diff** as the value, and **targettemp** as the group.

7. Restart the Livy interpreter to exit the application. To do so, open interpreter settings by clicking the logged in user name from the top-right corner, and then click **Interpreter**.



8. Scroll to Livy interpreter settings and then click **Restart**.

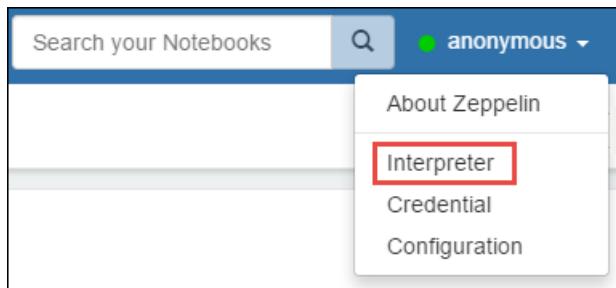


How do I use external packages with the notebook?

You can configure the Zeppelin notebook in Apache Spark cluster on HDInsight (Linux) to use external, community-contributed packages that are not included out-of-the-box in the cluster. You can search the [Maven repository](#) for the complete list of packages that are available. You can also get a list of available packages from other sources. For example, a complete list of community-contributed packages is available at [Spark Packages](#).

In this article, you will see how to use the [spark-csv](#) package with the Jupyter notebook.

1. Open interpreter settings. From the top-right corner, click the logged in user name, and then click **Interpreter**.



2. Scroll to Livy interpreter settings and then click **Edit**.

Livy %livy, %livy.pyspark, %livy.sql

edit restart remove

Option

shared ▾ Interpreter for note

3. Add a new key, called **livy.spark.jars.packages** and set its value in the format `groupId:id:version`. So, if you want to use the [spark-csv](#) package, you must set the value of the key to `com.databricks:spark-csv_2.10:1.4.0`.

zeppelin.livy.url http://hn0-02f3b0.linchan-2v5-2016091
60951-ssh.d3.internal.cloudapp.net:899
8

livy.spark.jars.packages com.databricks:spark-csv_2.10:1.4.0 +

Dependencies

These dependencies will be added to classpath when interpreter process starts.

artifact	exclude	action
groupId:artifactId:version or local file pat	(Optional) comma separated grou	+

Save Cancel

Click **Save** and then restart the Livy interpreter.

4. **Tip:** If you want to understand how to arrive at the value of the key entered above, here's how.

- Locate the package in the Maven Repository. For this tutorial, we used [spark-csv](#).
- From the repository, gather the values for **GroupId**, **ArtifactId**, and **Version**.

Project Information

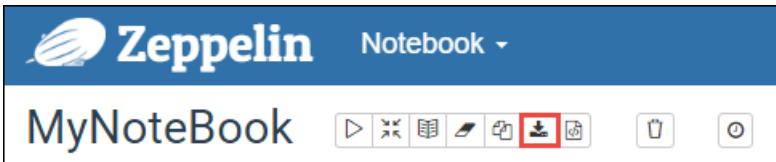
GroupId:	com.databricks
ArtifactId:	spark-csv_2.10
Version:	1.4.0

- Concatenate the three values, separated by a colon (:).

com.databricks:spark-csv_2.10:1.4.0

Where are the Zeppelin notebooks saved?

The Zeppelin notebooks are saved to the cluster headnodes. So, if you delete the cluster, the notebooks will be deleted as well. If you want to preserve your notebooks for later use on other clusters, you must export them after you have finished running the jobs. To export a notebook, click the **Export** icon as shown in the image below.



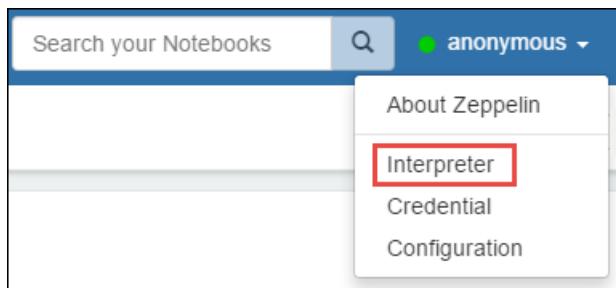
This saves the notebook as a JSON file in your download location.

Livy session management

When you run the first code paragraph in your Zeppelin notebook, a new Livy session is created in your HDInsight Spark cluster. This session is shared across all Zeppelin notebooks that you subsequently create. If for some reason the Livy session is killed (cluster reboot, etc.), you will not be able to run jobs from the Zeppelin notebook.

In such a case, you must perform the following steps before you can start running jobs from a Zeppelin notebook.

1. Restart the Livy interpreter from the Zeppelin notebook. To do so, open interpreter settings by clicking the logged in user name from the top-right corner, and then click **Interpreter**.



2. Scroll to Livy interpreter settings and then click **Restart**.



3. Run a code cell from an existing Zeppelin notebook. This creates a new Livy session in the HDInsight cluster.

See also

- [Overview: Apache Spark on Azure HDInsight](#)

Scenarios

- [Spark with BI: Perform interactive data analysis using Spark in HDInsight with BI tools](#)
- [Spark with Machine Learning: Use Spark in HDInsight for analyzing building temperature using HVAC data](#)
- [Spark with Machine Learning: Use Spark in HDInsight to predict food inspection results](#)
- [Spark Streaming: Use Spark in HDInsight for building real-time streaming applications](#)
- [Website log analysis using Spark in HDInsight](#)

Create and run applications

- [Create a standalone application using Scala](#)

- Run jobs remotely on a Spark cluster using Livy

Tools and extensions

- Use HDInsight Tools Plugin for IntelliJ IDEA to create and submit Spark Scala applications
- Use HDInsight Tools Plugin for IntelliJ IDEA to debug Spark applications remotely
- Kernels available for Jupyter notebook in Spark cluster for HDInsight
- Use external packages with Jupyter notebooks
- Install Jupyter on your computer and connect to an HDInsight Spark cluster

Manage resources

- Manage resources for the Apache Spark cluster in Azure HDInsight
- Track and debug jobs running on an Apache Spark cluster in HDInsight

Kernels for Jupyter notebook on Spark clusters in Azure HDInsight

8/16/2017 • 8 min to read • [Edit Online](#)

HDInsight Spark clusters provide kernels that you can use with the Jupyter notebook on Spark for testing your applications. A kernel is a program that runs and interprets your code. The three kernels are:

- **PySpark** - for applications written in Python2
- **PySpark3** - for applications written in Python3
- **Spark** - for applications written in Scala

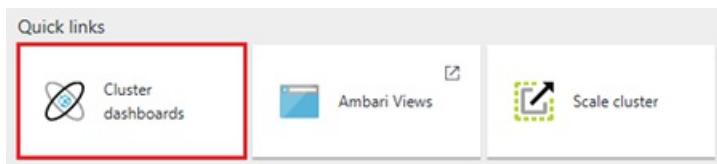
In this article, you learn how to use these kernels and the benefits of using them.

Prerequisites

- An Apache Spark cluster in HDInsight. For instructions, see [Create Apache Spark clusters in Azure HDInsight](#).

Create a Jupyter notebook on Spark HDInsight

1. From the [Azure portal](#), open your cluster. See [List and show clusters](#) for the instructions. The cluster is opened in a new portal blade.
2. From the **Quick links** section, click **Cluster dashboards** to open the **Cluster dashboards** blade. If you don't see **Quick Links**, click **Overview** from the left menu on the blade.



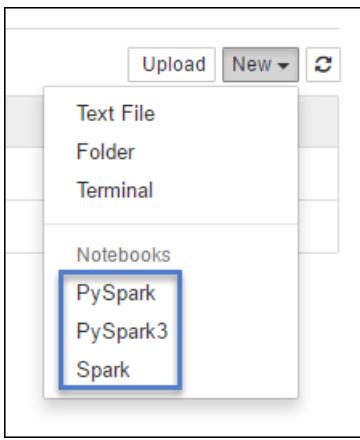
3. Click **Jupyter Notebook**. If prompted, enter the admin credentials for the cluster.

NOTE

You may also reach the Jupyter notebook on Spark cluster by opening the following URL in your browser. Replace **CLUSTERNAME** with the name of your cluster:

```
https://CLUSTERNAME.azurehdinsight.net/jupyter
```

4. Click **New**, and then click either **Pyspark**, **PySpark3**, or **Spark** to create a notebook. Use the Spark kernel for Scala applications, PySpark kernel for Python2 applications, and PySpark3 kernel for Python3 applications.



5. A notebook opens with the kernel you selected.

Benefits of using the kernels

Here are a few benefits of using the new kernels with Jupyter notebook on Spark HDInsight clusters.

- **Preset contexts.** With **PySpark**, **PySpark3**, or the **Spark** kernels, you do not need to set the Spark or Hive contexts explicitly before you start working with your applications. These are available by default. These contexts are:
 - **sc** - for Spark context
 - **sqlContext** - for Hive context

So, you don't have to run statements like the following to set the contexts:

```
sc = SparkContext('yarn-client') sqlContext = HiveContext(sc)
```

Instead, you can directly use the preset contexts in your application.

- **Cell magics.** The PySpark kernel provides some predefined "magics", which are special commands that you can call with `%%` (for example, `%%MAGIC`). The magic command must be the first word in a code cell and allow for multiple lines of content. The magic word should be the first word in the cell. Adding anything before the magic, even comments, causes an error. For more information on magics, see [here](#).

The following table lists the different magics available through the kernels.

MAGIC	EXAMPLE	DESCRIPTION
help	<code>%%help</code>	Generates a table of all the available magics with example and description
info	<code>%%info</code>	Outputs session information for the current Livy endpoint
configure	<code>%%configure -f</code> <code>{"executorMemory": "1000M",</code> <code>"executorCores": 4 }</code>	Configures the parameters for creating a session. The force flag (-f) is mandatory if a session has already been created, which ensures that the session is dropped and recreated. Look at Livy's POST /sessions Request Body for a list of valid parameters. Parameters must be passed in as a JSON string and must be on the next line after the magic, as shown in the example column.

MAGIC	EXAMPLE	DESCRIPTION
sql	<pre>%%sql -o <variable name> SHOW TABLES</pre>	Executes a Hive query against the sqlContext. If the <code>-o</code> parameter is passed, the result of the query is persisted in the %local Python context as a Pandas dataframe.
local	<pre>%%local a=1</pre>	All the code in subsequent lines is executed locally. Code must be valid Python2 code even irrespective of the kernel you are using. So, even if you selected PySpark3 or Spark kernels while creating the notebook, if you use the <code>%%local</code> magic in a cell, that cell must only have valid Python2 code..
logs	<pre>%%logs</pre>	Outputs the logs for the current Livy session.
delete	<pre>%%delete -f -s <session number></pre>	Deletes a specific session of the current Livy endpoint. Note that you cannot delete the session that is initiated for the kernel itself.
cleanup	<pre>%%cleanup -f</pre>	Deletes all the sessions for the current Livy endpoint, including this notebook's session. The force flag <code>-f</code> is mandatory.

NOTE

In addition to the magics added by the PySpark kernel, you can also use the [built-in IPython magics](#), including `%%sh`. You can use the `%%sh` magic to run scripts and block of code on the cluster headnode.

- **Auto visualization.** The **Pyspark** kernel automatically visualizes the output of Hive and SQL queries. You can choose between several different types of visualizations including Table, Pie, Line, Area, Bar.

Parameters supported with the `%%sql` magic

The `%%sql` magic supports different parameters that you can use to control the kind of output that you receive when you run queries. The following table lists the output.

PARAMETER	EXAMPLE	DESCRIPTION
<code>-o</code>	<pre>-o <VARIABLE NAME></pre>	Use this parameter to persist the result of the query, in the %local Python context, as a Pandas dataframe. The name of the dataframe variable is the variable name you specify.

PARAMETER	EXAMPLE	DESCRIPTION
-q	-q	Use this to turn off visualizations for the cell. If you don't want to auto-visualize the content of a cell and just want to capture it as a dataframe, then use -q -o <VARIABLE>. If you want to turn off visualizations without capturing the results (for example, for running a SQL query, like a CREATE TABLE statement), use -q without specifying a -o argument.
-m	-m <METHOD>	Where METHOD is either take or sample (default is take). If the method is take , the kernel picks elements from the top of the result data set specified by MAXROWS (described later in this table). If the method is sample , the kernel randomly samples elements of the data set according to -r parameter, described next in this table.
-r	-r <FRACTION>	Here FRACTION is a floating-point number between 0.0 and 1.0. If the sample method for the SQL query is sample , then the kernel randomly samples the specified fraction of the elements of the result set for you. For example, if you run a SQL query with the arguments -m sample -r 0.01, then 1% of the result rows are randomly sampled.
-n	-n <MAXROWS>	MAXROWS is an integer value. The kernel limits the number of output rows to MAXROWS . If MAXROWS is a negative number such as -1, then the number of rows in the result set is not limited.

Example:

```
%%sql -q -m sample -r 0.1 -n 500 -o query2
SELECT * FROM hivesamplatable
```

The statement above does the following:

- Selects all records from **hivesamplatable**.
- Because we use -q, it turns off auto-visualization.
- Because we use -m sample -r 0.1 -n 500 it randomly samples 10% of the rows in the hivesamplatable and limits the size of the result set to 500 rows.
- Finally, because we used -o query2 it also saves the output into a dataframe called **query2**.

Considerations while using the new kernels

Whichever kernel you use, leaving the notebooks running consumes the cluster resources. With these kernels, because the contexts are preset, simply exiting the notebooks does not kill the context and hence the cluster

resources continue to be in use. A good practice is to use the **Close and Halt** option from the notebook's **File** menu when you are finished using the notebook, which kills the context and then exits the notebook.

Show me some examples

When you open a Jupyter notebook, you see two folders available at the root level.

- The **PySpark** folder has sample notebooks that use the new **Python** kernel.
- The **Scala** folder has sample notebooks that use the new **Spark** kernel.

You can open the **00 - [READ ME FIRST] Spark Magic Kernel Features** notebook from the **PySpark** or **Spark** folder to learn about the different magics available. You can also use the other sample notebooks available under the two folders to learn how to achieve different scenarios using Jupyter notebooks with HDInsight Spark clusters.

Where are the notebooks stored?

Jupyter notebooks are saved to the storage account associated with the cluster under the **/HdiNotebooks** folder. Notebooks, text files, and folders that you create from within Jupyter are accessible from the storage account. For example, if you use Jupyter to create a folder **myfolder** and a notebook **myfolder/mynotebook.ipynb**, you can access that notebook at `/HdiNotebooks/myfolder/mynotebook.ipynb` within the storage account. The reverse is also true, that is, if you upload a notebook directly to your storage account at `/HdiNotebooks/mynotebook1.ipynb`, the notebook is visible from Jupyter as well. Notebooks remain in the storage account even after the cluster is deleted.

The way notebooks are saved to the storage account is compatible with HDFS. So, if you SSH into the cluster you can use file management commands as shown in the following snippet:

```
hdfs dfs -ls /HdiNotebooks          # List everything at the root directory – everything  
in this directory is visible to Jupyter from the home page  
hdfs dfs -copyToLocal /HdiNotebooks    # Download the contents of the HdiNotebooks folder  
hdfs dfs -copyFromLocal example.ipynb /HdiNotebooks  # Upload a notebook example.ipynb to the root folder so  
it's visible from Jupyter
```

In case there are issues accessing the storage account for the cluster, the notebooks are also saved on the headnode `/var/lib/jupyter`.

Supported browser

Jupyter notebooks on Spark HDInsight clusters are supported only on Google Chrome.

Feedback

The new kernels are in evolving stage and will mature over time. This could also mean that APIs could change as these kernels mature. We would appreciate any feedback that you have while using these new kernels. This is useful in shaping the final release of these kernels. You can leave your comments/feedback under the **Comments** section at the bottom of this article.

See also

- [Overview: Apache Spark on Azure HDInsight](#)

Scenarios

- [Spark with BI: Perform interactive data analysis using Spark in HDInsight with BI tools](#)
- [Spark with Machine Learning: Use Spark in HDInsight for analyzing building temperature using HVAC data](#)

- Spark with Machine Learning: Use Spark in HDInsight to predict food inspection results
- Spark Streaming: Use Spark in HDInsight for building real-time streaming applications
- Website log analysis using Spark in HDInsight

Create and run applications

- Create a standalone application using Scala
- Run jobs remotely on a Spark cluster using Livy

Tools and extensions

- Use HDInsight Tools Plugin for IntelliJ IDEA to create and submit Spark Scala applications
- Use HDInsight Tools Plugin for IntelliJ IDEA to debug Spark applications remotely
- Use Zeppelin notebooks with a Spark cluster on HDInsight
- Use external packages with Jupyter notebooks
- Install Jupyter on your computer and connect to an HDInsight Spark cluster

Manage resources

- Manage resources for the Apache Spark cluster in Azure HDInsight
- Track and debug jobs running on an Apache Spark cluster in HDInsight

Use external packages with Jupyter notebooks in Apache Spark clusters on HDInsight

8/16/2017 • 3 min to read • [Edit Online](#)

Learn how to configure a Jupyter notebook in Apache Spark cluster on HDInsight to use external, community-contributed **maven** packages that are not included out-of-the-box in the cluster.

You can search the [Maven repository](#) for the complete list of packages that are available. You can also get a list of available packages from other sources. For example, a complete list of community-contributed packages is available at [Spark Packages](#).

In this article, you will learn how to use the [spark-csv](#) package with the Jupyter notebook.

Prerequisites

You must have the following:

- An Apache Spark cluster on HDInsight. For instructions, see [Create Apache Spark clusters in Azure HDInsight](#).

Use external packages with Jupyter notebooks

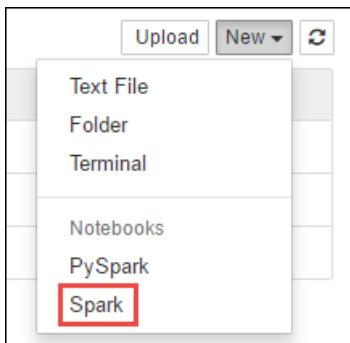
1. From the [Azure Portal](#), from the startboard, click the tile for your Spark cluster (if you pinned it to the startboard). You can also navigate to your cluster under **Browse All > HDInsight Clusters**.
2. From the Spark cluster blade, click **Quick Links**, and then from the **Cluster Dashboard** blade, click **Jupyter Notebook**. If prompted, enter the admin credentials for the cluster.

NOTE

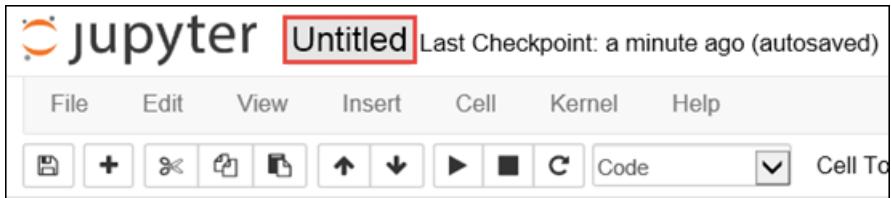
You may also reach the Jupyter Notebook for your cluster by opening the following URL in your browser. Replace **CLUSTERNAME** with the name of your cluster:

```
https://CLUSTERNAME.azurehdinsight.net/jupyter
```

3. Create a new notebook. Click **New**, and then click **Spark**.



4. A new notebook is created and opened with the name Untitled.pynb. Click the notebook name at the top, and enter a friendly name.



5. You will use the `%%configure` magic to configure the notebook to use an external package. In notebooks that use external packages, make sure you call the `%%configure` magic in the first code cell. This ensures that the kernel is configured to use the package before the session starts.

IMPORTANT

If you forget to configure the kernel in the first cell, you can use the `%%configure` with the `-f` parameter, but that will restart the session and all progress will be lost.

HDINSIGHT VERSION	COMMAND
For HDInsight 3.3 and HDInsight 3.4	<pre>%%configure { "packages": ["com.databricks:spark-csv_2.10:1.4.0"] }</pre>
For HDInsight 3.5	<pre>%%configure { "conf": { "spark.jars.packages": "com.databricks:spark-csv_2.10:1.4.0" } }</pre>

6. The snippet above expects the maven coordinates for the external package in Maven Central Repository. In this snippet, `com.databricks:spark-csv_2.10:1.4.0` is the maven coordinate for **spark-csv** package. Here's how you construct the coordinates for a package.

- Locate the package in the Maven Repository. For this tutorial, we use [spark-csv](#).
- From the repository, gather the values for **GroupId**, **ArtifactId**, and **Version**. Make sure that the values you gather match your cluster. In this case, we are using a Scala 2.10 and Spark 1.4.0 package, but you may need to select different versions for the appropriate Scala or Spark version in your cluster. You can find out the Scala version on your cluster by running `scala.util.Properties.versionString` on the Spark Jupyter kernel or on Spark submit. You can find out the Spark version on your cluster by running `sc.version` on Jupyter notebooks.

Project Information

GroupId:	com.databricks
ArtifactId:	spark-csv_2.10
Version:	1.4.0

- Concatenate the three values, separated by a colon (:).

```
com.databricks:spark-csv_2.10:1.4.0
```

7. Run the code cell with the `%%configure` magic. This will configure the underlying Livy session to use the package you provided. In the subsequent cells in the notebook, you can now use the package, as shown below.

```
val df = sqlContext.read.format("com.databricks.spark.csv").  
option("header", "true").  
option("inferSchema", "true").  
load("wasb:///HdiSamples/HdiSamples/SensorSampleData/hvac/HVAC.csv")
```

8. You can then run the snippets, like shown below, to view the data from the dataframe you created in the previous step.

```
df.show()  
  
df.select("Time").count()
```

See also

- [Overview: Apache Spark on Azure HDInsight](#)

Scenarios

- [Spark with BI: Perform interactive data analysis using Spark in HDInsight with BI tools](#)
- [Spark with Machine Learning: Use Spark in HDInsight for analyzing building temperature using HVAC data](#)
- [Spark with Machine Learning: Use Spark in HDInsight to predict food inspection results](#)
- [Spark Streaming: Use Spark in HDInsight for building real-time streaming applications](#)
- [Website log analysis using Spark in HDInsight](#)

Create and run applications

- [Create a standalone application using Scala](#)
- [Run jobs remotely on a Spark cluster using Livy](#)

Tools and extensions

- [Use external python packages with Jupyter notebooks in Apache Spark clusters on HDInsight Linux](#)
- [Use HDInsight Tools Plugin for IntelliJ IDEA to create and submit Spark Scala applications](#)
- [Use HDInsight Tools Plugin for IntelliJ IDEA to debug Spark applications remotely](#)
- [Use Zeppelin notebooks with a Spark cluster on HDInsight](#)
- [Kernels available for Jupyter notebook in Spark cluster for HDInsight](#)
- [Install Jupyter on your computer and connect to an HDInsight Spark cluster](#)

Manage resources

- [Manage resources for the Apache Spark cluster in Azure HDInsight](#)
- [Track and debug jobs running on an Apache Spark cluster in HDInsight](#)

Use Script Action to install external Python packages for Jupyter notebooks in Apache Spark clusters on HDInsight

8/16/2017 • 2 min to read • [Edit Online](#)

Learn how to use Script Actions to configure an Apache Spark cluster on HDInsight (Linux) to use external, community-contributed **python** packages that are not included out-of-the-box in the cluster.

NOTE

You can also configure a Jupyter notebook by using `%%configure` magic to use external packages. For instructions, see [Use external packages with Jupyter notebooks in Apache Spark clusters on HDInsight](#).

You can search the [package index](#) for the complete list of packages that are available. You can also get a list of available packages from other sources. For example, you can install packages made available through [Anaconda](#) or [conda-forge](#).

In this article, you will learn how to install the [TensorFlow](#) package using Script Action on your cluster and use it via the Jupyter notebook.

Prerequisites

You must have the following:

- An Azure subscription. See [Get Azure free trial](#).
- An Apache Spark cluster on HDInsight. For instructions, see [Create Apache Spark clusters in Azure HDInsight](#).

NOTE

If you do not already have a Spark cluster on HDInsight Linux, you can run script actions during cluster creation. Visit the documentation on [how to use custom script actions](#).

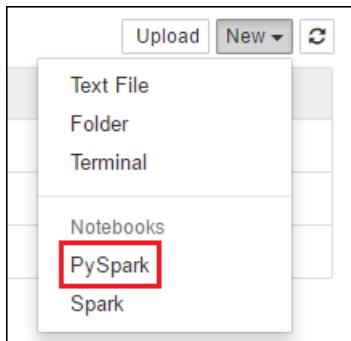
Use external packages with Jupyter notebooks

1. From the [Azure Portal](#), from the startboard, click the tile for your Spark cluster (if you pinned it to the startboard). You can also navigate to your cluster under **Browse All > HDInsight Clusters**.
2. From the Spark cluster blade, click **Script Actions** under **Usage**. Run the custom action that installs TensorFlow in the head nodes and the worker nodes. The bash script can be referenced from: <https://hdiconfigactions.blob.core.windows.net/linuxtensorflow/tensorflowinstall.sh> Visit the documentation on [how to use custom script actions](#).

NOTE

There are two python installations in the cluster. Spark will use the Anaconda python installation located at `/usr/bin/anaconda/bin`. Reference that installation in your custom actions via `/usr/bin/anaconda/bin/pip` and `/usr/bin/anaconda/bin/conda`.

3. Open a PySpark Jupyter notebook



4. A new notebook is created and opened with the name Untitled.pynb. Click the notebook name at the top, and enter a friendly name.



5. You will now `import tensorflow` and run a hello world example.

Code to copy:

```
import tensorflow as tf
hello = tf.constant('Hello, TensorFlow!')
sess = tf.Session()
print(sess.run(hello))
```

The result will look like this:

```
In [1]: import tensorflow as tf
hello = tf.constant('Hello, TensorFlow!')
sess = tf.Session()
print(sess.run(hello))
```

```
Creating SparkContext as 'sc'
Creating HiveContext as 'sqlContext'
Hello, TensorFlow!
```

See also

- [Overview: Apache Spark on Azure HDInsight](#)

Scenarios

- [Spark with BI: Perform interactive data analysis using Spark in HDInsight with BI tools](#)
- [Spark with Machine Learning: Use Spark in HDInsight for analyzing building temperature using HVAC data](#)
- [Spark with Machine Learning: Use Spark in HDInsight to predict food inspection results](#)
- [Spark Streaming: Use Spark in HDInsight for building real-time streaming applications](#)

- Website log analysis using Spark in HDInsight

Create and run applications

- Create a standalone application using Scala
- Run jobs remotely on a Spark cluster using Livy

Tools and extensions

- Use external packages with Jupyter notebooks in Apache Spark clusters on HDInsight
- Use HDInsight Tools Plugin for IntelliJ IDEA to create and submit Spark Scala applications
- Use HDInsight Tools Plugin for IntelliJ IDEA to debug Spark applications remotely
- Use Zeppelin notebooks with a Spark cluster on HDInsight
- Kernels available for Jupyter notebook in Spark cluster for HDInsight
- Install Jupyter on your computer and connect to an HDInsight Spark cluster

Manage resources

- Manage resources for the Apache Spark cluster in Azure HDInsight
- Track and debug jobs running on an Apache Spark cluster in HDInsight

Use Azure Toolkit for IntelliJ to create Spark applications for HDInsight cluster

8/16/2017 • 10 min to read • [Edit Online](#)

Use the Azure Toolkit for IntelliJ plug-in to develop Spark applications written in Scala and submit them to an HDInsight Spark cluster, directly from the IntelliJ IDE. You can use the plug-in in a few different ways:

- To develop and submit a Scala Spark application on an HDInsight Spark cluster
- To access your Azure HDInsight Spark cluster resources
- To develop and run a Scala Spark application locally

You can follow a [video](#) to create your project.

IMPORTANT

This plug-in can be used to create and submit applications only for an HDInsight Spark cluster on Linux.

Prerequisites

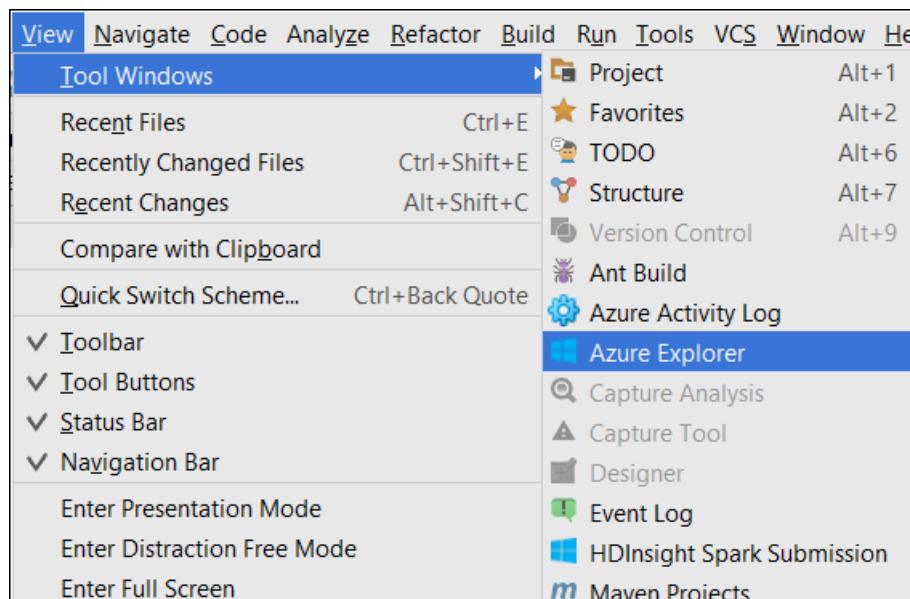
- An Apache Spark cluster on HDInsight Linux. For instructions, see [Create Apache Spark clusters in Azure HDInsight](#).
- Oracle Java Development Kit. You can install it from the [Oracle website](#).
- IntelliJ IDEA. This article uses version 2017.1. You can install it from the [JetBrains website](#).

Install Azure Toolkit for IntelliJ

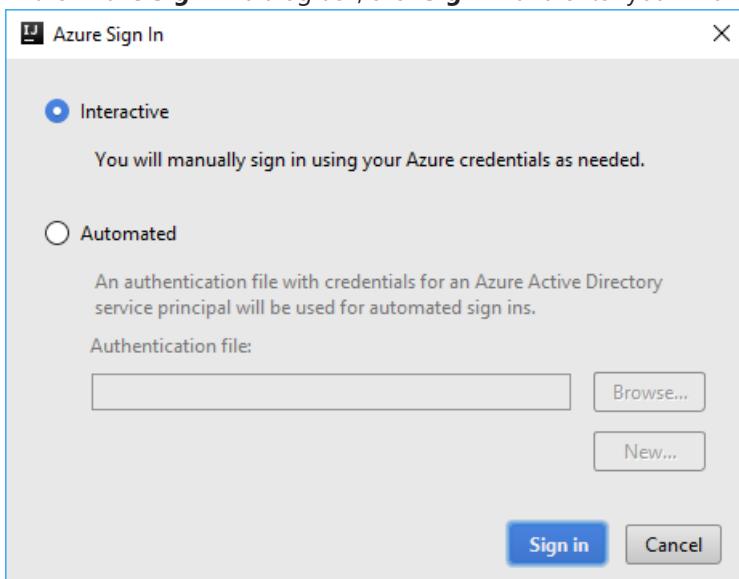
For installation instructions, see [Installing the Azure Toolkit for IntelliJ](#).

Sign in to your Azure subscription

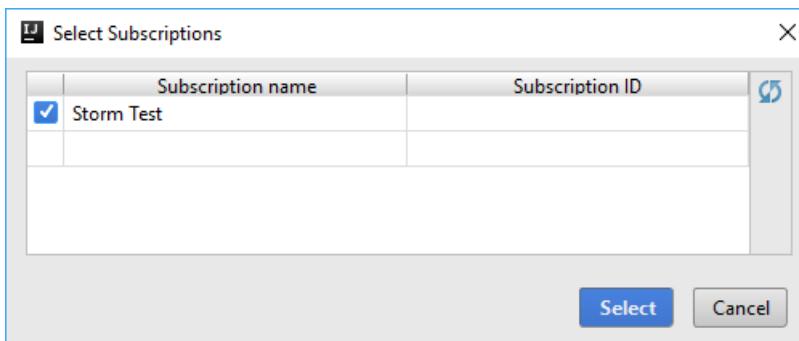
1. Start the IntelliJ IDE and open Azure Explorer. On the **View** menu, click **Tool Windows**, and then click **Azure Explorer**.



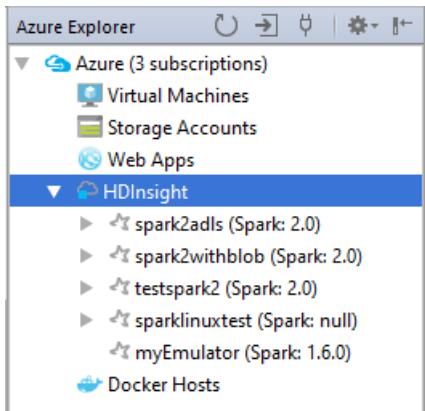
2. Right-click the **Azure** node, and then click **Sign In**.
3. In the **Azure Sign In** dialog box, click **Sign in** and enter your Azure credentials.



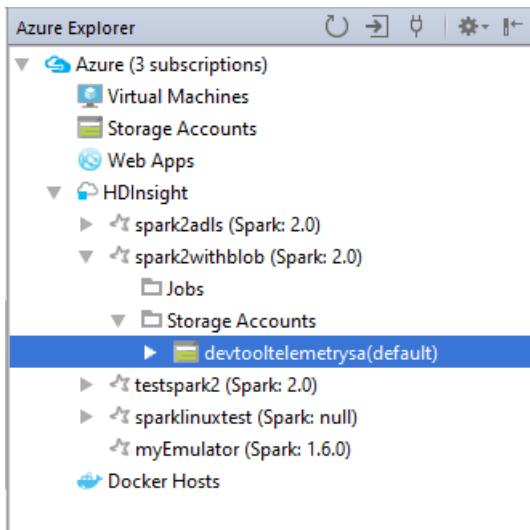
4. After you're signed in, the **Select Subscriptions** dialog box lists all the Azure subscriptions associated with the credentials. Click **Select** to close the dialog box.



5. On the **Azure Explorer** tab, expand **HDInsight** to see the HDInsight Spark clusters under your subscription.

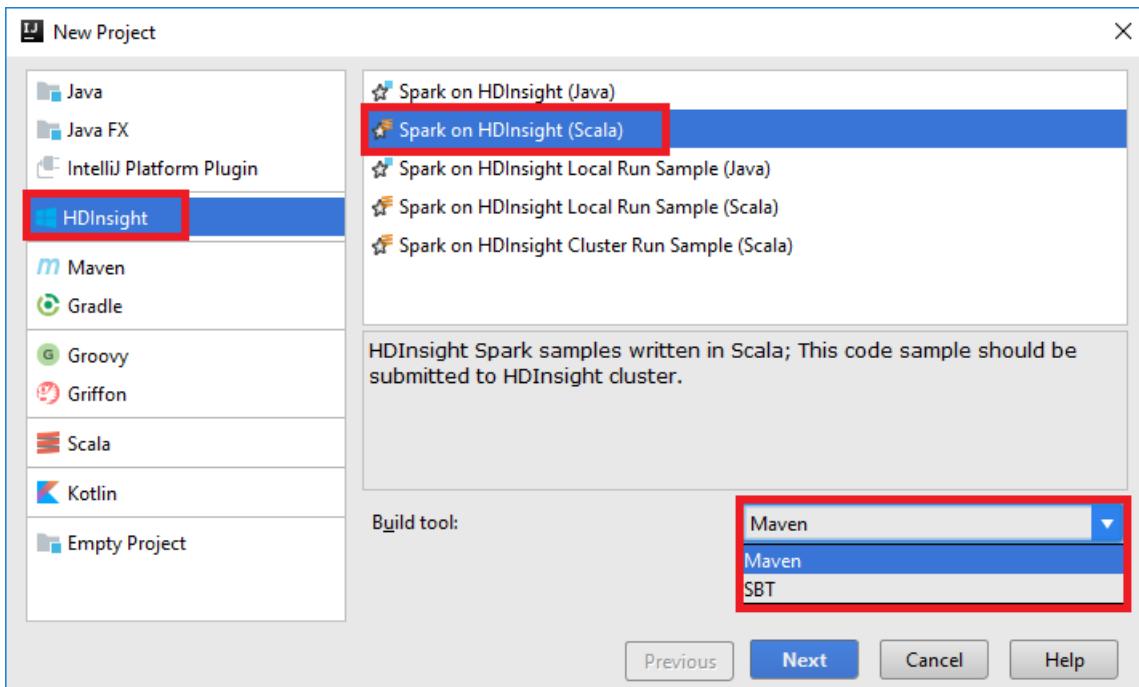


6. You can further expand a cluster name node to see the resources (for example, storage accounts) associated with the cluster.



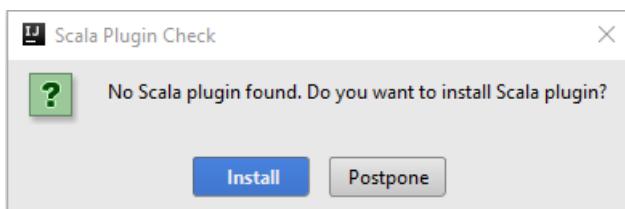
Run a Spark Scala application on an HDInsight Spark cluster

1. Start IntelliJ IDEA and create a project. In the **New Project** dialog box, make the following choices, and then click **Next**.

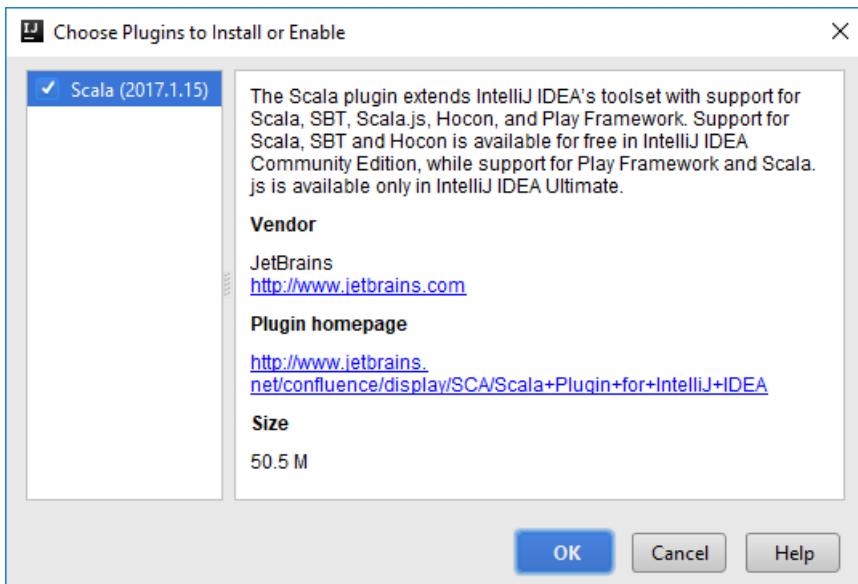


- In the left pane, select **HDInsight**.
- In the right pane, select **Spark on HDInsight (Scala)**.
- Build tool: Scala project creation wizard support Maven or SBT managing the dependencies and building for scala project. You select one according to need.

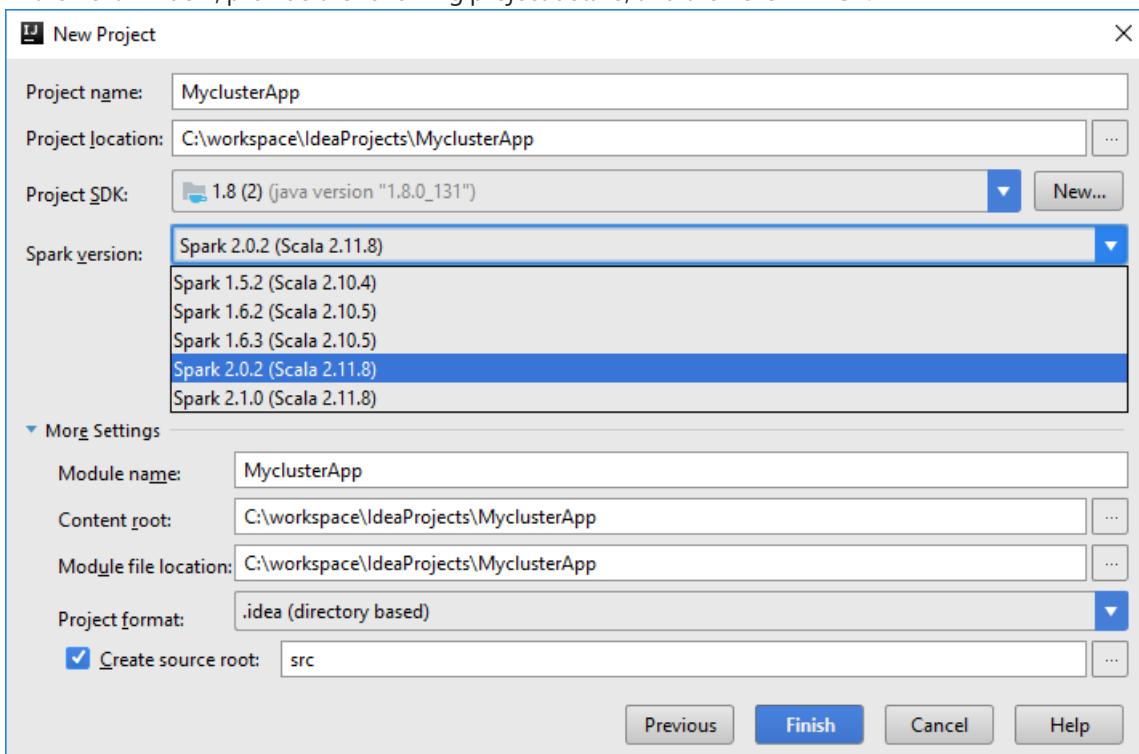
2. The Scala project creation wizard auto detects whether you installed Scala plugin or not. Click the **Install** to continue.



3. Click **OK** to download the Scala plugin. Follow the instructions to restart IntelliJ.



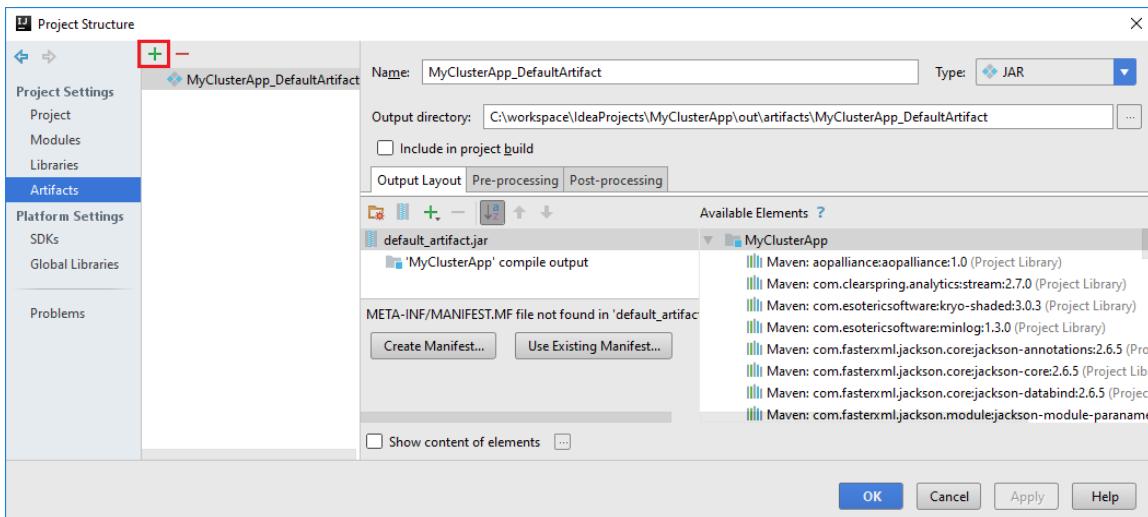
4. In the next window, provide the following project details, and then click **Finish**.



- Provide a project name and project location.
- For **Project SDK**, Use Java 1.8 for spark 2.x cluster, Java 1.7 for spark 1.x cluster.
- For **Spark Version**, Scala project creation wizard integrates proper version for Spark SDK and Scala SDK. If the spark cluster version is lower 2.0, choose spark 1.x. Otherwise, you should select spark2.x. This example uses Spark2.0.2(Scala 2.11.8).

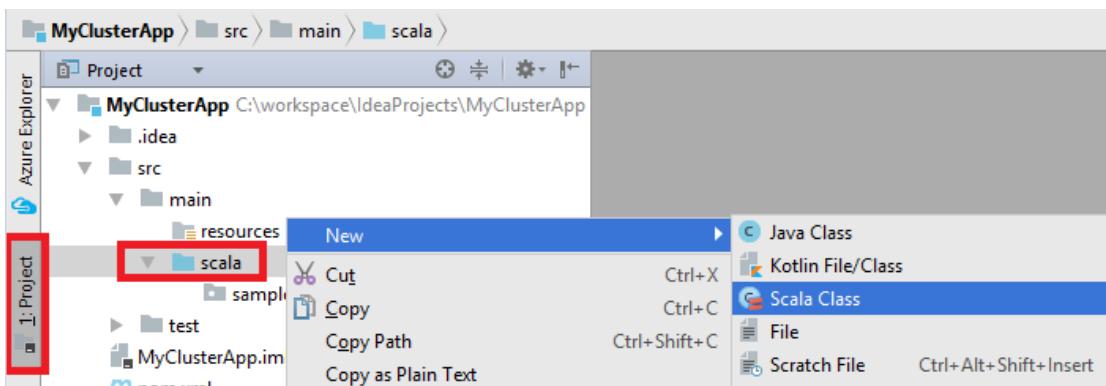
5. The Spark project automatically creates an artifact for you. To see the artifact, follow these steps:

- a. On the **File** menu, click **Project Structure**.
- b. In the **Project Structure** dialog box, click **Artifacts** to see the default artifact that is created. You can also create your own artifact by clicking the + icon.

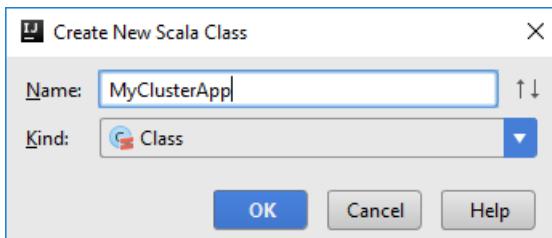


6. Add your application source code.

a. In Project Explorer, right-click **src**, point to **New**, and then click **Scala Class**.



b. In the **Create New Scala Class** dialog box, provide a name, select **Object** in the **Kind** box, and then click **OK**.



c. In the **MyClusterApp.scala** file, paste the following code. This code reads the data from HVAC.csv (available on all HDInsight Spark clusters), retrieves the rows that have only one digit in the seventh column in the CSV file, and writes the output to **/HVACOut** under the default storage container for the cluster.

```

import org.apache.spark.SparkConf
import org.apache.spark.SparkContext

object MyClusterApp{
  def main (arg: Array[String]): Unit = {
    val conf = new SparkConf().setAppName("MyClusterApp")
    val sc = new SparkContext(conf)

    val rdd = sc.textFile("wasb:///HdiSamples/HdiSamples/SensorSampleData/hvac/HVAC.csv")

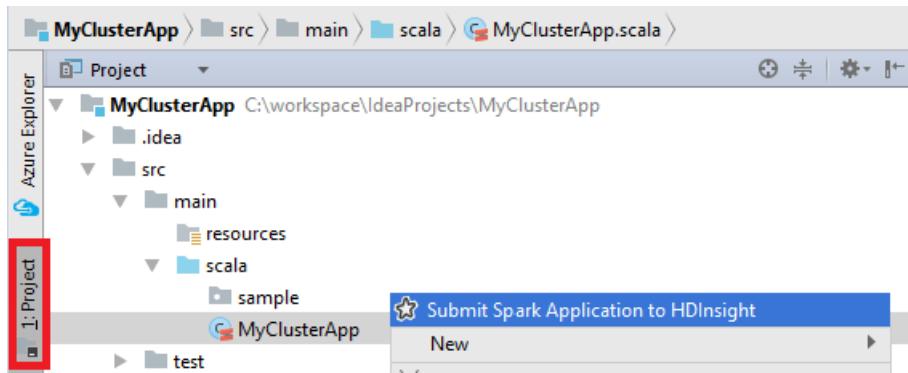
    //find the rows that have only one digit in the seventh column in the CSV file
    val rdd1 = rdd.filter(s => s.split(",")(6).length() == 1)

    rdd1.saveAsTextFile("wasb:///HVACOut")
  }
}

```

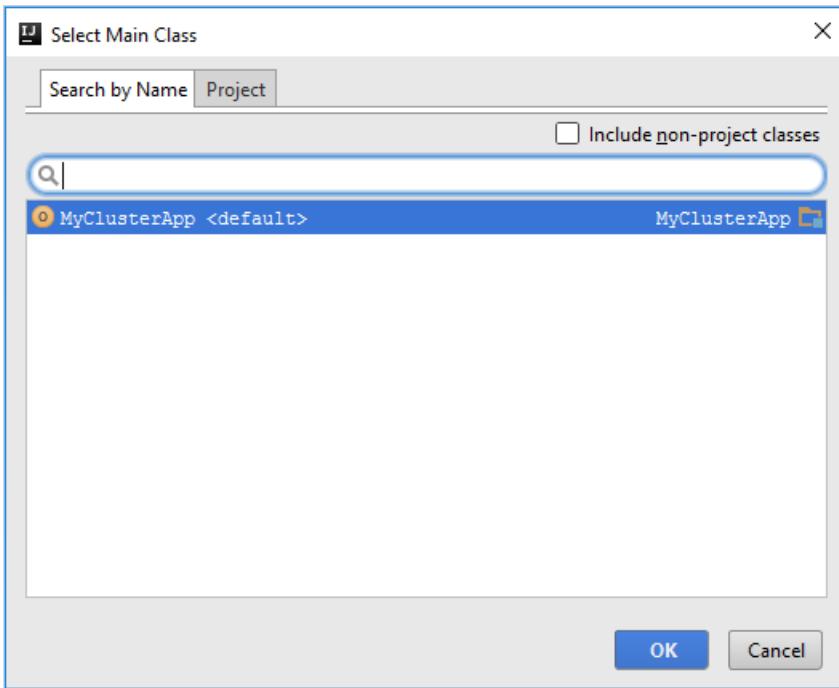
7. Run the application on an HDInsight Spark cluster.

- In Project Explorer, right-click the project name, and then select **Submit Spark Application to HDInsight**.

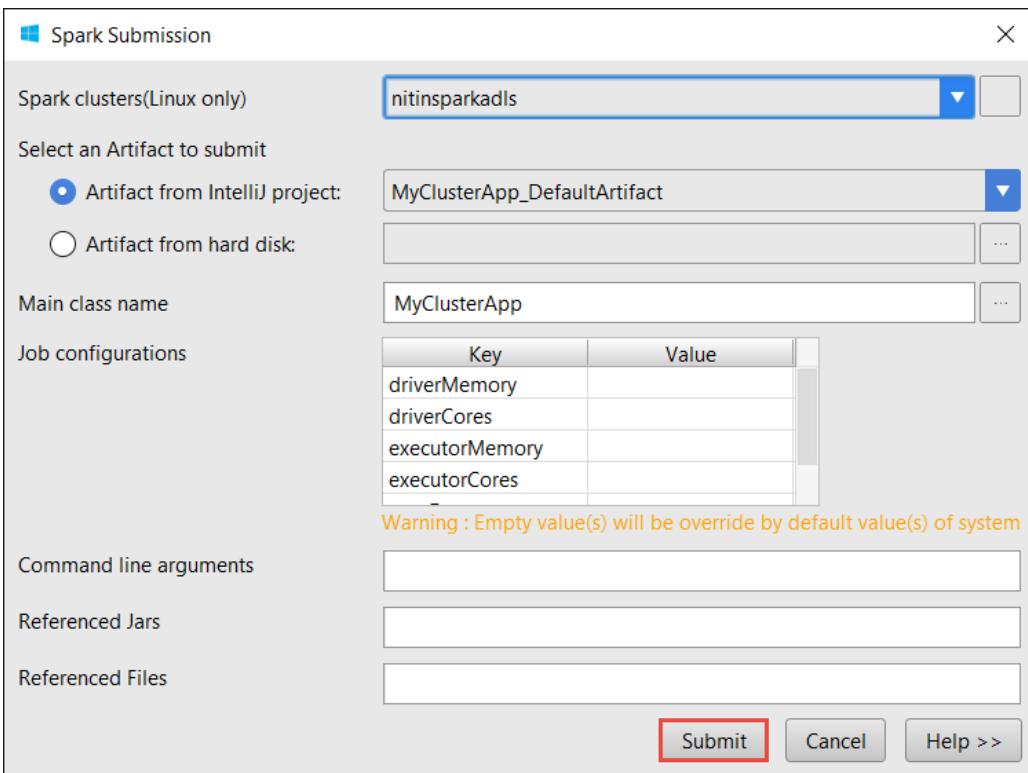


- You are prompted to enter your Azure subscription credentials. In the **Spark Submission** dialog box, provide the following values, and then click **Submit**.

- For **Spark clusters (Linux only)**, select the HDInsight Spark cluster on which you want to run your application.
- Select an artifact from the IntelliJ project, or select one from the hard drive.
- In the **Main class name** box, click the ellipsis (...), select the main class in your application source code, and then click **OK**.



- Because the application code in this example does not require any command-line arguments or reference JARs or files, you can leave the remaining boxes empty. After you provide all the inputs, the dialog box should resemble the following image.



- The **Spark Submission** tab at the bottom of the window should start displaying the progress. You can also stop the application by clicking the red button in the **Spark Submission** window.

The screenshot shows the 'Spark Submission' tab of the Azure Toolkit for IntelliJ. It displays a log of INFO Client messages. The log includes details such as client token, diagnostics, ApplicationMaster host, ApplicationMaster RPC port, queue, start time, final status, tracking URL, and user information. The log entries are timestamped from 16/06/01 20:22:31 to 16/06/01 20:22:37.

```

16/06/01 20:22:31 INFO Client: Application report for application_1464568490700_0005 (state: ACCEPTED)
16/06/01 20:22:32 INFO Client: Application report for application_1464568490700_0005 (state: ACCEPTED)
16/06/01 20:22:33 INFO Client: Application report for application_1464568490700_0005 (state: ACCEPTED)
16/06/01 20:22:34 INFO Client: Application report for application_1464568490700_0005 (state: ACCEPTED)
16/06/01 20:22:35 INFO Client: Application report for application_1464568490700_0005 (state: ACCEPTED)
16/06/01 20:22:36 INFO Client: Application report for application_1464568490700_0005 (state: ACCEPTED)
16/06/01 20:22:37 INFO Client: Application report for application_1464568490700_0005 (state: ACCEPTED)

```

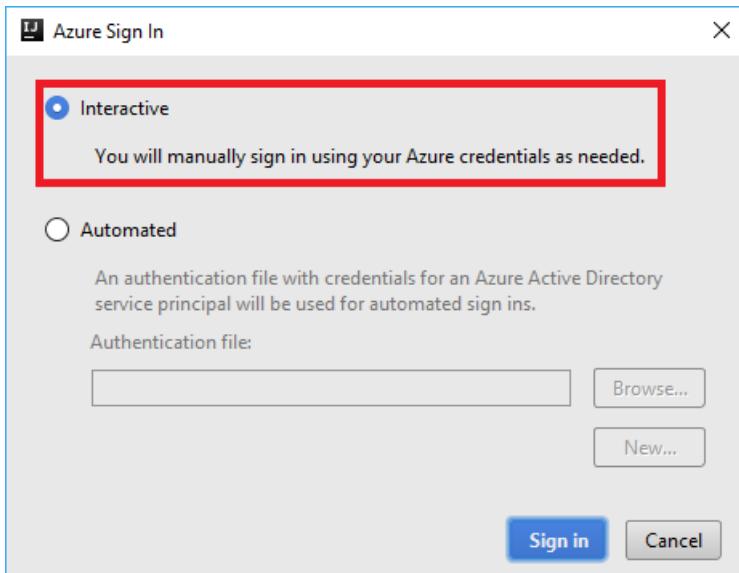
In the "Access and manage HDInsight Spark clusters by using Azure Toolkit for IntelliJ" section later in this article, you'll learn how to access the job output.

Run as or Debug as a Spark Scala application on an HDInsight Spark cluster

We also recommend another way of submitting Spark application to the cluster. They are by setting the parameters in **Run/Debug configurations** IDE. For more information, see [Remotely debug Spark applications on an HDInsight cluster with Azure Toolkit for IntelliJ through SSH](#)

Choose Azure Data Lake Store as Spark Scala application storage

If you want to submit an application to Azure Data Lake Store, you must choose **Interactive** mode during the Azure sign-in process.



If you select **Automated** mode, you get the following error:

```

List spark clusters ...
List spark clusters successfully
List additional spark clusters successfully
Info : Build null successfully.
Info : Get target jar from C:/workspace/IdeaProjects/MyLocalApp/out/artifacts/MyLocalApp_DefaultArtifact/default_artifact.jar.
Info : Begin uploading file C:/workspace/IdeaProjects/MyLocalApp/out/artifacts/MyLocalApp_DefaultArtifact/default_artifact.jar to Azure Datalake store
adls/devtooltelemetryadls.azuredatalakestore.net/spark2adls/SparkSubmission ...
Error : Failed to submit application to spark cluster. Exception : Forbidden. This problem could be: 1. Attached Azure DataLake Store is not supported in Automated login model. Please logout first and try Interactive login model2. Login account have no write permission on attached ADLS storage. Please grant write access from storage account admin(or other roles who have permission to do it)

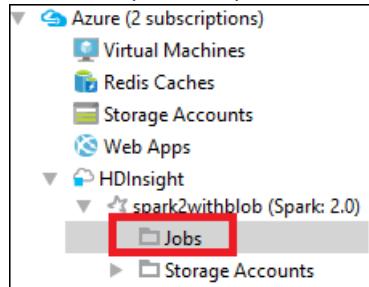
```

Access and manage HDInsight Spark clusters by using Azure Toolkit for IntelliJ

You can perform various operations by using Azure Toolkit for IntelliJ.

Access the job view

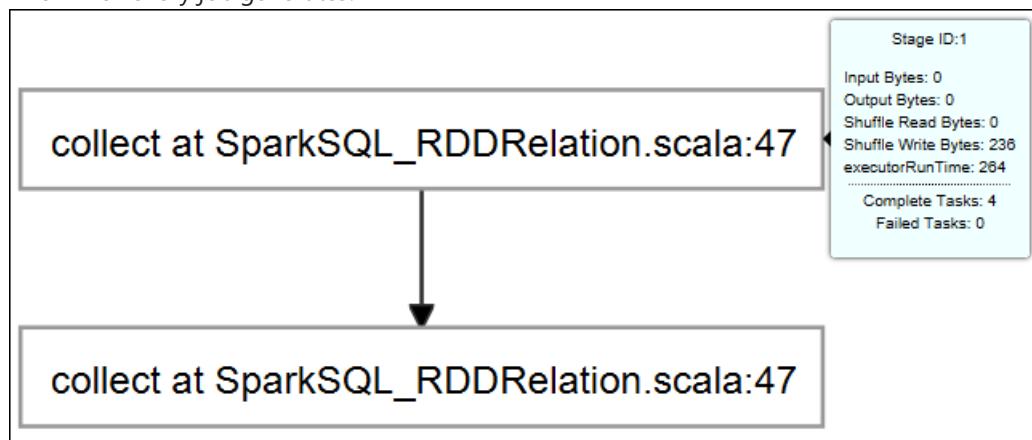
1. In Azure Explorer, expand **HDInsight**, expand the Spark cluster name, and then click **Jobs**.



2. In the right pane, the **Spark Job View** tab displays all the applications that were run on the cluster. Click the name of the application for which you want to see more details.

A screenshot of the 'spark2withblob: Job View' tab. On the left, a table lists 15 applications with columns: State, Application ID, Job Name, Start Time, Attempt Times, and Spark User. One row, 'application_14992986_62215_0007', is highlighted with a yellow background. On the right, a detailed view for 'application_14992986_62215_0007' is shown. It includes tabs for Application Graph, Stage Summary, Executors, Task Summary, and Logs. The Application Graph shows a 'Driver' node connected to eight 'task' nodes labeled 'task 0' through 'task 7'. A tooltip for the 'Driver' node provides job statistics: Job ID: 0, collect at: SparkSQL_RDDRelation.scala:44, Time Duration (ms): 0.02, Completed Tasks: 4, Failed Tasks: 0, Completed Stages: 1, Failed Stages: 0.

3. Hover on job graph, it displays basic running job info. Click on job graph, you can see the stages graph and info which every job generates.



4. Frequently-used log including Driver Stderr, Driver Stdout, Directory Info are listed in **Log** tab.

The screenshot shows the Azure Toolkit for IntelliJ interface with several tabs at the top: Application Graph, Stage Summary, Executors, Task Summary, and Logs. The Logs tab is selected, showing two sections: Driver Stderr and Driver Stdout. The Driver Stderr section contains a large block of log output from a Spark application, detailing the startup of various services like SparkEnv, MapOutputTracker, BlockManagerMaster, and MemoryStore. The Driver Stdout section is currently empty. Below these, a Directory Info section shows the terminal command 'ls -l' executed in a directory, listing files such as yarn_hadoop.jar, container_tokens, default_container_executor_session.sh, default_container_executor.sh, launch_container.sh, spark_conf.zip, and launch_container.shcrc.

```

17/07/06 03:36:34 INFO SparkEnv: Successfully started service 'sparkEnv' on port 50501
17/07/06 03:36:34 INFO SparkEnv: Registering MapOutputTracker
17/07/06 03:36:34 INFO SparkEnv: Registering BlockManagerMaster
17/07/06 03:36:34 INFO DiskBlockManager: Created local directory at /mnt/resource/hadoop/yarn/local/usercache/livy/appcache/application_1499298662215_0007/blockmgr-ff8e6103-1d1c-44ca-bc3e-92134091955c
17/07/06 03:36:34 INFO MemoryStore: MemoryStore started with capacity 366.3 MB
17/07/06 03:36:34 INFO SparkEnv: Registering OutputCommitCoordinator
17/07/06 03:36:34 INFO log: Logging initialized @36785ms
17/07/06 03:36:34 INFO JettyUtils: Adding filter: org.apache.hadoop.yarn.server.webproxy.amfilter.AmIpFilter
17/07/06 03:36:34 INFO Server: jetty-9.2.z-SNAPSHOT
17/07/06 03:36:34 INFO ContextHandler: Started o.s.j.s.ServletContextHandler@8b7b1d{/jobs,null,AVAILABLE}
17/07/06 03:36:34 INFO ContextHandler: Started o.s.j.s.ServletContextHandler@1c9558de{/jobs/json,null,AVAILABLE}
17/07/06 03:36:34 INFO ContextHandler: Started o.s.j.s.ServletContextHandler@2fd6a91{/jobs/job,null,AVAILABLE}
17/07/06 03:36:34 INFO ContextHandler: Started o.s.j.s.ServletContextHandler@9af2b1{/jobs/job/json,null,AVAILABLE}
17/07/06 03:36:34 INFO ContextHandler: Started o.s.j.s.ServletContextHandler@5b92222c{/stages,null,AVAILABLE}
17/07/06 03:36:34 INFO ContextHandler: Started o.s.j.s.ServletContextHandler@6d77c228{/stages/json,null,AVAILABLE}
17/07/06 03:36:34 INFO ContextHandler: Started o.s.j.s.ServletContextHandler@348784f5{/stages/stage,null,AVAILABLE}
17/07/06 03:36:34 INFO ContextHandler: Started o.s.j.s.ServletContextHandler@63dff7ae{/stages/stage/json,null,AVAILABLE}
17/07/06 03:36:34 INFO ContextHandler: Started o.s.j.s.ServletContextHandler@29163abb{/stages/pool,null,AVAILABLE}
17/07/06 03:36:34 INFO ContextHandler: Started
Driver Stdout
```

Driver Stdout

Directory Info

```

ls -l:
total 32
lrwxrwxrwx 1 yarn hadoop 66 Jul 6 03:35 __app__.jar ->
/mnt/resource/hadoop/yarn/local/filecache/227/default_artifact.jar
-rw-r--r-- 1 yarn hadoop 74 Jul 6 03:35 container_tokens
-rwx----- 1 yarn hadoop 705 Jul 6 03:35 default_container_executor_session.sh
-rwx----- 1 yarn hadoop 759 Jul 6 03:35 default_container_executor.sh
-rwx----- 1 yarn hadoop 5678 Jul 6 03:35 launch_container.sh
lrwxrwxrwx 1 yarn hadoop 79 Jul 6 03:35 __spark_conf__ ->
/mnt/resource/hadoop/yarn/local/usercache/livy/filecache/224/__spark_conf__.zip
drwx---x- 2 yarn hadoop 4096 Jul 6 03:35 tmp
find -L . -maxdepth 5 -ls:
20581517 4 drwx--x--- 3 yarn      hadoop      4096 Jul 6 03:35 .
20581523 4 -rw-r--r-- 1 yarn      hadoop      56 Jul 6 03:35 ./launch_container.sh.crc
20581520 4 -rw-r--r-- 1 yarn      hadoop      74 Jul 6 03:35 ./container_tokens
```

- You can also open the Spark history UI and the YARN UI (at the application level) by clicking the respective hyperlink at the top of the window.

Access the Spark history server

- In Azure Explorer, expand **HDInsight**, right-click your Spark cluster name, and then select **Open Spark History UI**. When you're prompted, enter the admin credentials for the cluster. You must have specified these while provisioning the cluster.
- In the Spark history server dashboard, you can use the application name to look for the application that you just finished running. In the preceding code, you set the application name by using `val conf = new SparkConf().setAppName("MyClusterApp")`. Hence, your Spark application name was **MyClusterApp**.

Start the Ambari portal

- In Azure Explorer, expand **HDInsight**, right-click your Spark cluster name, and then select **Open Cluster Management Portal (Ambari)**.
- When you're prompted, enter the admin credentials for the cluster. You specified these credentials during the cluster provisioning process.

Manage Azure subscriptions

By default, Azure Toolkit for IntelliJ lists the Spark clusters from all your Azure subscriptions. If necessary, you can specify the subscriptions for which you want to access the cluster.

- In Azure Explorer, right-click the **Azure** root node, and then click **Manage Subscriptions**.
- In the dialog box, clear the check boxes for the subscription that you don't want to access, and then click **Close**. You can also click **Sign Out** if you want to sign out of your Azure subscription.

Run a Spark Scala application locally

You can use Azure Toolkit for IntelliJ to run Spark Scala applications locally on your workstation. Typically, these applications don't need access to cluster resources such as a storage container, and you can run and test them locally.

Prerequisite

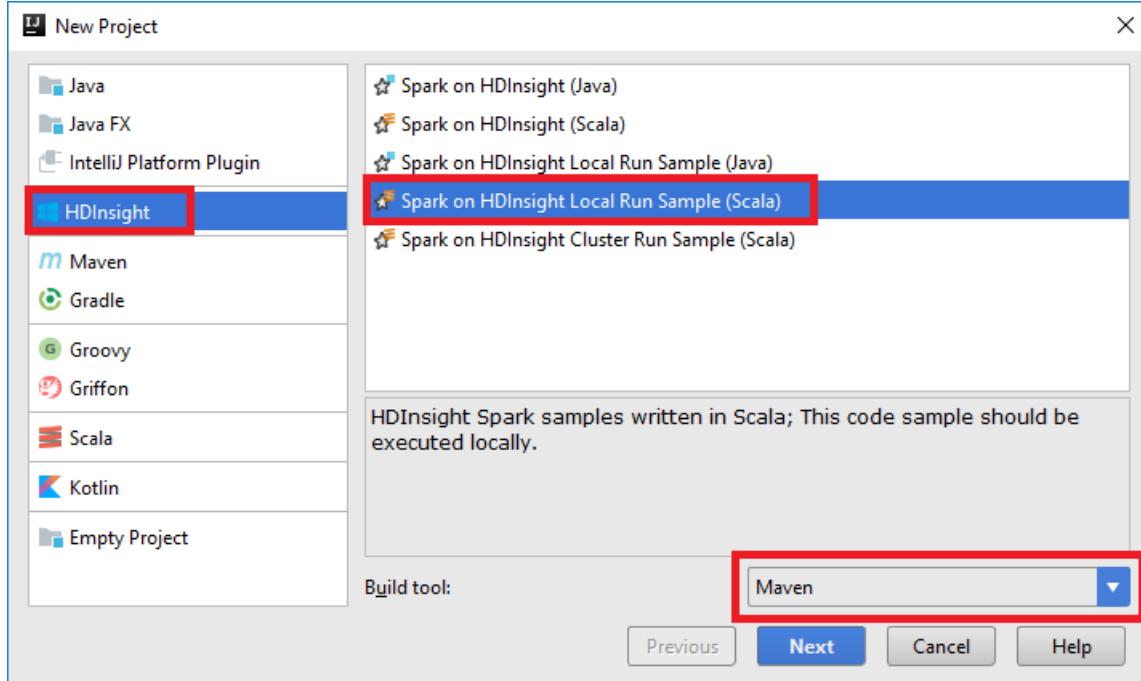
While you're running the local Spark Scala application on a Windows computer, you might get an exception as explained in [SPARK-2356](#). This exception occurs because WinUtils.exe is missing on Windows.

To resolve this error, you must [download the executable](#) to a location like **C:\WinUtils\bin**. Then, add the environment variable **HADOOP_HOME** and set the value of the variable to **C\WinUtils**.

Run a local Spark Scala application

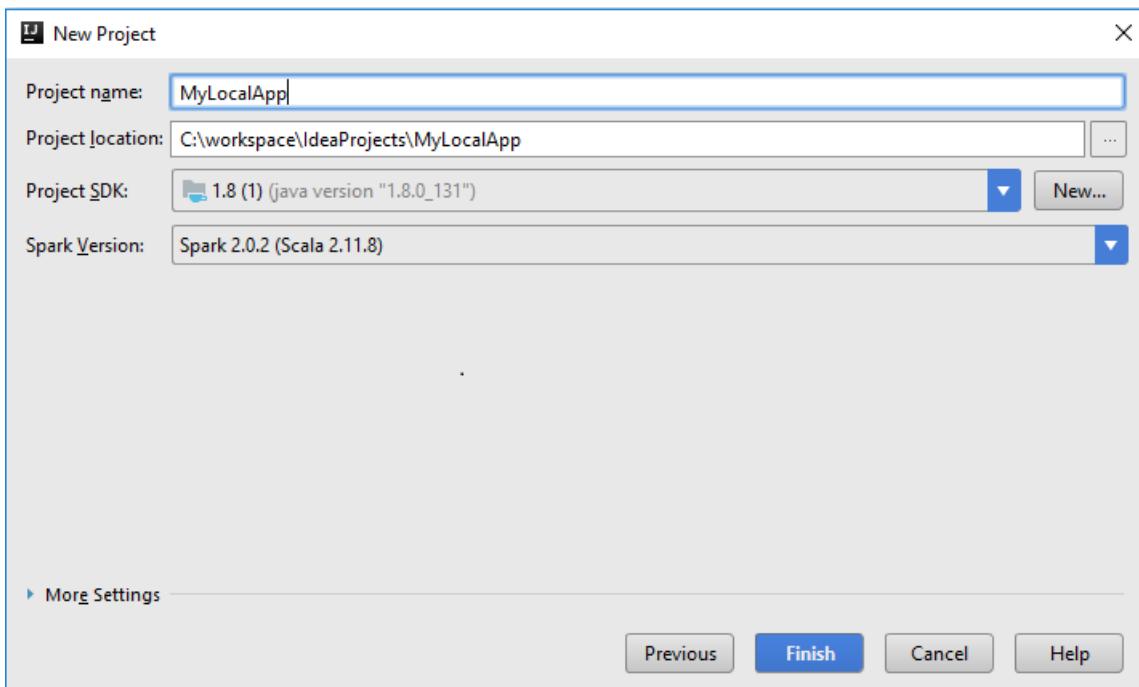
1. Start IntelliJ IDEA and create a project. In the **New Project** dialog box, make the following choices, and then click **Next**.

- In the left pane, select **HDInsight**.
- In the right pane, select **Spark on HDInsight Local Run Sample (Scala)**.
- Build tool: Scala project creation wizard support Maven or SBT managing the dependencies and building for scala project. You select one according need.

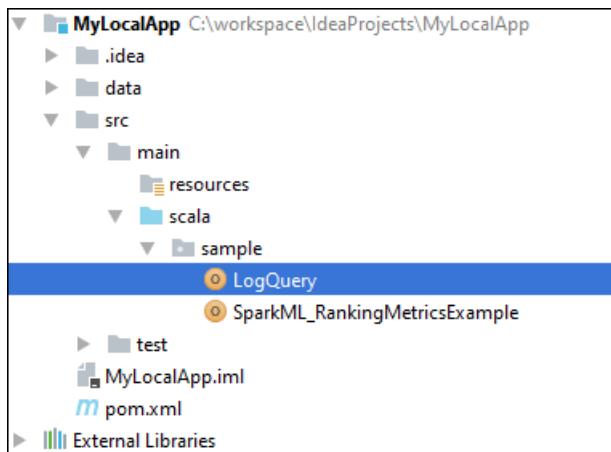


2. In the next window, provide the following project details, and then click **Finish**.

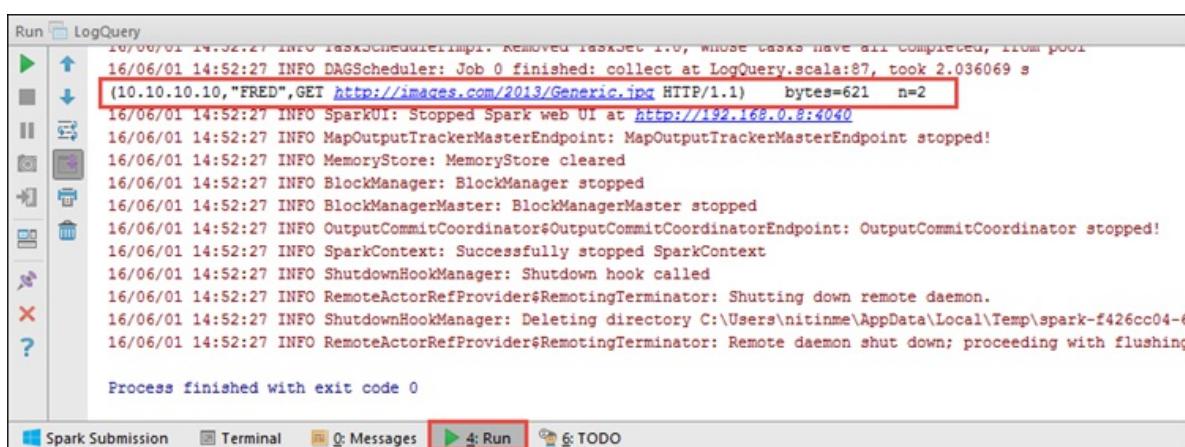
- Provide a project name and project location.
- For **Project SDK**, make sure that you provide a Java version later than 7.
- For **Spark Version**, select the version of Scala to use: Scala 2.11.x for Spark 2.0, and Scala 2.10.x for Spark 1.6.



3. The template adds a sample code (**LogQuery**) under the **src** folder that you can run locally on your computer.



4. Right-click the **LogQuery** application, and then click **Run 'LogQuery'**. On the **Run** tab at the bottom, you see an output like the following.



Convert existing IntelliJ IDEA applications to use Azure Toolkit for IntelliJ

You can convert your existing Spark Scala applications created in IntelliJ IDEA to be compatible with Azure Toolkit for IntelliJ. You can then use the plug-in to submit the applications to an HDInsight Spark cluster.

1. For an existing Spark Scala application created through IntelliJ IDEA, open the associated .iml file.
2. At the root level, you see a **module** element like this:

```
<module org.jetbrains.idea.maven.project.MavenProjectsManager.isMavenModule="true" type="JAVA_MODULE"
version="4">
```

Edit the element to add `UniqueKey="HDInsightTool"` so that the **module** element looks like the following:

```
<module org.jetbrains.idea.maven.project.MavenProjectsManager.isMavenModule="true" type="JAVA_MODULE"
version="4" UniqueKey="HDInsightTool">
```

3. Save the changes. Your application should now be compatible with Azure Toolkit for IntelliJ. You can test it by right-clicking the project name in Project Explorer. The pop-up menu now has the option **Submit Spark Application to HDInsight**.

Troubleshooting

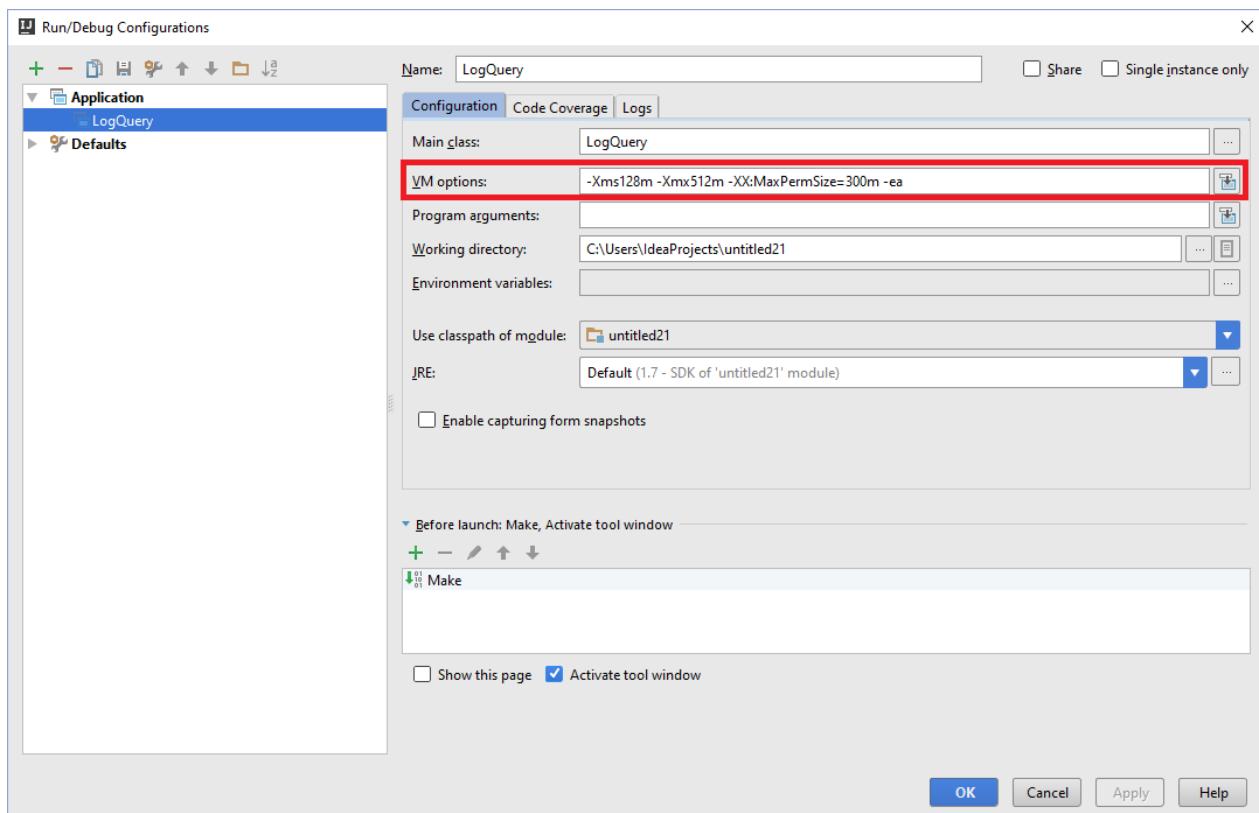
"Please use a larger heap size" error in local run

In Spark 1.6, if you're using a 32-bit Java SDK during local run, you might encounter the following errors:

```
Exception in thread "main" java.lang.OutOfMemoryError: System memory 259522560 must be at least
4.718592E8. Please use a larger heap size.
    at org.apache.spark.memory.UnifiedMemoryManager$.getMaxMemory(UnifiedMemoryManager.scala:193)
    at org.apache.spark.memory.UnifiedMemoryManager$.apply(UnifiedMemoryManager.scala:175)
    at org.apache.spark.SparkEnv$.create(SparkEnv.scala:354)
    at org.apache.spark.SparkEnv$.createDriverEnv(SparkEnv.scala:193)
    at org.apache.spark.SparkContext.createSparkEnv(SparkContext.scala:288)
    at org.apache.spark.SparkContext.<init>(SparkContext.scala:457)
    at LogQuery$.main(LogQuery.scala:53)
    at LogQuery.main(LogQuery.scala)
    at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
    at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:57)
    at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
    at java.lang.reflect.Method.invoke(Method.java:606)
    at com.intellij.rt.execution.application.AppMain.main(AppMain.java:144)
```

These errors happen because the heap size is not large enough for Spark to run. (Spark requires at least 471 MB. You can get more details from [SPARK-12081](#)). One simple solution is to use a 64-bit Java SDK. You can also change the JVM settings in IntelliJ by adding the following options:

```
-Xms128m -Xmx512m -XX:MaxPermSize=300m -ea
```



Feedback and known issues

Currently, viewing Spark outputs directly is not supported.

If you have any suggestions or feedback, or if you encounter any problems when using this plug-in, send us an email at hdivstool@microsoft.com.

See also

- [Overview: Apache Spark on Azure HDInsight](#)

Demo

- Create Scala Project (Video): [Create Spark Scala Applications](#)
- Remote Debug (Video): [Use Azure Toolkit for IntelliJ to debug Spark applications remotely on HDInsight Cluster](#)

Scenarios

- [Spark with BI: Perform interactive data analysis using Spark in HDInsight with BI tools](#)
- [Spark with Machine Learning: Use Spark in HDInsight for analyzing building temperature using HVAC data](#)
- [Spark with Machine Learning: Use Spark in HDInsight to predict food inspection results](#)
- [Spark Streaming: Use Spark in HDInsight for building real-time streaming applications](#)
- [Website log analysis using Spark in HDInsight](#)

Creating and running applications

- [Create a standalone application using Scala](#)
- [Run jobs remotely on a Spark cluster using Livy](#)

Tools and extensions

- [Use Azure Toolkit for IntelliJ to debug Spark applications remotely through VPN](#)
- [Use Azure Toolkit for IntelliJ to debug Spark applications remotely through SSH](#)
- [Use HDInsight Tools for IntelliJ with Hortonworks Sandbox](#)

- Use HDInsight Tools in Azure Toolkit for Eclipse to create Spark applications
- Use Zeppelin notebooks with a Spark cluster on HDInsight
- Kernels available for Jupyter notebook in Spark cluster for HDInsight
- Use external packages with Jupyter notebooks
- Install Jupyter on your computer and connect to an HDInsight Spark cluster

Managing resources

- Manage resources for the Apache Spark cluster in Azure HDInsight
- Track and debug jobs running on an Apache Spark cluster in HDInsight

Use Azure Toolkit for IntelliJ to debug applications remotely on HDInsight Spark through VPN

8/16/2017 • 9 min to read • [Edit Online](#)

We recommend the way of debugging spark applicaltion remotely through ssh. For instructions, see [Remotely debug Spark applications on an HDInsight cluster with Azure Toolkit for IntelliJ through SSH](#).

This article provides step-by-step guidance on how to use the HDInsight Tools in Azure Toolkit for IntelliJ to submit a Spark job on HDInsight Spark cluster and then debug it remotely from your desktop computer. To do so, you must perform the following high-level steps:

1. Create a site-to-site or point-to-site Azure Virtual Network. The steps in this document assume that you use a site-to-site network.
2. Create a Spark cluster in Azure HDInsight that is part of the site-to-site Azure Virtual Network.
3. Verify the connectivity between the cluster headnode and your desktop.
4. Create a Scala application in IntelliJ IDEA and configure it for remote debugging.
5. Run and debug the application.

Prerequisites

- An Azure subscription. See [Get Azure free trial](#).
- An Apache Spark cluster on HDInsight. For instructions, see [Create Apache Spark clusters in Azure HDInsight](#).
- Oracle Java Development kit. You can install it from [here](#).
- IntelliJ IDEA. This article uses version 2017.1. You can install it from [here](#).
- HDInsight Tools in Azure Toolkit for IntelliJ. HDInsight tools for IntelliJ are available as part of the Azure Toolkit for IntelliJ. For instructions on how to install the Azure Toolkit, see [Installing the Azure Toolkit for IntelliJ](#).
- Log into your Azure Subscription from IntelliJ IDEA. Follow the instructions [here](#).
- While running Spark Scala application for remote debugging on a Windows computer, you might get an exception as explained in [SPARK-2356](#) that occurs due to a missing WinUtils.exe on Windows. To work around this error, you must [download the executable from here](#) to a location like **C:\WinUtils\bin**. You must then add an environment variable **HADOOP_HOME** and set the value of the variable to **C\WinUtils**.

Step 1: Create an Azure Virtual Network

Follow the instructions from the below links to create an Azure Virtual Network and then verify the connectivity between the desktop and Azure Virtual Network.

- [Create a VNet with a site-to-site VPN connection using Azure Portal](#)
- [Create a VNet with a site-to-site VPN connection using PowerShell](#)
- [Configure a point-to-site connection to a virtual network using PowerShell](#)

Step 2: Create an HDInsight Spark cluster

You should also create an Apache Spark cluster on Azure HDInsight that is part of the Azure Virtual Network that you created. Use the information available at [Create Linux-based clusters in HDInsight](#). As part of optional configuration, select the Azure Virtual Network that you created in the previous step.

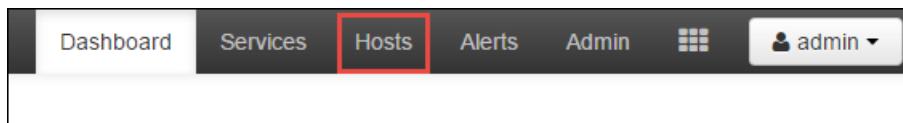
Step 3: Verify the connectivity between the cluster headnode and your

desktop

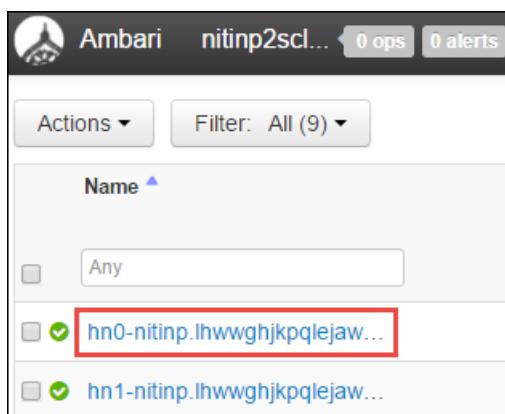
- Get the IP address of the headnode. Open Ambari UI for the cluster. From the cluster blade, click **Dashboard**.



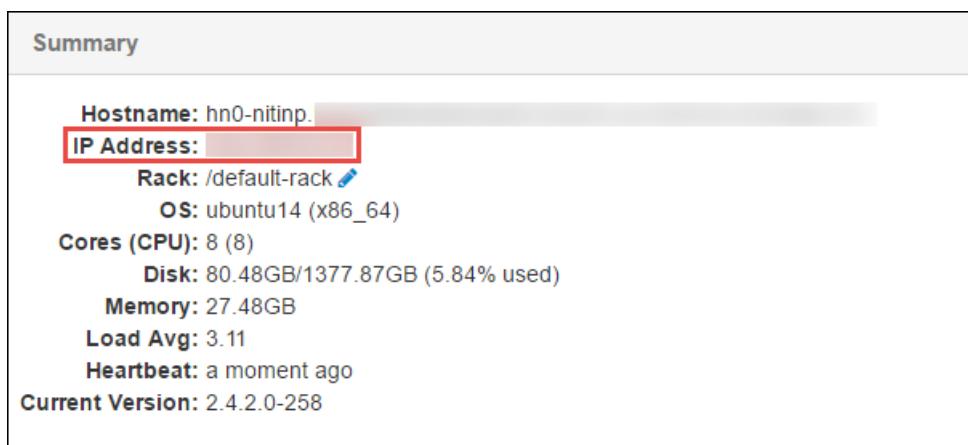
- From the Ambari UI, from the top-right corner, click **Hosts**.



- You should see a list of headnodes, worker nodes, and zookeeper nodes. The headnodes have the **hn*** prefix. Click the first headnode.



- At the bottom of the page that opens, from the **Summary** box, copy the IP address of the headnode and the host name.



- Include the IP address and the host name of the headnode to the **hosts** file on the computer from where you want to run and remotely debug the Spark jobs. This will enable you to communicate with the headnode using the IP address as well as the hostname.

- Open a notepad with elevated permissions. From the file menu, click **Open** and then navigate to the location of the hosts file. On a Windows computer, it is `C:\Windows\System32\Drivers\etc\hosts`.
- Add the following to the **hosts** file.

```

# For headnode0
192.xxx.xx.xx hn0-nitinp
192.xxx.xx.xx hn0-nitinp.lhwgghjkpquejawpbwcdyp3.gx.internal.cloudapp.net

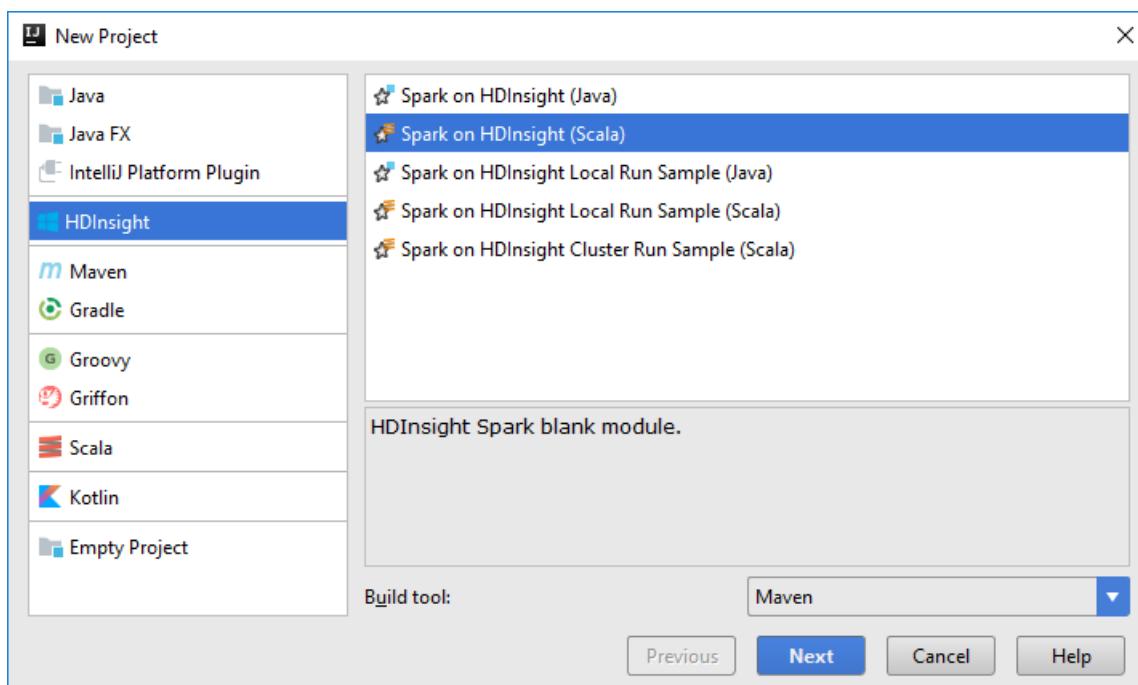
# For headnode1
192.xxx.xx.xx hn1-nitinp
192.xxx.xx.xx hn1-nitinp.lhwgghjkpquejawpbwcdyp3.gx.internal.cloudapp.net

```

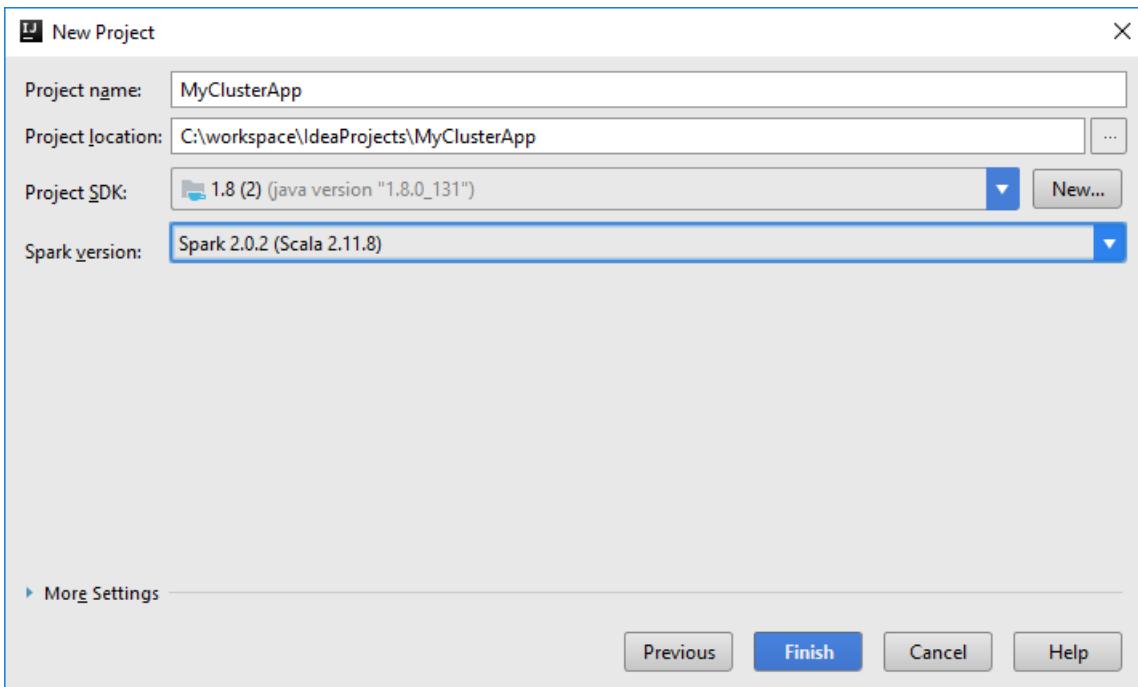
6. From the computer that you connected to the Azure Virtual Network that is used by the HDInsight cluster, verify that you can ping both the headnodes using the IP address as well as the hostname.
7. SSH into the cluster headnode using the instructions at [Connect to an HDInsight cluster using SSH](#). From the cluster headnode, ping the IP address of the desktop computer. You should test connectivity to both the IP addresses assigned to the computer, one for the network connection and the other for the Azure Virtual Network that the computer is connected to.
8. Repeat the steps for the other headnode as well.

Step 4: Create a Spark Scala application using the HDInsight Tools in Azure Toolkit for IntelliJ and configure it for remote debugging

1. Launch IntelliJ IDEA and create a new project. In the new project dialog box, make the following choices, and then click **Next**.

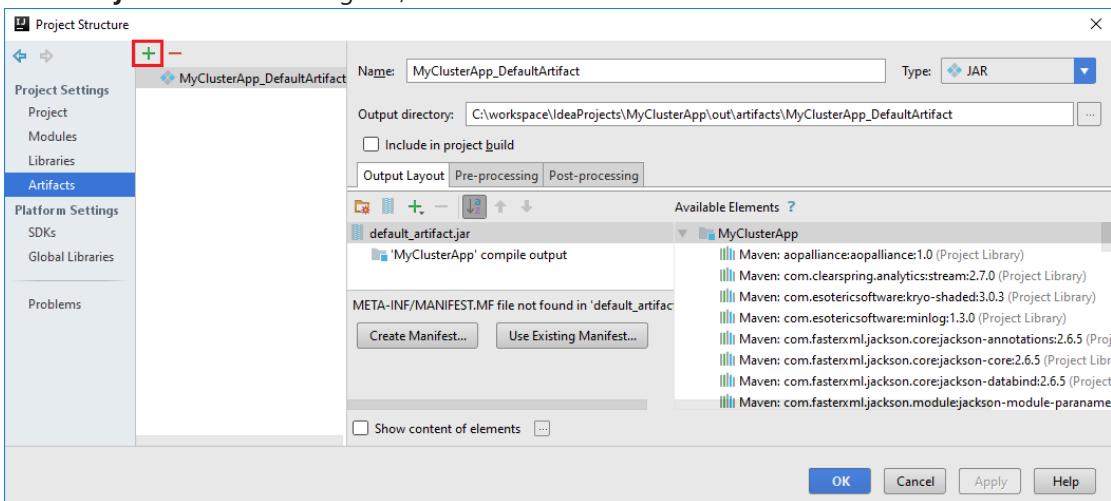


- From the left pane, select **HDInsight**.
 - From the right pane, select **Spark on HDInsight (Scala)**.
 - Click **Next**.
2. In the next window, provide the following project details, and then click **Finish**.
 - Provide a project name and project location.
 - For **Project SDK**, Use Java 1.8 for spark 2.x cluster, Java 1.7 for spark 1.x cluster.
 - For **Spark Version**, Scala project creation wizard integrates proper version for Spark SDK and Scala SDK. If the spark cluster version is lower 2.0, choose spark 1.x. Otherwise, you should select spark2.x. This example uses Spark2.0.2(Scala 2.11.8).



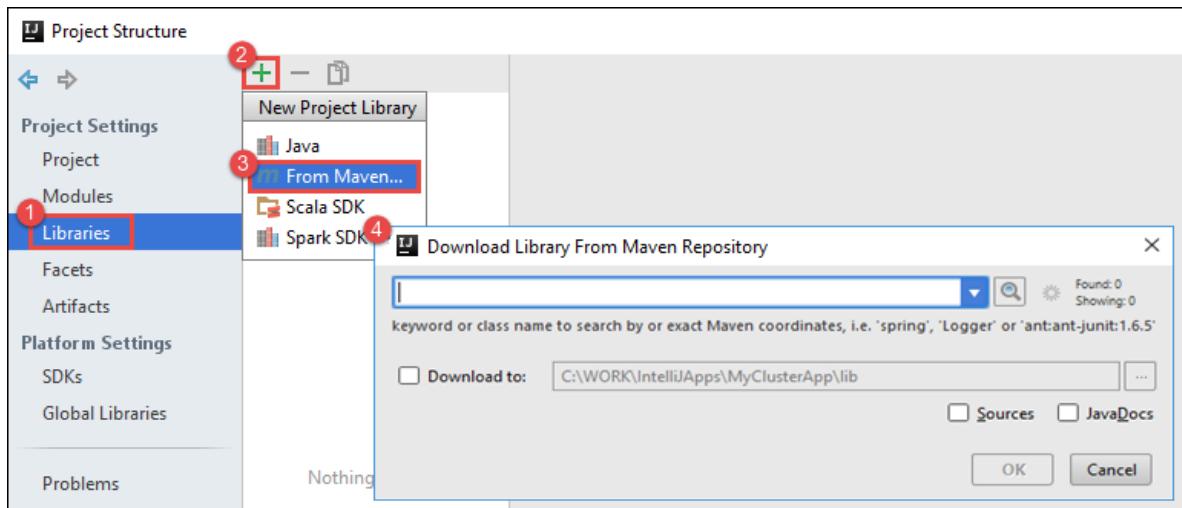
3. The Spark project will automatically create an artifact for you. To see the artifact, follow these steps.

- From the **File** menu, click **Project Structure**.
- In the **Project Structure** dialog box, click **Artifacts** to see the default artifact that is created.



You can also create your own artifact by clicking on the + icon, highlighted in the image above.

- Add libraries to your project. To add a library, right-click the project name in the project tree, and then click **Open Module Settings**. In the **Project Structure** dialog box, from the left pane, click **Libraries**, click the (+) symbol, and then click **From Maven**.



In the **Download Library from Maven Repository** dialog box, search and add the following libraries.

- org.scalatest:scalatest_2.10:2.2.1
- org.apache.hadoop:hadoop-azure:2.7.1

5. Copy `yarn-site.xml` and `core-site.xml` from the cluster headnode and add it to the project. Use the following commands to copy the files. You can use [Cygwin](#) to run the following `scp` commands to copy the files from the cluster headnodes.

```
scp <ssh user name>@<headnode IP address or host name>:/etc/hadoop/conf/core-site.xml .
```

Because we already added the cluster headnode IP address and hostnames to the hosts file on the desktop, we can use the `scp` commands in the following manner.

```
scp sshuser@hn0-nitinp:/etc/hadoop/conf/core-site.xml .
scp sshuser@hn0-nitinp:/etc/hadoop/conf/yarn-site.xml .
```

Add these files to your project by copying them under the `/src` folder in your project tree, for example

```
<your project directory>\src .
```

6. Update the `core-site.xml` to make the following changes.

- `core-site.xml` includes the encrypted key to the storage account associated with the cluster. In the `core-site.xml` that you added to the project, replace the encrypted key with the actual storage key associated with the default storage account. See [Manage your storage access keys](#).

```
<property>
  <name>fs.azure.account.key.hdistoragecentral.blob.core.windows.net</name>
  <value>access-key-associated-with-the-account</value>
</property>
```

- Remove the following entries from the `core-site.xml`.

```

<property>
    <name>fs.azure.account.keyprovider.hdstoragecentral.blob.core.windows.net</name>
    <value>org.apache.hadoop.fs.azure.ShellDecryptionKeyProvider</value>
</property>

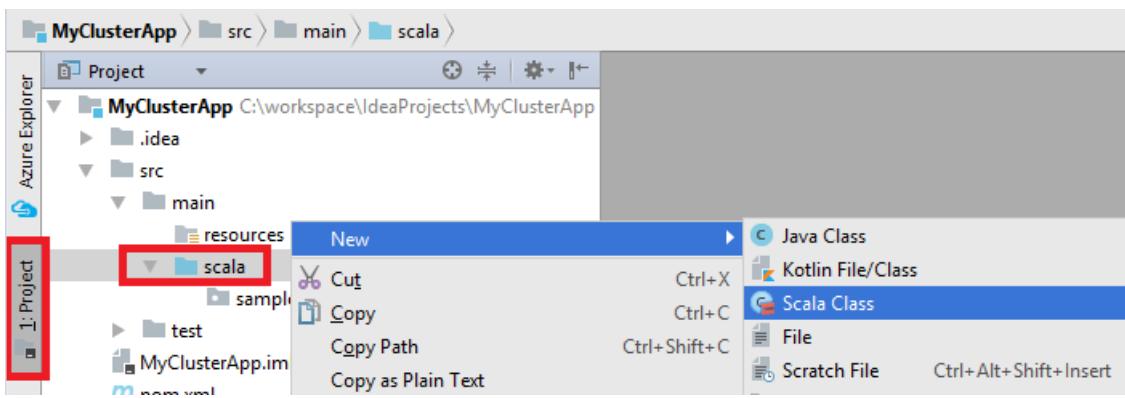
<property>
    <name>fs.azure.shellkeyprovider.script</name>
    <value>/usr/lib/python2.7/dist-packages/hdinsight_common/decrypt.sh</value>
</property>

<property>
    <name>net.topology.script.file.name</name>
    <value>/etc/hadoop/conf/topology_script.py</value>
</property>

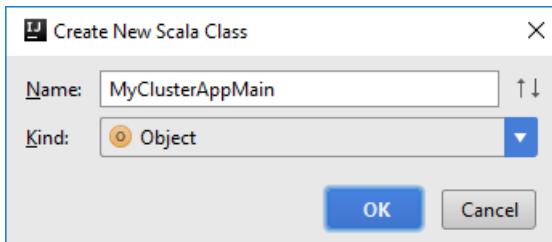
```

c. Save the file.

- Add the Main class for your application. From the **Project Explorer**, right-click **src**, point to **New**, and then click **Scala class**.



- In the **Create New Scala Class** dialog box, provide a name, for **Kind** select **Object**, and then click **OK**.



- In the `MyClusterAppMain.scala` file, paste the following code. This code creates the Spark context and launches an `executeJob` method from the `SparkSample` object.

```

import org.apache.spark.{SparkConf, SparkContext}

object SparkSampleMain {
  def main (arg: Array[String]): Unit = {
    val conf = new SparkConf().setAppName("SparkSample")
      .set("spark.hadoop.validateOutputSpecs", "false")
    val sc = new SparkContext(conf)

    SparkSample.executeJob(sc,
      "wasb://HdiSamples/HdiSamples/SensorSampleData/hvac/HVAC.csv",
      "wasb:///HVACOut")
  }
}

```

- Repeat steps 8 and 9 above to add a new Scala object called `SparkSample`. To this class add the following code. This code reads the data from the HVAC.csv (available on all HDInsight Spark clusters), retrieves the

rows that only have one digit in the seventh column in the CSV, and writes the output to **/HVACOut** under the default storage container for the cluster.

```
import org.apache.spark.SparkContext

object SparkSample {
  def executeJob (sc: SparkContext, input: String, output: String): Unit = {
    val rdd = sc.textFile(input)

    //find the rows which have only one digit in the 7th column in the CSV
    val rdd1 = rdd.filter(s => s.split(",")(6).length() == 1)

    val s = sc.parallelize(rdd.take(5)).cartesian(rdd).count()
    println(s)

    rdd1.saveAsTextFile(output)
    //rdd1.collect().foreach(println)
  }
}
```

11. Repeat steps 8 and 9 above to add a new class called `RemoteClusterDebugging`. This class implements the Spark test framework that is used for debugging applications. Add the following code to the `RemoteClusterDebugging` class.

```
import org.apache.spark.{SparkConf, SparkContext}
import org.scalatest.FunSuite

class RemoteClusterDebugging extends FunSuite {

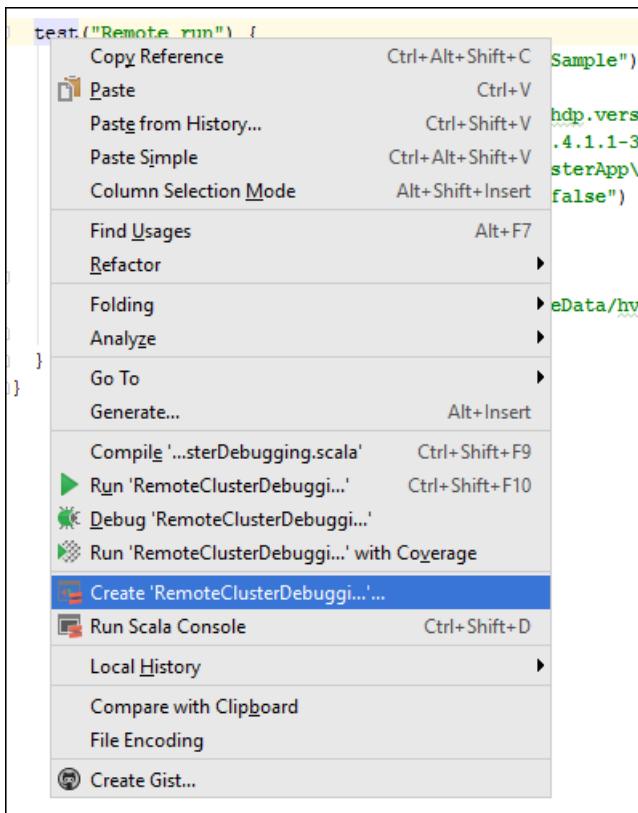
  test("Remote run") {
    val conf = new SparkConf().setAppName("SparkSample")
      .setMaster("yarn-client")
      .set("spark.yarn.am.extraJavaOptions", "-Dhdp.version=2.4")
      .set("spark.yarn.jar", "wasb://hdp/apps/2.4.2.0-258/spark-assembly-
1.6.1.2.4.2.0-258-hadoop2.7.1.2.4.2.0-258.jar")

      .setJars(Seq("""C:\workspace\IdeaProjects\MyClusterApp\out\artifacts\MyClusterApp_DefaultArtifact\defa
ult_artifact.jar"""))
      .set("spark.hadoop.validateOutputSpecs", "false")
    val sc = new SparkContext(conf)

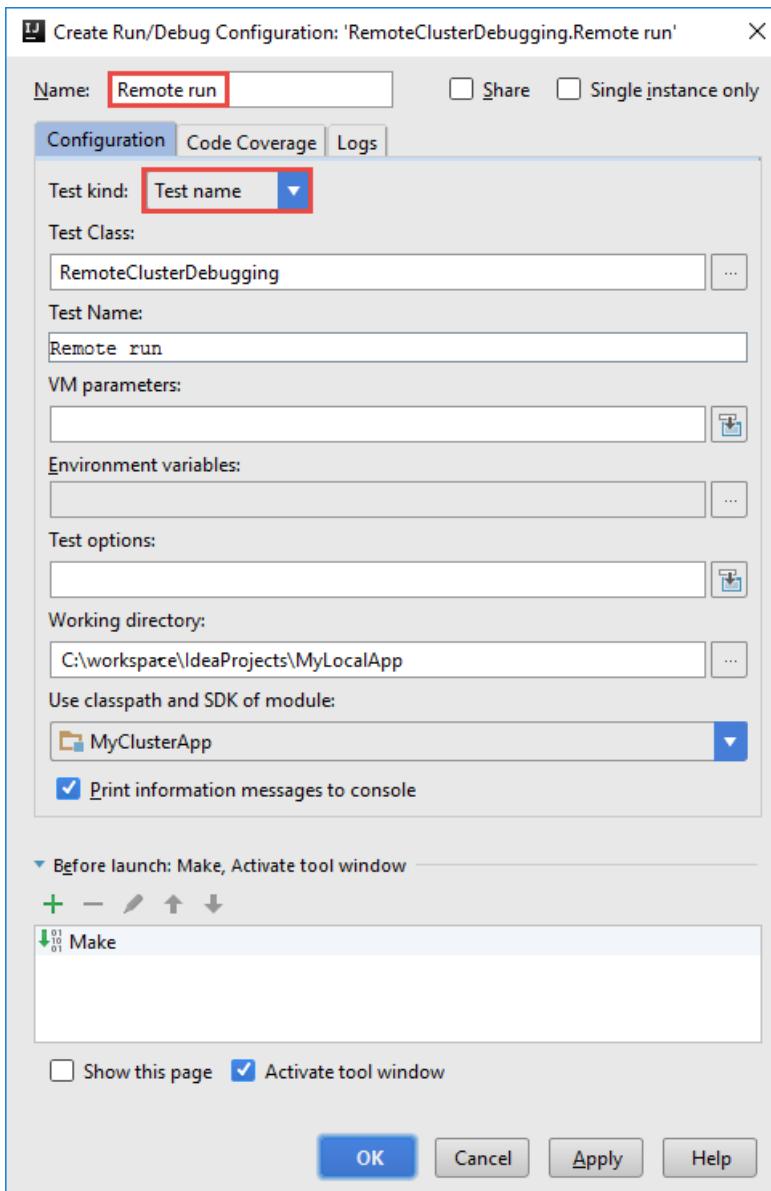
    SparkSample.executeJob(sc,
      "wasb://HdiSamples/HdiSamples/SensorSampleData/hvac/HVAC.csv",
      "wasb:///HVACOut")
  }
}
```

Couple of important things to note here:

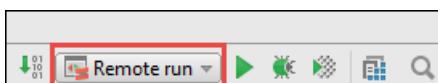
- For
`.set("spark.yarn.jar", "wasb://hdp/apps/2.4.2.0-258/spark-assembly-1.6.1.2.4.2.0-258-
hadoop2.7.1.2.4.2.0-258.jar")`, make sure the Spark assembly JAR is available on the cluster storage at the specified path.
 - For `setJars`, specify the location where the artifact jar will be created. Typically it is
`<Your IntelliJ project directory>\out\<project name>_DefaultArtifact\default_artifact.jar`.
12. In the `RemoteClusterDebugging` class, right-click the `test` keyword and select **Create RemoteClusterDebugging Configuration**.



13. In the dialog box, provide a name for the configuration, and select the **Test kind** as **Test name**. Leave all other values as default, click **Apply**, and then click **OK**.

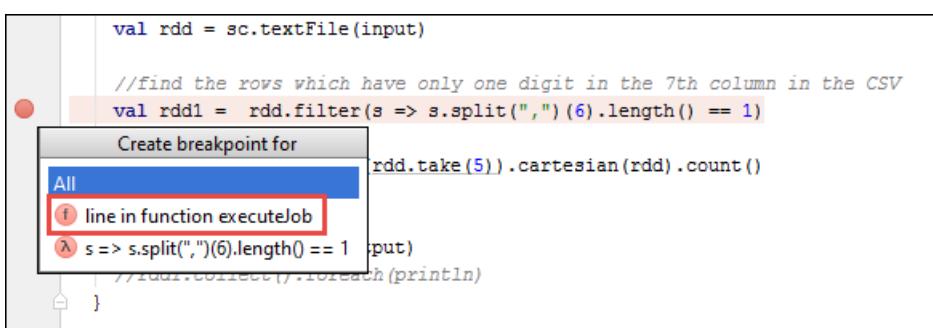


14. You should now see a **Remote Run** configuration drop-down in the menu bar.

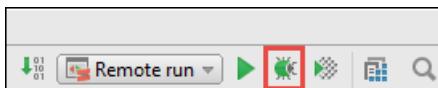


Step 5: Run the application in debug mode

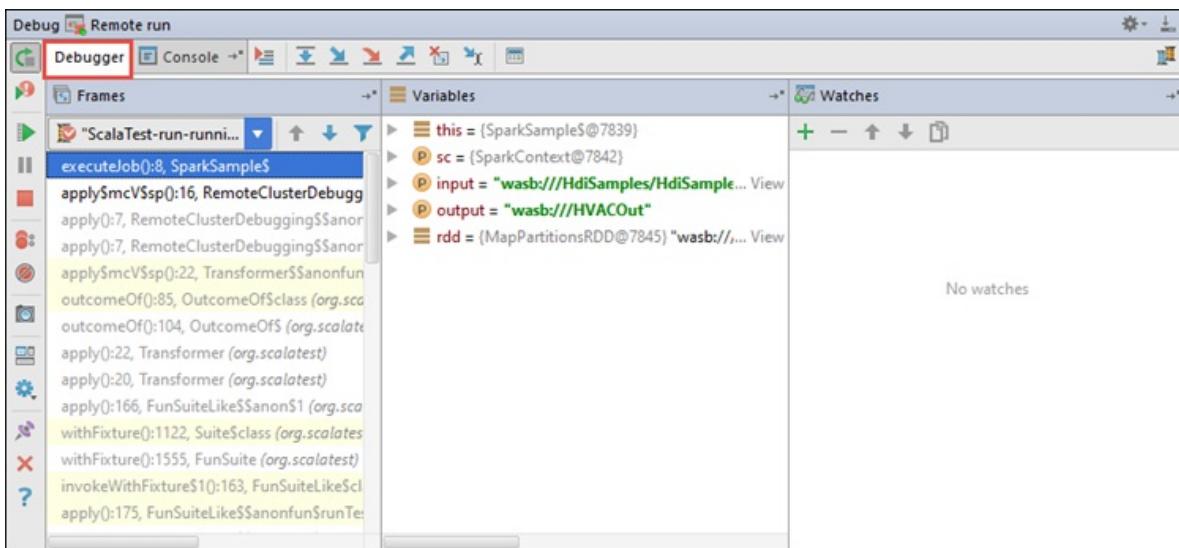
- In your IntelliJ IDEA project, open `SparkSample.scala` and create a breakpoint next to `'val rdd1'`. In the pop-up menu for creating a breakpoint, select **line in function executeJob**.



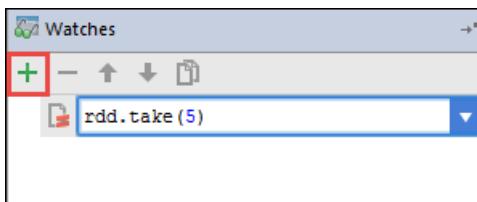
- Click the **Debug Run** button next to the **Remote Run** configuration drop-down to start running the application.



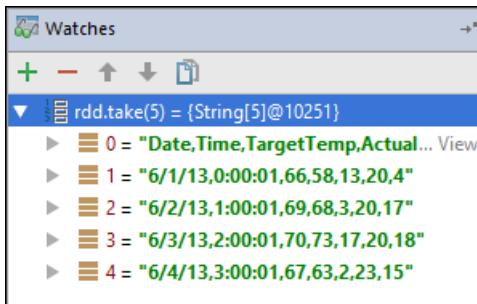
3. When the program execution reaches the breakpoint, you should see a **Debugger** tab in the bottom pane.



4. Click the (+) icon to add a watch as shown in the image below.

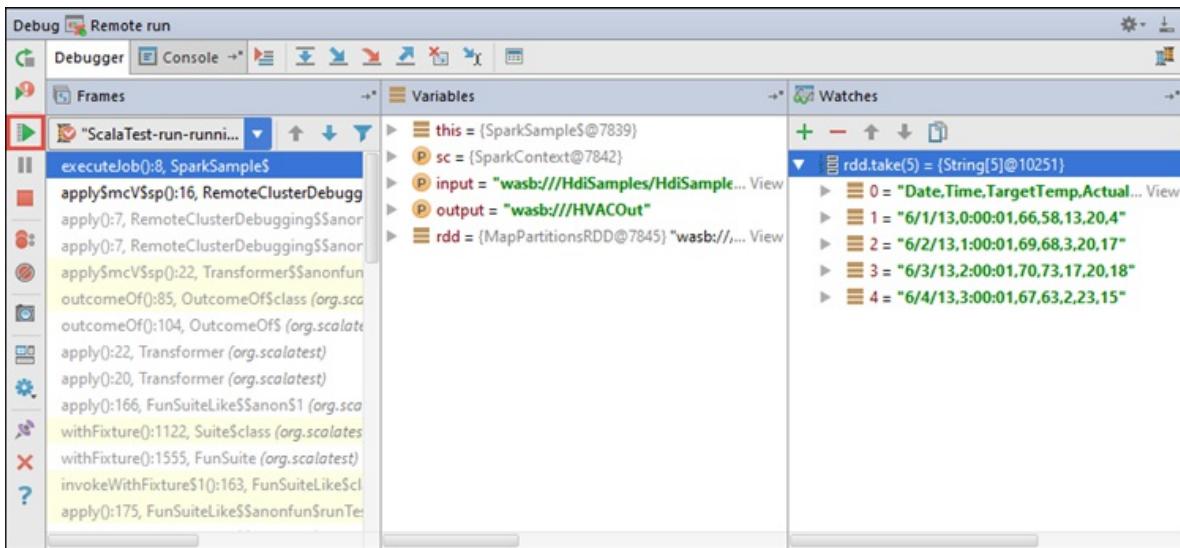


Here, because the application broke before the variable `rdd1` was created, using this watch we can see what are the first 5 rows in the variable `rdd`. Press **ENTER**.

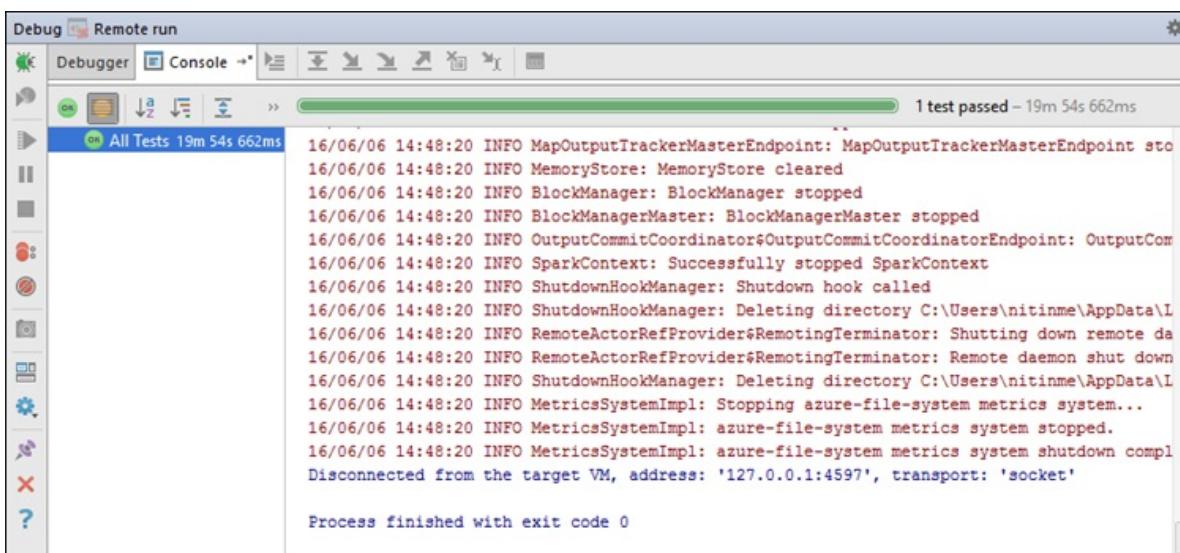


What you see in the image above is that at runtime, you could query terabytes of data and debug how your application progresses. For example, in the output shown in the image above, you can see that the first row of the output is a header. Based on this, you can modify your application code to skip the header row if required.

5. You can now click the **Resume Program** icon to proceed with your application run.



- If the application completes successfully, you should see an output like the following.



See also

- [Overview: Apache Spark on Azure HDInsight](#)

Demo

- Create Scala Project (Video): [Create Spark Scala Applications](#)
- Remote Debug (Video): [Use Azure Toolkit for IntelliJ to debug Spark applications remotely on HDInsight Cluster](#)

Scenarios

- Spark with BI: Perform interactive data analysis using Spark in HDInsight with BI tools
- Spark with Machine Learning: Use Spark in HDInsight for analyzing building temperature using HVAC data
- Spark with Machine Learning: Use Spark in HDInsight to predict food inspection results
- Spark Streaming: Use Spark in HDInsight for building real-time streaming applications
- Website log analysis using Spark in HDInsight

Create and run applications

- [Create a standalone application using Scala](#)
- [Run jobs remotely on a Spark cluster using Livy](#)

Tools and extensions

- Use HDInsight Tools in Azure Toolkit for IntelliJ to create and submit Spark Scala applications
- Use Azure Toolkit for IntelliJ to debug Spark applications remotely through SSH
- Use HDInsight Tools for IntelliJ with Hortonworks Sandbox
- Use HDInsight Tools in Azure Toolkit for Eclipse to create Spark applications
- Use Zeppelin notebooks with a Spark cluster on HDInsight
- Kernels available for Jupyter notebook in Spark cluster for HDInsight
- Use external packages with Jupyter notebooks
- Install Jupyter on your computer and connect to an HDInsight Spark cluster

Manage resources

- Manage resources for the Apache Spark cluster in Azure HDInsight
- Track and debug jobs running on an Apache Spark cluster in HDInsight

Analyze Application Insights telemetry logs with Spark on HDInsight

8/16/2017 • 12 min to read • [Edit Online](#)

Learn how to use Spark on HDInsight to analyze Application Insight telemetry data.

[Visual Studio Application Insights](#) is an analytics service that monitors your web applications. Telemetry data generated by Application Insights can be exported to Azure Storage. Once the data is in Azure Storage, HDInsight can be used to analyze it.

Prerequisites

- An application that is configured to use Application Insights.
- Familiarity with creating a Linux-based HDInsight cluster. For more information, see [Create Spark on HDInsight](#).

IMPORTANT

The steps in this document require an HDInsight cluster that uses Linux. Linux is the only operating system used on HDInsight version 3.4 or greater. For more information, see [HDInsight retirement on Windows](#).

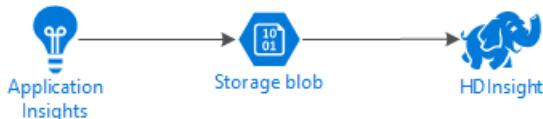
- A web browser.

The following resources were used in developing and testing this document:

- Application Insights telemetry data was generated using a [Node.js web app configured to use Application Insights](#).
- A Linux-based Spark on HDInsight cluster version 3.5 was used to analyze the data.

Architecture and planning

The following diagram illustrates the service architecture of this example:



Azure storage

Application Insights can be configured to continuously export telemetry information to blobs. HDInsight can then read data stored in the blobs. However, there are some requirements that you must follow:

- **Location:** If the Storage Account and HDInsight are in different locations, it may increase latency. It also increases cost, as egress charges are applied to data moving between regions.

WARNING

Using a Storage Account in a different location than HDInsight is not supported.

- **Blob type:** HDInsight only supports block blobs. Application Insights defaults to using block blobs, so should

work by default with HDInsight.

For information on adding additional storage to an existing HDInsight cluster, see the [Add additional storage accounts](#) document.

Data schema

Application Insights provides [export data model](#) information for the telemetry data format exported to blobs. The steps in this document use Spark SQL to work with the data. Spark SQL can automatically generate a schema for the JSON data structure logged by Application Insights.

Export telemetry data

Follow the steps in [Configure Continuous Export](#) to configure your Application Insights to export telemetry information to an Azure storage blob.

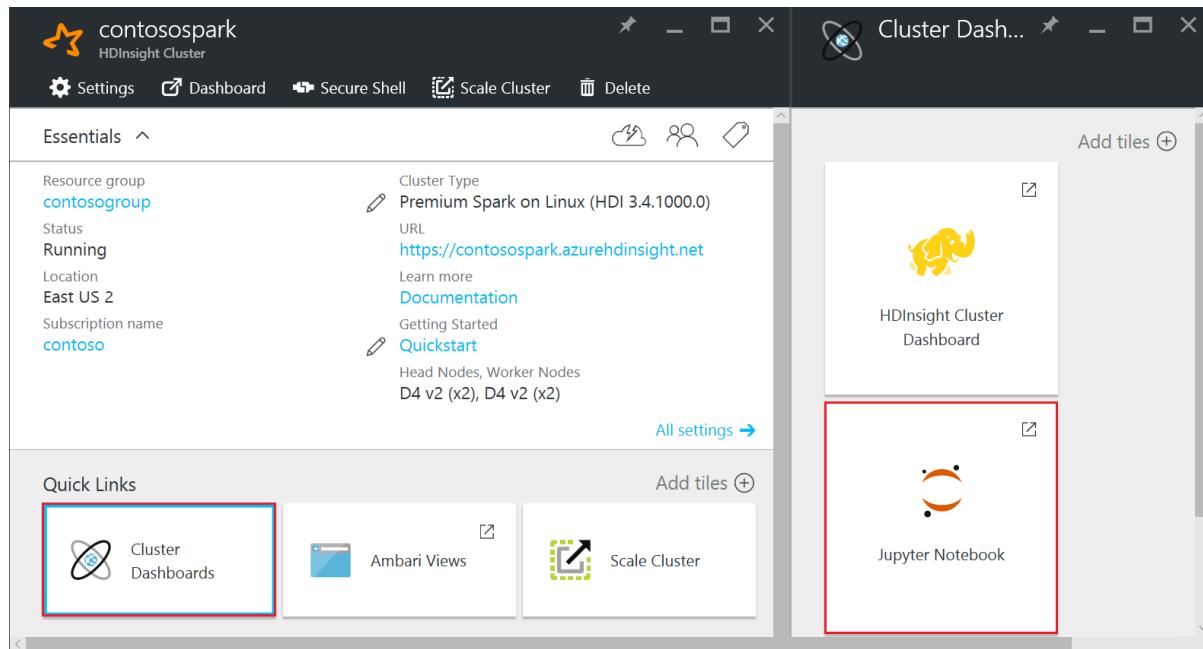
Configure HDInsight to access the data

If you are creating an HDInsight cluster, add the storage account during cluster creation.

To add the Azure Storage Account to an existing cluster, use the information in the [Add additional Storage Accounts](#) document.

Analyze the data: PySpark

- From the [Azure portal](#), select your Spark on HDInsight cluster. From the **Quick Links** section, select **Cluster Dashboards**, and then select **Jupyter Notebook** from the Cluster Dashboard blade.



- In the upper right corner of the Jupyter page, select **New**, and then **PySpark**. A new browser tab containing a Python-based Jupyter Notebook opens.
- In the first field (called a **cell**) on the page, enter the following text:

```
sc._jsc.hadoopConfiguration().set('mapreduce.input.fileinputformat.input.dir.recursive', 'true')
```

This code configures Spark to recursively access the directory structure for the input data. Application Insights telemetry is logged to a directory structure similar to the `/{telemetry type}/YYYY-MM-DD/{##}/`.

- Use **SHIFT+ENTER** to run the code. On the left side of the cell, an '*' appears between the brackets to

indicate that the code in this cell is being executed. Once it completes, the '*' changes to a number, and output similar to the following text is displayed below the cell:

```
Creating SparkContext as 'sc'

ID      YARN Application ID      Kind      State      Spark UI      Driver log      Current session?
3      application_1468969497124_0001      pyspark      idle      Link      Link      ✓

Creating HiveContext as 'sqlContext'
SparkContext and HiveContext created. Executing user code ...
```

5. A new cell is created below the first one. Enter the following text in the new cell. Replace `CONTAINER` and `STORAGEACCOUNT` with the Azure storage account name and blob container name that contains Application Insights data.

```
%%bash
hdfs dfs -ls wasb://CONTAINER@STORAGEACCOUNT.blob.core.windows.net/
```

Use **SHIFT+ENTER** to execute this cell. You see a result similar to the following text:

```
Found 1 items
drwxrwxrwx -          0 1970-01-01 00:00
wasb://appinsights@contosostore.blob.core.windows.net/contosoappinsights_2bededa61bc741fbdee6b556571a483
1
```

The wasb path returned is the location of the Application Insights telemetry data. Change the `hdfs dfs -ls` line in the cell to use the wasb path returned, and then use **SHIFT+ENTER** to run the cell again. This time, the results should display the directories that contain telemetry data.

NOTE

For the remainder of the steps in this section, the `wasb://appinsights@contosostore.blob.core.windows.net/contosoappinsights_{ID}/Requests` directory was used. Your directory structure may be different.

6. In the next cell, enter the following code: Replace `WASB_PATH` with the path from the previous step.

```
jsonFiles = sc.textFile('WASB_PATH')
jsonData = sqlContext.read.json(jsonFiles)
```

This code creates a dataframe from the JSON files exported by the continuous export process. Use **SHIFT+ENTER** to run this cell.

7. In the next cell, enter and run the following to view the schema that Spark created for the JSON files:

```
jsonData.printSchema()
```

The schema for each type of telemetry is different. The following example is the schema that is generated for web requests (data stored in the `Requests` subdirectory):

```

root
|-- context: struct (nullable = true)
|   |-- application: struct (nullable = true)
|   |   |-- version: string (nullable = true)
|   |-- custom: struct (nullable = true)
|   |   |-- dimensions: array (nullable = true)
|   |   |   |-- element: string (containsNull = true)
|   |-- metrics: array (nullable = true)
|   |   |-- element: string (containsNull = true)
|-- data: struct (nullable = true)
|   |-- eventTime: string (nullable = true)
|   |-- isSynthetic: boolean (nullable = true)
|   |-- samplingRate: double (nullable = true)
|   |-- syntheticSource: string (nullable = true)
|-- device: struct (nullable = true)
|   |-- browser: string (nullable = true)
|   |-- browserVersion: string (nullable = true)
|   |-- deviceModel: string (nullable = true)
|   |-- deviceName: string (nullable = true)
|   |-- id: string (nullable = true)
|   |-- osVersion: string (nullable = true)
|   |-- type: string (nullable = true)
|-- location: struct (nullable = true)
|   |-- city: string (nullable = true)
|   |-- clientip: string (nullable = true)
|   |-- continent: string (nullable = true)
|   |-- country: string (nullable = true)
|   |-- province: string (nullable = true)
|-- operation: struct (nullable = true)
|   |-- name: string (nullable = true)
|-- session: struct (nullable = true)
|   |-- id: string (nullable = true)
|   |-- isFirst: boolean (nullable = true)
|-- user: struct (nullable = true)
|   |-- anonId: string (nullable = true)
|   |-- isAuthenticated: boolean (nullable = true)
|-- internal: struct (nullable = true)
|   |-- data: struct (nullable = true)
|   |   |-- documentVersion: string (nullable = true)
|   |   |-- id: string (nullable = true)
|-- request: array (nullable = true)
|   |-- element: struct (containsNull = true)
|   |   |-- count: long (nullable = true)
|   |   |-- durationMetric: struct (nullable = true)
|   |   |   |-- count: double (nullable = true)
|   |   |   |-- max: double (nullable = true)
|   |   |   |-- min: double (nullable = true)
|   |   |   |-- sampledValue: double (nullable = true)
|   |   |   |-- stdDev: double (nullable = true)
|   |   |   |-- value: double (nullable = true)
|   |   |-- id: string (nullable = true)
|   |   |-- name: string (nullable = true)
|   |   |-- responseCode: long (nullable = true)
|   |   |-- success: boolean (nullable = true)
|   |   |-- url: string (nullable = true)
|   |   |-- urlData: struct (nullable = true)
|   |   |   |-- base: string (nullable = true)
|   |   |   |-- hashTag: string (nullable = true)
|   |   |   |-- host: string (nullable = true)
|   |   |   |-- protocol: string (nullable = true)

```

8. Use the following to register the dataframe as a temporary table and run a query against the data:

```
jsonData.registerTempTable("requests")
df = sqlContext.sql("select context.location.city from requests where context.location.city is not
null")
df.show()
```

This query returns the city information for the top 20 records where context.location.city is not null.

NOTE

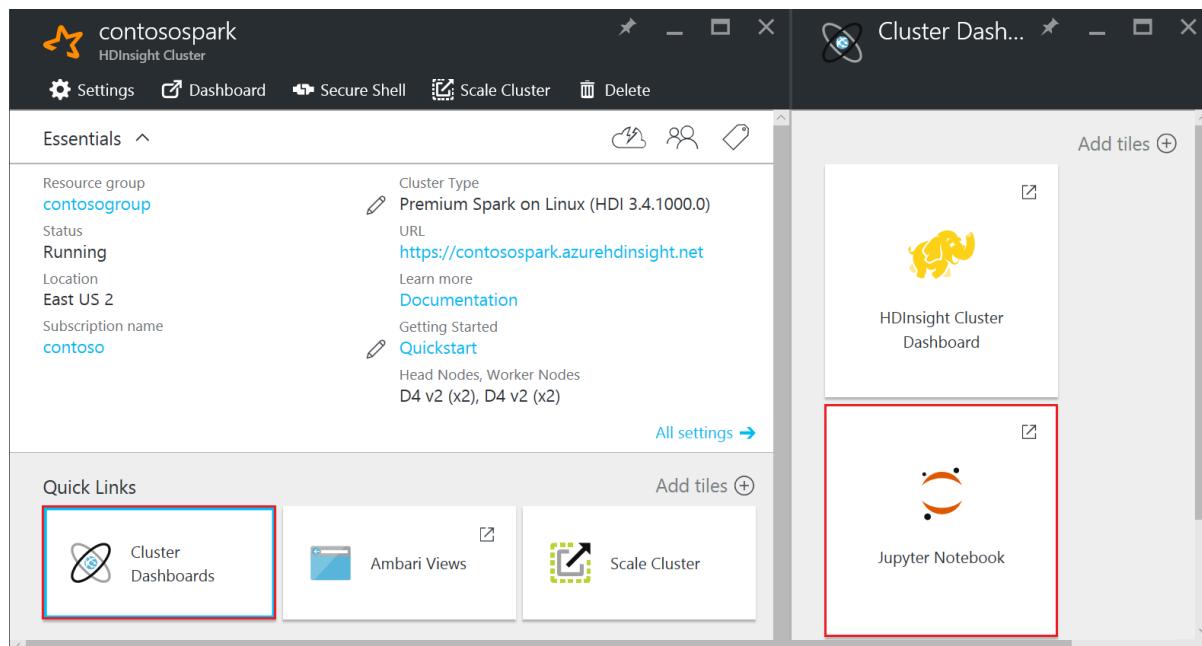
The context structure is present in all telemetry logged by Application Insights. The city element may not be populated in your logs. Use the schema to identify other elements that you can query that may contain data for your logs.

This query returns information similar to the following text:

```
+-----+
|    city|
+-----+
| Bellevue|
| Redmond|
| Seattle|
|Charlotte|
...
+-----+
```

Analyze the data: Scala

- From the [Azure portal](#), select your Spark on HDInsight cluster. From the **Quick Links** section, select **Cluster Dashboards**, and then select **Jupyter Notebook** from the Cluster Dashboard blade.



- In the upper right corner of the Jupyter page, select **New**, and then **Scala**. A new browser tab containing a Scala-based Jupyter Notebook appears.
- In the first field (called a **cell**) on the page, enter the following text:

```
sc.hadoopConfiguration.set("mapreduce.input.fileinputformat.input.dir.recursive", "true")
```

This code configures Spark to recursively access the directory structure for the input data. Application Insights telemetry is logged to a directory structure similar to `/{{telemetry type}}/{{YYYY-MM-DD}}/{{##}}/`.

4. Use **SHIFT+ENTER** to run the code. On the left side of the cell, an '*' appears between the brackets to indicate that the code in this cell is being executed. Once it completes, the '*' changes to a number, and output similar to the following text is displayed below the cell:

```
Creating SparkContext as 'sc'

ID      YARN Application ID      Kind      State      Spark UI      Driver log      Current session?
3      application_1468969497124_0001      spark      idle      Link      Link      ✓

Creating HiveContext as 'sqlContext'
SparkContext and HiveContext created. Executing user code ...
```

5. A new cell is created below the first one. Enter the following text in the new cell. Replace `CONTAINER` and `STORAGEACCOUNT` with the Azure storage account name and blob container name that contains Application Insights logs.

```
%%bash
hdfs dfs -ls wasb://CONTAINER@STORAGEACCOUNT.blob.core.windows.net/
```

Use **SHIFT+ENTER** to execute this cell. You see a result similar to the following text:

```
Found 1 items
drwxrwxrwx -          0 1970-01-01 00:00
wasb://appinsights@contosostore.blob.core.windows.net/contosoappinsights_2bededa61bc741fbdee6b556571a483
1
```

The wasb path returned is the location of the Application Insights telemetry data. Change the `hdfs dfs -ls` line in the cell to use the wasb path returned, and then use **SHIFT+ENTER** to run the cell again. This time, the results should display the directories that contain telemetry data.

NOTE

For the remainder of the steps in this section, the

`wasb://appinsights@contosostore.blob.core.windows.net/contosoappinsights_{ID}/Requests` directory was used. This directory may not exist unless your telemetry data is for a web app.

6. In the next cell, enter the following code: Replace `WASB_PATH` with the path from the previous step.

```
var jsonFiles = sc.textFile('WASB_PATH')
val sqlContext = new org.apache.spark.sql.SQLContext(sc)
var jsonData = sqlContext.read.json(jsonFiles)
```

This code creates a dataframe from the JSON files exported by the continuous export process. Use **SHIFT+ENTER** to run this cell.

7. In the next cell, enter and run the following to view the schema that Spark created for the JSON files:

```
jsonData.printSchema
```

The schema for each type of telemetry is different. The following example is the schema that is generated for

web requests (data stored in the `Requests` subdirectory):

```
root
|-- context: struct (nullable = true)
|   |-- application: struct (nullable = true)
|   |   |-- version: string (nullable = true)
|   |-- custom: struct (nullable = true)
|   |   |-- dimensions: array (nullable = true)
|   |   |   |-- element: string (containsNull = true)
|   |   |-- metrics: array (nullable = true)
|   |   |   |-- element: string (containsNull = true)
|   |-- data: struct (nullable = true)
|   |   |-- eventTime: string (nullable = true)
|   |   |-- isSynthetic: boolean (nullable = true)
|   |   |-- samplingRate: double (nullable = true)
|   |   |-- syntheticSource: string (nullable = true)
|   |-- device: struct (nullable = true)
|   |   |-- browser: string (nullable = true)
|   |   |-- browserVersion: string (nullable = true)
|   |   |-- deviceModel: string (nullable = true)
|   |   |-- deviceName: string (nullable = true)
|   |   |-- id: string (nullable = true)
|   |   |-- osVersion: string (nullable = true)
|   |   |-- type: string (nullable = true)
|   |-- location: struct (nullable = true)
|   |   |-- city: string (nullable = true)
|   |   |-- clientip: string (nullable = true)
|   |   |-- continent: string (nullable = true)
|   |   |-- country: string (nullable = true)
|   |   |-- province: string (nullable = true)
|   |-- operation: struct (nullable = true)
|   |   |-- name: string (nullable = true)
|   |-- session: struct (nullable = true)
|   |   |-- id: string (nullable = true)
|   |   |-- isFirst: boolean (nullable = true)
|   |-- user: struct (nullable = true)
|   |   |-- anonId: string (nullable = true)
|   |   |-- isAuthenticated: boolean (nullable = true)
|-- internal: struct (nullable = true)
|   |-- data: struct (nullable = true)
|   |   |-- documentVersion: string (nullable = true)
|   |   |-- id: string (nullable = true)
|-- request: array (nullable = true)
|   |-- element: struct (containsNull = true)
|   |   |-- count: long (nullable = true)
|   |   |-- durationMetric: struct (nullable = true)
|   |   |   |-- count: double (nullable = true)
|   |   |   |-- max: double (nullable = true)
|   |   |   |-- min: double (nullable = true)
|   |   |   |-- sampledValue: double (nullable = true)
|   |   |   |-- stdDev: double (nullable = true)
|   |   |   |-- value: double (nullable = true)
|   |   |-- id: string (nullable = true)
|   |   |-- name: string (nullable = true)
|   |   |-- responseCode: long (nullable = true)
|   |   |-- success: boolean (nullable = true)
|   |   |-- url: string (nullable = true)
|   |   |-- urlData: struct (nullable = true)
|   |   |   |-- base: string (nullable = true)
|   |   |   |-- hashTag: string (nullable = true)
|   |   |   |-- host: string (nullable = true)
|   |   |   |-- protocol: string (nullable = true)
```

8. Use the following to register the dataframe as a temporary table and run a query against the data:

```
jsonData.registerTempTable("requests")
var city = sqlContext.sql("select context.location.city from requests where context.location.city is not
null limit 10").show()
```

This query returns the city information for the top 20 records where context.location.city is not null.

NOTE

The context structure is present in all telemetry logged by Application Insights. The city element may not be populated in your logs. Use the schema to identify other elements that you can query that may contain data for your logs.

This query returns information similar to the following text:

```
+-----+
|   city|
+-----+
| Bellevue|
| Redmond|
| Seattle|
|Charlotte|
...
+-----+
```

Next steps

For more examples of using Spark to work with data and services in Azure, see the following documents:

- [Spark with BI: Perform interactive data analysis using Spark in HDInsight with BI tools](#)
- [Spark with Machine Learning: Use Spark in HDInsight for analyzing building temperature using HVAC data](#)
- [Spark with Machine Learning: Use Spark in HDInsight to predict food inspection results](#)
- [Spark Streaming: Use Spark in HDInsight for building streaming applications](#)
- [Website log analysis using Spark in HDInsight](#)

For information on creating and running Spark applications, see the following documents:

- [Create a standalone application using Scala](#)
- [Run jobs remotely on a Spark cluster using Livy](#)

Analyze website logs using a custom Python library with Spark cluster on HDInsight

8/16/2017 • 7 min to read • [Edit Online](#)

This notebook demonstrates how to analyze log data using a custom library with Spark on HDInsight. The custom library we use is a Python library called **iislogparser.py**.

TIP

This tutorial is also available as a Jupyter notebook on a Spark (Linux) cluster that you create in HDInsight. The notebook experience lets you run the Python snippets from the notebook itself. To perform the tutorial from within a notebook, create a Spark cluster, launch a Jupyter notebook (<https://CLUSTERNAME.azurehdinsight.net/jupyter>), and then run the notebook **Analyze logs with Spark using a custom library.ipynb** under the **PySpark** folder.

Prerequisites:

You must have the following:

- An Azure subscription. See [Get Azure free trial](#).
- An Apache Spark cluster on HDInsight. For instructions, see [Create Apache Spark clusters in Azure HDInsight](#).

Save raw data as an RDD

In this section, we use the [Jupyter](#) notebook associated with an Apache Spark cluster in HDInsight to run jobs that process your raw sample data and save it as a Hive table. The sample data is a .csv file (hvac.csv) available on all clusters by default.

Once your data is saved as a Hive table, in the next section we will connect to the Hive table using BI tools such as Power BI and Tableau.

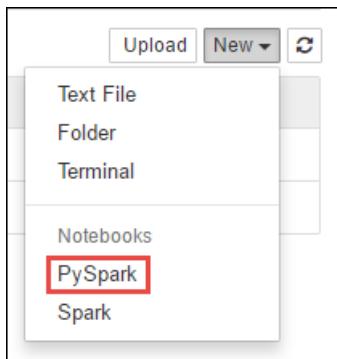
1. From the [Azure portal](#), from the startboard, click the tile for your Spark cluster (if you pinned it to the startboard). You can also navigate to your cluster under **Browse All > HDInsight Clusters**.
2. From the Spark cluster blade, click **Cluster Dashboard**, and then click **Jupyter Notebook**. If prompted, enter the admin credentials for the cluster.

NOTE

You may also reach the Jupyter Notebook for your cluster by opening the following URL in your browser. Replace **CLUSTERNAME** with the name of your cluster:

<https://CLUSTERNAME.azurehdinsight.net/jupyter>

3. Create a new notebook. Click **New**, and then click **PySpark**.



4. A new notebook is created and opened with the name Untitled.pynb. Click the notebook name at the top, and enter a friendly name.



5. Because you created a notebook using the PySpark kernel, you do not need to create any contexts explicitly. The Spark and Hive contexts will be automatically created for you when you run the first code cell. You can start by importing the types that are required for this scenario. Paste the following snippet in an empty cell, and then press **SHIFT + ENTER**.

```
from pyspark.sql import Row
from pyspark.sql.types import *
```

6. Create an RDD using the sample log data already available on the cluster. You can access the data in the default storage account associated with the cluster at
\HdiSamples\HdiSamples\WebsiteLogSampleData\SampleLog\909f2b.log.

```
logs = sc.textFile('wasb:///HdiSamples/HdiSamples/WebsiteLogSampleData/SampleLog/909f2b.log')
```

7. Retrieve a sample log set to verify that the previous step completed successfully.

```
logs.take(5)
```

You should see an output similar to the following:

```

# -----
# THIS IS AN OUTPUT
# -----


[u'#Software: Microsoft Internet Information Services 8.0',
 u'#Fields: date time s-sitename cs-method cs-uri-stem cs-uri-query s-port cs-username c-ip cs(User-Agent) cs(Cookie) cs(Referer) cs-host sc-status sc-substatus sc-win32-status sc-bytes cs-bytes time-taken',
 u'2014-01-01 02:01:09 SAMPLEWEBSITE GET /blogposts/mvc4/step2.png X-ARR-LOG-ID=2ec4b8ad-3cf0-4442-93ab-837317ece6a1 80 - 1.54.23.196 Mozilla/5.0+(Windows+NT+6.3;+WOW64)+AppleWebKit/537.36+(KHTML,+like+Gecko)+Chrome/31.0.1650.63+Safari/537.36 -
http://weblogs.asp.net/sample/archive/2007/12/09/asp-net-mvc-framework-part-4-handling-form-edit-and-post-scenarios.aspx www.sample.com 200 0 0 53175 871 46',
 u'2014-01-01 02:01:09 SAMPLEWEBSITE GET /blogposts/mvc4/step3.png X-ARR-LOG-ID=9eace870-2f49-4efd-b204-0d170da46b4a 80 - 1.54.23.196 Mozilla/5.0+(Windows+NT+6.3;+WOW64)+AppleWebKit/537.36+(KHTML,+like+Gecko)+Chrome/31.0.1650.63+Safari/537.36 -
http://weblogs.asp.net/sample/archive/2007/12/09/asp-net-mvc-framework-part-4-handling-form-edit-and-post-scenarios.aspx www.sample.com 200 0 0 51237 871 32',
 u'2014-01-01 02:01:09 SAMPLEWEBSITE GET /blogposts/mvc4/step4.png X-ARR-LOG-ID=4bea5b3d-8ac9-46c9-9b8c-ec3e9500cbea 80 - 1.54.23.196 Mozilla/5.0+(Windows+NT+6.3;+WOW64)+AppleWebKit/537.36+(KHTML,+like+Gecko)+Chrome/31.0.1650.63+Safari/537.36 -
http://weblogs.asp.net/sample/archive/2007/12/09/asp-net-mvc-framework-part-4-handling-form-edit-and-post-scenarios.aspx www.sample.com 200 0 0 72177 871 47']

```

Analyze log data using a custom Python library

- In the output above, the first couple lines include the header information and each remaining line matches the schema described in that header. Parsing such logs could be complicated. So, we use a custom Python library (**iislogparser.py**) that makes parsing such logs much easier. By default, this library is included with your Spark cluster on HDInsight at [/HdiSamples/HdiSamples/WebsiteLogSampleData/iislogparser.py](#).

However, this library is not in the `PYTHONPATH` so we cannot use it by using an import statement like `import iislogparser`. To use this library, we must distribute it to all the worker nodes. Run the following snippet.

```
sc.addPyFile('wasb:///HdiSamples/HdiSamples/WebsiteLogSampleData/iislogparser.py')
```

- `iislogparser` provides a function `parse_log_line` that returns `None` if a log line is a header row, and returns an instance of the `LogLine` class if it encounters a log line. Use the `LogLine` class to extract only the log lines from the RDD:

```

def parse_line(l):
    import iislogparser
    return iislogparser.parse_log_line(l)
logLines = logs.map(parse_line).filter(lambda p: p is not None).cache()

```

- Retrieve a couple of extracted log lines to verify that the step completed successfully.

```
logLines.take(2)
```

The output should be similar to the following:

```

# -----
# THIS IS AN OUTPUT
# -----


[2014-01-01 02:01:09 SAMPLEWEBSITE GET /blogposts/mvc4/step2.png X-ARR-LOG-ID=2ec4b8ad-3cf0-4442-93ab-
837317ece6a1 80 - 1.54.23.196 Mozilla/5.0+(Windows+NT+6.3;+WOW64)+AppleWebKit/537.36+
(KHTML,+like+Gecko)+Chrome/31.0.1650.63+Safari/537.36 -
http://weblogs.asp.net/sample/archive/2007/12/09/asp-net-mvc-framework-part-4-handling-form-edit-and-
post-scenarios.aspx www.sample.com 200 0 0 53175 871 46,
2014-01-01 02:01:09 SAMPLEWEBSITE GET /blogposts/mvc4/step3.png X-ARR-LOG-ID=9eace870-2f49-4efd-b204-
0d170da46b4a 80 - 1.54.23.196 Mozilla/5.0+(Windows+NT+6.3;+WOW64)+AppleWebKit/537.36+
(KHTML,+like+Gecko)+Chrome/31.0.1650.63+Safari/537.36 -
http://weblogs.asp.net/sample/archive/2007/12/09/asp-net-mvc-framework-part-4-handling-form-edit-and-
post-scenarios.aspx www.sample.com 200 0 0 51237 871 32]

```

4. The `LogLine` class, in turn, has some useful methods, like `is_error()`, which returns whether a log entry has an error code. Use this to compute the number of errors in the extracted log lines, and then log all the errors to a different file.

```

errors = logLines.filter(lambda p: p.is_error())
numLines = logLines.count()
numErrors = errors.count()
print 'There are', numErrors, 'errors and', numLines, 'log entries'
errors.map(lambda p:
str(p)).saveAsTextFile('wasb:///HdiSamples/HdiSamples/WebsiteLogSampleData/SampleLog/909f2b-2.log')

```

You should see an output like the following:

```

# -----
# THIS IS AN OUTPUT
# -----


There are 30 errors and 646 log entries

```

5. You can also use **Matplotlib** to construct a visualization of the data. For example, if you want to isolate the cause of requests that run for a long time, you might want to find the files that take the most time to serve on average. The snippet below retrieves the top 25 resources that took most time to serve a request.

```

def avgTimeTakenByKey(rdd):
    return rdd.combineByKey(lambda line: (line.time_taken, 1),
                           lambda x, line: (x[0] + line.time_taken, x[1] + 1),
                           lambda x, y: (x[0] + y[0], x[1] + y[1]))
    .map(lambda x: (x[0], float(x[1][0]) / float(x[1][1])))

avgTimeTakenByKey(logLines.map(lambda p: (p.cs_uri_stem, p))).top(25, lambda x: x[1])

```

You should see an output like the following:

```

# -----
# THIS IS AN OUTPUT
# -----


[(u'/blogposts/mvc4/step13.png', 197.5),
 (u'/blogposts/mvc2/step10.jpg', 179.5),
 (u'/blogposts/extractusercontrol/step5.png', 170.0),
 (u'/blogposts/mvc4/step8.png', 159.0),
 (u'/blogposts/mvcrouting/step22.jpg', 155.0),
 (u'/blogposts/mvcrouting/step3.jpg', 152.0),
 (u'/blogposts/linqsproc1/step16.jpg', 138.75),
 (u'/blogposts/linqsproc1/step26.jpg', 137.3333333333334),
 (u'/blogposts/vs2008javascript/step10.jpg', 127.0),
 (u'/blogposts/nested/step2.jpg', 126.0),
 (u'/blogposts/adminpack/step1.png', 124.0),
 (u'/BlogPosts/datalistpaging/step2.png', 118.0),
 (u'/blogposts/mvc4/step35.png', 117.0),
 (u'/blogposts/mvcrouting/step2.jpg', 116.5),
 (u'/blogposts/aboutme/basketball.jpg', 109.0),
 (u'/blogposts/anonymoustypes/step11.jpg', 109.0),
 (u'/blogposts/mvc4/step12.png', 106.0),
 (u'/blogposts/linq8/step0.jpg', 105.5),
 (u'/blogposts/mvc2/step18.jpg', 104.0),
 (u'/blogposts/mvc2/step11.jpg', 104.0),
 (u'/blogposts/mvcrouting/step1.jpg', 104.0),
 (u'/blogposts/extractusercontrol/step1.png', 103.0),
 (u'/blogposts/sqlvideos/sqlvideos.jpg', 102.0),
 (u'/blogposts/mvcrouting/step21.jpg', 101.0),
 (u'/blogposts/mvc4/step1.png', 98.0)]

```

6. You can also present this information in the form of plot. As a first step to create a plot, let us first create a temporary table **AverageTime**. The table groups the logs by time to see if there were any unusual latency spikes at any particular time.

```

avgTimeTakenByMinute = avgTimeTakenByKey(logLines.map(lambda p: (p.datetime.minute, p))).sortByKey()
schema = StructType([StructField('Minutes', IntegerType(), True),
                     StructField('Time', FloatType(), True)])

avgTimeTakenByMinuteDF = sqlContext.createDataFrame(avgTimeTakenByMinute, schema)
avgTimeTakenByMinuteDF.registerTempTable('AverageTime')

```

7. You can then run the following SQL query to get all the records in the **AverageTime** table.

```

%%sql -o averagetime
SELECT * FROM AverageTime

```

The `%%sql` magic followed by `-o averagetime` ensures that the output of the query is persisted locally on the Jupyter server (typically the headnode of the cluster). The output is persisted as a [Pandas](#) dataframe with the specified name **averagetime**.

You should see an output like the following:

Type:

Minutes	Time
1	52.333332
2	66.125000
3	60.000000
4	45.062500
5	43.000000
6	35.750000
7	48.500000
8	57.153847
9	37.500000

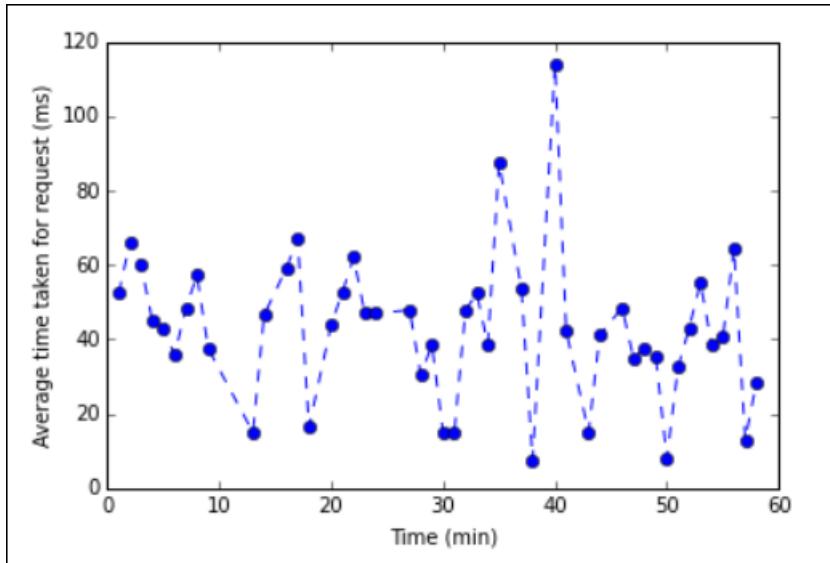
For more information about the `%%sql` magic, see [Parameters supported with the %%sql magic](#).

8. You can now use Matplotlib, a library used to construct visualization of data, to create a plot. Because the plot must be created from the locally persisted **averagetime** dataframe, the code snippet must begin with the `%%local` magic. This ensures that the code is run locally on the Jupyter server.

```
%%local
%matplotlib inline
import matplotlib.pyplot as plt

plt.plot(averagetime['Minutes'], averagetime['Time'], marker='o', linestyle='--')
plt.xlabel('Time (min)')
plt.ylabel('Average time taken for request (ms)')
```

You should see an output like the following:



9. After you have finished running the application, you should shutdown the notebook to release the resources. To do so, from the **File** menu on the notebook, click **Close and Halt**. This will shutdown and close the notebook.

See also

- [Overview: Apache Spark on Azure HDInsight](#)

Scenarios

- Spark with BI: Perform interactive data analysis using Spark in HDInsight with BI tools
- Spark with Machine Learning: Use Spark in HDInsight for analyzing building temperature using HVAC data
- Spark with Machine Learning: Use Spark in HDInsight to predict food inspection results
- Spark Streaming: Use Spark in HDInsight for building real-time streaming applications

Create and run applications

- Create a standalone application using Scala
- Run jobs remotely on a Spark cluster using Livy

Tools and extensions

- Use HDInsight Tools Plugin for IntelliJ IDEA to create and submit Spark Scala applications
- Use HDInsight Tools Plugin for IntelliJ IDEA to debug Spark applications remotely
- Use Zeppelin notebooks with a Spark cluster on HDInsight
- Kernels available for Jupyter notebook in Spark cluster for HDInsight
- Use external packages with Jupyter notebooks
- Install Jupyter on your computer and connect to an HDInsight Spark cluster

Manage resources

- Manage resources for the Apache Spark cluster in Azure HDInsight
- Track and debug jobs running on an Apache Spark cluster in HDInsight

Creating Spark ML Pipelines

8/16/2017 • 4 min to read • [Edit Online](#)

Overview

The Apache Spark [spark.ml](#) package provides a uniform set of high-level APIs built on top of data frames that can help you create and tune practical machine learning pipelines. MLlib is Spark's scalable machine learning library, which brings modeling capabilities to this distributed environment. The moniker "Spark ML" is a term that refers to the MLlib DataFrame-based API, as opposed to the older RDD-based pipeline API, which is now in maintenance mode.

The primary function of an ML pipeline is to create a complete workflow by combining multiple machine learning algorithms together. There are oftentimes many steps required to process and learn from data, which requires applying a sequence of algorithms. Pipelines introduce a nice way to define the stages and order of your process. The stages of a pipeline are represented by a sequence of `PipelineStage`s, where a `Transformer` and `Estimator` perform tasks, and are executed in a specific order.

A `Transformer` is an algorithm that transforms one `DataFrame` to another by using the `transform()` method. For example, a feature transformer could read a column of a `DataFrame`, map it to another column, and output a new `DataFrame` with the mapped column appended to it.

`Estimator`s, an abstraction of learning algorithms, are responsible for fitting or training on a `Dataset` to produce a `Transformer`. To do this, an `Estimator` implements a method called `fit()`, which accepts a `DataFrame` and produces a `Model`, which is a `Transformer`.

Each stateless instance of a `Transformer` and `Estimator` has its own unique id, which can be used for specifying parameters. Both use a uniform API for specifying these parameters.

Pipeline example

To demonstrate a practical use of an ML pipeline, we will use the sample HVAC.csv data file that comes pre-loaded on the Azure Storage or Data Lake Store configured as the default storage for your HDInsight cluster. To view the contents of the file, navigate to the following location: [/HdiSamples/HdiSamples/SensorSampleData/hvac](#).

We'll start out with a custom parser to extract the data (HVAC.csv) we want to train our model. The function checks whether the building is "hot" by comparing its actual temperature to the target temperature. We'll store the parsed information within a `LabeledDocument`, which stores the `BuildingID`, `SystemInfo` (comprising the system's Id and age), and the `label` (1.0 if the building is not, 0.0 if not). The last step of this segment is to save the data into a new `DataFrame`.

```

# List the structure of data for better understanding. Because the data will be
# loaded as an array, this structure makes it easy to understand what each element
# in the array corresponds to

# 0 Date
# 1 Time
# 2 TargetTemp
# 3 ActualTemp
# 4 System
# 5 SystemAge
# 6 BuildingID

LabeledDocument = Row("BuildingID", "SystemInfo", "label")

# Define a function that parses the raw CSV file and returns an object of type LabeledDocument

def parseDocument(line):
    values = [str(x) for x in line.split(',')]
    if (values[3] > values[2]):
        hot = 1.0
    else:
        hot = 0.0

    textView = str(values[4]) + " " + str(values[5])

    return LabeledDocument((values[6]), textView, hot)

# Load the raw HVAC.csv file, parse it using the function
data = sc.textFile("wasbs://HdiSamples/HdiSamples/SensorSampleData/hvac/HVAC.csv")

documents = data.filter(lambda s: "Date" not in s).map(parseDocument)
training = documents.toDF()

```

The pipeline we'll be building consists of three stages: `Tokenizer` and `HashingTF` (both `Transformers`), and `Logistic Regression`, which is an `Estimator`.

Our data (extracted from the CSV and mapped to our `LabeledDocument` type), which is a `DataFrame` flows through the pipeline when `pipeline.fit(training)` is called. The first stage, `Tokenizer`, splits the `SystemInfo` input column (consisting of two "words": the system Id and age values) into a "words" output column. This new "words" column is added to the `DataFrame`. The next stage, `HashingTF`, converts the new "words" column into feature vectors. This new column, "features", is added to the `DataFrame`. Remember, these first two stages are `Transformer`s. Since `LogisticRegression` is an `Estimator`, our pipeline calls the `LogisticRegression.fit()` method to produce a `LogisticRegressionModel`.

```

tokenizer = Tokenizer(inputCol="SystemInfo", outputCol="words")
hashingTF = HashingTF(inputCol=tokenizer.getOutputCol(), outputCol="features")
lr = LogisticRegression(maxIter=10, regParam=0.01)

# Build the pipeline with our tokenizer, hashingTF, and logistic regression stages
pipeline = Pipeline(stages=[tokenizer, hashingTF, lr])

model = pipeline.fit(training)

```

To take a look at the new columns added by the `Tokenizer` and `HashingTF` transformers ("words" and "features"), as well as a sample of the `LogisticRegression` estimator, we'll run a `PipelineModel.transform()` method on our original `DataFrame`. * This is just for illustrative purposes. Typically, the next step would be to pass in a test `DataFrame` to validate our training:

```

peek = model.transform(training)
peek.show()

# Outputs the following:
+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+
|BuildingID|SystemInfo|label| words | features | rawPrediction |
|probability|prediction|
+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+
|     4| 13 20| 0.0|[13, 20]|(262144,[250802,2...|[0.11943986671420...|[0.52982451901740...| |
|0.0|      17| 3 20| 0.0|[3, 20]|(262144,[89074,25...|[0.17511205617446...|[0.54366648775222...|
|0.0|      18| 17 20| 1.0|[17, 20]|(262144,[64358,25...|[0.14620993833623...|[0.53648750722548...|
|0.0|      15| 2 23| 0.0|[2, 23]|(262144,[31351,21...|[-0.0361327091023...|[0.49096780538523...|
|1.0|      3| 16 9| 1.0|[16, 9]|(262144,[153779,1...|[-0.0853679939336...|[0.47867095324139...|
|1.0|      4| 13 28| 0.0|[13, 28]|(262144,[69821,25...|[0.14630166986618...|[0.53651031790592...|
|0.0|      2| 12 24| 0.0|[12, 24]|(262144,[187043,2...|[-0.0509556393066...|[0.48726384581522...|
|1.0|      16| 20 26| 1.0|[20, 26]|(262144,[128319,2...|[0.33829638728900...|[0.58377663577684...|
|0.0|      9| 16 9| 1.0|[16, 9]|(262144,[153779,1...|[-0.0853679939336...|[0.47867095324139...|
|1.0|      12| 6 5| 0.0|[6, 5]|(262144,[18659,89...|[0.07513008136562...|[0.51877369045183...|
|0.0|      15| 10 17| 1.0|[10, 17]|(262144,[64358,25...|[-0.0291988646553...|[0.49270080242078...|
|1.0|      7| 2 11| 0.0|[2, 11]|(262144,[212053,2...|[0.03678030020834...|[0.50919403860812...|
|0.0|      15| 14 2| 1.0|[14, 2]|(262144,[109681,2...|[0.06216423725633...|[0.51553605651806...|
|0.0|      6| 3 2| 0.0|[3, 2]|(262144,[89074,21...|[0.00565582077537...|[0.50141395142468...|
|0.0|      20| 19 22| 0.0|[19, 22]|(262144,[139093,2...|[-0.0769288695989...|[0.48077726176073...|
|1.0|      8| 19 11| 0.0|[19, 11]|(262144,[139093,2...|[0.04988910033929...|[0.51246968885151...|
|0.0|      6| 15 7| 0.0|[15, 7]|(262144,[77099,20...|[0.14854929135994...|[0.53706918109610...|
|0.0|      13| 12 5| 0.0|[12, 5]|(262144,[89689,25...|[-0.0519932532562...|[0.48700461408785...|
|1.0|      4| 8 22| 0.0|[8, 22]|(262144,[98962,21...|[-0.0120753606650...|[0.49698119651572...|
|1.0|      7| 17 5| 0.0|[17, 5]|(262144,[64358,89...|[-0.0721054054871...|[0.48198145477106...|
|1.0|
+-----+-----+-----+-----+-----+
+
only showing top 20 rows

```

From this point, the `model` object can be used to make predictions. The full sample of this machine learning application, and step-by-step instructions for running it, can be found [here](#).

Next steps

This article introduced the key concepts behind Spark ML Pipelines, and illustrated a sample use case of pipelines through code examples.

- See a [full example machine learning application](#) that incorporates Spark ML Pipelines.

- Learn more about [creating Spark ML models in notebooks](#).

Data Science using Scala and Spark on Azure

8/14/2017 • 34 min to read • [Edit Online](#)

This article shows you how to use Scala for supervised machine learning tasks with the Spark scalable MLlib and Spark ML packages on an Azure HDInsight Spark cluster. It walks you through the tasks that constitute the [Data Science process](#): data ingestion and exploration, visualization, feature engineering, modeling, and model consumption. The models in the article include logistic and linear regression, random forests, and gradient-boosted trees (GBTs), in addition to two common supervised machine learning tasks:

- Regression problem: Prediction of the tip amount (\$) for a taxi trip
- Binary classification: Prediction of tip or no tip (1/0) for a taxi trip

The modeling process requires training and evaluation on a test data set and relevant accuracy metrics. In this article, you can learn how to store these models in Azure Blob storage and how to score and evaluate their predictive performance. This article also covers the more advanced topics of how to optimize models by using cross-validation and hyper-parameter sweeping. The data used is a sample of the 2013 NYC taxi trip and fare data set available on GitHub.

[Scala](#), a language based on the Java virtual machine, integrates object-oriented and functional language concepts. It's a scalable language that is well suited to distributed processing in the cloud, and runs on Azure Spark clusters.

[Spark](#) is an open-source parallel-processing framework that supports in-memory processing to boost the performance of big data analytics applications. The Spark processing engine is built for speed, ease of use, and sophisticated analytics. Spark's in-memory distributed computation capabilities make it a good choice for iterative algorithms in machine learning and graph computations. The [spark.ml](#) package provides a uniform set of high-level APIs built on top of data frames that can help you create and tune practical machine learning pipelines. [MLlib](#) is Spark's scalable machine learning library, which brings modeling capabilities to this distributed environment.

[HDInsight Spark](#) is the Azure-hosted offering of open-source Spark. It also includes support for Jupyter Scala notebooks on the Spark cluster, and can run Spark SQL interactive queries to transform, filter, and visualize data stored in Azure Blob storage. The Scala code snippets in this article that provide the solutions and show the relevant plots to visualize the data run in Jupyter notebooks installed on the Spark clusters. The modeling steps in these topics have code that shows you how to train, evaluate, save, and consume each type of model.

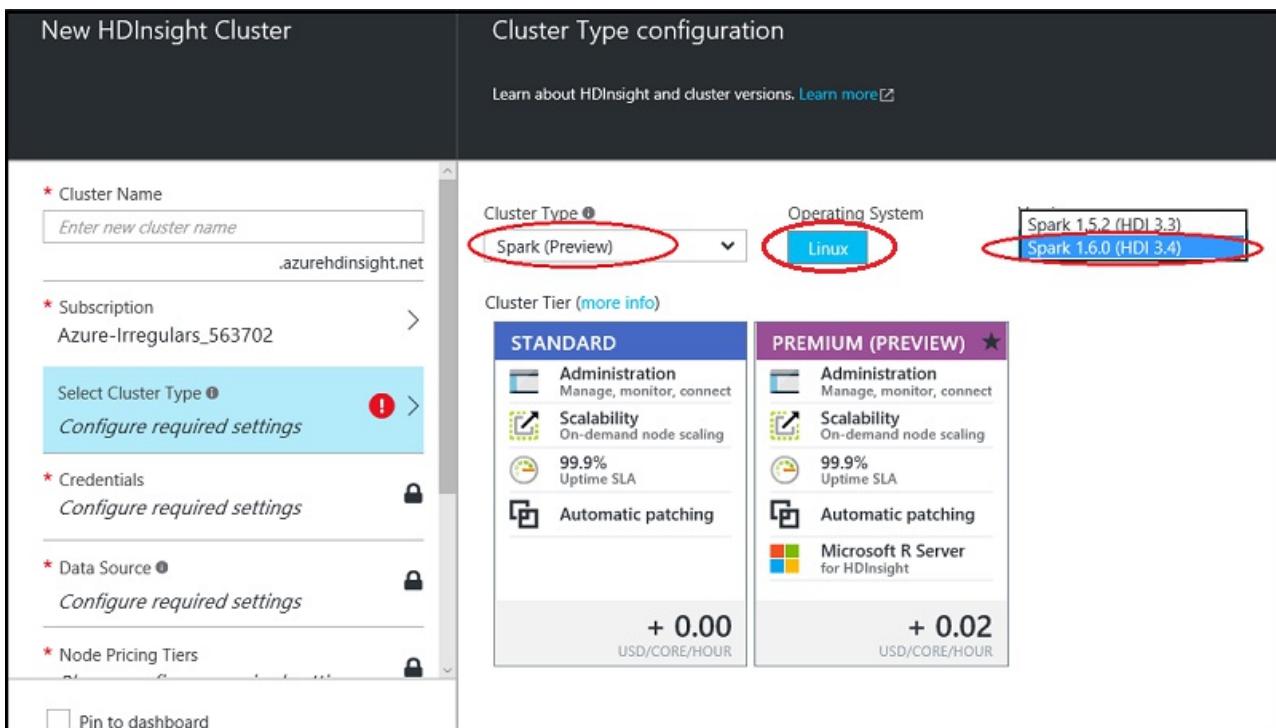
The setup steps and code in this article are for Azure HDInsight 3.4 Spark 1.6. However, the code in this article and in the [Scala Jupyter Notebook](#) are generic and should work on any Spark cluster. The cluster setup and management steps might be slightly different from what is shown in this article if you are not using HDInsight Spark.

NOTE

For a topic that shows you how to use Python rather than Scala to complete tasks for an end-to-end Data Science process, see [Data Science using Spark on Azure HDInsight](#).

Prerequisites

- You must have an Azure subscription. If you do not already have one, [get an Azure free trial](#).
- You need an Azure HDInsight 3.4 Spark 1.6 cluster to complete the following procedures. To create a cluster, see the instructions in [Get started: Create Apache Spark on Azure HDInsight](#). Set the cluster type and version on the **Select Cluster Type** menu.



WARNING

Billing for HDInsight clusters is prorated per minute, whether you are using them or not. Be sure to delete your cluster after you have finished using it. For more information, see [How to delete an HDInsight cluster](#).

For a description of the NYC taxi trip data and instructions on how to execute code from a Jupyter notebook on the Spark cluster, see the relevant sections in [Overview of Data Science using Spark on Azure HDInsight](#).

Execute Scala code from a Jupyter notebook on the Spark cluster

You can launch a Jupyter notebook from the Azure portal. Find the Spark cluster on your dashboard, and then click it to enter the management page for your cluster. Next, click **Cluster Dashboards**, and then click **Jupyter Notebook** to open the notebook associated with the Spark cluster.

The screenshot shows two blades side-by-side. The left blade is the 'HDInsight Cluster' blade, showing cluster details like 'Resource group' (redacted), 'Status' (Running), 'Location' (South Central US), and 'Subscription name' (redacted). It also shows 'Cluster Type' (Standard Spark on Linux), 'URL' (<https://<clustername>.azurehdinsight.net>), and 'Quickstart' options. A red circle highlights the 'Cluster Dashboards' link in the 'Quick Links' section. Below it is a donut chart titled 'Cores in South Central US for subscription' showing resource usage.

The right blade is the 'Cluster Dashboards' blade, listing tiles for 'HDInsight Cluster Dashboard', 'Jupyter Notebook' (redacted), 'Spark History Server', and 'Yarn'. The 'Jupyter Notebook' tile is circled in red.

You also can access Jupyter notebooks at <https://<clustername>.azurehdinsight.net/jupyter>. Replace *clustername* with the name of your cluster. You need the password for your administrator account to access the Jupyter notebooks.

The screenshot shows the Jupyter notebook interface. The top navigation bar includes 'Files', 'Running', and 'Clusters'. The main area displays a file explorer with items 'PySpark' and 'Scala'. A red circle highlights the 'Scala' folder. In the top right corner, there is an 'Upload' button, which is also circled in red.

Select **Scala** to see a directory that has a few examples of prepackaged notebooks that use the PySpark API. The Exploration Modeling and Scoring using Scala.ipynb notebook that contains the code samples for this suite of Spark topics is available on [GitHub](#).

You can upload the notebook directly from GitHub to the Jupyter Notebook server on your Spark cluster. On your Jupyter home page, click the **Upload** button. In the file explorer, paste the GitHub (raw content) URL of the Scala notebook, and then click **Open**. The Scala notebook is available at the following URL:

[Exploration-Modeling-and-Scoring-using-Scala.ipynb](#)

Setup: Preset Spark and Hive contexts, Spark magics, and Spark libraries

Preset Spark and Hive contexts

```
# SET THE START TIME
import java.util.Calendar
val beginningTime = Calendar.getInstance().getTime()
```

The Spark kernels that are provided with Jupyter notebooks have preset contexts. You don't need to explicitly set the Spark or Hive contexts before you start working with the application you are developing. The preset contexts are:

- `sc` for `SparkContext`
- `sqlContext` for `HiveContext`

Spark magics

The Spark kernel provides some predefined "magics," which are special commands that you can call with `%%`. Two of these commands are used in the following code samples.

- `%%local` specifies that the code in subsequent lines will be executed locally. The code must be valid Scala code.
- `%%sql -o <variable name>` executes a Hive query against `sqlContext`. If the `-o` parameter is passed, the result of the query is persisted in the `%%local` Scala context as a Spark data frame.

For more information about the kernels for Jupyter notebooks and their predefined "magics" that you call with `%%` (for example, `%%local`), see [Kernels available for Jupyter notebooks with HDInsight Spark Linux clusters on HDInsight](#).

Import libraries

Import the Spark, MLlib, and other libraries you'll need by using the following code.

```

# IMPORT SPARK AND JAVA LIBRARIES
import org.apache.spark.sql.SQLContext
import org.apache.spark.sql.functions._
import java.text.SimpleDateFormat
import java.util.Calendar
import sqlContext.implicits._
import org.apache.spark.sql.Row

# IMPORT SPARK SQL FUNCTIONS
import org.apache.spark.sql.types.{StructType, StructField, StringType, IntegerType, FloatType, DoubleType}
import org.apache.spark.sql.functions.rand

# IMPORT SPARK ML FUNCTIONS
import org.apache.spark.ml.Pipeline
import org.apache.spark.ml.feature.{StringIndexer, VectorAssembler, OneHotEncoder, VectorIndexer, Binarizer}
import org.apache.spark.ml.tuning.{ParamGridBuilder, TrainValidationSplit, CrossValidator}
import org.apache.spark.ml.regression.{LinearRegression, LinearRegressionModel, RandomForestRegressor,
RandomForestRegressionModel, GBTRegressor, GBTRegressionModel}
import org.apache.spark.ml.classification.{LogisticRegression, LogisticRegressionModel, RandomForestClassifier,
RandomForestClassificationModel, GBTClassifier, GBTClassificationModel}
import org.apache.spark.ml.evaluation.{BinaryClassificationEvaluator, RegressionEvaluator,
MulticlassClassificationEvaluator}

# IMPORT SPARK MLLIB FUNCTIONS
import org.apache.spark.mllib.linalg.{Vector, Vectors}
import org.apache.spark.mllib.util.MLUtils
import org.apache.spark.mllib.classification.{LogisticRegressionWithLBFGS, LogisticRegressionModel}
import org.apache.spark.mllib.regression.{LabeledPoint, LinearRegressionWithSGD, LinearRegressionModel}
import org.apache.spark.mllib.tree.{GradientBoostedTrees, RandomForest}
import org.apache.spark.mllib.tree.configuration.BoostingStrategy
import org.apache.spark.mllib.tree.model.{GradientBoostedTreesModel, RandomForestModel, Predict}
import org.apache.spark.mllib.evaluation.{BinaryClassificationMetrics, MulticlassMetrics, RegressionMetrics}

# SPECIFY SQLCONTEXT
val sqlContext = new SQLContext(sc)

```

Data ingestion

The first step in the Data Science process is to ingest the data that you want to analyze. You bring the data from external sources or systems where it resides into your data exploration and modeling environment. In this article, the data you ingest is a joined 0.1% sample of the taxi trip and fare file (stored as a .tsv file). The data exploration and modeling environment is Spark. This section contains the code to complete the following series of tasks:

1. Set directory paths for data and model storage.
2. Read in the input data set (stored as a .tsv file).
3. Define a schema for the data and clean the data.
4. Create a cleaned data frame and cache it in memory.
5. Register the data as a temporary table in SQLContext.
6. Query the table and import the results into a data frame.

Set directory paths for storage locations in Azure Blob storage

Spark can read and write to Azure Blob storage. You can use Spark to process any of your existing data, and then store the results again in Blob storage.

To save models or files in Blob storage, you need to properly specify the path. Reference the default container attached to the Spark cluster by using a path that begins with `wasb:///`. Reference other locations by using `wasb://`.

The following code sample specifies the location of the input data to be read and the path to Blob storage that is attached to the Spark cluster where the model will be saved.

```

# SET PATHS TO DATA AND MODEL FILE LOCATIONS
# INGEST DATA AND SPECIFY HEADERS FOR COLUMNS
val taxi_train_file =
sc.textFile("wasb://mllibwalkthroughs@cdsparksamples.blob.core.windows.net/Data/NYCTaxi/JoinedTaxiTripFare.Po
int1Pct.Train.tsv")
val header = taxi_train_file.first;

# SET THE MODEL STORAGE DIRECTORY PATH
# NOTE THAT THE FINAL BACKSLASH IN THE PATH IS REQUIRED.
val modelDir = "wasb:///user/remoteuser/NYCTaxi/Models/";

```

Import data, create an RDD, and define a data frame according to the schema

```

# RECORD THE START TIME
val starttime = Calendar.getInstance().getTime()

# DEFINE THE SCHEMA BASED ON THE HEADER OF THE FILE
val sqlContext = new SQLContext(sc)
val taxi_schema = StructType(
  Array(
    StructField("medallion", StringType, true),
    StructField("hack_license", StringType, true),
    StructField("vendor_id", StringType, true),
    StructField("rate_code", DoubleType, true),
    StructField("store_and_fwd_flag", StringType, true),
    StructField("pickup_datetime", StringType, true),
    StructField("dropoff_datetime", StringType, true),
    StructField("pickup_hour", DoubleType, true),
    StructField("pickup_week", DoubleType, true),
    StructField("weekday", DoubleType, true),
    StructField("passenger_count", DoubleType, true),
    StructField("trip_time_in_secs", DoubleType, true),
    StructField("trip_distance", DoubleType, true),
    StructField("pickup_longitude", DoubleType, true),
    StructField("pickup_latitude", DoubleType, true),
    StructField("dropoff_longitude", DoubleType, true),
    StructField("dropoff_latitude", DoubleType, true),
    StructField("direct_distance", StringType, true),
    StructField("payment_type", StringType, true),
    StructField("fare_amount", DoubleType, true),
    StructField("surcharge", DoubleType, true),
    StructField("mta_tax", DoubleType, true),
    StructField("tip_amount", DoubleType, true),
    StructField("tolls_amount", DoubleType, true),
    StructField("total_amount", DoubleType, true),
    StructField("tipped", DoubleType, true),
    StructField("tip_class", DoubleType, true)
  )
)

# CAST VARIABLES ACCORDING TO THE SCHEMA
val taxi_temp = (taxi_train_file.map(_.split("\t"))
  .filter((r) => r(0) != "medallion")
  .map(p => Row(p(0), p(1), p(2),
    p(3).toDouble, p(4), p(5), p(6), p(7).toDouble, p(8).toDouble, p(9).toDouble,
    p(10).toDouble,
    p(11).toDouble, p(12).toDouble, p(13).toDouble, p(14).toDouble, p(15).toDouble,
    p(16).toDouble,
    p(17), p(18), p(19).toDouble, p(20).toDouble, p(21).toDouble, p(22).toDouble,
    p(23).toDouble, p(24).toDouble, p(25).toDouble, p(26).toDouble)))
)

# CREATE AN INITIAL DATA FRAME AND DROP COLUMNS, AND THEN CREATE A CLEANED DATA FRAME BY FILTERING FOR UNWANTED
VALUES OR OUTLIERS
val taxi_train_df = sqlContext.createDataFrame(taxi_temp, taxi_schema)

```

```

val taxi_df_train_cleaned = (taxi_train_df.drop(taxi_train_df.col("medallion"))
    .drop(taxi_train_df.col("hack_license")).drop(taxi_train_df.col("store_and_fwd_flag"))
    .drop(taxi_train_df.col("pickup_datetime")).drop(taxi_train_df.col("dropoff_datetime"))
    .drop(taxi_train_df.col("pickup_longitude")).drop(taxi_train_df.col("pickup_latitude"))
    .drop(taxi_train_df.col("dropoff_longitude")).drop(taxi_train_df.col("dropoff_latitude"))
    .drop(taxi_train_df.col("surcharge")).drop(taxi_train_df.col("mta_tax"))
    .drop(taxi_train_df.col("direct_distance")).drop(taxi_train_df.col("tolls_amount"))
    .drop(taxi_train_df.col("total_amount")).drop(taxi_train_df.col("tip_class"))
    .filter("passenger_count > 0 AND passenger_count < 8 AND payment_type in ('CSH', 'CRD') AND tip_amount
    >= 0 AND tip_amount < 30 AND fare_amount >= 1 AND fare_amount < 150 AND trip_distance > 0 AND trip_distance <
    100 AND trip_time_in_secs > 30 AND trip_time_in_secs < 7200"));

# CACHE AND MATERIALIZE THE CLEANED DATA FRAME IN MEMORY
taxi_df_train_cleaned.cache()
taxi_df_train_cleaned.count()

# REGISTER THE DATA FRAME AS A TEMPORARY TABLE IN SQLCONTEXT
taxi_df_train_cleaned.registerTempTable("taxi_train")

# GET THE TIME TO RUN THE CELL
val endtime = Calendar.getInstance().getTime()
val elapsedtime = ((endtime.getTime() - starttime.getTime()) / 1000).toString;
println("Time taken to run the above cell: " + elapsedtime + " seconds.");

```

Output:

Time to run the cell: 8 seconds.

Query the table and import results in a data frame

Next, query the table for fare, passenger, and tip data; filter out corrupt and outlying data; and print several rows.

```

# QUERY THE DATA
val sqlStatement = """
    SELECT fare_amount, passenger_count, tip_amount, tipped
    FROM taxi_train
    WHERE passenger_count > 0 AND passenger_count < 7
    AND fare_amount > 0 AND fare_amount < 200
    AND payment_type in ('CSH', 'CRD')
    AND tip_amount > 0 AND tip_amount < 25
"""
val sqlResultsDF = sqlContext.sql(sqlStatement)

# SHOW ONLY THE TOP THREE ROWS
sqlResultsDF.show(3)

```

Output:

FARE_AMOUNT	PASSENGER_COUNT	TIP_AMOUNT	TIPPED
13.5	1.0	2.9	1.0
16.0	2.0	3.4	1.0
10.5	2.0	1.0	1.0

Data exploration and visualization

After you bring the data into Spark, the next step in the Data Science process is to gain a deeper understanding of the data through exploration and visualization. In this section, you examine the taxi data by using SQL queries.

Then, import the results into a data frame to plot the target variables and prospective features for visual inspection

by using the auto-visualization feature of Jupyter.

Use local and SQL magic to plot data

By default, the output of any code snippet that you run from a Jupyter notebook is available within the context of the session that is persisted on the worker nodes. If you want to save a trip to the worker nodes for every computation, and if all the data that you need for your computation is available locally on the Jupyter server node (which is the head node), you can use the `%%local` magic to run the code snippet on the Jupyter server.

- **SQL magic** (`%sql`). The HDInsight Spark kernel supports easy inline HiveQL queries against `SQLContext`. The (`-o VARIABLE_NAME`) argument persists the output of the SQL query as a Pandas data frame on the Jupyter server. This means it'll be available in the local mode.
- **%%local magic**. The `%%local` magic runs the code locally on the Jupyter server, which is the head node of the HDInsight cluster. Typically, you use `%%local` magic in conjunction with the `%sql` magic with the `-o` parameter. The `-o` parameter would persist the output of the SQL query locally, and then `%%local` magic would trigger the next set of code snippet to run locally against the output of the SQL queries that is persisted locally.

Query the data by using SQL

This query retrieves the taxi trips by fare amount, passenger count, and tip amount.

```
# RUN THE SQL QUERY
%sql -q -o sqlResults
SELECT fare_amount, passenger_count, tip_amount, tipped FROM taxi_train WHERE passenger_count > 0 AND
passenger_count < 7 AND fare_amount > 0 AND fare_amount < 200 AND payment_type in ('CSH', 'CRD') AND tip_amount
> 0 AND tip_amount < 25
```

In the following code, the `%%local` magic creates a local data frame, `sqlResults`. You can use `sqlResults` to plot by using `matplotlib`.

TIP

Local magic is used multiple times in this article. If your data set is large, please sample to create a data frame that can fit in local memory.

Plot the data

You can plot by using Python code after the data frame is in local context as a Pandas data frame.

```
# RUN THE CODE LOCALLY ON THE JUPYTER SERVER
%%local

# USE THE JUPYTER AUTO-PLOTTING FEATURE TO CREATE INTERACTIVE FIGURES.
# CLICK THE TYPE OF PLOT TO GENERATE (LINE, AREA, BAR, ETC.)
sqlResults
```

The Spark kernel automatically visualizes the output of SQL (HiveQL) queries after you run the code. You can choose between several types of visualizations:

- Table
- Pie
- Line
- Area
- Bar

Here's the code to plot the data:

```

# RUN THE CODE LOCALLY ON THE JUPYTER SERVER AND IMPORT LIBRARIES
%%local
import matplotlib.pyplot as plt
%matplotlib inline

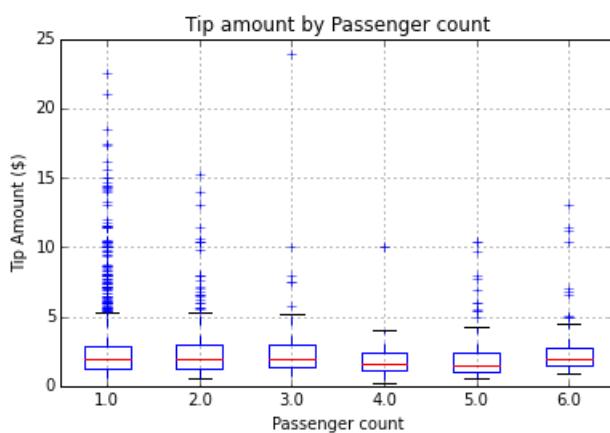
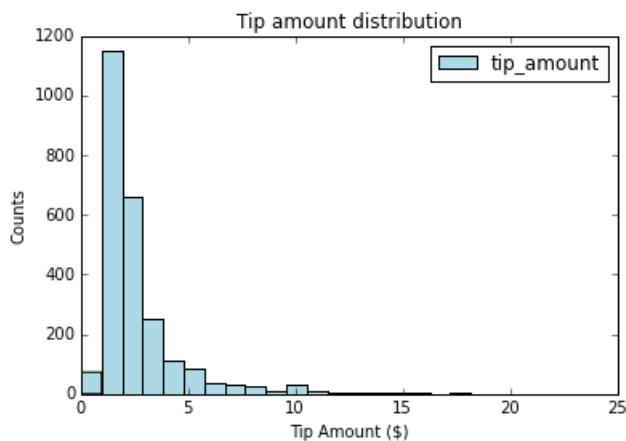
# PLOT TIP BY PAYMENT TYPE AND PASSENGER COUNT
ax1 = sqlResults[['tip_amount']].plot(kind='hist', bins=25, facecolor='lightblue')
ax1.set_title('Tip amount distribution')
ax1.set_xlabel('Tip Amount ($)')
ax1.set_ylabel('Counts')
plt.suptitle('')
plt.show()

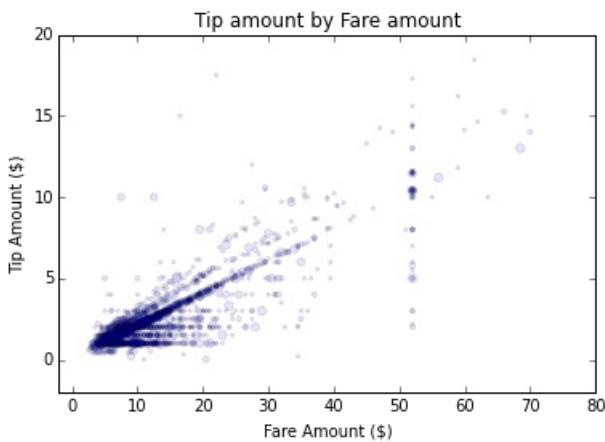
# PLOT TIP BY PASSENGER COUNT
ax2 = sqlResults.boxplot(column=['tip_amount'], by=['passenger_count'])
ax2.set_title('Tip amount by Passenger count')
ax2.set_xlabel('Passenger count')
ax2.set_ylabel('Tip Amount ($)')
plt.suptitle('')
plt.show()

# PLOT TIP AMOUNT BY FARE AMOUNT; SCALE POINTS BY PASSENGER COUNT
ax = sqlResults.plot(kind='scatter', x= 'fare_amount', y = 'tip_amount', c='blue', alpha = 0.10, s=5*
(sqlResults.passenger_count))
ax.set_title('Tip amount by Fare amount')
ax.set_xlabel('Fare Amount ($)')
ax.set_ylabel('Tip Amount ($)')
plt.axis([-2, 80, -2, 20])
plt.show()

```

Output:





Create features and transform features, and then prep data for input into modeling functions

For tree-based modeling functions from Spark ML and MLLib, you have to prepare target and features by using a variety of techniques, such as binning, indexing, one-hot encoding, and vectorization. Here are the procedures to follow in this section:

1. Create a new feature by **binning** hours into traffic time buckets.
2. Apply **indexing and one-hot encoding** to categorical features.
3. **Sample and split the data set** into training and test fractions.
4. **Specify training variable and features**, and then create indexed or one-hot encoded training and testing input labeled point resilient distributed datasets (RDDs) or data frames.
5. Automatically **categorize and vectorize features and targets** to use as inputs for machine learning models.

Create a new feature by binning hours into traffic time buckets

This code shows you how to create a new feature by binning hours into traffic time buckets and how to cache the resulting data frame in memory. Where RDDs and data frames are used repeatedly, caching leads to improved execution times. Accordingly, you'll cache RDDs and data frames at several stages in the following procedures.

```
# CREATE FOUR BUCKETS FOR TRAFFIC TIMES
val sqlStatement = """
    SELECT *,
    CASE
        WHEN (pickup_hour <= 6 OR pickup_hour >= 20) THEN "Night"
        WHEN (pickup_hour >= 7 AND pickup_hour <= 10) THEN "AMRush"
        WHEN (pickup_hour >= 11 AND pickup_hour <= 15) THEN "Afternoon"
        WHEN (pickup_hour >= 16 AND pickup_hour <= 19) THEN "PMRush"
    END as TrafficTimeBins
    FROM taxi_train
"""
val taxi_df_train_with_newFeatures = sqlContext.sql(sqlStatement)

# CACHE THE DATA FRAME IN MEMORY AND MATERIALIZE THE DATA FRAME IN MEMORY
taxi_df_train_with_newFeatures.cache()
taxi_df_train_with_newFeatures.count()
```

Indexing and one-hot encoding of categorical features

The modeling and predict functions of MLLib require features with categorical input data to be indexed or encoded prior to use. This section shows you how to index or encode categorical features for input into the modeling functions.

You need to index or encode your models in different ways, depending on the model. For example, logistic and linear regression models require one-hot encoding. For example, a feature with three categories can be expanded

into three feature columns. Each column would contain 0 or 1 depending on the category of an observation. MLlib provides the [OneHotEncoder](#) function for one-hot encoding. This encoder maps a column of label indices to a column of binary vectors with at most a single one-value. With this encoding, algorithms that expect numerical valued features, such as logistic regression, can be applied to categorical features.

Here you transform only four variables to show examples, which are character strings. You also can index other variables, such as weekday, represented by numerical values, as categorical variables.

For indexing, use `StringIndexer()`, and for one-hot encoding, use `OneHotEncoder()` functions from MLlib. Here is the code to index and encode categorical features:

```
# CREATE INDEXES AND ONE-HOT ENCODED VECTORS FOR SEVERAL CATEGORICAL FEATURES

# RECORD THE START TIME
val starttime = Calendar.getInstance().getTime()

# INDEX AND ENCODE VENDOR_ID
val stringIndexer = new
StringIndexer().setInputCol("vendor_id").setOutputCol("vendorIndex").fit(taxi_df_train_with_newFeatures)
val indexed = stringIndexer.transform(taxi_df_train_with_newFeatures)
val encoder = new OneHotEncoder().setInputCol("vendorIndex").setOutputCol("vendorVec")
val encoded1 = encoder.transform(indexed)

# INDEX AND ENCODE RATE_CODE
val stringIndexer = new StringIndexer().setInputCol("rate_code").setOutputCol("rateIndex").fit(encoded1)
val indexed = stringIndexer.transform(encoded1)
val encoder = new OneHotEncoder().setInputCol("rateIndex").setOutputCol("rateVec")
val encoded2 = encoder.transform(indexed)

# INDEX AND ENCODE PAYMENT_TYPE
val stringIndexer = new StringIndexer().setInputCol("payment_type").setOutputCol("paymentIndex").fit(encoded2)
val indexed = stringIndexer.transform(encoded2)
val encoder = new OneHotEncoder().setInputCol("paymentIndex").setOutputCol("paymentVec")
val encoded3 = encoder.transform(indexed)

# INDEX AND TRAFFIC TIME BINS
val stringIndexer = new
StringIndexer().setInputCol("TrafficTimeBins").setOutputCol("TrafficTimeBinsIndex").fit(encoded3)
val indexed = stringIndexer.transform(encoded3)
val encoder = new OneHotEncoder().setInputCol("TrafficTimeBinsIndex").setOutputCol("TrafficTimeBinsVec")
val encodedFinal = encoder.transform(indexed)

# GET THE TIME TO RUN THE CELL
val endtime = Calendar.getInstance().getTime()
val elapsedtime = ((endtime.getTime() - starttime.getTime())/1000).toString;
println("Time taken to run the above cell: " + elapsedtime + " seconds.");
```

Output:

Time to run the cell: 4 seconds.

Sample and split the data set into training and test fractions

This code creates a random sampling of the data (25%, in this example). Although sampling is not required for this example due to the size of the data set, the article shows you how you can sample so that you know how to use it for your own problems when needed. When samples are large, this can save significant time while you train models. Next, split the sample into a training part (75%, in this example) and a testing part (25%, in this example) to use in classification and regression modeling.

Add a random number (between 0 and 1) to each row (in a "rand" column) that can be used to select cross-validation folds during training.

```

# RECORD THE START TIME
val starttime = Calendar.getInstance().getTime()

# SPECIFY SAMPLING AND SPLITTING FRACTIONS
val samplingFraction = 0.25;
val trainingFraction = 0.75;
val testingFraction = (1-trainingFraction);
val seed = 1234;
val encodedFinalSampledTmp = encodedFinal.sample(withReplacement = false, fraction = samplingFraction, seed =
seed)
val sampledDFcount = encodedFinalSampledTmp.count().toInt

val generateRandomDouble = udf(() => {
    scala.util.Random.nextDouble
})

# ADD A RANDOM NUMBER FOR CROSS-VALIDATION
val encodedFinalSampled = encodedFinalSampledTmp.withColumn("rand", generateRandomDouble());

# SPLIT THE SAMPLED DATA FRAME INTO TRAIN AND TEST, WITH A RANDOM COLUMN ADDED FOR DOING CROSS-VALIDATION
# (SHOWN LATER)
# INCLUDE A RANDOM COLUMN FOR CREATING CROSS-VALIDATION FOLDS
val splits = encodedFinalSampled.randomSplit(Array(trainingFraction, testingFraction), seed = seed)
val trainData = splits(0)
val testData = splits(1)

# GET THE TIME TO RUN THE CELL
val endtime = Calendar.getInstance().getTime()
val elapsedtime = ((endtime.getTime() - starttime.getTime())/1000).toString;
println("Time taken to run the above cell: " + elapsedtime + " seconds.");

```

Output:

Time to run the cell: 2 seconds.

Specify training variable and features, and then create indexed or one-hot encoded training and testing input labeled point RDDs or data frames

This section contains code that shows you how to index categorical text data as a labeled point data type, and encode it so you can use it to train and test MLlib logistic regression and other classification models. Labeled point objects are RDDs that are formatted in a way that is needed as input data by most of machine learning algorithms in MLlib. A [labeled point](#) is a local vector, either dense or sparse, associated with a label/response.

In this code, you specify the target (dependent) variable and the features to use to train models. Then, you create indexed or one-hot encoded training and testing input labeled point RDDs or data frames.

```

# RECORD THE START TIME
val starttime = Calendar.getInstance().getTime()

# MAP NAMES OF FEATURES AND TARGETS FOR CLASSIFICATION AND REGRESSION PROBLEMS
val featuresIndOneHot = List("paymentVec", "vendorVec", "rateVec", "TrafficTimeBinsVec", "pickup_hour",
"weekday", "passenger_count", "trip_time_in_secs", "trip_distance",
"fare_amount").map(encodedFinalSampled.columns.indexOf(_))
val featuresIndIndex = List("paymentIndex", "vendorIndex", "rateIndex", "TrafficTimeBinsIndex", "pickup_hour",
"weekday", "passenger_count", "trip_time_in_secs", "trip_distance",
"fare_amount").map(encodedFinalSampled.columns.indexOf(_))

# SPECIFY THE TARGET FOR CLASSIFICATION ('tipped') AND REGRESSION ('tip_amount') PROBLEMS
val targetIndBinary = List("tipped").map(encodedFinalSampled.columns.indexOf(_))
val targetIndRegression = List("tip_amount").map(encodedFinalSampled.columns.indexOf(_))

# CREATE INDEXED LABELED POINT RDD OBJECTS
val indexedTRAINbinary = trainData.rdd.map(r => LabeledPoint(r.getDouble(targetIndBinary(0).toInt),
Vectors.dense(featuresIndIndex.map(r.getDouble(_)).toArray)))
val indexedTESTbinary = testData.rdd.map(r => LabeledPoint(r.getDouble(targetIndBinary(0).toInt),
Vectors.dense(featuresIndIndex.map(r.getDouble(_)).toArray)))
val indexedTRAINreg = trainData.rdd.map(r => LabeledPoint(r.getDouble(targetIndRegression(0).toInt),
Vectors.dense(featuresIndIndex.map(r.getDouble(_)).toArray)))
val indexedTESTreg = testData.rdd.map(r => LabeledPoint(r.getDouble(targetIndRegression(0).toInt),
Vectors.dense(featuresIndIndex.map(r.getDouble(_)).toArray)))

# CREATE INDEXED DATA FRAMES THAT YOU CAN USE TO TRAIN BY USING SPARK ML FUNCTIONS
val indexedTRAINbinaryDF = indexedTRAINbinary.toDF()
val indexedTESTbinaryDF = indexedTESTbinary.toDF()
val indexedTRAINregDF = indexedTRAINreg.toDF()
val indexedTESTregDF = indexedTESTreg.toDF()

# CREATE ONE-HOT ENCODED (VECTORIZED) DATA FRAMES THAT YOU CAN USE TO TRAIN BY USING SPARK ML FUNCTIONS
val assemblerOneHot = new VectorAssembler().setInputCols(Array("paymentVec", "vendorVec", "rateVec",
"TrafficTimeBinsVec", "pickup_hour", "weekday", "passenger_count", "trip_time_in_secs", "trip_distance",
"fare_amount")).setOutputCol("features")
val OneHotTRAIN = assemblerOneHot.transform(trainData)
val OneHotTEST = assemblerOneHot.transform(testData)

# GET THE TIME TO RUN THE CELL
val endtime = Calendar.getInstance().getTime()
val elapsedtime = ((endtime.getTime() - starttime.getTime()) / 1000).toString;
println("Time taken to run the above cell: " + elapsedtime + " seconds.");

```

Output:

Time to run the cell: 4 seconds.

Automatically categorize and vectorize features and targets to use as inputs for machine learning models

Use Spark ML to categorize the target and features to use in tree-based modeling functions. The code completes two tasks:

- Creates a binary target for classification by assigning a value of 0 or 1 to each data point between 0 and 1 by using a threshold value of 0.5.
- Automatically categorizes features. If the number of distinct numerical values for any feature is less than 32, that feature is categorized.

Here's the code for these two tasks.

```

# CATEGORIZE FEATURES AND BINARIZE THE TARGET FOR THE BINARY CLASSIFICATION PROBLEM

# TRAIN DATA
val indexer = new VectorIndexer().setInputCol("features").setOutputCol("featuresCat").setMaxCategories(32)
val indexerModel = indexer.fit(indexedTRAINbinaryDF)
val indexedTrainwithCatFeat = indexerModel.transform(indexedTRAINbinaryDF)
val binarizer: Binarizer = new Binarizer().setInputCol("label").setOutputCol("labelBin").setThreshold(0.5)
val indexedTRAINwithCatFeatBinTarget = binarizer.transform(indexedTrainwithCatFeat)

# TEST DATA
val indexerModel = indexer.fit(indexedTESTbinaryDF)
val indexedTrainwithCatFeat = indexerModel.transform(indexedTESTbinaryDF)
val binarizer: Binarizer = new Binarizer().setInputCol("label").setOutputCol("labelBin").setThreshold(0.5)
val indexedTESTwithCatFeatBinTarget = binarizer.transform(indexedTrainwithCatFeat)

# CATEGORIZE FEATURES FOR THE REGRESSION PROBLEM
# CREATE PROPERLY INDEXED AND CATEGORIZED DATA FRAMES FOR TREE-BASED MODELS

# TRAIN DATA
val indexer = new VectorIndexer().setInputCol("features").setOutputCol("featuresCat").setMaxCategories(32)
val indexerModel = indexer.fit(indexedTRAINregDF)
val indexedTRAINwithCatFeat = indexerModel.transform(indexedTRAINregDF)

# TEST DATA
val indexerModel = indexer.fit(indexedTESTbinaryDF)
val indexedTESTwithCatFeat = indexerModel.transform(indexedTESTregDF)

```

Binary classification model: Predict whether a tip should be paid

In this section, you create three types of binary classification models to predict whether or not a tip should be paid:

- A **logistic regression model** by using the Spark ML `LogisticRegression()` function
- A **random forest classification model** by using the Spark ML `RandomForestClassifier()` function
- A **gradient boosting tree classification model** by using the MLLib `GradientBoostedTrees()` function

Create a logistic regression model

Next, create a logistic regression model by using the Spark ML `LogisticRegression()` function. You create the model building code in a series of steps:

1. **Train the model** data with one parameter set.
2. **Evaluate the model** on a test data set with metrics.
3. **Save the model** in Blob storage for future consumption.
4. **Score the model** against test data.
5. **Plot the results** with receiver operating characteristic (ROC) curves.

Here's the code for these procedures:

```

# CREATE A LOGISTIC REGRESSION MODEL
val lr = new LogisticRegression().setLabelCol("tipped").setFeaturesCol("features").setMaxIter(10).setRegParam(0.3).setElasticNetParam(0.8)
val lrModel = lr.fit(OneHotTRAIN)

# PREDICT ON THE TEST DATA SET
val predictions = lrModel.transform(OneHotTEST)

# SELECT `BinaryClassificationEvaluator()` TO COMPUTE THE TEST ERROR
val evaluator = new BinaryClassificationEvaluator().setLabelCol("tipped").setRawPredictionCol("probability").setMetricName("areaUnderROC")
val ROC = evaluator.evaluate(predictions)
println("ROC on test data = " + ROC)

# SAVE THE MODEL
val timestamp = Calendar.getInstance().getTime().toString.replaceAll(" ", ".").replaceAll(":", "_");
val modelName = "LogisticRegression_"
val filename = modelDir.concat(modelName).concat(timestamp)
lrModel.save(filename);

```

Load, score, and save the results.

```

# RECORD THE START TIME
val starttime = Calendar.getInstance().getTime()

# LOAD THE SAVED MODEL AND SCORE THE TEST DATA SET
val savedModel = org.apache.spark.ml.classification.LogisticRegressionModel.load(filename)
println(s"Coefficients: ${savedModel.coefficients} Intercept: ${savedModel.intercept}")

# SCORE THE MODEL ON THE TEST DATA
val predictions = savedModel.transform(OneHotTEST).select("tipped","probability","rawPrediction")
predictions.registerTempTable("testResults")

# SELECT `BinaryClassificationEvaluator()` TO COMPUTE THE TEST ERROR
val evaluator = new BinaryClassificationEvaluator().setLabelCol("tipped").setRawPredictionCol("probability").setMetricName("areaUnderROC")
val ROC = evaluator.evaluate(predictions)

# GET THE TIME TO RUN THE CELL
val endtime = Calendar.getInstance().getTime()
val elapsedtime = ((endtime.getTime() - starttime.getTime())/1000).toString;
println("Time taken to run the above cell: " + elapsedtime + " seconds.");

# PRINT THE ROC RESULTS
println("ROC on test data = " + ROC)

```

Output:

ROC on test data = 0.9827381497557599

Use Python on local Pandas data frames to plot the ROC curve.

```

# QUERY THE RESULTS
%%sql -q -o sqlResults
SELECT tipped, probability from testResults

# RUN THE CODE LOCALLY ON THE JUPYTER SERVER AND IMPORT LIBRARIES
%%local
%matplotlib inline
from sklearn.metrics import roc_curve,auc

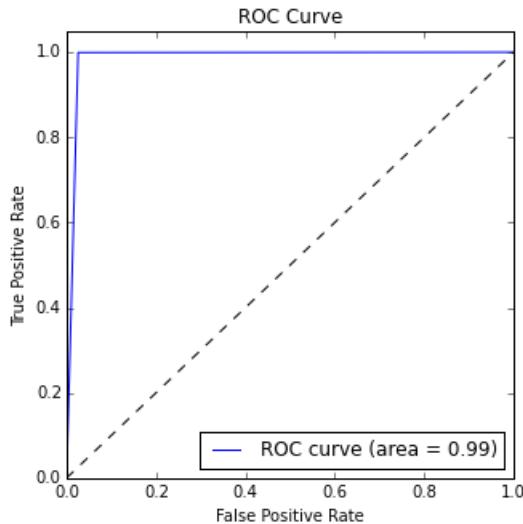
sqlResults['probFloat'] = sqlResults.apply(lambda row: row['probability'].values()[0][1], axis=1)
predictions_pddf = sqlResults[["tipped","probFloat"]]

# PREDICT THE ROC CURVE
# predictions_pddf = sqlResults.rename(columns={'_1': 'probability', 'tipped': 'label'})
prob = predictions_pddf["probFloat"]
fpr, tpr, thresholds = roc_curve(predictions_pddf['tipped'], prob, pos_label=1);
roc_auc = auc(fpr, tpr)

# PLOT THE ROC CURVE
plt.figure(figsize=(5,5))
plt.plot(fpr, tpr, label='ROC curve (area = %0.2f)' % roc_auc)
plt.plot([0, 1], [0, 1], 'k--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve')
plt.legend(loc="lower right")
plt.show()

```

Output:



Create a random forest classification model

Next, create a random forest classification model by using the Spark ML `RandomForestClassifier()` function, and then evaluate the model on test data.

```

# RECORD THE START TIME
val starttime = Calendar.getInstance().getTime()

# CREATE THE RANDOM FOREST CLASSIFIER MODEL
val rf = new
RandomForestClassifier().setLabelCol("labelBin").setFeaturesCol("featuresCat").setNumTrees(10).setSeed(1234)

# FIT THE MODEL
val rfModel = rf.fit(indexedTRAINwithCatFeatBinTarget)
val predictions = rfModel.transform(indexedTESTwithCatFeatBinTarget)

# EVALUATE THE MODEL
val evaluator = new
MulticlassClassificationEvaluator().setLabelCol("label").setPredictionCol("prediction").setMetricName("f1")
val Test_f1Score = evaluator.evaluate(predictions)
println("F1 score on test data: " + Test_f1Score);

# GET THE TIME TO RUN THE CELL
val endtime = Calendar.getInstance().getTime()
val elapsedtime = ((endtime.getTime() - starttime.getTime())/1000).toString;
println("Time taken to run the above cell: " + elapsedtime + " seconds.");

# CALCULATE BINARY CLASSIFICATION EVALUATION METRICS
val evaluator = new
BinaryClassificationEvaluator().setLabelCol("label").setRawPredictionCol("probability").setMetricName("areaUnderROC")
val ROC = evaluator.evaluate(predictions)
println("ROC on test data = " + ROC)

```

Output:

ROC on test data = 0.9847103571552683

Create a GBT classification model

Next, create a GBT classification model by using MLlib's `GradientBoostedTrees()` function, and then evaluate the model on test data.

```

# TRAIN A GBT CLASSIFICATION MODEL BY USING MLLIB AND A LABELED POINT

# RECORD THE START TIME
val starttime = Calendar.getInstance().getTime()

# DEFINE THE GBT CLASSIFICATION MODEL
val boostingStrategy = BoostingStrategy.defaultParams("Classification")
boostingStrategy.numIterations = 20
boostingStrategy.treeStrategy.numClasses = 2
boostingStrategy.treeStrategy.maxDepth = 5
boostingStrategy.treeStrategy.categoricalFeaturesInfo = Map[Int, Int]((0,2),(1,2),(2,6),(3,4))

# TRAIN THE MODEL
val gbtModel = GradientBoostedTrees.train(indexedTRAINbinary, boostingStrategy)

# SAVE THE MODEL IN BLOB STORAGE
val datestamp = Calendar.getInstance().getTime().toString.replaceAll(" ", ".").replaceAll(":", "_");
val modelName = "GBT_Classification_"
val filename = modelDir.concat(modelName).concat(datestamp)
gbtModel.save(sc, filename);

# EVALUATE THE MODEL ON TEST INSTANCES AND THE COMPUTE TEST ERROR
val labelAndPreds = indexedTESTbinary.map { point =>
    val prediction = gbtModel.predict(point.features)
    (point.label, prediction)
}
val testErr = labelAndPreds.filter(r => r._1 != r._2).count.toDouble / indexedTRAINbinary.count()
//println("Learned classification GBT model:\n" + gbtModel.toDebugString)
println("Test Error = " + testErr)

# USE BINARY AND MULTICLASS METRICS TO EVALUATE THE MODEL ON THE TEST DATA
val metrics = new MulticlassMetrics(labelAndPreds)
println(s"Precision: ${metrics.precision}")
println(s"Recall: ${metrics.recall}")
println(s"F1 Score: ${metrics.fMeasure}")

val metrics = new BinaryClassificationMetrics(labelAndPreds)
println(s"Area under PR curve: ${metrics.areaUnderPR}")
println(s"Area under ROC curve: ${metrics.areaUnderROC}")

# GET THE TIME TO RUN THE CELL
val endtime = Calendar.getInstance().getTime()
val elapsedtime = ((endtime.getTime() - starttime.getTime()) / 1000).toString;
println("Time taken to run the above cell: " + elapsedtime + " seconds.");

# PRINT THE ROC METRIC
println(s"Area under ROC curve: ${metrics.areaUnderROC}")

```

Output:

Area under ROC curve: 0.9846895479241554

Regression model: Predict tip amount

In this section, you create two types of regression models to predict the tip amount:

- A **regularized linear regression model** by using the Spark ML `LinearRegression()` function. You'll save the model and evaluate the model on test data.
- A **gradient-boosting tree regression model** by using the Spark ML `GBTRegressor()` function.

Create a regularized linear regression model

```

# RECORD THE START TIME
val starttime = Calendar.getInstance().getTime()

# CREATE A REGULARIZED LINEAR REGRESSION MODEL BY USING THE SPARK ML FUNCTION AND DATA FRAMES
val lr = new
LinearRegression().setLabelCol("tip_amount").setFeaturesCol("features").setMaxIter(10).setRegParam(0.3).setElasticNetParam(0.8)

# FIT THE MODEL BY USING DATA FRAMES
val lrModel = lr.fit(OneHotTRAIN)
println(s"Coefficients: ${lrModel.coefficients} Intercept: ${lrModel.intercept}")

# SUMMARIZE THE MODEL OVER THE TRAINING SET AND PRINT METRICS
val trainingSummary = lrModel.summary
println(s"numIterations: ${trainingSummary.totalIterations}")
println(s"objectiveHistory: ${trainingSummary.objectiveHistory.toList}")
trainingSummary.residuals.show()
println(s"RMSE: ${trainingSummary.rootMeanSquaredError}")
println(s"r2: ${trainingSummary.r2}")

# SAVE THE MODEL IN AZURE BLOB STORAGE
val timestamp = Calendar.getInstance().getTime().toString.replaceAll(" ", ".").replaceAll(":", "_");
val modelName = "LinearRegression_"
val filename = modelDir.concat(modelName).concat(timestamp)
lrModel.save(filename);

# PRINT THE COEFFICIENTS
println(s"Coefficients: ${lrModel.coefficients} Intercept: ${lrModel.intercept}")

# SCORE THE MODEL ON TEST DATA
val predictions = lrModel.transform(OneHotTEST)

# EVALUATE THE MODEL ON TEST DATA
val evaluator = new
RegressionEvaluator().setLabelCol("tip_amount").setPredictionCol("prediction").setMetricName("r2")
val r2 = evaluator.evaluate(predictions)
println("R-sqr on test data = " + r2)

# GET THE TIME TO RUN THE CELL
val endtime = Calendar.getInstance().getTime()
val elapsedtime = ((endtime.getTime() - starttime.getTime()) / 1000).toString;
println("Time taken to run the above cell: " + elapsedtime + " seconds.");

```

Output:

Time to run the cell: 13 seconds.

```

# LOAD A SAVED LINEAR REGRESSION MODEL FROM BLOB STORAGE AND SCORE A TEST DATA SET

# RECORD THE START TIME
val starttime = Calendar.getInstance().getTime()

# LOAD A SAVED LINEAR REGRESSION MODEL FROM AZURE BLOB STORAGE
val savedModel = org.apache.spark.ml.regression.LinearRegressionModel.load(filename)
println(s"Coefficients: ${savedModel.coefficients} Intercept: ${savedModel.intercept}")

# SCORE THE MODEL ON TEST DATA
val predictions = savedModel.transform(OneHotTEST).select("tip_amount","prediction")
predictions.registerTempTable("testResults")

# EVALUATE THE MODEL ON TEST DATA
val evaluator = new
RegressionEvaluator().setLabelCol("tip_amount").setPredictionCol("prediction").setMetricName("r2")
val r2 = evaluator.evaluate(predictions)
println("R-sqr on test data = " + r2)

# GET THE TIME TO RUN THE CELL
val endtime = Calendar.getInstance().getTime()
val elapsedtime = ((endtime.getTime() - starttime.getTime()) / 1000).toString;
println("Time taken to run the above cell: " + elapsedtime + " seconds.")

# PRINT THE RESULTS
println("R-sqr on test data = " + r2)

```

Output:

R-sqr on test data = 0.5960320470835743

Next, query the test results as a data frame and use AutoVizWidget and matplotlib to visualize it.

```

# RUN A SQL QUERY
%%sql -q -o sqlResults
select * from testResults

# RUN THE CODE LOCALLY ON THE JUPYTER SERVER
%%local

# USE THE JUPYTER AUTO-PLOTTING FEATURE TO CREATE INTERACTIVE FIGURES
# CLICK THE TYPE OF PLOT TO GENERATE (LINE, AREA, BAR, AND SO ON)
sqlResults

```

The code creates a local data frame from the query output and plots the data. The `%%local` magic creates a local data frame, `sqlResults`, which you can use to plot with matplotlib.

NOTE

This Spark magic is used multiple times in this article. If the amount of data is large, you should sample to create a data frame that can fit in local memory.

Create plots by using Python matplotlib.

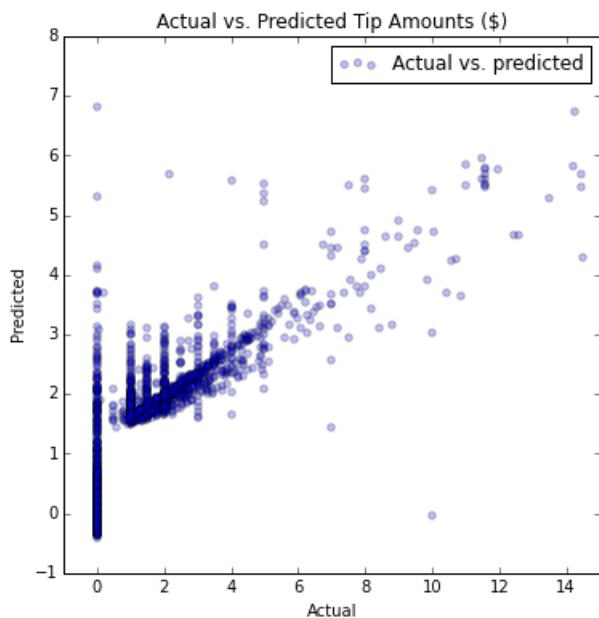
```

# RUN THE CODE LOCALLY ON THE JUPYTER SERVER AND IMPORT LIBRARIES
%%local
sqlResults
%matplotlib inline
import numpy as np

# PLOT THE RESULTS
ax = sqlResults.plot(kind='scatter', figsize = (6,6), x='tip_amount', y='prediction', color='blue', alpha = 0.25, label='Actual vs. predicted');
fit = np.polyfit(sqlResults['tip_amount'], sqlResults['prediction'], deg=1)
ax.set_title('Actual vs. Predicted Tip Amounts ($)')
ax.set_xlabel("Actual")
ax.set_ylabel("Predicted")
#ax.plot(sqlResults['tip_amount'], fit[0] * sqlResults['prediction'] + fit[1], color='magenta')
plt.axis([-1, 15, -1, 8])
plt.show(ax)

```

Output:



Create a GBT regression model

Create a GBT regression model by using the Spark ML `GBTRegressor()` function, and then evaluate the model on test data.

[Gradient-boosted trees](#) (GBTs) are ensembles of decision trees. GBTs train decision trees iteratively to minimize a loss function. You can use GBTs for regression and classification. They can handle categorical features, do not require feature scaling, and can capture nonlinearities and feature interactions. You also can use them in a multiclass-classification setting.

```

# RECORD THE START TIME
val starttime = Calendar.getInstance().getTime()

# TRAIN A GBT REGRESSION MODEL
val gbt = new GBTRegressor().setLabelCol("label").setFeaturesCol("featuresCat").setMaxIter(10)
val gbtModel = gbt.fit(indexedTRAINwithCatFeat)

# MAKE PREDICTIONS
val predictions = gbtModel.transform(indexedTESTwithCatFeat)

# COMPUTE TEST SET R2
val evaluator = new
RegressionEvaluator().setLabelCol("label").setPredictionCol("prediction").setMetricName("r2")
val Test_R2 = evaluator.evaluate(predictions)

# GET THE TIME TO RUN THE CELL
val endtime = Calendar.getInstance().getTime()
val elapsedtime = ((endtime.getTime() - starttime.getTime())/1000).toString;
println("Time taken to run the above cell: " + elapsedtime + " seconds.");

# PRINT THE RESULTS
println("Test R-sqr is: " + Test_R2);

```

Output:

Test R-sqr is: 0.7655383534596654

Advanced modeling utilities for optimization

In this section, you use machine learning utilities that developers frequently use for model optimization. Specifically, you can optimize machine learning models three different ways by using parameter sweeping and cross-validation:

- Split the data into train and validation sets, optimize the model by using hyper-parameter sweeping on a training set, and evaluate on a validation set (linear regression)
- Optimize the model by using cross-validation and hyper-parameter sweeping by using Spark ML's CrossValidator function (binary classification)
- Optimize the model by using custom cross-validation and parameter-sweeping code to use any machine learning function and parameter set (linear regression)

Cross-validation is a technique that assesses how well a model trained on a known set of data will generalize to predict the features of data sets on which it has not been trained. The general idea behind this technique is that a model is trained on a data set of known data, and then the accuracy of its predictions is tested against an independent data set. A common implementation is to divide a data set into k -folds, and then train the model in a round-robin fashion on all but one of the folds.

Hyper-parameter optimization is the problem of choosing a set of hyper-parameters for a learning algorithm, usually with the goal of optimizing a measure of the algorithm's performance on an independent data set. A hyper-parameter is a value that you must specify outside the model training procedure. Assumptions about hyper-parameter values can affect the flexibility and accuracy of the model. Decision trees have hyper-parameters, for example, such as the desired depth and number of leaves in the tree. You must set a misclassification penalty term for a support vector machine (SVM).

A common way to perform hyper-parameter optimization is to use a grid search, also called a **parameter sweep**. In a grid search, an exhaustive search is performed through the values of a specified subset of the hyper-parameter space for a learning algorithm. Cross-validation can supply a performance metric to sort out the optimal results produced by the grid search algorithm. If you use cross-validation hyper-parameter sweeping, you can help limit

problems like overfitting a model to training data. This way, the model retains the capacity to apply to the general set of data from which the training data was extracted.

Optimize a linear regression model with hyper-parameter sweeping

Next, split data into train and validation sets, use hyper-parameter sweeping on a training set to optimize the model, and evaluate on a validation set (linear regression).

```
# RECORD THE START TIME
val starttime = Calendar.getInstance().getTime()

# RENAME `tip_amount` AS A LABEL
val OneHotTRAINLabeled =
OneHotTRAIN.select("tip_amount","features").withColumnRenamed(existingName="tip_amount",newName="label")
val OneHotTESTLabeled =
OneHotTEST.select("tip_amount","features").withColumnRenamed(existingName="tip_amount",newName="label")
OneHotTRAINLabeled.cache()
OneHotTESTLabeled.cache()

# DEFINE THE ESTIMATOR FUNCTION: `THE LinearRegression()` FUNCTION
val lr = new LinearRegression().setLabelCol("label").setFeaturesCol("features").setMaxIter(10)

# DEFINE THE PARAMETER GRID
val paramGrid = new ParamGridBuilder().addGrid(lr.regParam, Array(0.1, 0.01,
0.001)).addGrid(lr.fitIntercept).addGrid(lr.elasticNetParam, Array(0.1, 0.5, 0.9)).build()

# DEFINE THE PIPELINE WITH A TRAIN/TEST VALIDATION SPLIT (75% IN THE TRAINING SET), AND THEN THE SPECIFY
ESTIMATOR, EVALUATOR, AND PARAMETER GRID
val trainPct = 0.75
val trainValidationSplit = new TrainValidationSplit().setEstimator(lr).setEvaluator(new
RegressionEvaluator).setEstimatorParamMaps(paramGrid).setTrainRatio(trainPct)

# RUN THE TRAIN VALIDATION SPLIT AND CHOOSE THE BEST SET OF PARAMETERS
val model = trainValidationSplit.fit(OneHotTRAINLabeled)

# MAKE PREDICTIONS ON THE TEST DATA BY USING THE MODEL WITH THE COMBINATION OF PARAMETERS THAT PERFORMS THE
BEST
val testResults = model.transform(OneHotTESTLabeled).select("label", "prediction")

# COMPUTE TEST SET R2
val evaluator = new
RegressionEvaluator().setLabelCol("label").setPredictionCol("prediction").setMetricName("r2")
val Test_R2 = evaluator.evaluate(testResults)

# GET THE TIME TO RUN THE CELL
val endtime = Calendar.getInstance().getTime()
val elapsedtime = ((endtime.getTime() - starttime.getTime())/1000).toString;
println("Time taken to run the above cell: " + elapsedtime + " seconds.");

println("Test R-sqr is: " + Test_R2);
```

Output:

Test R-sqr is: 0.6226484708501209

Optimize the binary classification model by using cross-validation and hyper-parameter sweeping

This section shows you how to optimize a binary classification model by using cross-validation and hyper-parameter sweeping. This uses the Spark ML `CrossValidator` function.

```

# RECORD THE START TIME
val starttime = Calendar.getInstance().getTime()

# CREATE DATA FRAMES WITH PROPERLY LABELED COLUMNS TO USE WITH THE TRAIN AND TEST SPLIT
val indexedTRAINwithCatFeatBinTargetRF =
indexedTRAINwithCatFeatBinTarget.select("labelBin","featuresCat").withColumnRenamed(existingName="labelBin",newName="label").withColumnRenamed(existingName="featuresCat",newName="features")
val indexedTESTwithCatFeatBinTargetRF =
indexedTESTwithCatFeatBinTarget.select("labelBin","featuresCat").withColumnRenamed(existingName="labelBin",newName="label").withColumnRenamed(existingName="featuresCat",newName="features")
indexedTRAINwithCatFeatBinTargetRF.cache()
indexedTESTwithCatFeatBinTargetRF.cache()

# DEFINE THE ESTIMATOR FUNCTION
val rf = new RandomForestClassifier().setLabelCol("label").setFeaturesCol("features").setImpurity("gini").setSeed(1234).setFeatureSubsetStrategy("auto").setMaxBins(32)

# DEFINE THE PARAMETER GRID
val paramGrid = new ParamGridBuilder().addGrid(rf.maxDepth, Array(4,8)).addGrid(rf.numTrees,Array(5,10)).addGrid(rf.minInstancesPerNode, Array(100,300)).build()

# SPECIFY THE NUMBER OF FOLDS
val numFolds = 3

# DEFINE THE TRAIN/TEST VALIDATION SPLIT (75% IN THE TRAINING SET)
val CrossValidator = new CrossValidator().setEstimator(rf).setEvaluator(new BinaryClassificationEvaluator).setEstimatorParamMaps(paramGrid).setNumFolds(numFolds)

# RUN THE TRAIN VALIDATION SPLIT AND CHOOSE THE BEST SET OF PARAMETERS
val model = CrossValidator.fit(indexedTRAINwithCatFeatBinTargetRF)

# MAKE PREDICTIONS ON THE TEST DATA BY USING THE MODEL WITH THE COMBINATION OF PARAMETERS THAT PERFORMS THE BEST
val testResults = model.transform(indexedTESTwithCatFeatBinTargetRF).select("label", "prediction")

# COMPUTE THE TEST F1 SCORE
val evaluator = new MulticlassClassificationEvaluator().setLabelCol("label").setPredictionCol("prediction").setMetricName("f1")
val Test_f1Score = evaluator.evaluate(testResults)

# GET THE TIME TO RUN THE CELL
val endtime = Calendar.getInstance().getTime()
val elapsedtime = ((endtime.getTime() - starttime.getTime())/1000).toString;
println("Time taken to run the above cell: " + elapsedtime + " seconds.");

```

Output:

Time to run the cell: 33 seconds.

Optimize the linear regression model by using custom cross-validation and parameter-sweeping code

Next, optimize the model by using custom code, and identify the best model parameters by using the criterion of highest accuracy. Then, create the final model, evaluate the model on test data, and save the model in Blob storage. Finally, load the model, score test data, and evaluate accuracy.

```

# RECORD THE START TIME
val starttime = Calendar.getInstance().getTime()

# DEFINE THE PARAMETER GRID AND THE NUMBER OF FOLDS
val paramGrid = new ParamGridBuilder().addGrid(rf.maxDepth, Array(5,10)).addGrid(rf.numTrees,Array(10,25,50)).build()

val nFolds = 3
val numModels = paramGrid.size
val numParamsinGrid = 2

```

```

# SPECIFY THE NUMBER OF CATEGORIES FOR CATEGORICAL VARIABLES
val categoricalFeaturesInfo = Map[Int, Int]((0,2),(1,2),(2,6),(3,4))

var maxDepth = -1
var numTrees = -1
var param = ""
var paramval = -1
var validateLB = -1.0
var validateUB = -1.0
val h = 1.0 / nFolds;
val RMSE = Array.fill(numModels)(0.0)

# CREATE K-FOLDS
val splits = MLUtils.kFold(indexedTRAINbinary, numFolds = nFolds, seed=1234)

# LOOP THROUGH K-FOLDS AND THE PARAMETER GRID TO GET AND IDENTIFY THE BEST PARAMETER SET BY LEVEL OF ACCURACY
for (i <- 0 to (nFolds-1)) {
    validateLB = i * h
    validateUB = (i + 1) * h
    val validationCV = trainData.filter($"rand" >= validateLB && $"rand" < validateUB)
    val trainCV = trainData.filter($"rand" < validateLB || $"rand" >= validateUB)
    val validationLabPt = validationCV.rdd.map(r => LabeledPoint(r.getDouble(targetIndRegression(0).toInt),
    Vectors.dense(featuresIndIndex.map(r.getDouble(_)).toArray)));
    val trainCVLabPt = trainCV.rdd.map(r => LabeledPoint(r.getDouble(targetIndRegression(0).toInt),
    Vectors.dense(featuresIndIndex.map(r.getDouble(_)).toArray)));
    validationLabPt.cache()
    trainCVLabPt.cache()

    for (nParamSets <- 0 to (numModels-1)) {
        for (nParams <- 0 to (numParamsinGrid-1)) {
            param = paramGrid(nParamSets).toSeq(nParams).param.toString.split("__")(1)
            paramval = paramGrid(nParamSets).toSeq(nParams).value.toString.toInt
            if (param == "maxDepth") {maxDepth = paramval}
            if (param == "numTrees") {numTrees = paramval}
        }
        val rfModel = RandomForest.trainRegressor(trainCVLabPt,
categoricalFeaturesInfo=categoricalFeaturesInfo,
                                         numTrees=numTrees, maxDepth=maxDepth,
                                         featureSubsetStrategy="auto", impurity="variance", maxBins=32)
        val labelAndPreds = validationLabPt.map { point =>
            val prediction = rfModel.predict(point.features)
            ( prediction, point.label )
        }
        val validMetrics = new RegressionMetrics(labelAndPreds)
        val rmse = validMetrics.rootMeanSquaredError
        RMSE(nParamSets) += rmse
    }
    validationLabPt.unpersist();
    trainCVLabPt.unpersist();
}
val minRMSEindex = RMSE.indexOf(RMSE.min)

# GET THE BEST PARAMETERS FROM A CROSS-VALIDATION AND PARAMETER SWEEP
var best_maxDepth = -1
var best_numTrees = -1
for (nParams <- 0 to (numParamsinGrid-1)) {
    param = paramGrid(minRMSEindex).toSeq(nParams).param.toString.split("__")(1)
    paramval = paramGrid(minRMSEindex).toSeq(nParams).value.toString.toInt
    if (param == "maxDepth") {best_maxDepth = paramval}
    if (param == "numTrees") {best_numTrees = paramval}
}

# CREATE THE BEST MODEL WITH THE BEST PARAMETERS AND A FULL TRAINING DATA SET
val best_rfModel = RandomForest.trainRegressor(indexedTRAINreg,
categoricalFeaturesInfo=categoricalFeaturesInfo,
                                         numTrees=best_numTrees, maxDepth=best_maxDepth,
                                         featureSubsetStrategy="auto", impurity="variance", maxBins=32)

```

```

# SAVE THE BEST RANDOM FOREST MODEL IN BLOB STORAGE
val datestamp = Calendar.getInstance().getTime().toString.replaceAll(" ", ".").replaceAll(":", "_");
val modelName = "BestCV_RF_Regression_"
val filename = modelDir.concat(modelName).concat(datestamp)
best_rfModel.save(sc, filename);

# PREDICT ON THE TRAINING SET WITH THE BEST MODEL AND THEN EVALUATE
val labelAndPreds = indexedTESTreg.map { point =>
    val prediction = best_rfModel.predict(point.features)
    ( prediction, point.label )
}

val test_rmse = new RegressionMetrics(labelAndPreds).rootMeanSquaredError
val test_rsqr = new RegressionMetrics(labelAndPreds).r2

# GET THE TIME TO RUN THE CELL
val endtime = Calendar.getInstance().getTime()
val elapsedtime = ((endtime.getTime() - starttime.getTime()) / 1000).toString;
println("Time taken to run the above cell: " + elapsedtime + " seconds.");

# LOAD THE MODEL
val savedRFModel = RandomForestModel.load(sc, filename)

val labelAndPreds = indexedTESTreg.map { point =>
    val prediction = savedRFModel.predict(point.features)
    ( prediction, point.label )
}

# TEST THE MODEL
val test_rmse = new RegressionMetrics(labelAndPreds).rootMeanSquaredError
val test_rsqr = new RegressionMetrics(labelAndPreds).r2

```

Output:

Time to run the cell: 61 seconds.

Consume Spark-built machine learning models automatically with Scala

For an overview of topics that walk you through the tasks that comprise the Data Science process in Azure, see [Team Data Science Process](#).

[Team Data Science Process walkthroughs](#) describes other end-to-end walkthroughs that demonstrate the steps in the Team Data Science Process for specific scenarios. The walkthroughs also illustrate how to combine cloud and on-premises tools and services into a workflow or pipeline to create an intelligent application.

[Score Spark-built machine learning models](#) shows you how to use Scala code to automatically load and score new data sets with machine learning models built in Spark and saved in Azure Blob storage. You can follow the instructions provided there, and simply replace the Python code with Scala code in this article for automated consumption.

Use Caffe on Azure HDInsight Spark for distributed deep learning

8/16/2017 • 12 min to read • [Edit Online](#)

Introduction

Deep learning is impacting everything from healthcare to transportation to manufacturing, and more. Companies are turning to deep learning to solve hard problems, like [image classification](#), [speech recognition](#), object recognition, and machine translation.

There are [many popular frameworks](#), including [Microsoft Cognitive Toolkit](#), [Tensorflow](#), MXNet, Theano, etc. Caffe is one of the most famous non-symbolic (imperative) neural network frameworks, and widely used in many areas including computer vision. Furthermore, [CaffeOnSpark](#) combines Caffe with Apache Spark, in which case deep learning can be easily used on an existing Hadoop cluster together with Spark ETL pipelines, reducing system complexity and latency for end-to-end learning.

[HDInsight](#) is the only fully-managed cloud Hadoop offering that provides optimized open source analytic clusters for Spark, Hive, MapReduce, HBase, Storm, Kafka, and R Server backed by a 99.9% SLA. Each of these big data technologies and ISV applications are easily deployable as managed clusters with enterprise-level security and monitoring.

Some users are asking us about how to use deep learning on HDInsight, which is Microsoft's PaaS Hadoop product. We will have more to share in the future, but today we want to summarize a technical blog on how to use Caffe on HDInsight Spark.

If you have installed Caffe before, you will notice that installing this framework is a little bit challenging. In this blog, we will first illustrate how to install [Caffe on Spark](#) for an HDInsight cluster, then use the built-in MNIST demo to demonstrate how to use Distributed Deep Learning using HDInsight Spark on CPUs.

There are four major steps to get it work on HDInsight.

1. Install the required dependencies on all the nodes
2. Build Caffe on Spark for HDInsight on the head node
3. Distribute the required libraries to all the worker nodes
4. Compose a Caffe model and run it distributely

Since HDInsight is a PaaS solution, it offers great platform features - so it is quite easy to perform some tasks. One of the features that we heavily use in this blog post is called [Script Action](#), with which you can execute shell commands to customize cluster nodes (head node, worker node, or edge node).

Step 1: Install the required dependencies on all the nodes

To get started, we need to install the dependencies we need. The Caffe site and [CaffeOnSpark site](#) offers some very useful wiki for installing the dependencies for Spark on YARN mode (which is the mode for HDInsight Spark), but we need to add a few more dependencies for HDInsight platform. We will use the script action as below and run it on all the head nodes and worker nodes. This script action will take about 20 minutes, as those dependencies also depend on other packages. You should put it in some location that is accessible to your HDInsight cluster, such as a GitHub location or the default BLOB storage account.

```

#!/bin/bash
#Please be aware that installing the below will add additional 20 mins to cluster creation because of the
dependencies
#installing all dependencies, including the ones mentioned in http://caffe.berkeleyvision.org/install_apt.html,
as well a few packages that are not included in HDInsight, such as gflags, glog, lmdb, numpy
#It seems numpy will only needed during compilation time, but for safety purpose we install them on all the
nodes

sudo apt-get install -y libprotobuf-dev libleveldb-dev libsnappy-dev libopencv-dev libhdf5-serial-dev protobuf-
compiler maven libatlas-base-dev libgflags-dev libgoogle-glog-dev liblmdb-dev build-essential libboost-all-dev
python-numpy python-scipy python-matplotlib ipython ipython-notebook python-pandas python-sympy python-nose

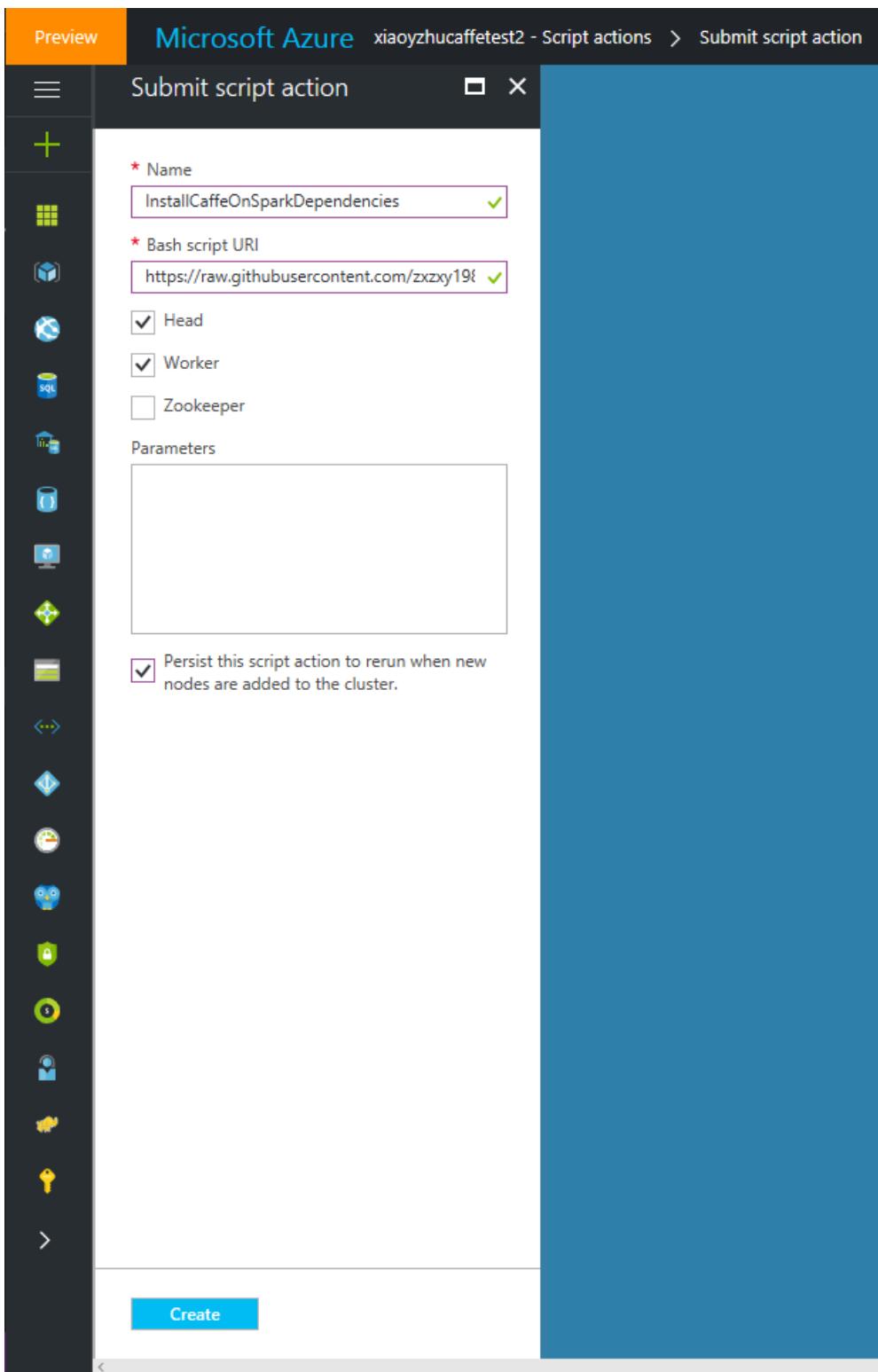
#install protobuf
wget https://github.com/google/protobuf/releases/download/v2.5.0/protobuf-2.5.0.tar.gz
sudo tar xzvf protobuf-2.5.0.tar.gz -C /tmp/
cd /tmp/protobuf-2.5.0/
sudo ./configure
sudo make
sudo make check
sudo make install
sudo ldconfig
echo "protobuf installation done"

```

There are two steps in the script action above. The first step is to install all the required libraries. Those libraries include the necessary libraries for both compiling Caffe(such as gflags, glog) and running Caffe (such as numpy). We are using libatlas for CPU optimization, but you can always follow the CaffeOnSpark wiki on installing other optimization libraries, such as MKL or CUDA (for GPU).

The second step is to download, compile, and install protobuf 2.5.0 for Caffe during runtime. Protobuf 2.5.0 [is required](#), however this version is not available as a package on Ubuntu 16, so we need to compile it from the source code. There are also a few resources on the Internet on how to compile it, such as [this](#)

To simply get started, you can just run this script action against your cluster to all the worker nodes and head nodes (for HDInsight 3.5). You can either run the script actions for a running cluster, or you can also run the script actions during the cluster provision time. For more details on the script actions, please see the documentation [here](#)



Step 2: Build Caffe on Spark for HDInsight on the head node

The second step is to build Caffe on the headnode, and then distribute the compiled libraries to all the worker nodes. In this step, you will need to [ssh into your headnode](#), then simply follow the [CaffeOnSpark build process](#), and below is the script you can use to build CaffeOnSpark with a few additional steps.

```

#!/bin/bash
git clone https://github.com/yahoo/CaffeOnSpark.git --recursive
export CAFFE_ON_SPARK=$(pwd)/CaffeOnSpark

pushd ${CAFFE_ON_SPARK}/caffe-public/
cp Makefile.config.example Makefile.config
echo "INCLUDE_DIRS += ${JAVA_HOME}/include" >> Makefile.config
#Below configurations might need to be updated based on actual cases. For example, if you are using GPU, or
using a different BLAS library, you may want to update those settings accordingly.
echo "CPU_ONLY := 1" >> Makefile.config
echo "BLAS := atlas" >> Makefile.config
echo "INCLUDE_DIRS += /usr/include/hdf5/serial/" >> Makefile.config
echo "LIBRARY_DIRS += /usr/lib/x86_64-linux-gnu/hdf5/serial/" >> Makefile.config
popd

#compile CaffeOnSpark
pushd ${CAFFE_ON_SPARK}
#always clean up the environment before building (especially when rebuiding), or there will be errors such as
"failed to execute goal org.apache.maven.plugins:maven-antrun-plugin:1.7:run (proto) on project caffe-distri:
An Ant BuildException has occured: exec returned: 2"
make clean
#the build step usually takes 20~30 mins, since it has a lot maven dependencies
make build
popd
export LD_LIBRARY_PATH=${CAFFE_ON_SPARK}/caffe-public/distribute/lib:${CAFFE_ON_SPARK}/caffe-
distri/distribute/lib

hadoop fs -mkdir -p wasb:///projects/machine_learning/image_dataset

${CAFFE_ON_SPARK}/scripts/setup-mnist.sh
hadoop fs -put -f ${CAFFE_ON_SPARK}/data/mnist_*_lmdb wasb:///projects/machine_learning/image_dataset/

${CAFFE_ON_SPARK}/scripts/setup-cifar10.sh
hadoop fs -put -f ${CAFFE_ON_SPARK}/data/cifar10_*_lmdb wasb:///projects/machine_learning/image_dataset/

#put the already compiled CaffeOnSpark libraries to wasb storage, then read back to each node using script
actions. This is because CaffeOnSpark requires all the nodes have the libaries
hadoop fs -mkdir -p /CaffeOnSpark/caffe-public/distribute/lib/
hadoop fs -mkdir -p /CaffeOnSpark/caffe-distri/distribute/lib/
hadoop fs -put CaffeOnSpark/caffe-distri/distribute/lib/* /CaffeOnSpark/caffe-distri/distribute/lib/
hadoop fs -put CaffeOnSpark/caffe-public/distribute/lib/* /CaffeOnSpark/caffe-public/distribute/lib/

```

You may need to do more than what the documentation of CaffeOnSpark says. The changes are:

- Change to CPU only and use libatlas for this particular purpose.
- Put the datasets to the BLOB storage, which is a shared location that is accessible to all worker nodes for later use.
- Put the compiled Caffe libraries to BLOB storage, and later you will copy those libraries to all the nodes using script actions to avoid additional compilation time.

Troubleshooting: An Ant BuildException has occured: exec returned: 2

When first trying to build CaffeOnSpark, sometimes it will say

```

failed to execute goal org.apache.maven.plugins:maven-antrun-plugin:1.7:run (proto) on project caffe-distri: An
Ant BuildException has occured: exec returned: 2

```

Simply clean the code repository by "make clean" and then run "make build" will solve this issue, as long as you have the correct dependencies.

Troubleshooting: Maven repository connection time out

Sometimes maven gives me the connection time out error, similar to below:

```
Retry:  
[INFO] Downloading: https://repo.maven.apache.org/maven2/com/twitter/chill_2.11/0.8.0/chill_2.11-0.8.0.jar  
Feb 01, 2017 5:14:49 AM org.apache.maven.providers.http.httpclient.impl.execchain.RetryExec execute  
INFO: I/O exception (java.net.SocketException) caught when processing request to {s}-  
>https://repo.maven.apache.org:443: Connection timed out (Read failed)
```

It will be OK after waiting for a few minutes and then just try to rebuild the code, so it might be Maven somehow limits the traffic from a given IP address.

Troubleshooting: Test failure for Caffe

You probably will see a test failure when doing the final check for CaffeOnSpark, similar with below. This is probably related with UTF-8 encoding, but should not impact the usage of Caffe

```
Run completed in 32 seconds, 78 milliseconds.  
Total number of tests run: 7  
Suites: completed 5, aborted 0  
Tests: succeeded 6, failed 1, canceled 0, ignored 0, pending 0  
*** 1 TEST FAILED ***
```

Step 3: Distribute the required libraries to all the worker nodes

The next step is to distribute the libraries (basically the libraries in CaffeOnSpark/caffe-public/distribute/lib/ and CaffeOnSpark/caffe-distri/distribute/lib/) to all the nodes. In Step 2, we put those libraries on BLOB storage, and in this step, we will use script actions to copy it to all the head nodes and worker nodes.

To do this, simple run a script action as below (you need to point to the right location specific to your cluster):

```
#!/bin/bash  
hadoop fs -get wasb:///CaffeOnSpark /home/changetoyourusername/
```

Because in step 2, we put it on the BLOB storage which is accessible to all the nodes, in this step we just simply copy it to all the nodes.

Step 4: Compose a Caffe model and run it distributely

After running the above steps, Caffe is already installed on the headnode and we are good to go. The next step is to write a Caffe model.

Caffe is using an "expressive architecture", where for composing a model, you just need to define a configuration file, and without coding at all (in most cases). So let's take a look there.

The model we will train today is a sample model for MNIST training. The MNIST database of handwritten digits has a training set of 60,000 examples, and a test set of 10,000 examples. It is a subset of a larger set available from NIST. The digits have been size-normalized and centered in a fixed-size image. CaffeOnSpark has some scripts to download the dataset and convert it into the right format.

CaffeOnSpark provides some network topologies example for MNIST training. It has a nice design of splitting the network architecture (the topology of the network) and optimization. In this case, There are two files required:

the "Solver" file (`$(CAFFE_ON_SPARK)/data/lenet_memory_solver.prototxt`) is used for overseeing the optimization and generating parameter updates. For example, it defines whether CPU or GPU will be used, what's the momentum, how many iterations will be, etc. It also defines which neuron network topology should the program use (which is the second file we need). For more information about Solver, please refer to [Caffe documentation](#).

For this example, since we are using CPU rather than GPU, we should change the last line to:

```
# solver mode: CPU or GPU  
solver_mode: CPU
```

```
# The train/test net protocol buffer definition  
net: "lenet_memory_train_test.prototxt"  
# test_iter specifies how many forward passes the test should carry out.  
test_iter: 10  
# Carry out testing every 100 training iterations.  
test_interval: 100  
# The base learning rate, momentum and the weight decay of the network.  
base_lr: 0.01  
momentum: 0.9  
weight_decay: 0.0005  
# The learning rate policy  
lr_policy: "inv"  
gamma: 0.0001  
power: 0.75  
# Display every 100 iterations  
display: 100  
# The maximum number of iterations  
max_iter: 2000  
# snapshot intermediate results  
snapshot: 5000  
snapshot_prefix: "mnist_lenet"  
# solver mode: CPU or GPU  
solver_mode: CPU
```

You can change other lines as needed.

The second file (`$(CAFFE_ON_SPARK)/data/lenet_memory_train_test.prototxt`) defines how the neuron network looks like, and the relevant input and output file. We also need to update the file to reflect the training data location. Change the following part in `lenet_memory_train_test.prototxt` (you need to point to the right location specific to your cluster):

- change the "file:/Users/mridul/bigml/demodl/mnist_train_lmdb" to "wasb://projects/machine_learning/image_dataset/mnist_train_lmdb"
- change "file:/Users/mridul/bigml/demodl/mnist_test_lmdb/" to "wasb://projects/machine_learning/image_dataset/mnist_test_lmdb"

```
name: "LeNet"  
layer {  
    name: "data"  
    type: "MemoryData"  
    top: "data"  
    top: "label"  
    include {  
        phase: TRAIN  
    }  
    source_class: "com.yahoo.ml.caffe.LMDB"  
    memory_data_param {  
        source: "wasb:///projects/machine_learning/image_dataset/mnist_train_lmdb"  
        batch_size: 64  
        channels: 1  
        height: 28  
        width: 28  
        share_in_parallel: false  
    }  
    transform_param {  
        scale: 0.00390625  
    }  
}  
layer {  
<feOnSpark/data/lenet_memory_train_test.prototxt" 176L, 2568C 1,1
```

For more information on how to define the network, please check the [Caffe documentation on MNIST dataset](#)

For the purpose of this blog, we just use this simple MNIST example. You should run the command below from the

head node:

```
spark-submit --master yarn --deploy-mode cluster --num-executors 8 --files
${CAFFE_ON_SPARK}/data/lenet_memory_solver.prototxt,${CAFFE_ON_SPARK}/data/lenet_memory_train_test.prototxt --
conf spark.driver.extraLibraryPath="${LD_LIBRARY_PATH}" --conf
spark.executorEnv.LD_LIBRARY_PATH="${LD_LIBRARY_PATH}" --class com.yahoo.ml.caffe.CaffeOnSpark
${CAFFE_ON_SPARK}/caffe-grid/target/caffe-grid-0.1-SNAPSHOT-jar-with-dependencies.jar -train -features
accuracy,loss -label label -conf lenet_memory_solver.prototxt -devices 1 -connection ethernet -model
wasb:///mnist.model -output wasb:///mnist_features_result
```

Basically it distributes the required files (lenet_memory_solver.prototxt and lenet_memory_train_test.prototxt) to each YARN container, and also set the relevant PATH of each Spark driver/executor to LD_LIBRARY_PATH, which is defined in the previous code snippet and points to the location that has CaffeOnSpark libraries.

Monitoring and troubleshooting

Since we are using YARN cluster mode, in which case the Spark driver will be scheduled to an arbitrary container (and an arbitrary worker node) you should only see in the console outputting something like:

```
17/02/01 23:22:16 INFO Client: Application report for application_1485916338528_0015 (state: RUNNING)
```

If you want to know what happened, you usually need to get the Spark driver's log, which has more information. In this case, you need to go to the YARN UI to find the relevant YARN logs. You can get the YARN UI by this URL:

<https://yourclustername.azurehdinsight.net/yarnui>

ID	User	Name	Application Type	Queue	Application Priority	StartTime	FinishTime	State	FinalStatus	Running Containers	Allocated CPU	Allocated Memory	% of Queue	% of Cluster	Progress
application_1485916338528_0014	xiaoyzhu	com.yahoo.ml.caffe.CaffeOnSpark	SPARK	default	0	Tue Jan 31 23:30:12 2017	Tue Jan 31 23:35:24 -0800 2017	FINISHED	SUCCEEDED	N/A	N/A	N/A	0.0	0.0	
application_1485916338528_0013	xiaoyzhu	com.yahoo.ml.caffe.CaffeOnSpark	SPARK	default	0	Tue Jan 31 23:27:58 -0800 2017	Tue Jan 31 23:29:55 -0800 2017	KILLED	KILLED	N/A	N/A	N/A	0.0	0.0	
application_1485916338528_0012	xiaoyzhu	com.yahoo.ml.caffe.CaffeOnSpark	SPARK	default	0	Tue Jan 31 23:27:08 -0800 2017	Tue Jan 31 23:29:49 -0800 2017	KILLED	KILLED	N/A	N/A	N/A	0.0	0.0	
application_1485916338528_0011	xiaoyzhu	com.yahoo.ml.caffe.CaffeOnSpark	SPARK	default	0	Tue Jan 31 23:26:34 -0800 2017	Tue Jan 31 23:28:56 -0800 2017	FINISHED	FAILED	N/A	N/A	N/A	0.0	0.0	
application_1485916338528_0010	xiaoyzhu	com.yahoo.ml.caffe.CaffeOnSpark	SPARK	default	0	Tue Jan 31 23:26:34 -0800 2017	Tue Jan 31 23:28:56 -0800 2017	FINISHED	FAILED	N/A	N/A	N/A	0.0	0.0	

You can take a look at how many resources are allocated for this particular application. You can click the "Scheduler" link, and then you will see that for this application, there are 9 containers running. We ask YARN to provide 8 executors, and another container is for driver process.

NEW,NEW_SAVING,SUBMITTED,ACCEPTED,RUNNING,FINISHED,FAILED,KILLED

Scheduler Metrics

Nodes	Apps Submitted	Apps Pending	Apps Running	Containers Running	Memory Used	Memory Total	Memory Reserved	Vcores Used	Vcores Total	Vcores Reserved	Active Nodes	Decommissioned Nodes	Lost Nodes	Unred
13	0	0	3	11	44.50 GB	100 GB	0 B	11	60	0	4	0	0	0

Scheduler Type: Capacity Scheduler **Scheduling Resource Type**: [MEMORY] **Minimum Allocation**: <memory:512, vCores:1> **Maximum Allocation**: <memory:25600, vCores:15>

Dump scheduler logs: 1 min

Application Queues

Legend: Capacity : Used : Used (over capacity) : Max Capacity :

- Queues root
- Queue: thriftsvr
- Queue: default

44.5% used
6.0% used
83.3% used

Show 20 entries

ID	User	Name	Application Type	Queue	Application Priority	StartTime	FinishTime	State	FinalStatus	Running Containers	Allocated CPU	Allocated Memory MB	% of Queue	% of Cluster	Progress
application_1485916338528_0015	xiaoyzhu	com.yahoo.ml.caffe.CaffeOnSpark	SPARK	default	0	Wed Feb 1 19:21:58 2017	N/A	RUNNING	UNDEFINED	9	9	42496	83.0	41.5	
application_1485916338528_0002	hive	org.apache.spark.sql.hive.thriftserver.HiveThriftServer2	SPARK	thriftsvr	0	Tue Jan 31 18:33:47 2017	N/A	RUNNING	UNDEFINED	1	1	1536	3.0	1.5	
application_1485916338528_0001	hive	org.apache.spark.sql.hive.thriftserver.HiveThriftServer2	SPARK	thriftsvr	0	Tue Jan 31 18:33:46 2017	N/A	RUNNING	UNDEFINED	1	1	1536	3.0	1.5	

Showing 1 to 3 of 3 entries

Aggregate scheduler counts

Total Container Allocations(count)	Total Container Releases(count)	Total Fulfilled Reservations(count)	Total Container Preempted(count)
375	364	13	6

Last scheduler run

Time	Allocations(count - resources)	Reservations(count - resources)	Releases(count)
Wed Feb 01 23:26:42 +0000 2017	0 - <memory:0, vCores:0>	0 - <memory:0, vCores:0>	0 - <memory:0, vCores:0>

Last Preemption

You may want to check the driver logs or container logs if there are failures. For driver logs, you can click the application ID in YARN UI, then click the "Logs" button. The driver logs are written into stderr.

Application application_1485916338528_0008

Driver stacktrace:

```

Dlog4jspark.log.file=sparkexecutor.log"-Dlog4j.configuration=file:/usr/hdp/current/spark2-client/conf/log4j.properties"-Djava.xml.parsers.SAXParserFactory=com.sun.org.apache
Djava.io.tmpdir=/mnt/resource/hadoop/yarn/local/usercache/xiaoyzhu/appcache/application_1485916338528_0008/container/_1485916338528_0008_05_000011/tmp`-Dspark.driver
Dspark.ui.port=0`-Dspark.yarn.app.container.log.dir=/mnt/resource/hadoop/yarn/log/application_1485916338528_0008/container_1485916338528_0008_05_000011-XX`OnOutC
org.apache.spark.executor.CoarseGrainedExecutorBackend--driver-uri spark://CoarseGrainedScheduler@10.0.0.13.43942--executor-id 7--hostname 10.0.0.13--cores 3--app
file:/mnt/resource/hadoop/yarn/local/usercache/xiaoyzhu/appcache/application_1485916338528_0008/container_1485916338528_0008_05_000011/_app__.jar>
/mnt/resource/hadoop/yarn/log/application_1485916338528_0008/container_1485916338528_0008_05_000011/stdout>
/mnt/resource/hadoop/yarn/log/application_1485916338528_0008/container_1485916338528_0008_05_000011/stderr>
at org.apache.hadoop.util.Shell runCommand(Shell.java:933)
at org.apache.hadoop.util.Shell run(Shell.java:844)
at org.apache.hadoop.util.Shell$ShellCommandExecutor execute(Shell.java:1123)
at org.apache.hadoop.yarn.server.nodemanager.DefaultContainerExecutor.launchContainer(DefaultContainerExecutor.java:225)
at org.apache.hadoop.yarn.server.nodemanager.ContainerManager.launchContainer(ContainerManager.java:317)
at org.apache.hadoop.yarn.server.nodemanager.ContainerManager.launchContainer(ContainerManager.java:63)
at java.util.concurrent.FutureTask.run(FutureTask.java:266)
at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1142)
at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:617)
at java.lang.Thread.run(Thread.java:745)
Container exited with a non-zero exit code 134
Driver stacktrace:
false

```

Unmanaged Application: <Not set>

Application Node: <DEFAULT_PARTITION>

Label expression: <DEFAULT_PARTITION>

Total Resource Preempted: <memory:0, vCores:0>
Total Number of Non-AM Containers Preempted: 0
Total Number of AM Containers Preempted: 0
Resource Preempted from Current Attempt: <memory:0, vCores:0>
Number of Non-AM Containers Preempted from Current Attempt: 0
Aggregate Resource Allocation: 4651331 MB-seconds, 1052 vcore-second

Show 20 entries

Attempt ID	Started	Node	Logs
appattempt_1485916338528_0008_000005	Tue Jan 31 23:11:47 -0800 2017	http://10.0.0.15.30060	Logs
appattempt_1485916338528_0008_000004	Tue Jan 31 23:11:11 -0800 2017	http://10.0.0.15.30060	Logs
appattempt_1485916338528_0008_000003	Tue Jan 31 23:10:29 -0800 2017	http://10.0.0.15.30060	Logs
appattempt_1485916338528_0008_000002	Tue Jan 31 23:09:52 -0800 2017	http://10.0.0.15.30060	Logs
appattempt_1485916338528_0008_000001	Tue Jan 31 23:08:08 -0800 2017	http://10.0.0.13.30060	Logs

Showing 1 to 5 of 5 entries

For example, you might see some of the error below from the driver logs, indicating you allocate too many executors.

```
17/02/01 07:26:06 ERROR ApplicationMaster: User class threw exception: java.lang.IllegalStateException:  
Insufficient training data. Please adjust hyperparameters or increase dataset.  
java.lang.IllegalStateException: Insufficient training data. Please adjust hyperparameters or increase dataset.  
at com.yahoo.ml.caffe.CaffeOnSpark.trainWithValidation(CaffeOnSpark.scala:261)  
at com.yahoo.ml.caffe.CaffeOnSpark$.main(CaffeOnSpark.scala:42)  
at com.yahoo.ml.caffe.CaffeOnSpark.main(CaffeOnSpark.scala)  
at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)  
at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)  
at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)  
at java.lang.reflect.Method.invoke(Method.java:498)  
at org.apache.spark.deploy.yarn.ApplicationMaster$$anon$2.run(ApplicationMaster.scala:627)
```

Sometimes, the issue can happen in executors rather than drivers. In this case, you need to check the container logs. You can always get the container logs, and then get the failed container. For example, you might meet this failure when running Caffe.

```

17/02/01 07:12:05 WARN YarnAllocator: Container marked as failed: container_1485916338528_0008_05_000005 on
host: 10.0.0.14. Exit status: 134. Diagnostics: Exception from container-launch.
Container id: container_1485916338528_0008_05_000005
Exit code: 134
Exception message: /bin/bash: line 1: 12230 Aborted (core dumped)
LD_LIBRARY_PATH=/usr/hdp/current/hadoop-client/lib/native:/usr/hdp/current/hadoop-client/lib/native/Linux-
amd64-64:/home/xiaoyzhu/CaffeOnSpark/caffe-public/distribute/lib:/home/xiaoyzhu/CaffeOnSpark/caffe-
distri/distribute/lib /usr/lib/jvm/java-8-openjdk-amd64/bin/java -server -Xmx4608m '-Dhdp.version=' '-
Detwlogger.component=sparkexecutor' '-DlogFilter.filename=SparkLogFilters.xml' '-
DpatternGroup.filename=SparkPatternGroups.xml' '-Dlog4jspark.root.logger=INFO,console,RFA,ETW,Anonymizer' '-
Dlog4jspark.log.dir=/var/log/sparkapp/${user.name}' '-Dlog4jspark.log.file=sparkexecutor.log' '-
Dlog4j.configuration=file:/usr/hdp/current/spark2-client/conf/log4j.properties' '-
Djavax.xml.parsers.SAXParserFactory=com.sun.org.apache.xerces.internal.jaxp.SAXParserFactoryImpl' -
Djava.io.tmpdir=/mnt/resource/hadoop/yarn/local/usercache/xiaoyzhu/appcache/application_1485916338528_0008/cont
ainer_1485916338528_0008_05_000005/tmp '-Dspark.driver.port=43942' '-Dspark.history.ui.port=18080' '-
Dspark.ui.port=0' -
Dspark.yarn.app.container.log.dir=/mnt/resource/hadoop/yarn/log/application_1485916338528_0008/container_148591
6338528_0008_05_000005 -XX:OnOutOfMemoryError='kill %p' org.apache.spark.executor.CoarseGrainedExecutorBackend
--driver-url spark://CoarseGrainedScheduler@10.0.0.13:43942 --executor-id 4 --hostname 10.0.0.14 --cores 3 --
app-id application_1485916338528_0008 --user-class-path
file:/mnt/resource/hadoop/yarn/local/usercache/xiaoyzhu/appcache/application_1485916338528_0008/container_14859
16338528_0008_05_000005/_app_.jar >
/mnt/resource/hadoop/yarn/log/application_1485916338528_0008/container_1485916338528_0008_05_000005/stdout 2>
/mnt/resource/hadoop/yarn/log/application_1485916338528_0008/container_1485916338528_0008_05_000005/stderr

Stack trace: ExitCodeException exitCode=134: /bin/bash: line 1: 12230 Aborted (core dumped)
LD_LIBRARY_PATH=/usr/hdp/current/hadoop-client/lib/native:/usr/hdp/current/hadoop-client/lib/native/Linux-
amd64-64:/home/xiaoyzhu/CaffeOnSpark/caffe-public/distribute/lib:/home/xiaoyzhu/CaffeOnSpark/caffe-
distri/distribute/lib /usr/lib/jvm/java-8-openjdk-amd64/bin/java -server -Xmx4608m '-Dhdp.version=' '-
Detwlogger.component=sparkexecutor' '-DlogFilter.filename=SparkLogFilters.xml' '-
DpatternGroup.filename=SparkPatternGroups.xml' '-Dlog4jspark.root.logger=INFO,console,RFA,ETW,Anonymizer' '-
Dlog4jspark.log.dir=/var/log/sparkapp/${user.name}' '-Dlog4jspark.log.file=sparkexecutor.log' '-
Dlog4j.configuration=file:/usr/hdp/current/spark2-client/conf/log4j.properties' '-
Djavax.xml.parsers.SAXParserFactory=com.sun.org.apache.xerces.internal.jaxp.SAXParserFactoryImpl' -
Djava.io.tmpdir=/mnt/resource/hadoop/yarn/local/usercache/xiaoyzhu/appcache/application_1485916338528_0008/cont
ainer_1485916338528_0008_05_000005/tmp '-Dspark.driver.port=43942' '-Dspark.history.ui.port=18080' '-
Dspark.ui.port=0' -
Dspark.yarn.app.container.log.dir=/mnt/resource/hadoop/yarn/log/application_1485916338528_0008/container_148591
6338528_0008_05_000005 -XX:OnOutOfMemoryError='kill %p' org.apache.spark.executor.CoarseGrainedExecutorBackend
--driver-url spark://CoarseGrainedScheduler@10.0.0.13:43942 --executor-id 4 --hostname 10.0.0.14 --cores 3 --
app-id application_1485916338528_0008 --user-class-path
file:/mnt/resource/hadoop/yarn/local/usercache/xiaoyzhu/appcache/application_1485916338528_0008/container_14859
16338528_0008_05_000005/_app_.jar >
/mnt/resource/hadoop/yarn/log/application_1485916338528_0008/container_1485916338528_0008_05_000005/stdout 2>
/mnt/resource/hadoop/yarn/log/application_1485916338528_0008/container_1485916338528_0008_05_000005/stderr

at org.apache.hadoop.util.Shell.runCommand(Shell.java:933)
at org.apache.hadoop.util.Shell.run(Shell.java:844)
at org.apache.hadoop.util.Shell$ShellCommandExecutor.execute(Shell.java:1123)
at
org.apache.hadoop.yarn.server.nodemanager.DefaultContainerExecutor.launchContainer(DefaultContainerExecutor.jav
a:225)
at
org.apache.hadoop.yarn.server.nodemanager.containermanager.launcher.ContainerLaunch.call(ContainerLaunch.java:3
17)
at
org.apache.hadoop.yarn.server.nodemanager.containermanager.launcher.ContainerLaunch.call(ContainerLaunch.java:8
3)
at java.util.concurrent.FutureTask.run(FutureTask.java:266)
at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1142)
at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:617)
at java.lang.Thread.run(Thread.java:745)

```

Container exited with a non-zero exit code 134

In this case, you need to get the failed container ID (in the above case, it is container_1485916338528_0008_05_000005). Then you need to run

```
yarn logs -containerId container_1485916338528_0008_03_000005
```

from the headnode. After checking container failure, it is caused by using GPU mode (where you should use CPU mode instead) in lenet_memory_solver.prototxt.

```
17/02/01 07:10:48 INFO LMDB: Batch size:100
WARNING: Logging before InitGoogleLogging() is written to STDERR
F0201 07:10:48.309725 11624 common.cpp:79] Cannot use GPU in CPU-only Caffe: check mode.
```

Getting results

Since we are allocating 8 executors, and the network topology is simple, it should only take around 30 minutes to run the result. From the command line, you can see that we put the model to wasb://mnist.model, and put the results to a folder named wasb://mnist_features_result.

You can get the results by running

```
hadoop fs -cat hdfs://mnist_features_result/*
```

and the result looks like:

```
{"SampleID":"00009597","accuracy":[1.0],"loss": [0.028171852],"label": [2.0]}
 {"SampleID":"00009598","accuracy": [1.0],"loss": [0.028171852],"label": [6.0]}
 {"SampleID":"00009599","accuracy": [1.0],"loss": [0.028171852],"label": [1.0]}
 {"SampleID":"00009600","accuracy": [0.97],"loss": [0.0677709],"label": [5.0]}
 {"SampleID":"00009601","accuracy": [0.97],"loss": [0.0677709],"label": [0.0]}
 {"SampleID":"00009602","accuracy": [0.97],"loss": [0.0677709],"label": [1.0]}
 {"SampleID":"00009603","accuracy": [0.97],"loss": [0.0677709],"label": [2.0]}
 {"SampleID":"00009604","accuracy": [0.97],"loss": [0.0677709],"label": [3.0]}
 {"SampleID":"00009605","accuracy": [0.97],"loss": [0.0677709],"label": [4.0]}
```

The SampleID represents the ID in the MNIST dataset, and the label is the number that the model identifies.

Conclusion

In this documentation, you have tried to install CaffeOnSpark with running a simple example. HDInsight is a full managed cloud distributed compute platform, and is the best place for running machine learning and advanced analytics workloads on large data set, and for distributed deep learning, you can use Caffe on HDInsight Spark to perform deep learning tasks.

See also

- [Overview: Apache Spark on Azure HDInsight](#)

Scenarios

- [Spark with Machine Learning: Use Spark in HDInsight for analyzing building temperature using HVAC data](#)
- [Spark with Machine Learning: Use Spark in HDInsight to predict food inspection results](#)

Manage resources

- [Manage resources for the Apache Spark cluster in Azure HDInsight](#)

Introduction to R Server and open-source R capabilities on HDInsight

8/16/2017 • 8 min to read • [Edit Online](#)

Microsoft R Server is available as a deployment option when you create HDInsight clusters in Azure. This new capability provides data scientists, statisticians, and R programmers with on-demand access to scalable, distributed methods of analytics on HDInsight.

Clusters can be sized appropriately to the projects and tasks at hand and then torn down when they're no longer needed. Since they're part of Azure HDInsight, these clusters come with enterprise-level 24/7 support, an SLA of 99.9% up-time, and the ability to integrate with other components in the Azure ecosystem.

R Server on HDInsight provides the latest capabilities for R-based analytics on datasets of virtually any size, loaded to either Azure Blob or Data Lake storage. Since R Server is built on open source R, the R-based applications you build can leverage any of the 8000+ open source R packages. The routines in ScaleR, Microsoft's big data analytics package included with R Server, are also available.

The edge node of a cluster provides a convenient place to connect to the cluster and to run your R scripts. With an edge node, you have the option of running the parallelized distributed functions of ScaleR across the cores of the edge node server. You can also run them across the nodes of the cluster by using ScaleR's Hadoop Map Reduce or Spark compute contexts.

The models or predictions that result from analyses can be downloaded for use on-premises. They can also be operationalized elsewhere in Azure, in particular through [Azure Machine Learning Studio web service](#).

Get started with R on HDInsight

To include R Server in an HDInsight cluster, you must select the R Server cluster type when creating an HDInsight cluster using the Azure portal. The R Server cluster type includes R Server on the data nodes of the cluster and on an edge node, which serves as a landing zone for R Server-based analytics. See [Getting Started with R Server on HDInsight](#) for a walkthrough on how to create the cluster.

Learn about data storage options

Default storage for the HDFS file system of HDInsight clusters can be associated with either an Azure Storage account or an Azure Data Lake store. This association ensures that whatever data is uploaded to the cluster storage during analysis is made persistent. There are various tools for handling the data transfer to the storage option that you select, including the portal-based upload facility of the storage account and the [AzCopy](#) utility.

You have the option of adding access to additional Blob and Data lake stores during the cluster provisioning process regardless of the primary storage option in use. See [Getting started with R Server on HDInsight](#) article to learn more about using multiple storage accounts.

You can also use [Azure Files](#) as a storage option for use on the edge node. Azure Files enables you to mount a file share that was created in Azure Storage to the Linux file system. For more information about these data storage options for R Server on HDInsight cluster, see [Azure Storage options for R Server on HDInsight clusters](#).

Access R Server on the cluster

You can connect to R Server on the edge node using a browser, provided you've chosen to include RStudio Server during the provisioning process. If you did not install it when provisioning the cluster, you can add it later. For

information about installing RStudio Server after a cluster is created, see [Installing RStudio Server on HDInsight clusters](#). You can also connect to the R Server by using SSH/PuTTY to access the R console.

Develop and run R scripts

The R scripts you create and run can use any of the 8000+ open source R packages in addition to the parallelized and distributed routines available in the ScaleR library. In general, a script that is run with R Server on the edge node runs within the R interpreter on that node. The exceptions are those steps that need to call a ScaleR function with a compute context that is set to Hadoop Map Reduce (RxHadoopMR) or Spark (RxSpark). In this case, the function runs in a distributed fashion across those data (task) nodes of the cluster that are associated with the data referenced. For more information about the different compute context options, see [Compute context options for R Server on HDInsight](#).

Operationalize a model

When your data modeling is complete, you can operationalize the model to make predictions for new data either from Azure and on-premises. This process is known as scoring. Scoring can be done in HDInsight, Azure Machine Learning, or on-premises.

Score in HDInsight

To score in HDInsight, write an R function that calls your model to make predictions for a new data file that you've loaded to your storage account. Then save the predictions back to the storage account. You can run the routine on-demand on the edge node of your cluster or by using a scheduled job.

Score in Azure Machine Learning (AML)

To score using an AML web service, use the open source Azure Machine Learning R package known as [AzureML](#) to publish your model as an Azure web service. For convenience, this package is pre-installed on the edge node. Next, use the facilities in Machine Learning to create a user interface for the web service, and then call the web service as needed for scoring.

If you choose this option, you need to convert any ScaleR model objects to equivalent open-source model objects for use with the web service. Use ScaleR coercion functions, such as `as.randomForest()` for ensemble-based models, for this conversion.

Score on-premises

To score on-premises after creating your model, you can serialize the model in R, download it, de-serialize it, and then use it for scoring new data. You can score new data by using the approach described earlier in [Scoring in HDInsight](#) or by using [DeployR](#).

Maintain the cluster

Install and maintain R packages

Most of the R packages that you use are required on the edge node since most steps of your R scripts run there. To install additional R packages on the edge node, you can use the usual `install.packages()` method in R.

If you are just using routines from the ScaleR library across the cluster, you do not usually need to install additional R packages on the data nodes. However, you might need additional packages to support the use of **rxExec** or **RxDatapartition** execution on the data nodes.

In these cases, the additional packages can be installed with a script action after you create the cluster. For more information, see [Creating an HDInsight cluster with R Server](#).

Change Hadoop Map Reduce memory settings

A cluster can be modified to change the amount of memory that's available to R Server when it's running a Map

Reduce job. To modify a cluster, use the Apache Ambari UI that's available through the Azure portal blade for your cluster. For instructions about how to access the Ambari UI for your cluster, see [Manage HDInsight clusters using the Ambari Web UI](#).

It's also possible to change the amount of memory that's available to R Server by using Hadoop switches in the call to **RxHadoopMR** as follows:

```
hadoopSwitches = "-libjars /etc/hadoop/conf -Dmapred.job.map.memory.mb=6656"
```

Scale your cluster

An existing cluster can be scaled up or down through the portal. By scaling, you can gain the additional capacity that you might need for larger processing tasks, or you can scale back a cluster when it is idle. For instructions about how to scale a cluster, see [Manage HDInsight clusters](#).

Maintain the system

Maintenance to apply OS patches and other updates is performed on the underlying Linux VMs in an HDInsight cluster during off-hours. Typically, maintenance is done at 3:30 AM (based on the local time for the VM) every Monday and Thursday. Updates are performed in such a way that they don't impact more than a quarter of the cluster at a time.

Since the head nodes are redundant and not all data nodes are impacted, any jobs that are running during this time might slow down. They should still run to completion, however. Any custom software or local data that you have is preserved across these maintenance events unless a catastrophic failure occurs that requires a cluster rebuild.

Learn about IDE options for R Server on an HDInsight cluster

The Linux edge node of an HDInsight cluster is the landing zone for R-based analysis. Recent versions of HDInsight provide a default option for installing the community version of [RStudio Server](#) on the edge node as a browser-based IDE. Use of RStudio Server as an IDE for the development and execution of R scripts can be considerably more productive than just using the R console. If you chose not to add RStudio Server when creating the cluster but would like to add it later, then see [Installing R Studio Server on HDInsight clusters](#).+

Another full IDE option is to install a desktop IDE and use it to access the cluster through use of a remote Map Reduce or Spark compute context. Options include Microsoft's [R Tools for Visual Studio](#) (RTVS), RStudio, and Walware's Eclipse-based [StatET](#).

Lastly, you can access the R Server console on the edge node by typing **R** at the Linux command prompt after connecting via SSH or PuTY. When using the console interface, it is convenient to run a text editor for R script development in another window, and cut and paste sections of your script into the R console as needed.

Learn about pricing

The fees that are associated with an HDInsight cluster with R Server are structured similarly to the fees for the standard HDInsight clusters. They are based on the sizing of the underlying VMs across the name, data, and edge nodes, with the addition of a core-hour uplift. For more information about HDInsight pricing, and the availability of a 30-day free trial, see [HDInsight pricing](#).

Next steps

To learn more about how to use R Server with HDInsight clusters, see the following topics:

- [Getting started with R Server on HDInsight](#)
- [Add RStudio Server to HDInsight \(if not installed during cluster creation\)](#)

- Compute context options for R Server on HDInsight
- Azure Storage options for R Server on HDInsight

Compute context options for R Server on HDInsight

8/16/2017 • 4 min to read • [Edit Online](#)

Microsoft R Server on Azure HDInsight provides the latest capabilities for R-based analytics. It uses data that's stored in HDFS in a container in your [Azure Blob](#) storage account, a Data Lake store or the local Linux file system. Since R Server is built on open source R, the R-based applications you build can leverage any of the 8000+ open source R packages. They can also leverage the routines in [ScaleR](#), Microsoft's big data analytics package that's included with R Server.

The edge node of a cluster provides a convenient place to connect to the cluster and run your R scripts. With an edge node, you have the option of running ScaleR's parallelized distributed functions across the cores of the edge node server. You also have the option to run them across the nodes of the cluster by using ScaleR's Hadoop Map Reduce or Spark compute contexts.

Compute contexts for an edge node

In general, an R script that's run in R Server on the edge node runs within the R interpreter on that node. The exceptions are those steps that call a ScaleR function. The ScaleR calls run in a compute environment that's determined by how you set the ScaleR compute context. When you run your R script from an edge node, the possible values of the compute context are local sequential ('local'), local parallel ('localpar'), Map Reduce, and Spark.

The 'local' and 'localpar' options differ only in how rxExec calls are executed. They both execute other rx-function calls in a parallel manner across all available cores unless specified otherwise through use of the ScaleR numCoresToUse option, e.g. rxOptions(numCoresToUse=6). The following summarizes the various compute context options

COMPUTE CONTEXT	HOW TO SET	EXECUTION CONTEXT
Local sequential	rxSetComputeContext('local')	Parallelized execution across the cores of the edge node server, except for rxExec calls which are executed serially
Local parallel	rxSetComputeContext('localpar')	Parallelized execution across the cores of the edge node server
Spark	RxSpark()	Parallelized distributed execution via Spark across the nodes of the HDI cluster
Map Reduce	RxHadoopMR()	Parallelized distributed execution via Map Reduce across the nodes of the HDI cluster

Assuming that you'd like parallelized execution for the purposes of performance, then there are three options. Which option you choose depends on the nature of your analytics work, and the size and location of your data.

Guidelines for deciding on a compute context

Currently, there is no formula that tells you which compute context to use. There are, however, some guiding principles that can help you make the right choice, or at least help you narrow down your choices before you run a

benchmark. These guiding principles include:

1. The local Linux file system is faster than HDFS.
2. Repeated analyses are faster if the data is local, and if it's in XDF.
3. It's preferable to stream small amounts of data from a text data source; if the amount of data is larger, convert it to XDF prior to analysis.
4. The overhead of copying or streaming the data to the edge node for analysis becomes unmanageable for very large amounts of data.
5. Spark is faster than Map Reduce for analysis in Hadoop by running compute at in-memory speeds using Spark RDDs.
6. The Spark compute context leverages the Spark DAG for distributing work across the nodes of the cluster, and provides a number of options for persisting those tasks. Because spawning these tasks is an expensive process, we can see performance increases over Map Reduce for many types of tasks.
7. Spark runs under YARN for resource management, providing greater flexibility on selecting the number of nodes on which to run tasks.

Given these principles, some general rules of thumb for selecting a compute context are:

Local

- If the amount of data to analyze is small and does not require repeated analysis, then stream it directly into the analysis routine and use 'local' or 'localpar'.
- If the amount of data to analyze is small or medium-sized and requires repeated analysis, then copy it to the local file system, import it to XDF, and analyze it via 'local' or 'localpar'.

Hadoop Spark

- If the amount of data to analyze is large, then import it to a Spark DataFrame using RxHiveData or RxParquetData, or to XDF in HDFS (unless storage is an issue), and analyze it via 'Spark'.
- SparkR provides access to native Spark capabilities, including a growing number of predictive analytics algorithms available in Spark.

Hadoop Map Reduce

- Use only if you encounter an insurmountable problem with use of the Spark compute context since generally it will be slower.

Inline help on rxSetComputeContext

For more information and examples of ScaleR compute contexts, see the inline help in R on the rxSetComputeContext method, for example:

```
> ?rxSetComputeContext
```

You can also refer to the "[ScaleR Distributed Computing Guide](#)" that's available from the [R Server MSDN](#) library.

Next steps

In this article, you learned how to create a new HDInsight cluster that includes R Server. You also learned the basics of using the R console from an SSH session. Now you can read the following articles to discover other ways of working with R Server on HDInsight:

- [Overview of R Server for Hadoop](#)
- [Get started with R Server for Hadoop](#)
- [Add RStudio Server to HDInsight \(if not added during cluster creation\)](#)
- [Azure Storage options for R Server on HDInsight](#)

Azure Storage solutions for R Server on HDInsight

8/16/2017 • 6 min to read • [Edit Online](#)

Microsoft R Server on HDInsight has a variety of storage solutions to persist data, code, or objects that contain results from analysis. These include the following options:

- [Azure Blob](#)
- [Azure Data Lake Storage](#)
- [Azure File storage](#)

You also have the option of accessing multiple Azure storage accounts or containers with your HDI cluster. Azure File storage is a convenient data storage option for use on the edge node that enables you to mount an Azure Storage file share to, for example, the Linux file system. But Azure File shares can be mounted and used by any system that has a supported OS such as Windows or Linux.

When you create an Hadoop cluster in HDInsight, you specify either an **Azure storage** account or a **Data Lake store**. A specific storage container from that account holds the file system for the cluster that you create (for example, the Hadoop Distributed File System). For more information and guidance, see:

- [Use Azure storage with HDInsight](#)
- [Use Data Lake Store with Azure HDInsight clusters](#).

For more information on the Azure storage solutions, see [Introduction to Microsoft Azure Storage](#).

For guidance on selecting the most appropriate storage option to use for your scenario, see [Deciding when to use Azure Blobs, Azure Files, or Azure Data Disks](#)

Use Azure Blob storage accounts with R Server

If necessary, you can access multiple Azure storage accounts or containers with your HDI cluster. To do so, you need to specify the additional storage accounts in the UI when you create the cluster, and then follow these steps to use them with R Server.

WARNING

For performance purposes, the HDInsight cluster is created in the same data center as the primary storage account that you specify. Using a storage account in a different location than the HDInsight cluster is not supported.

1. Create an HDInsight cluster with a storage account name of **storage1** and a default container called **container1**.
2. Specify an additional storage account called **storage2**.
3. Copy the mycsv.csv file to the /share directory, and perform analysis on that file.

```
hadoop fs -mkdir /share  
hadoop fs -copyFromLocal mycsv.csv /share
```

4. In R code, set the name node to **default**, and set your directory and file to process.

```

myNameNode <- "default"
myPort <- 0

#Location of the data:
bigDataDirRoot <- "/share"

#Define Spark compute context:
mySparkCluster <- RxSpark(consoleOutput=TRUE)

#Set compute context:
rxSetComputeContext(mySparkCluster)

#Define the Hadoop Distributed File System (HDFS) file system:
hdfsFS <- RxHdfsFileSystem(hostName=myNameNode, port=myPort)

#Specify the input file to analyze in HDFS:
inputFile <- file.path(bigDataDirRoot,"mycsv.csv")

```

All the directory and file references point to the storage account

wasb://container1@storage1.blob.core.windows.net. This is the **default storage account** that's associated with the HDInsight cluster.

Now, suppose you want to process a file called mySpecial.csv that's located in the /private directory of **container2** in **storage2**.

In your R code, point the name node reference to the **storage2** storage account.

```

myNameNode <- "wasb://container2@storage2.blob.core.windows.net"
myPort <- 0

#Location of the data:
bigDataDirRoot <- "/private"

#Define Spark compute context:
mySparkCluster <- RxSpark(consoleOutput=TRUE, nameNode=myNameNode, port=myPort)

#Set compute context:
rxSetComputeContext(mySparkCluster)

#Define HDFS file system:
hdfsFS <- RxHdfsFileSystem(hostName=myNameNode, port=myPort)

#Specify the input file to analyze in HDFS:
inputFile <- file.path(bigDataDirRoot,"mySpecial.csv")

```

All of the directory and file references now point to the storage account

wasb://container2@storage2.blob.core.windows.net. This is the **Name Node** that you've specified.

You have to configure the /user/RevoShare/ directory on **storage2** as follows:

```

hadoop fs -mkdir wasb://container2@storage2.blob.core.windows.net/user
hadoop fs -mkdir wasb://container2@storage2.blob.core.windows.net/user/RevoShare
hadoop fs -mkdir wasb://container2@storage2.blob.core.windows.net/user/RevoShare/<RDP username>

```

Use an Azure Data Lake store with R Server

To use Data Lake stores with your HDInsight account, you need to give your cluster access to each Azure Data Lake store that you want to use. For instructions on how to use the Azure portal to create a HDInsight cluster with an Azure Data Lake Store account as the default storage or as an additional store, see [Create an HDInsight cluster](#)

with Data Lake Store using Azure portal.

You then use the store in your R script much like you did a secondary Azure storage account as described in the previous procedure.

Add cluster access to your Azure Data Lake stores

You access a Data Lake store by using an Azure Active Directory (Azure AD) Service Principal that's associated with your HDInsight cluster.

To add an Azure AD Service Principal:

1. When you create your HDInsight cluster, select **Cluster AAD Identity** from the **Data Source** tab.
2. In the **Cluster AAD Identity** dialog box, under **Select AD Service Principal**, select **Create new**.

After you give the Service Principal a name and create a password for it, click **Manage ADLS Access** to associate the Service Principal with your Data Lake stores.

It's also possible to add cluster access to one or more Data Lake stores following cluster creation. Open the Azure portal entry for a Data Lake store and go to **Data Explorer > Access > Add**.

How to access the Data Lake store from R Server

Once you've given access to a Data Lake store, you can use the store in R Server on HDInsight the way you would a secondary Azure storage account. The only difference is that the prefix **wasb://** changes to **adl://** as follows:

```
# Point to the ADL store (e.g. ADLtest)
myNameNode <- "adl://rkadl1.azuredatalakestore.net"
myPort <- 0

# Location of the data (assumes a /share directory on the ADL account)
bigDataDirRoot <- "/share"

# Define Spark compute context
mySparkCluster <- RxSpark(consoleOutput=TRUE, nameNode=myNameNode, port=myPort)

# Set compute context
rxSetComputeContext(mySparkCluster)

# Define HDFS file system
hdfsFS <- RxHdfsFileSystem(hostName=myNameNode, port=myPort)

# Specify the input file in HDFS to analyze
inputFile <- file.path(bigDataDirRoot, "AirlineDemoSmall.csv")

# Create factors for days of the week
colInfo <- list(DayOfWeek = list(type = "factor",
    levels = c("Monday", "Tuesday", "Wednesday", "Thursday",
              "Friday", "Saturday", "Sunday")))

# Define the data source
airDS <- RxTextData(file = inputFile, missingValueString = "M",
                     colInfo = colInfo, fileSystem = hdfsFS)

# Run a linear regression
model <- rxLinMod(ArrDelay~CRSDepTime+DayOfWeek, data = airDS)
```

The following commands are used to configure the Data Lake storage account with the RevoShare directory and add the sample .csv file from the previous example:

```
hadoop fs -mkdir adl://rkadl1.azuredatalakestore.net/user  
hadoop fs -mkdir adl://rkadl1.azuredatalakestore.net/user/RevoShare  
hadoop fs -mkdir adl://rkadl1.azuredatalakestore.net/user/RevoShare/<user>  
  
hadoop fs -mkdir adl://rkadl1.azuredatalakestore.net/share  
  
hadoop fs -copyFromLocal /usr/lib64/R Server-7.4.1/library/RevoScaleR/SampleData/AirlineDemoSmall.csv  
adl://rkadl1.azuredatalakestore.net/share  
  
hadoop fs -ls adl://rkadl1.azuredatalakestore.net/share
```

Use Azure File storage with R Server

There is also a convenient data storage option for use on the edge node called [Azure Files](#). It enables you to mount an Azure Storage file share to the Linux file system. This option can be handy for storing data files, R scripts, and result objects that might be needed later, especially when it makes sense to use the native file system on the edge node rather than HDFS.

A major benefit of Azure Files is that the file shares can be mounted and used by any system that has a supported OS such as Windows or Linux. For example, it can be used by another HDInsight cluster that you or someone on your team has, by an Azure VM, or even by an on-premises system. For more information, see:

- [How to use Azure File storage with Linux](#)
- [How to use Azure File storage on Windows](#)

Next steps

Now that you understand the Azure storage options, use the following links to discover ways of getting data science tasks done with R Server on HDInsight.

- [Overview of R Server on HDInsight](#)
- [Get started with R server on Hadoop](#)
- [Add RStudio Server to HDInsight \(if not added during cluster creation\)](#)
- [Compute context options for R Server on HDInsight](#)

Submit Jobs from R Tools for Visual Studio

8/16/2017 • 4 min to read • [Edit Online](#)

Overview

R Tools for Visual Studio (RTVS) is a free, open-source extension for the Community (free), Professional, and Enterprise editions of both [Visual Studio 2017](#), as well as [Visual Studio 2015 Update 3](#) or higher. Most of the critical features of RStudio are included in RTVS, with more being added as the tool matures. If there is a missing feature that you'd like included in RTVS, consider filling out the [RTVS survey](#).

RTVS enhances your R workflow by offering tools such as the [R Interactive window](#) (REPL), intellisense (code completion), [plot visualization](#) through R libraries such as ggplot2 and ggviz, [R code debugging](#), and more.

Setting up your environment

To get started, [install R Tools for Visual Studio](#).

The screenshot shows the Visual Studio workload selection dialog. On the left, two workloads are listed: "Node.js development" and "Data science and analytical applications". The "Data science and analytical applications" workload is highlighted with a red border. On the right, a "Summary" section lists optional components under "Data science and analytical applications". Components include F# language support, Python language support, R language support, Runtime support for R development, Anaconda3 64-bit (4.3.0.1), Microsoft R Client (3.3.2), Cookiecutter template support, Python web support, and Python native development tools. The "R language support", "Runtime support for R development", "Anaconda3 64-bit (4.3.0.1)", and "Microsoft R Client (3.3.2)" checkboxes are also highlighted with a red border.

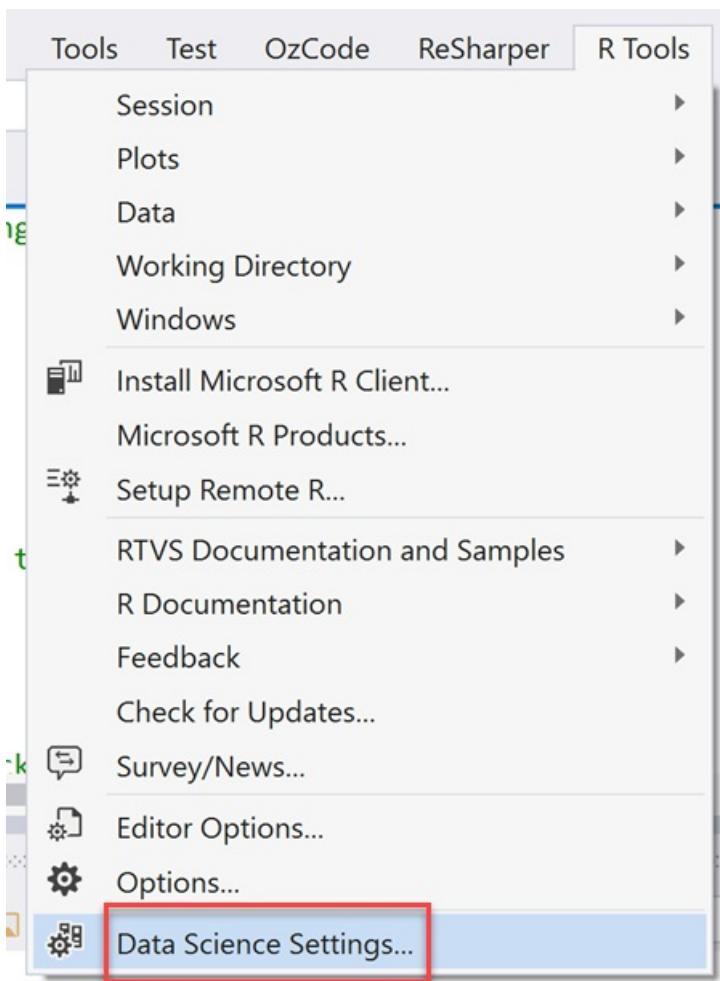
When installing the tools for Visual Studio 2017, make sure you select the **R language support**, **Runtime support for R development**, and **Microsoft R Client** options after selecting the *Data science and analytical applications* workload.

You will need to have public and private keys set up to use SSH with HDInsight for authentication.

To run the `RevoScaler` and `RxSpark` functions, you will need to [install R Server](#) on your machine.

To create a compute context that runs `RevoScaler` functions from your local client to your HDInsight cluster, you will need to install [PuTTY](#).

If you desire, apply the Data Science Settings to your Visual Studio environment, which provides a new layout for your workspace that makes it easier to work with the R tools. You do this by going to the new **R Tools** menu item, then selecting **Data Science Settings...**



Your workspace will now have a new layout optimized for working with R in Visual Studio. Refer to the screenshot in the next section to see the layout.

To revert to other Visual Studio settings later on, first use the **Tools > Import and Export Settings** command, selecting **Export selected environment settings**, and specifying a file name. To restore those settings, use the same command and select **Import selected environment settings**. You can also use the same commands if you change the data scientist layout and want to return to it later on, rather than using the **Data Science Settings** command directly.

Execute local R methods

Once you have [provisioned your R Server HDInsight cluster](#) and installed the [RTVS extension](#), download the [samples zip file](#) to get started.

1. Open `examples/Examples.sln`, to launch the solution in Visual Studio.
2. Open the `1-Getting Started with R.R` file underneath the `A first look at R` solution folder.
3. Starting at the top of the file, press **Ctrl+Enter** to send each line, one at a time, to the R Interactive window which will display the results. Some lines might take a while as they install packages. Alternatively, you can select all lines in the R file (**Ctrl+A**), then execute all (**Ctrl+Enter**), or click the Execute Interactive icon on the toolbar: 

After running all of the lines in the script, you should have an output similar to this:

The screenshot shows the Microsoft Visual Studio interface with the R Interactive window open. The code in the editor is:

```
# inspect predictions
head(predicted_values)

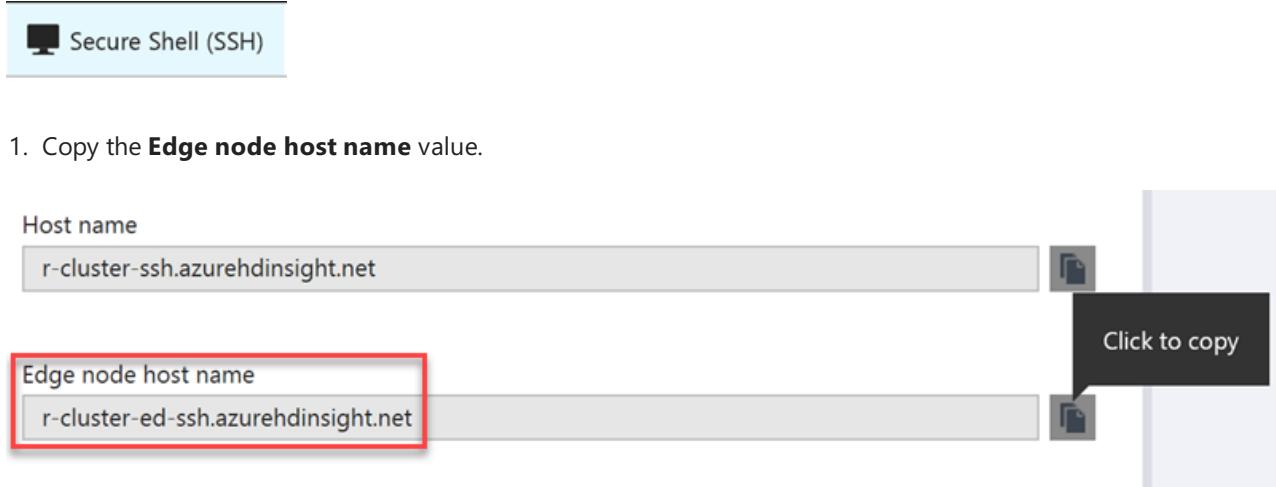
# Create plot of actuals vs predictions
ggplot(predicted_values, aes(x = actual, y = predicted)) +
  geom_point(colour = "blue", alpha = 0.01) +
  geom_smooth(colour = "red") +
  coord_equal(ylim = c(0, 20000)) + # force equal scale
  ggtitle("Linear model of diamonds data")
```

The R Interactive window displays the results of the code execution, including a scatter plot titled "Linear model of diamonds data". The plot shows a strong positive linear relationship between the actual and predicted values of diamonds.

Submitting jobs to the HDInsight R cluster

Since you are running Microsoft R Server/Microsoft R Client from a Windows computer equipped with PuTTY, you can create a compute context that will run distributed `RevoScaleR` functions from your local client to your HDInsight cluster. To do this, we need to use `RxSpark` to create the compute context, specifying our username, the Hadoop cluster's edge node, ssh switches, etc.

1. To find your edge node's host name, open your HDInsight R cluster blade on Azure, then select **Secure Shell (SSH)** on the top menu of the Overview pane.



1. Paste the following code into the R Interactive window in Visual Studio, altering the values of the setup variables to match your environment:

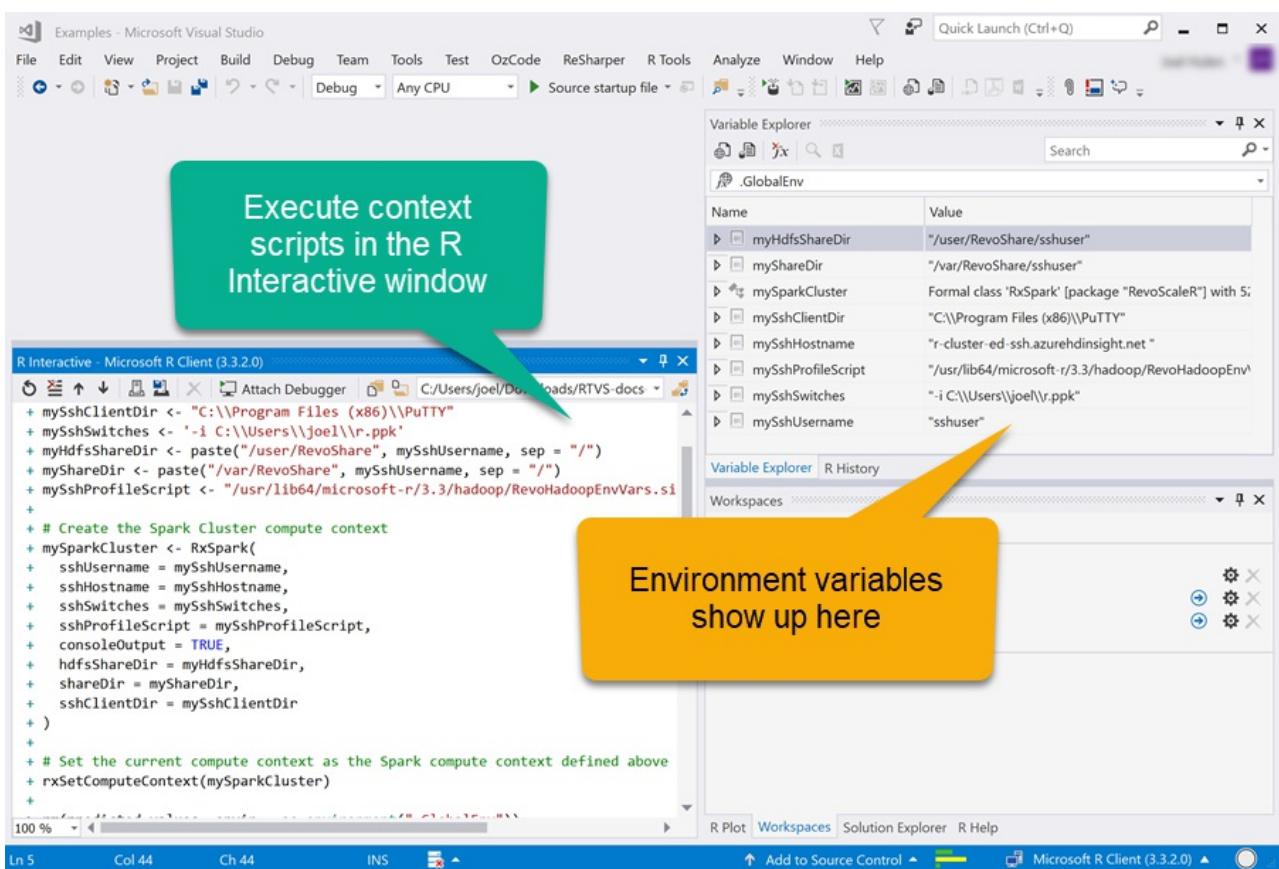
```

# Setup variables that connect the compute context to your HDInsight cluster
mySshHostname <- 'r-cluster-ed-ssh.azurehdinsight.net' # HDI secure shell hostname
mySshUsername <- 'sshuser' # HDI SSH username
mySshClientDir <- "C:\\Program Files (x86)\\PuTTY"
mySshSwitches <- '-i C:\\Users\\azureuser\\r.ppk' # Path to your private ssh key
myHdfsShareDir <- paste("/user/RevoShare", mySshUsername, sep = "/")
myShareDir <- paste("/var/RevoShare", mySshUsername, sep = "/")
mySshProfileScript <- "/usr/lib64/microsoft-r/3.3/hadoop/RevoHadoopEnvVars.site"

# Create the Spark Cluster compute context
mySparkCluster <- RxSpark(
  sshUsername = mySshUsername,
  sshHostname = mySshHostname,
  sshSwitches = mySshSwitches,
  sshProfileScript = mySshProfileScript,
  consoleOutput = TRUE,
  hdfsShareDir = myHdfsShareDir,
  shareDir = myShareDir,
  sshClientDir = mySshClientDir
)

# Set the current compute context as the Spark compute context defined above
rxSetComputeContext(mySparkCluster)

```



- Now that the compute context has been set to your cluster, execute the following commands in the R Interactive window:

```

rxHadoopCommand("version") # should return version information
rxHadoopMakeDir("/user/RevoShare/newUser") # creates a new folder in your storage account
rxHadoopCopy("/example/data/people.json", "/user/RevoShare/newUser") # copies file to new folder

```

The above commands should produce an output similar to the following:

R Interactive - Microsoft R Client (3.3.2.0)

Attach Debugger | C:/Users/joel/Downloads/RTVS-docs

```
> rxHadoopCommand("version") # should return version information
+ rxHadoopMakeDir("/user/RevoShare/newUser")
+ rxHadoopCopy("/example/data/people.json", "/user/RevoShare/newUser")
Hadoop 2.7.3.2.5.4.2-7
Subversion git@github.com:hortonworks/hadoop.git -r e1f139118ccc75d6610c176bf7e
Compiled by jenkins on 2017-04-04T10:53Z
Compiled with protoc 2.5.0
From source with checksum 10f7279f54afc13eb8389214075eb51
This command was run using /usr/hdp/2.5.4.2-7/hadoop/hadoop-common-2.7.3.2.5.4.
[1] TRUE
[1] TRUE
[1] TRUE
>
```

After running the `rxHadoopCopy` command that copies the `people.json` file from the example data folder to our newly created `/user/RevoShare/newUser` folder, let's browse to that location to make sure the file copy command was successful.

1. From your HDInsight R cluster blade in Azure, select **Storage accounts** from the left-hand menu.

PROPERTIES

Properties

Storage accounts

Data Lake Store access

1. Select the default storage account for your cluster, making note of the container/directory name.
2. Select **Containers** from the left-hand menu on your storage account blade.

BLOB SERVICE

Containers

CORS

Custom domain

1. Select your cluster's container name, browse to the `user` folder (you might have to click *Load more* at the bottom of the list), then select `RevoShare`, then `newUser`. The `people.json` file should be displayed in the `newUser` folder.



Upload Refresh

Location: r-cluster2 / user / RevoShare / newUser

Search blobs by prefix (case-sensitive)

NAME

[..]

people.json

1. Make sure you stop your Spark context. You cannot run multiple contexts at once:

```
rxStopEngine(mySparkCluster)
```

Next steps

In this article, we've walked through the steps to use R Tools for Visual Studio, and how to create a compute context that allows you to execute commands on your HDInsight cluster.

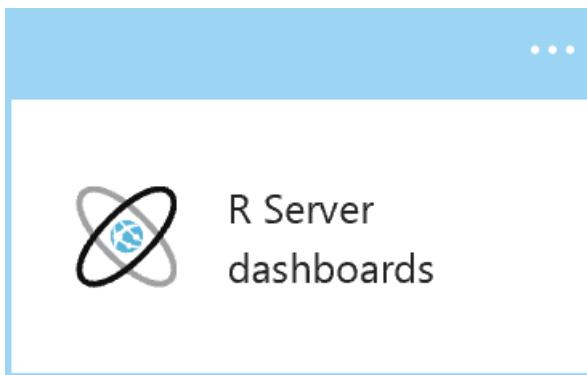
- Learn more about [compute context options for R Server on HDInsight](#).
- Walk through an example of [combining ScaleR and SparkR](#) for airline flight delay predictions.
- Read about an alternative way of submitting R jobs, using [R Studio Server](#)

Submit Jobs from R Studio Server

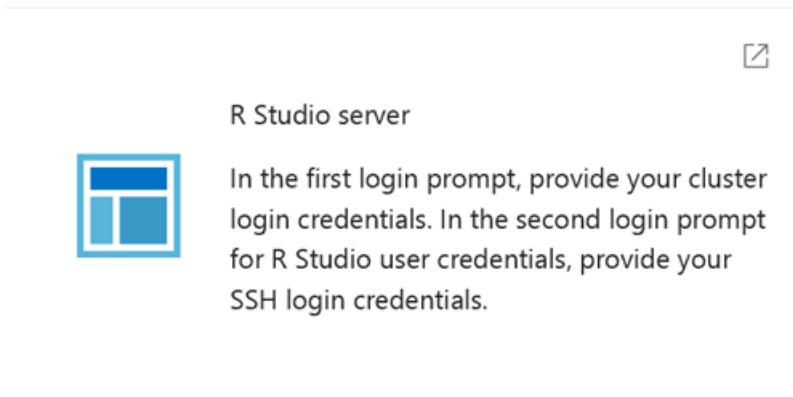
8/16/2017 • 5 min to read • [Edit Online](#)

R Studio Server is a popular, web-based integrated development environment (IDE) for developing and executing R scripts with R Server on an HDInsight cluster. It includes several features to speed up and ease development, such as a syntax-highlighting editor that supports direct code execution, a console for interactively running R commands, a workspace to view objects in the global environment, and much more.

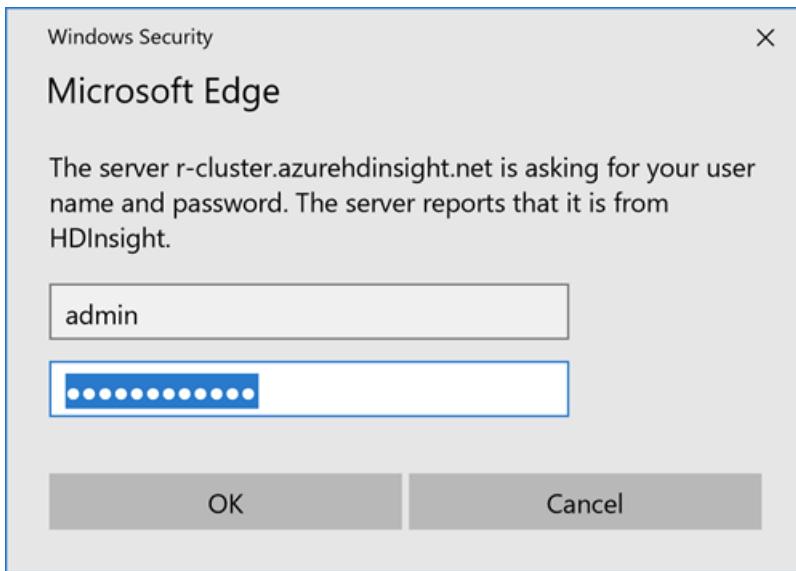
To open R Studio Server from your HDInsight cluster in Azure, navigate to the Overview pane, and click on **R Server dashboards**.



From the cluster dashboard, click on **R Studio server**.



You will be prompted twice to enter a password. The first prompt is displayed as a popup by your browser, in which you need to enter your cluster login credentials.



The second login prompt will ask for your R Studio user credentials. Supply the SSH login credentials here.



After successfully logging in, you should see the R Studio web interface.

A screenshot of the R Studio web interface. The left pane shows the R console output:

```
R version 3.3.2 (2016-10-31) -- "Sincere Pumpkin Patch"
Copyright (C) 2016 The R Foundation for Statistical Computing
Platform: x86_64-pc-linux-gnu (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

Microsoft R Open 3.3.2
The enhanced R distribution from Microsoft
Microsoft packages Copyright (C) 2017 Microsoft

Loading Microsoft R Server packages, version 9.0.1.
Type 'readme()' for release notes, 'privacy()' for privacy policy, or
'Revicense()' for licensing information.

Using the Intel MKL for parallel mathematical computing(using 4 cores).
Default CRAN mirror snapshot taken on 2016-11-01.
See: https://mran.microsoft.com/.
```

The right pane shows the "Environment" tab of the RStudio interface, which is currently empty.

Install R Studio Server

If the **R Studio Server** option is not available on the cluster dashboard, as outlined above, you must run the R Studio Server install script, following [these instructions](#).

Cluster configuration

* Cluster type ⓘ R Server	* Operating system Linux	* Version R Server 9.0 (HDI 3.5)
* Cluster tier ⓘ STANDARD PREMIUM		

R Server : Terabyte-scale, enterprise grade R analytics with transparent parallelization on top of Spark and Hadoop.

Configuration Options:

- R Server 9.0 on Spark 2.0 with Java 8
- R Server 8.0 on Spark 1.6 with Java 7

Adds 0.08 USD per Core-Hour.

Features

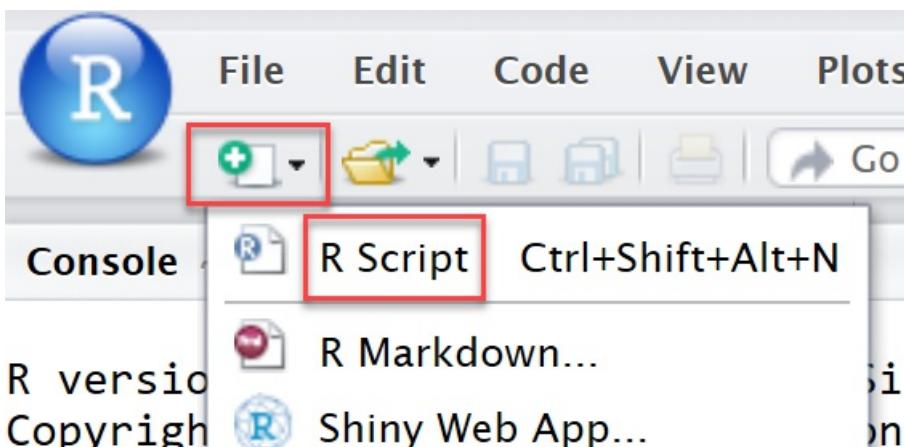
* denotes preview feature

Available	Not available
<input checked="" type="checkbox"/> R Studio community edition for R Server	+ Apache Ranger* (PREMIUM) ⓘ
+ Secure shell (SSH) access	+ Domain joining* (PREMIUM) ⓘ

Please note, R Studio community edition for R Server is automatically installed when you check the option to install it when provisioning your R cluster as shown in the screenshot above.

Submit your first job from R Studio Server

1. From the R Studio Server web interface, click on the new icon, then R Script.



1. Enter the following code to install the *ggplot2* package, plot random data on the chart, build a linear regression model, and compare actuals to predictions. Refer to the detailed comments to understand each step.

```
# To install a new package, use install.packages()
# Install the ggplot2 package for its plotting capability.
if (!require("ggplot2")){
  install.packages("ggplot2")
}
```

```

# Then load the package.
library("ggplot2")
search()
# Notice that package:ggplot2 is now added to the search list.

#### A Simple Regression Example

# Look at the data sets that come with the package.
data(package = "ggplot2")$results
# Note that the results in RTVS may pop up, or pop under, in a new window.

# ggplot2 contains a dataset called diamonds. Make this dataset available using the data() function.
data(diamonds, package = "ggplot2")

# Create a listing of all objects in the "global environment". Look for "diamonds" in the results.
ls()

# Now investigate the structure of diamonds, a data frame with 53,940 observations
str(diamonds)

# Print the first few rows.
head(diamonds)

# Print the last 6 lines.
tail(diamonds)

# Find out what kind of object it is.
class(diamonds)

# Look at the dimension of the data frame.
dim(diamonds)

#### Plots in R

# Create a random sample of the diamonds data.
diamondSample <- diamonds[sample(nrow(diamonds), 5000),]
dim(diamondSample)

# R has three systems for static graphics: base graphics, lattice and ggplot2.
# This example uses ggplot2

# Set the font size so that it will be clearly legible.
theme_set(theme_gray(base_size = 18))

# In this sample you use ggplot2.
ggplot(diamondSample, aes(x = carat, y = price)) +
  geom_point(colour = "blue")

# Add a log scale.
ggplot(diamondSample, aes(x = carat, y = price)) +
  geom_point(colour = "blue") +
  scale_x_log10()

# Add a log scale for both scales.
ggplot(diamondSample, aes(x = carat, y = price)) +
  geom_point(colour = "blue") +
  scale_x_log10() +
  scale_y_log10()

#### Linear Regression in R

# Now, build a simple regression model, examine the results of the model and plot the points and the
# regression line.

# Build the model. log of price explained by log of carat. This illustrates how linear regression works.
# Later we fit a model that includes the remaining variables

```

```

model <- lm(log(price) ~ log(carat) , data = diamondSample)

# Look at the results.
summary(model)
# R-squared = 0.9334, i.e. model explains 93.3% of variance

# Extract model coefficients.
coef(model)
coef(model)[1]
exp(coef(model)[1]) # exponentiate the log of price, to convert to original units

# Show the model in a plot.
ggplot(diamondSample, aes(x = carat, y = price)) +
geom_point(colour = "blue") +
geom_smooth(method = "lm", colour = "red", size = 2) +
scale_x_log10() +
scale_y_log10()

### Regression Diagnostics

# It is easy to get regression diagnostic plots. The same plot function that plots points either with a formula or with the coordinates also has a "method" for dealing with a model object.

# Look at some model diagnostics.
# check to see Q-Q plot to see linearity which means residuals are normally distributed

par(mfrow = c(2, 2)) # Set up for multiple plots on the same figure.
plot(model, col = "blue")
par(mfrow = c(1, 1)) # Rest plot layout to single plot on a 1x1 grid

### The Model Object

# Finally, let's look at the model object. R packs everything that goes with the model, e.g. the formula and results into the object. You can pick out what you need by indexing into the model object.
str(model)
model$coefficients # note this is the same as coef(model)

# Now fit a new model including more columns
model <- lm(log(price) ~ log(carat) + ., data = diamondSample) # Model log of price against all columns

summary(model)
# R-squared = 0.9824, i.e. model explains 98.2% of variance, i.e. a better model than previously

# Create data frame of actual and predicted price

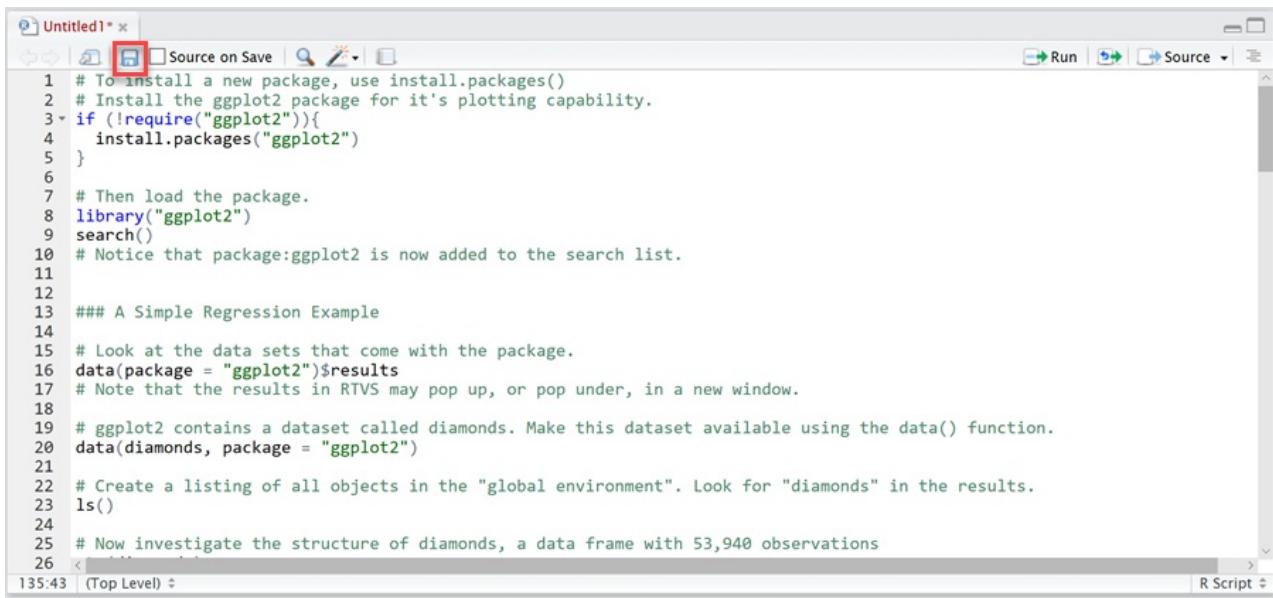
predicted_values <- data.frame(
actual = diamonds$price,
predicted = exp(predict(model, diamonds)) # anti-log of predictions
)

# Inspect predictions
head(predicted_values)

# Create plot of actuals vs predictions
ggplot(predicted_values, aes(x = actual, y = predicted)) +
geom_point(colour = "blue", alpha = 0.01) +
geom_smooth(colour = "red") +
coord_equal(ylim = c(0, 20000)) + # force equal scale
ggtitle("Linear model of diamonds data")

```

1. Click the Save button on top of the script window. When prompted, enter a name for your script, as well as location.



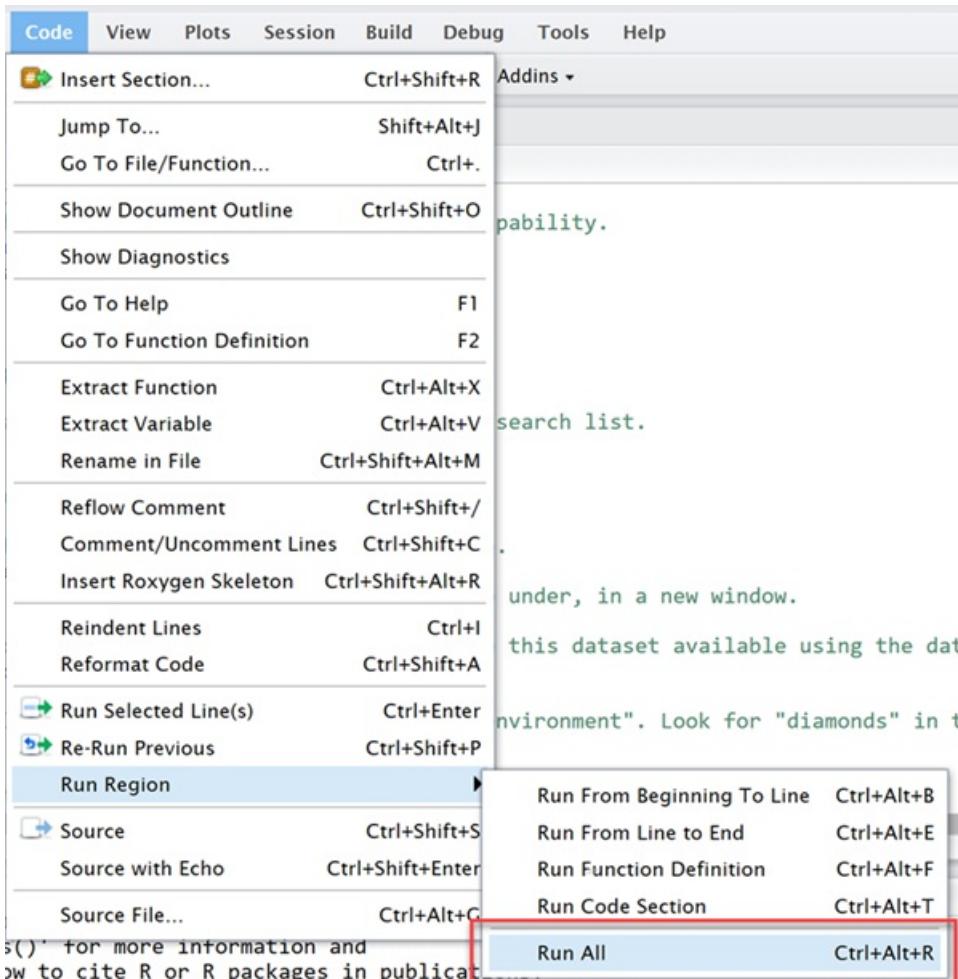
```

1 # To install a new package, use install.packages()
2 # Install the ggplot2 package for it's plotting capability.
3 if (!require("ggplot2")){
4   install.packages("ggplot2")
5 }
6
7 # Then load the package.
8 library("ggplot2")
9 search()
10 # Notice that package:ggplot2 is now added to the search list.
11
12
13 ### A Simple Regression Example
14
15 # Look at the data sets that come with the package.
16 data(package = "ggplot2")$results
17 # Note that the results in RTVS may pop up, or pop under, in a new window.
18
19 # ggplot2 contains a dataset called diamonds. Make this dataset available using the data() function.
20 data(diamonds, package = "ggplot2")
21
22 # Create a listing of all objects in the "global environment". Look for "diamonds" in the results.
23 ls()
24
25 # Now investigate the structure of diamonds, a data frame with 53,940 observations
26

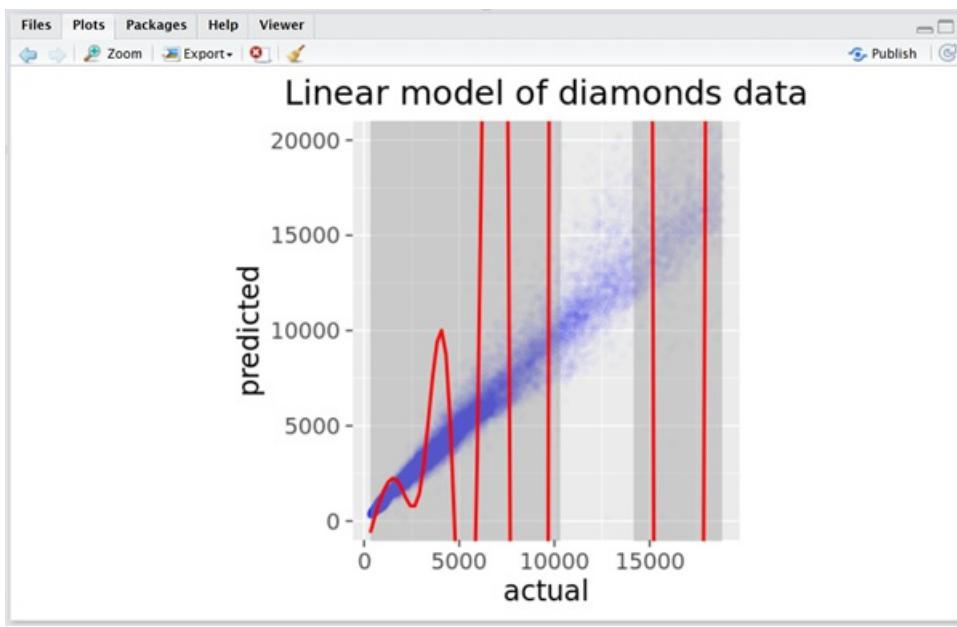
```

135:43 (Top Level) R Script

1. Execute entire script by entering **ctrl+Alt+R** (**Command+Alt+R** on Mac), through the menu by going to **Code**, **Run Region**, **Run All**. As the job executes, you will see information output within the Console beneath your script.



1. When execution is complete, the output of our graph will be displayed within the Plots tab. You may also view the environment data of the job within the Environment tab. Click on the table icon next to one of the data elements within the environment tab to view the data.



Environment History

Import Dataset...

Global Environment

Data

- diamonds 53940 obs. of 10 variables
- diamondSample 5000 obs. of 10 variables
- predicted_values 53940 obs. of 2 variables

Values

- model Large lm (13 elements, 2.3 Mb)

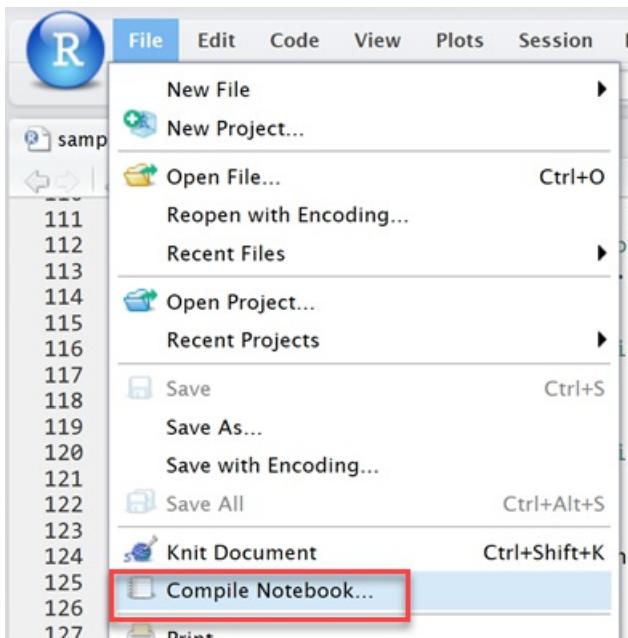

```
coefficients : Named num [1:25] 3.012 1.064 -0.469 0.12 -0.025 ...
      ... attr(*, "names")= chr [1:25] "(Intercept)" "log(carat)" "carat" "cut.L" ...
      residuals : Named num [1:5000] -0.1183 -0.2743 -0.0789 -0.1318 0.0683 ...
      ... attr(*, "names")= chr [1:5000] "876" "53781" "46852" "15215" ...
      effects : Named num [1:5000] -549.476 -68.756 -0.557 4.251 0.565 ...
```

sample.R x diamonds x Filter

	carat	cut	color	clarity	depth	table	price	x	y	z
1	0.23	Ideal	E	SI2	61.5	55.0	326	3.95	3.98	2.43
2	0.21	Premium	E	SI1	59.8	61.0	326	3.89	3.84	2.31
3	0.23	Good	E	VS1	56.9	65.0	327	4.05	4.07	2.31
4	0.29	Premium	I	VS2	62.4	58.0	334	4.20	4.23	2.63
5	0.31	Good	J	SI2	63.3	58.0	335	4.34	4.35	2.75
6	0.24	Very Good	J	VVS2	62.8	57.0	336	3.94	3.96	2.48
7	0.24	Very Good	I	VVS1	62.3	57.0	336	3.95	3.98	2.47
8	0.26	Very Good	H	SI1	61.9	55.0	337	4.07	4.11	2.53
9	0.22	Fair	E	VS2	65.1	61.0	337	3.87	3.78	2.49
10	0.23	Very Good	H	VS1	59.4	61.0	338	4.00	4.05	2.39
11	0.30	Good	J	SI1	64.0	55.0	339	4.25	4.28	2.73
12	0.23	Ideal	J	VS1	62.8	56.0	340	3.93	3.90	2.46
13	0.22	Premium	F	SI1	60.4	61.0	342	3.88	3.84	2.33
14	0.31	Ideal	J	SI2	62.2	54.0	344	4.35	4.37	2.71
15	0.20	Premium	E	SI2	60.2	62.0	345	3.79	3.75	2.27
16	0.32	Premium	F	II	60.9	58.0	345	4.38	4.42	2.68

Showing 1 to 16 of 53,940 entries

- View a nicely formatted report that includes all of the R commands, in-line with the output of each segment, by going to **File, Compile Notebook...** in the menu. There are a few options in the dropdown menu within the Compile Notebook from R Script dialog window, including HTML. Below is a sample portion of the compiled notebook generated from the script executed in this exercise.



```

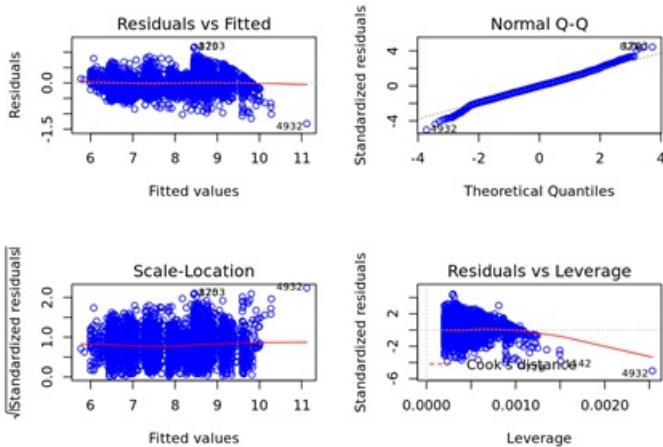
### Regression Diagnostics

# It is easy to get regression diagnostic plots. The same plot function that plots points either with a formula or with the
coordinates also has a "method" for dealing with a model object.

# Look at some model diagnostics.
# check to see Q-Q plot to see Linearity which means residuals are normally distributed

par(mfrow = c(2, 2)) # Set up for multiple plots on the same figure.
plot(model, col = "blue")

```



```

par(mfrow = c(1, 1)) # Rest plot layout to single plot on a 1x1 grid

### The Model Object

# Finally, let's look at the model object. R packs everything that goes with the model, e.g. the formula and results into th

```

Next steps

In this article, we've shown how easy it is to execute R scripts through the web-based R Studio Server.

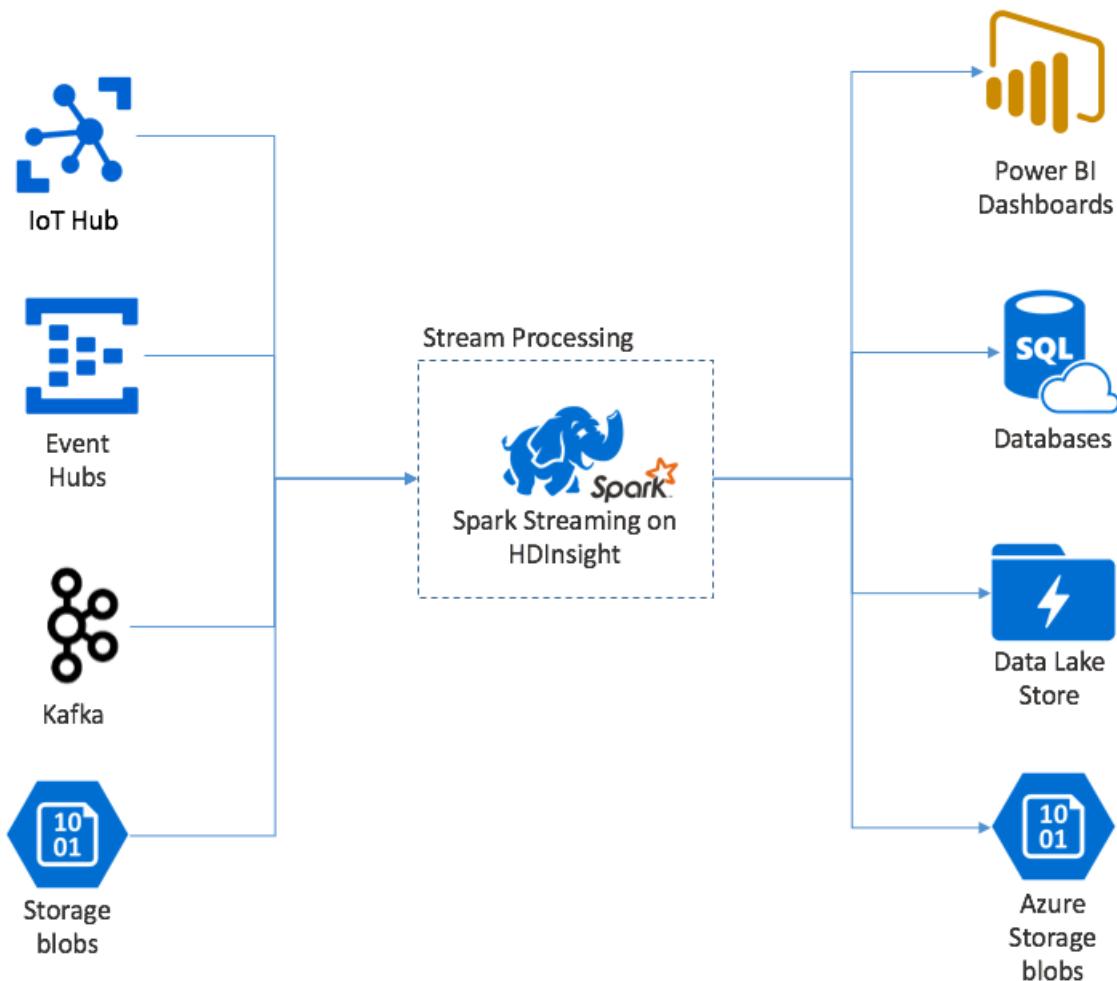
- Learn how to [install R Studio Server](#) if you didn't select the option to install it when provisioning your cluster.
- Walk through an example of [combining ScaleR and SparkR](#) for airline flight delay predictions.
- Learn about using [R Tools for Visual Studio](#) as an alternative to submitting jobs, as well as the nice features it offers, such as debugging and intellisense (code completion).

What is Spark Streaming?

8/16/2017 • 9 min to read • [Edit Online](#)

Spark Streaming enables you to implement scalable, high-throughput, fault-tolerant applications for the processing of data streams. You can run your Spark Streaming applications on HDInsight Spark clusters, and connect it to process data from a variety of sources such as Azure Event Hubs, Azure IoT Hub, Kafka, Flume, Twitter, ZeroMQ, raw TCP sockets or even by monitoring the HDFS filesystem for changes.

Spark Streaming creates a long running job during which you are able to apply transformations to the data (such as map, reduce, join and extract data by windows of time) and then push out the results to filesystems, databases, dashboards and the console.



Spark Streaming takes a micro-batch approach to how it processes data. This means that it must first wait to collect a time-defined batch of events (usually configured in the range of less than a second to a few seconds), before it sends the batch of events on for processing. This is in contrast to approaches that would receive a single event and process that single event immediately. The benefit of the micro-batch approach, as you will see, is it lets more efficiently process data that is rapidly ingested into your solution and gives you an abstraction that makes applying aggregate calculations on the events a little easier.

The design goals of Spark Streaming include low latency (measured in seconds) and linear scalability. However, what sets Spark Streaming apart are its support for fault tolerance with the guarantee that any given event would be processed exactly once, even in the face of a node failure. Additionally, Spark Streaming is integrated with the Spark core API, giving Spark developers a familiar programming model and new developers one less new framework to learn when first starting with Spark.

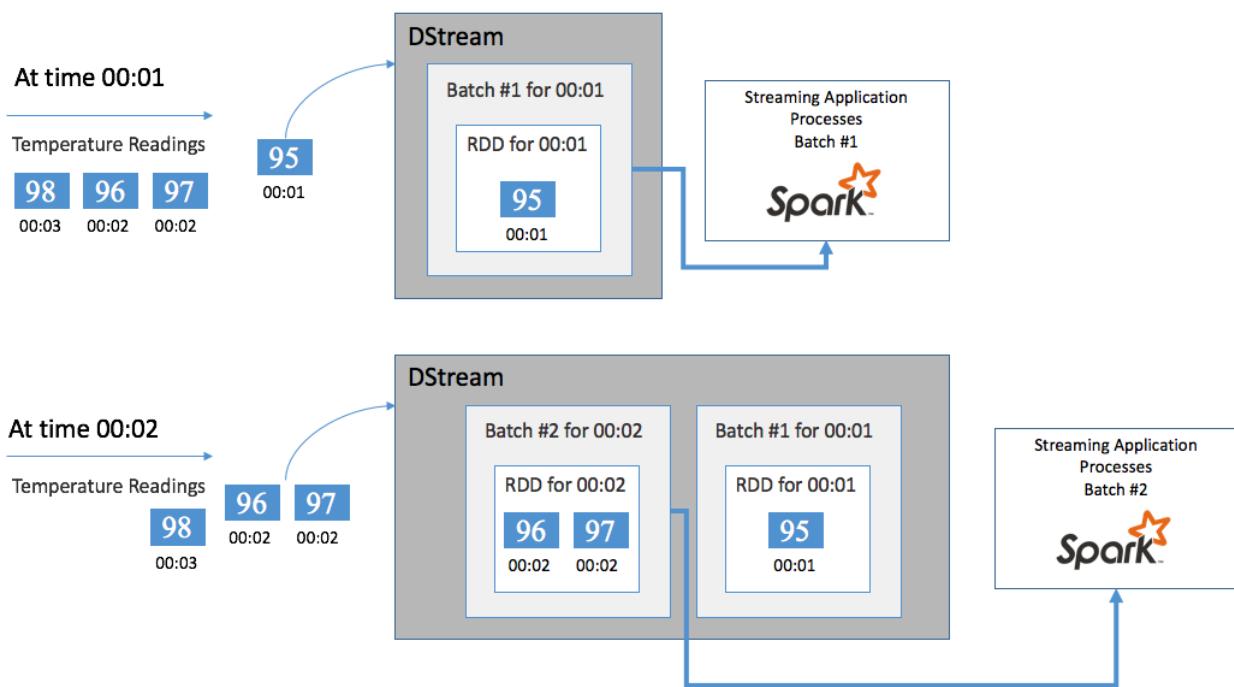
Introducing the DStream

Spark Streaming represents a continuous stream of data using a discretized stream or DStream. This DStream can be created from input sources like Event Hubs or Kafka, or by applying transformation on another DStream.

The DStream represents a few layers of abstraction on top of the raw event data. To understand how they work, it helps to build up a DStream from a single event.

Start with a single event, say a temperature reading from a connected thermostat. When this event arrives at your Spark Streaming application, the first thing that happens is the event is stored in a reliable way- it is replicated so that multiple nodes have a copy of your event. This ensures that the failure of any single node will not result in the loss of your event. Spark core has a data structure that distributes data across multiple nodes in the cluster, where each node generally maintains its data completely in-memory for best performance. This data structure is called a resilient distributed dataset or RDD. The temperature reading event will be stored in an RDD.

Each RDD represents events collected over some user defined timeframe called the batch interval. Everytime the batch interval elapses a new RDD is produced that contains all the data in the interval of time that just completed. It is this continuous set of RDD's that are collected into a DStream. So for example, if the batch interval was configured to be 1 second long, your DStream emits a batch every second containing one RDD that contains all the data ingested during that second. When processing the DStream, the temperature event would appear in one of these batches. A Spark Streaming application that processes these events, processes the batches that contains the events and ultimately acts on the data stored in the RDD each batch contains.



Structure of a Spark Streaming Application

A Spark Streaming application is a long running application that receives data from ingest sources, applies transformation to process the data and then pushes the data out to one or more destinations. The structure of a Spark Streaming application has two main parts. First, you define the processing logic that includes where the data comes from, what processing to do on the data and where the results should go. Second, you run the defined application indefinitely, waiting for any signals to stop the long running application.

To give you a sense for the structure of a Spark Streaming Application, we will show a simple application that receives a line of text over a TCP socket and counts the number of times each word appears.

Define the application

The application definition consists of four main steps: creating a StreamingContext, creating a DStream from the

StreamingContext, applying transformations to the DStream and outputting the results. During this phase you are just describing the application logic, no actual transformations are applied or is output emitted until you run the application (as shown in the second phase).

Create a StreamingContext

Create a StreamingContext from the SparkContext that points to your cluster. When creating a StreamingContext you specify the size of the batch in seconds. In this example, we create the StreamingContext so it has a batch size of one second.

```
val ssc = new StreamingContext(spark, Seconds(1))
```

Create a DStream

Using the StreamingContext instance you created, create an input DStream for your input source. In this case, we are opening watching for the appearance of new files in default storage attached to the HDInsight cluster.

```
val lines = ssc.textFileStream("/uploads/2017/01/")
```

Apply transformations

You implement the processing by applying transformations on the DStream. Our application will receive one line of text at a time from the file, split each line into words, and then follows the map reduce pattern to count the number of times each word appears.

```
val words = lines.flatMap(_.split(" "))
val pairs = words.map(word => (word, 1))
val wordCounts = pairs.reduceByKey(_ + _)
```

Output results

Push the transformation results out to the destination systems by applying output operations. In this case, we show the result of each run thru the computation in the console output.

```
wordCounts.print()
```

Run the application

Start the streaming application and run until a termination signal is received.

```
ssc.start()
ssc.awaitTermination()
```

For details on the Spark Stream API, along with the event sources, transformations and output operations it supports see [Spark Streaming Programming Guide](#).

Here is another sample application that is completely self-contained that you can run inside a [Jupyter Notebook](#). In the example below, we create a mock data source in the class DummySource that outputs the value of a counter and the current time in milliseconds every 5 seconds. We create a new StreamingContext object that has a batch interval of 30 seconds. Every time a batch is created, it examines the RDD produced, converts it to a Spark DataFrame and creates a temporary table over the DataFrame.

```

class DummySource extends org.apache.spark.streaming.receiver.Receiver[Int, Long]
(org.apache.spark.storage.StorageLevel.MEMORY_AND_DISK_2) {

    /** Start the thread that simulates receiving data */
    def onStart() {
        new Thread("Dummy Source") { override def run() { receive() } }.start()
    }

    def onStop() { }

    /** Periodically generate a random number from 0 to 9, and the timestamp */
    private def receive() {
        var counter = 0
        while(!isStopped()) {
            store(Iterator((counter, System.currentTimeMillis())))
            counter += 1
            Thread.sleep(5000)
        }
    }
}

// A batch is created every 30 seconds
val ssc = new org.apache.spark.streaming.StreamingContext(spark.sparkContext,
org.apache.spark.streaming.Seconds(30))

// Set the active SQLContext so that we can access it statically within the foreachRDD
org.apache.spark.sql.SQLContext.setActive(spark.sqlContext)

// Create the stream
val stream = ssc.receiverStream(new DummySource())

// Process RDDs in the batch
stream.foreachRDD { rdd =>

    // Access the SQLContext and create a table called demo_numbers we can query
    val _sqlContext = org.apache.spark.sql.SQLContext.getOrCreate(rdd.sparkContext)
    _sqlContext.createDataFrame(rdd).toDF("value", "time")
        .registerTempTable("demo_numbers")
}

// Start the stream processing
ssc.start()

```

We can then query the DataFrame periodically to see the current set of values present in the batch. In this case, we use the following SQL query.

```

%%sql
SELECT * FROM demo_numbers

```

The resulting output looks similar to the following:

VALUE	TIME
10	1497314465256
11	1497314470272
12	1497314475289
13	1497314480310

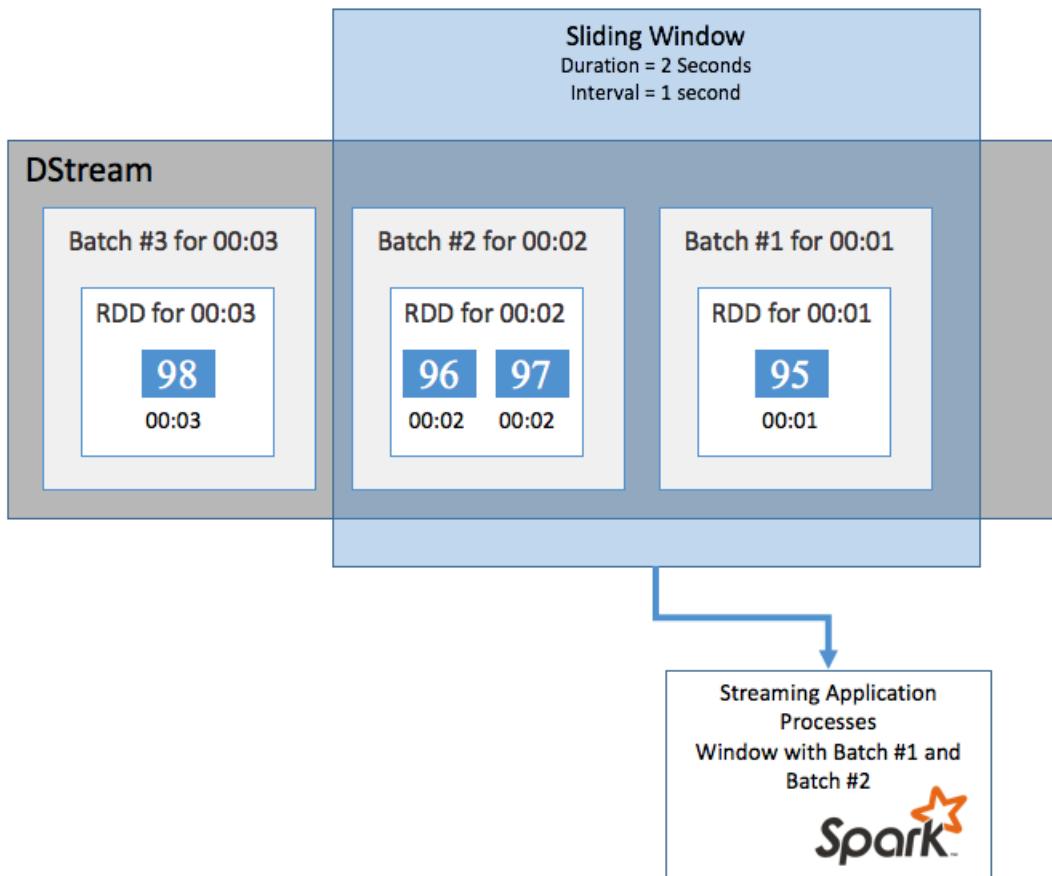
VALUE	TIME
14	1497314485327
15	1497314490346

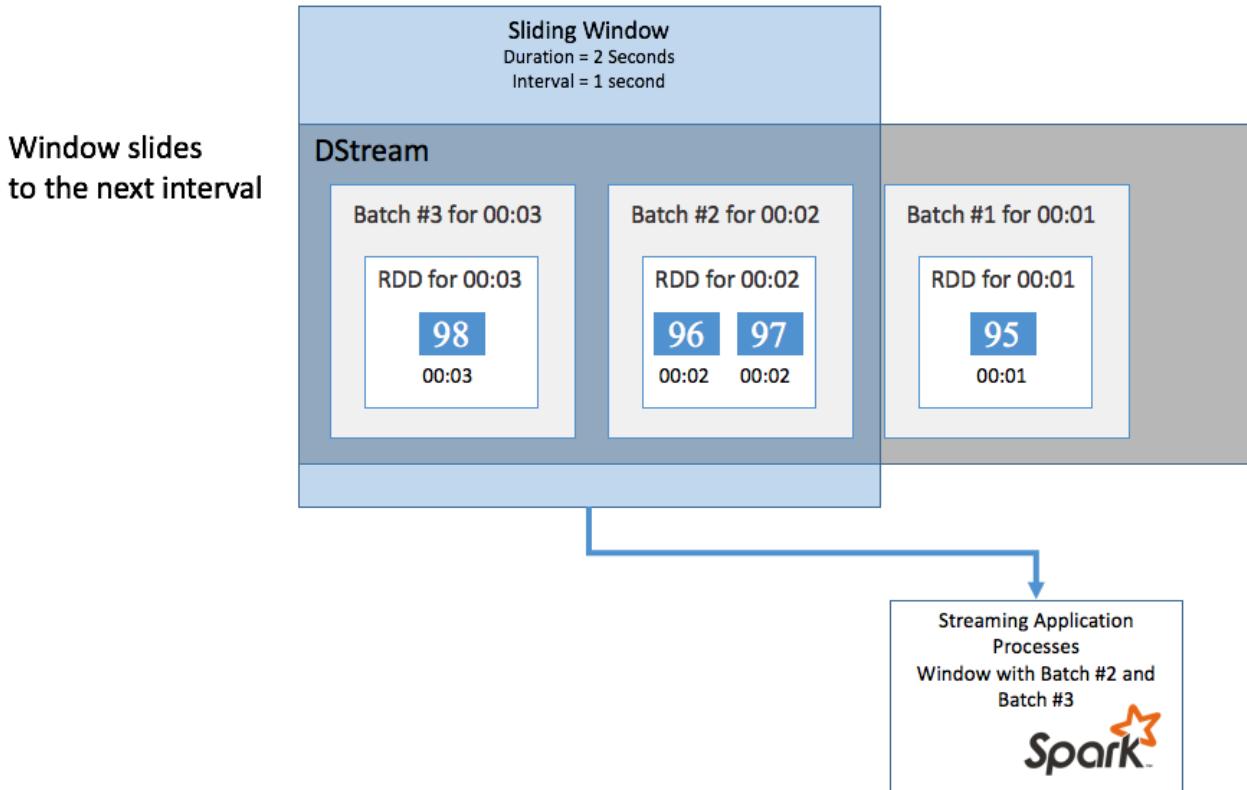
In the above expect six values in the typical case, because the DummySource creates a value every 5 seconds, and we emit a batch every 30 seconds.

Sliding Windows

If you want to perform aggregate calculations on your DStream over some time period, for example to get an average temperature over the last 2 seconds, you can use the sliding window operations included with Spark Streaming. A sliding window is defined as having a duration (referred to as the window length) and the interval at which the window's content are evaluated (referred to as the slide interval).

These sliding windows can overlap, for example you can define a window with a length of 2 seconds, that slides every 1 second. This means every time you perform an aggregation calculation, the window will include data from the last 1 second of the previous window as well as any new data in the next 1 second.





By way of example, we can enhance the code that uses the DummySource above to first collect the batches into a window with a 1 minute duration, that slides by 1 minute as well.

```
// A batch is created every 30 seconds
val ssc = new org.apache.spark.streaming.StreamingContext(spark.sparkContext,
org.apache.spark.streaming.Seconds(30))

// Set the active SQLContext so that we can access it statically within the foreachRDD
org.apache.spark.sql.SQLContext.setActive(spark.sqlContext)

// Create the stream
val stream = ssc.receiverStream(new DummySource())

// Process batches in 1 minute windows
stream.window(org.apache.spark.streaming.Minutes(1)).foreachRDD { rdd =>

    // Access the SQLContext and create a table called demo_numbers we can query
    val _sqlContext = org.apache.spark.sql.SQLContext.getOrCreate(rdd.sparkContext)
    _sqlContext.createDataFrame(rdd).toDF("value", "time")
    .registerTempTable("demo_numbers")
}

// Start the stream processing
ssc.start()
```

After the first minute, this will yield 12 entries or six entries from each of the two batches collected in the window.

VALUE	TIME
1	1497316294139
2	1497316299158
3	1497316304178

VALUE	TIME
4	1497316309204
5	1497316314224
6	1497316319243
7	1497316324260
8	1497316329278
9	1497316334293
10	1497316339314
11	1497316344339
12	1497316349361

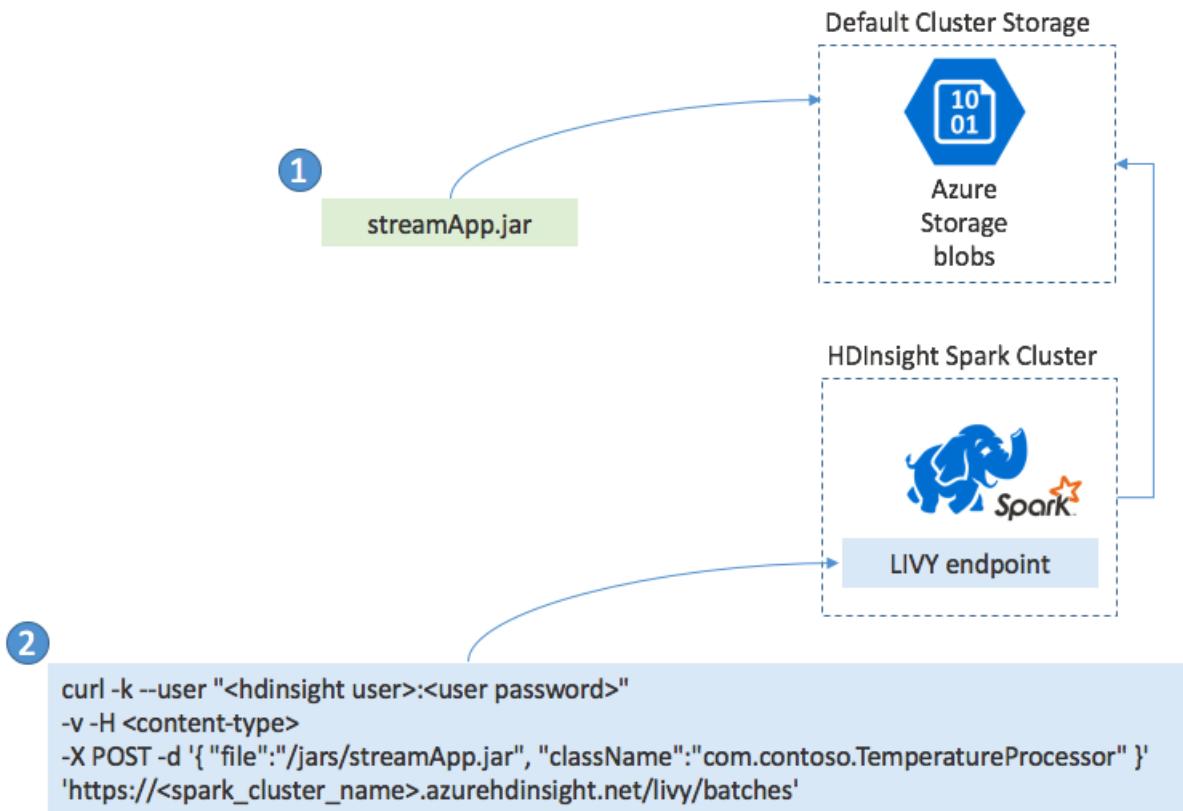
The sliding window functions available in the Spark Streaming API include `window`, `countByWindow`, `reduceByWindow` and `countByValueAndWindow`. For details on these functions see [Transformations on DStreams](#).

Checkpointing

In order to deliver resiliency and fault tolerance, Spark Streaming relies on checkpointing to insure that stream processing can continue uninterrupted, even in the face of node failures. In HDInsight, Spark creates checkpoints to durable storage (Azure Storage or Data Lake Store). These checkpoints store the metadata about the streaming application- such as the configuration, the operations defined by the application and any batches that were queued but not yet processed. In some cases, the checkpoints will also include the saving of the data in the RDD's to shorten the time it takes to rebuild the state of the data from what is present in the RDD's managed by Spark.

Deploying Spark Streaming Applications

You typically build your Spark Streaming application locally and then deploy it to Spark on HDInsight by copying the JAR file that contains your application to the default storage attached to your HDInsight cluster. Then you can start your application by using the LIVY REST API's available from your cluster. This is a POST operation where the body of the POST includes a JSON document that provides the path to your JAR, the name of the class whose main method defines and runs the streaming application, and optionally the resource requirements of the job (e.g., number of executors, memory and cores), and any configuration settings your application code requires.



The status of all applications can also be checked with a GET request against a LIVY endpoint. Finally, you can terminate a running application by issuing a DELETE request against the LIVY endpoint. For details on the LIVY API, see [Remote jobs with LIVY](#)

See also

- [Create an Apache Spark Cluster in HDInsight](#)
- [Spark Streaming Programming Guide](#)
- [Launch Spark jobs remotely with LIVY](#)

What is Spark Structured Streaming?

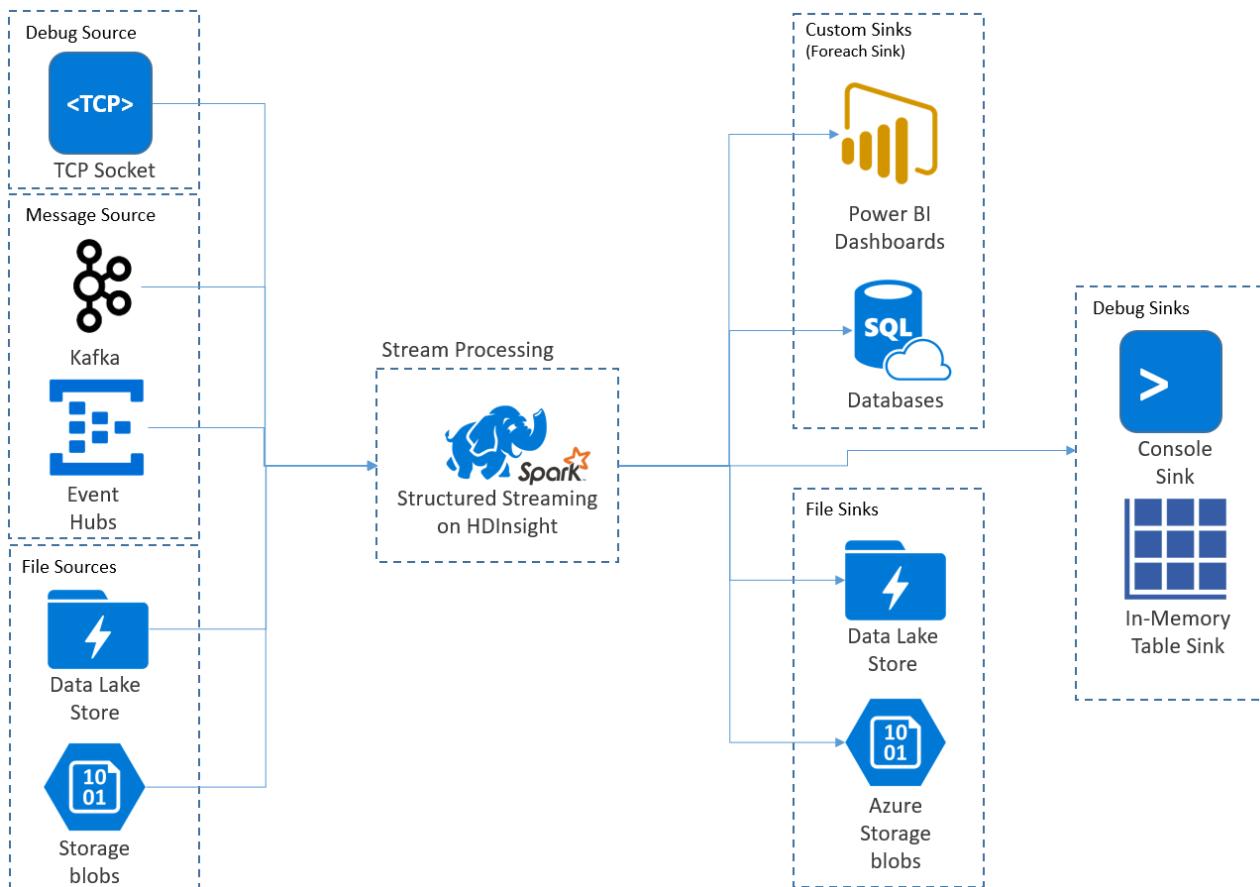
8/16/2017 • 10 min to read • [Edit Online](#)

Spark Structured Streaming enables you to implement scalable, high-throughput, fault-tolerant applications for the processing of data streams without having to build programs with specialized streaming constructs in mind.

Structured Streaming is built upon the Spark SQL engine, and improves upon the constructs from Spark SQL Data Frames and Datasets that enable you to write streaming queries in the same way you would write batch queries.

You can run your Structured Streaming applications on HDInsight Spark clusters, and connect it to process data in a streaming fashion from Kafka, a TCP socket (for debugging purposes), Azure Storage and Azure Data Lake Store. The latter two options which rely on external storage services enable you to watch for new files added into storage and process their content as if it were streamed.

Structured Streaming creates a long running query during which you are able to apply operations to the input data- such as selection, projection, aggregation, windowing and joining the streaming DataFrame with reference DataFrames and then output the result to file storage (Azure Storage Blobs or Data Lake Store) or to any datastore via custom code (such as SQL Database or Power BI). It also provides convenient output to the console which is useful when debugging locally and to an in-memory table which lets you peek at the data generated when debugging in HDInsight.

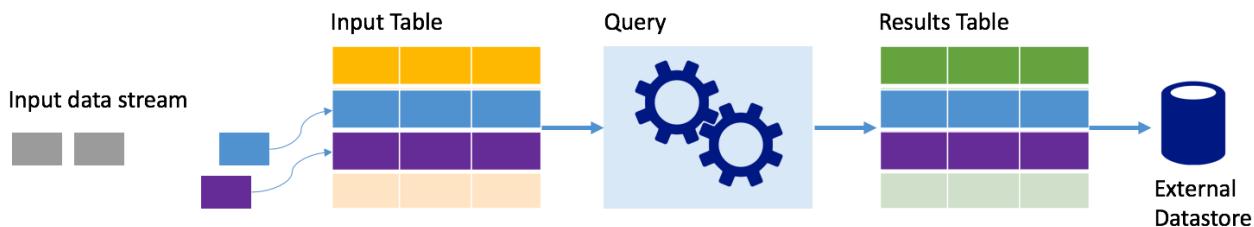


NOTE

Spark Structured Streaming is intended to be the replacement for Spark Streaming (DStreams). This means going forward, Structured Streaming will receive enhancements and maintenance, whereas DStreams will be in maintenance mode only. It is important to note, however, that Structured Streaming is currently not as feature complete as DStreams when it comes to the sources and sinks that it supports out of the box, so you should evaluate your requirements first before choosing the appropriate Spark stream processing option.

Streams as Tables

Spark Structured Streaming takes the perspective that a stream of data can be represented as a table that is unbounded in height- in other words it continues to grow as new data arrives. This Input Table is continuously processed by a long running query, and the results flushed out to an Output Table. This concept is best explained with the following illustration:



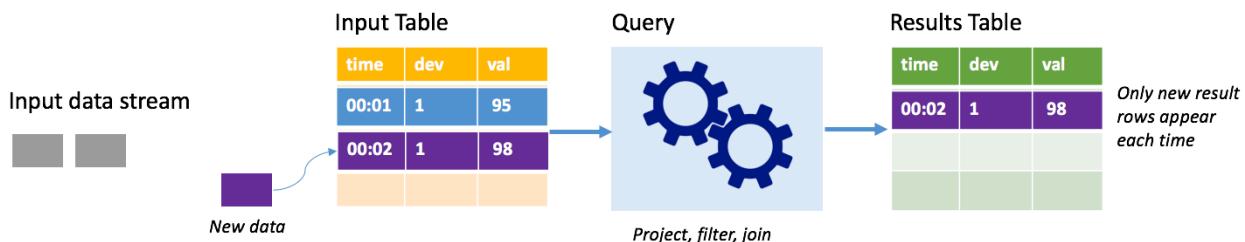
In Structured Streaming data arrives at the system and is immediately ingested into an Input Table. You write queries (using the DataFrame and Dataset APIs) that perform operations against this Input Table. The query output yields another table, called the Results Table. The Results Table contains results of your query from which you draw any data you would send to an external datastore (such a relational database). The timing of when data is processed from the Input Table is controlled by the trigger interval. By default Structured Streaming tries to process the data as soon as it arrives. In practice this means as soon as it is done processing the run of the previous query, it starts another processing run against any newly received data. However, you can also configure the trigger to run on a longer interval, so that the streaming data is processed according to time-based batches.

With regards to the output, the data in the Results Tables may be completely refreshed everytime there is new data so that it includes all of the output data since the streaming query began, or it may only contain just the data that is new since the last time the query was processed. This behavior is controlled by the output mode. Let's look at examples of each of these modes in turn.

Append Mode

In Append Mode, only the rows added to the Results Table since the last query run will be present in the Results Table and written to external storage. This is best explained by example. Suppose you have the simplest form of query that just copies all data from the Input Table to the Results Table unaltered. Every time a trigger happens, the new data is processed and the rows representing that new data appear in the Results Table.

Consider a scenario where you are processing telemetry from temperature sensors, like thermostats. Assume the first trigger processed one event at time 00:01 for device 1 having a temperature reading of 95 degrees. In the first trigger of the query, only the row with time 00:01 would appear in the Results Table. Consider what happens at time 00:02 when another event arrives. In this case, the only new row would be the row with time 00:02 and so the Results Table would contain only one row- the one with the time of 00:02 as shown in the illustration.

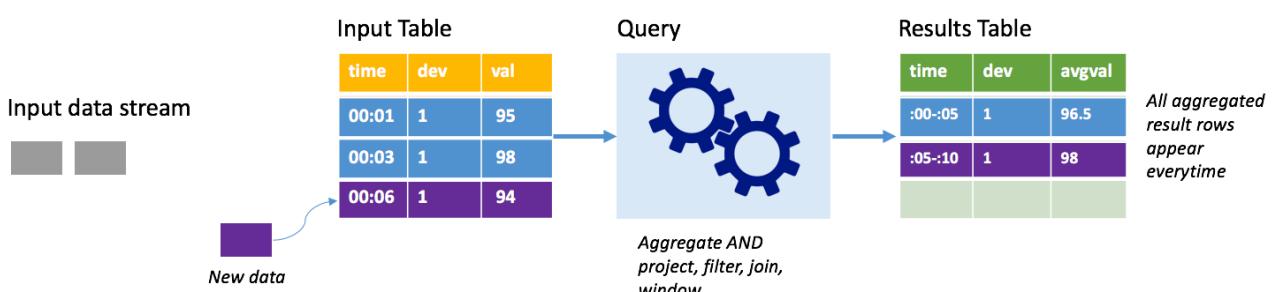


Of course, this is a trivial example query. Most likely when using the Append Mode in this way, your query would be applying projections (e.g., selecting the columns it cares about), filtering (e.g., picking only rows that match certain conditions) or joining (e.g., augmenting the data with data from a static lookup table). The Append Mode is useful, because it makes it easy to push only the relevant, new data points out to external storage.

Complete Mode

Now let's consider the same scenario, but apply the Complete Mode. In the Complete Mode, the entire Output Table is refreshed on every trigger so that it includes data not just from the most recent trigger run, but from all runs. In the trivial example we showed earlier, we could use the Complete Mode to copy the data unaltered from the Input Table to the Results Table. On every trigger run, the new result rows would appear along with all the previous rows. Naturally, this means that the Output Results table would end up storing all of the data collected since the query began- you would eventually run out of memory. This is why Complete Mode is intended for use with aggregate queries, which effectively summarize the data in some way and on every trigger the Results Table is updated with a new summary.

Let's use the following illustration as an example. Assume so far we have already processed 5 seconds worth of data and are looking at the result of processing the data for the sixth second. In our Input Table, we have collected events for time 00:01 and time 00:03. Let's assume the goal of our query is to tell us the average temperature of the device every five seconds. The implementation of this query has to apply an aggregate that takes all of the values that fall within each 5 second window of time, and averages the temperature and produces a row that represents the average temperature for that interval. At the end of the first 5 second window, we have two tuples that appear in the interval: (00:01, 1, 95) and (00:03, 1, 98). So for the window 00:00-00:05 we get a tuple with the average temperature of 96.5 degrees (the average of 95 and 98 is 96.5). Now let's consider what happens at the next 5 second window. In that window we only have one data point at time 00:06, to the resulting average temperature is 98 degrees. At time 00:10, when we use the Complete Mode, the Results Table will have the rows for both windows 00:00-00:05 and 00:05-00:10 because the query will output all the aggregated rows, not just the new ones. In other words, the Results Table will continue to grow as new windows are added.



It is important to realize that not all queries using Complete Mode will always cause the table to grow without bounds- and this is where Complete Mode is ideal. Consider in the above example that instead of averaging the temperature by time window, we averaged instead by the device ID. The Result Table would contain a fixed number of rows (one per device) with the average temperature for the device across all data points received from that device. As new temperatures are received, the Results Table would be updated so that the averages in the table are always current.

Components of a Spark Structured Streaming Application

Let's examine a simple example query that shows how the aforementioned concepts look in a Spark Structured Streaming query. In this case, we will build up a query that summarize the temperature readings by hour-long window. For simplicity, the data we use is stored in JSON files in Azure Storage (attached as the default storage for the HDInsight cluster). The data in one JSON files looks like the following:

```
{"time":1469501107,"temp":"95"}  
{"time":1469501147,"temp":"95"}  
{"time":1469501202,"temp":"95"}  
{"time":1469501219,"temp":"95"}  
{"time":1469501225,"temp":"95"}
```

These JSON files are stored in the "temps" subfolder, underneath the container used by the HDInsight cluster.

Define the input source

First you need to configure a DataFrame that describes the source of the data and any settings required by that source. In our example, we draw from the JSON files in Azure Storage and apply a schema to them at read time.

```
import org.apache.spark.sql.types._  
import org.apache.spark.sql.functions._  
  
//This is the cluster-local path to the folder containing the JSON files  
val inputPath = "/temps/"  
  
//Define the schema of the JSON files as having the "time" of type TimeStamp and the "temp" field of type String  
val jsonSchema = new StructType().add("time", TimestampType).add("temp", StringType)  
  
//Create a Streaming DataFrame by calling readStream and configuring it with schema and path  
val streamingInputDF = spark.readStream.schema(jsonSchema).json(inputPath)
```

Apply the query

Next, you apply a query that contains the operation you want against the Streaming DataFrame. In this case, we apply an aggregation that groups all the rows into 1 hour windows, and then computes the minimum temperature, average temperature and max temperature in that 1 hour window.

```
val streamingAggDF = streamingInputDF.groupBy(window($"time", "1 hour")).agg(min($"temp"), avg($"temp"), max($"temp"))
```

Define the output sink

Next, you define the destination for the rows that are added to the ResultsTable within each trigger interval. In this case, we want simply want to output all of the rows to an in-memory table called "temps" that we can later query with SparkSQL. Notice we use the output mode of Complete so that all rows for all windows are output, everytime.

```
val streamingOutDF = streamingAggDF.writeStream.format("memory").queryName("temps").outputMode("complete")
```

Start the query

Start the streaming query and run until a termination signal is received.

```
val query = streamingOutDF.start()
```

View the results

While the query is running, in the same SparkSession, we can run a SparkSQL query against the "temps" table in which the query results are being stored.

```
select * from temps
```

This would yield results similar to the following:

WINDOW	MIN(TEMP)	AVG(TEMP)	MAX(TEMP)
{u'start': u'2016-07-26T02:00:00.000Z', u'end': ...}	95	95.231579	99
{u'start': u'2016-07-26T03:00:00.000Z', u'end': ...}	95	96.023048	99
{u'start': u'2016-07-26T04:00:00.000Z', u'end': ...}	95	96.797133	99
{u'start': u'2016-07-26T05:00:00.000Z', u'end': ...}	95	96.984639	99
{u'start': u'2016-07-26T06:00:00.000Z', u'end': ...}	95	97.014749	99
{u'start': u'2016-07-26T07:00:00.000Z', u'end': ...}	95	96.980971	99
{u'start': u'2016-07-26T08:00:00.000Z', u'end': ...}	95	96.965997	99

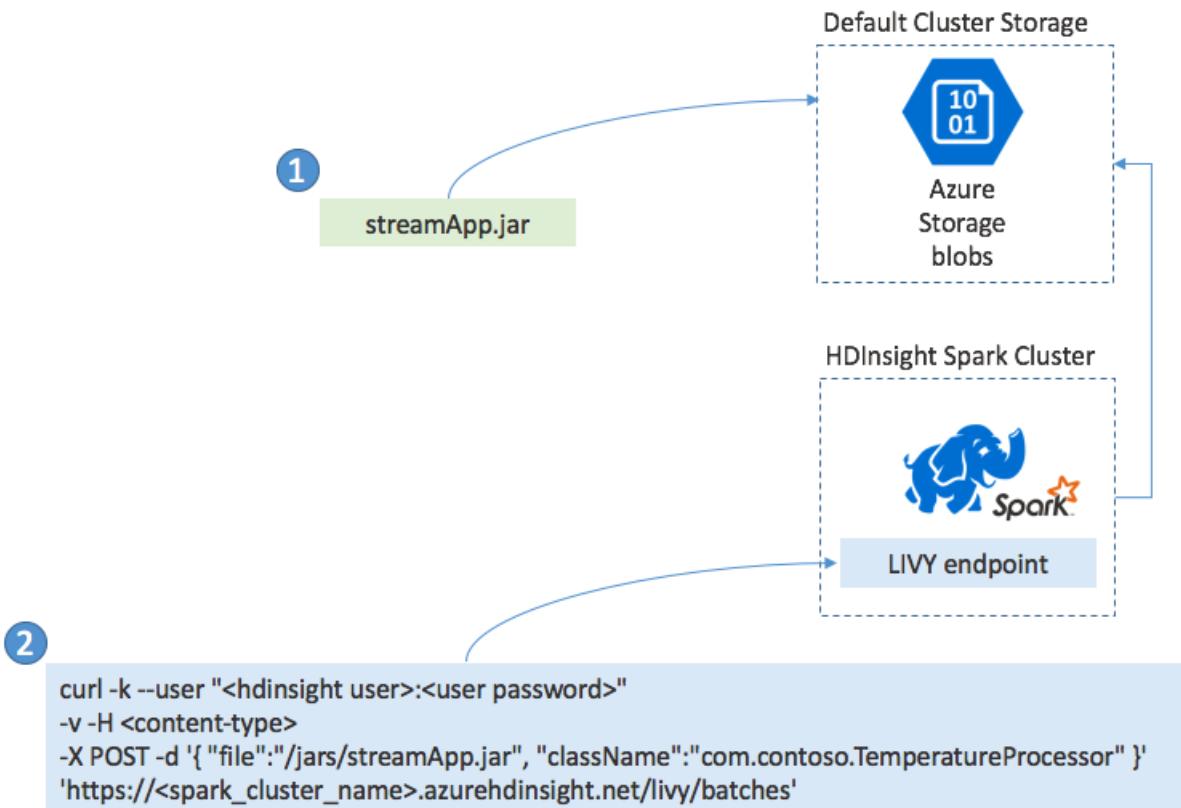
For details on the Spark Structured Stream API, along with the input data sources, operations and output sinks it supports see [Spark Structured Streaming Programming Guide](#).

Checkpointing and Write Ahead Logs

In order to deliver resiliency and fault tolerance, Structured Streaming relies on checkpointing to insure that stream processing can continue uninterrupted, even in the face of node failures. In HDInsight, Spark creates checkpoints to durable storage (Azure Storage or Data Lake Store). These checkpoints store the progress information about the streaming query. In addition, Structured Streaming utilizes a Write Ahead Log. The purpose of the WAL is to capture ingested data that has been received but not processed by a query, so if a failure occurs and processing is restarted the events received from the source are not lost.

Deploying Spark Streaming Applications

You typically build your Structured Streaming application locally and then deploy them to Spark on HDInsight by copying the JAR file that contains your application to the default storage attached to your HDInsight cluster. Then you can start your application by using the LIVY REST API's available from your cluster. This is a POST operation where the body of the POST includes a JSON document that provides the path to your JAR, the name of the class whose main method defines and runs the streaming application, and optionally the resource requirements of the job (e.g., number of executors, memory and cores), and any configuration settings your application code requires.



The status of all applications can also be checked with a GET request against a LIVY endpoint. Finally, you can terminate a running application by issuing a DELETE request against the LIVY endpoint. For details on the LIVY API, see [Remote jobs with LIVY](#)

See also

- [Create an Apache Spark Cluster in HDInsight](#)
- [Spark Structured Streaming Programming Guide](#)
- [Launch Spark jobs remotely with LIVY](#)

Apache Spark streaming (DStream) example with Kafka (preview) on HDInsight

8/16/2017 • 3 min to read • [Edit Online](#)

Learn how to use Spark Apache Spark to stream data into or out of Apache Kafka on HDInsight using DStreams. This example uses a Jupyter notebook that runs on the Spark cluster.

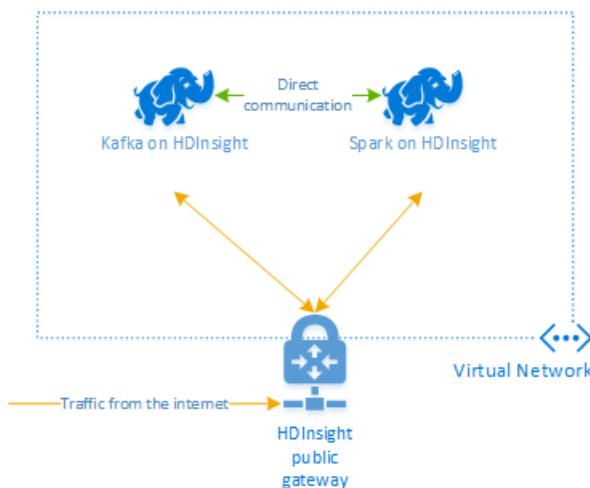
NOTE

The steps in this document create an Azure resource group that contains both a Spark on HDInsight and a Kafka on HDInsight cluster. These clusters are both located within an Azure Virtual Network, which allows the Spark cluster to directly communicate with the Kafka cluster.

When you are done with the steps in this document, remember to delete the clusters to avoid excess charges.

Create the clusters

Apache Kafka on HDInsight does not provide access to the Kafka brokers over the public internet. Anything that talks to Kafka must be in the same Azure virtual network as the nodes in the Kafka cluster. For this example, both the Kafka and Spark clusters are located in an Azure virtual network. The following diagram shows how communication flows between the clusters:



NOTE

Though Kafka itself is limited to communication within the virtual network, other services on the cluster such as SSH and Ambari can be accessed over the internet. For more information on the public ports available with HDInsight, see [Ports and URLs used by HDInsight](#).

While you can create an Azure virtual network, Kafka, and Spark clusters manually, it's easier to use an Azure Resource Manager template. Use the following steps to deploy an Azure virtual network, Kafka, and Spark clusters to your Azure subscription.

1. Use the following button to sign in to Azure and open the template in the Azure portal.

[Deploy to Azure >](#)

The Azure Resource Manager template is located at

<https://hditutorialdata.blob.core.windows.net/armtemplates/create-linux-based-kafka-spark-cluster-in-vnet-v2.1.json>.

WARNING

To guarantee availability of Kafka on HDInsight, your cluster must contain at least three worker nodes. This template creates a Kafka cluster that contains three worker nodes.

This template creates an HDInsight 3.6 cluster for both Kafka and Spark.

2. Use the following information to populate the entries on the **Custom deployment** blade:

TEMPLATE

 Customized template
4 resources

[Edit template](#) [Edit parameters](#) [Learn more](#)

BASICS

* Subscription

* Resource group Create new Use existing

* Location

SETTINGS

* Base Cluster Name ✓

Cluster Login User Name

* Cluster Login Password ⚡ ✓

Ssh User Name

* Ssh Password ⚡ ✓

TERMS AND CONDITIONS

[Azure Marketplace Terms](#) | [Azure Marketplace](#)

By clicking "Purchase," I (a) agree to the applicable legal terms associated with the offering; (b) authorize Microsoft to charge or bill my current payment method for the fees associated the offering(s), including applicable taxes, with the same billing frequency as my Azure subscription, until I discontinue use of the offering(s); and (c) agree that, if the deployment involves 3rd party offerings, Microsoft may share my contact information and other details of such deployment with the publisher of that offering.

Microsoft assumes no responsibility for any actions performed by third-party templates and does not provide rights for third-

I agree to the terms and conditions stated above

Pin to dashboard

Purchase

- **Resource group:** Create a group or select an existing one. This group contains the HDInsight cluster.
- **Location:** Select a location geographically close to you.

- **Base Cluster Name:** This value is used as the base name for the Spark and Kafka clusters. For example, entering **hdi** creates a Spark cluster named **spark-hdi** and a Kafka cluster named **kafka-hdi**.
- **Cluster Login User Name:** The admin user name for the Spark and Kafka clusters.
- **Cluster Login Password:** The admin user password for the Spark and Kafka clusters.
- **SSH User Name:** The SSH user to create for the Spark and Kafka clusters.
- **SSH Password:** The password for the SSH user for the Spark and Kafka clusters.

3. Read the **Terms and Conditions**, and then select **I agree to the terms and conditions stated above**.
4. Finally, check **Pin to dashboard** and then select **Purchase**. It takes about 20 minutes to create the clusters.

Once the resources have been created, you are redirected to a blade for the resource group that contains the clusters and web dashboard.

NAME	TYPE	LOCATION
kafka-hdi	HDInsight Clusters	North Central US
spark-hdi	HDInsight Clusters	North Central US
gateway-199eb825822c4b6d83bfee52fa1	Load balancer	North Central US
gateway-b546fc43f4d54289a39728c2398	Load balancer	North Central US
headnode-199eb825822c4b6d83bfee52fa	Load balancer	North Central US
headnode-b546fc43f4d54289a39728c239	Load balancer	North Central US
nic-gateway-0-b546fc43f4d54289a39728	Network inter...	North Central US
nic-gateway-1-199eb825822c4b6d83bfee	Network inter...	North Central US
nic-gateway-2-b546fc43f4d54289a39728	Network inter...	North Central US

IMPORTANT

Notice that the names of the HDInsight clusters are **spark-BASENAME** and **kafka-BASENAME**, where **BASENAME** is the name you provided to the template. You use these names in later steps when connecting to the clusters.

Use the notebooks

The code for the example described in this document is available at <https://github.com/Azure-Samples/hdinsight-spark-scala-kafka>.

Follow the steps in the `README.md` file to complete this example.

Delete the cluster

WARNING

Billing for HDInsight clusters is prorated per minute, whether you are using them or not. Be sure to delete your cluster after you have finished using it. For more information, see [How to delete an HDInsight cluster](#).

Since the steps in this document create both clusters in the same Azure resource group, you can delete the resource group in the Azure portal. Deleting the group removes all resources created by following this document, the Azure Virtual Network, and storage account used by the clusters.

Next steps

In this example, you learned how to use Spark to read and write to Kafka. Use the following links to discover other ways to work with Kafka:

- [Get started with Apache Kafka on HDInsight](#)
- [Use MirrorMaker to create a replica of Kafka on HDInsight](#)
- [Use Apache Storm with Kafka on HDInsight](#)

Apache Spark streaming: Process data from Azure Event Hubs with Spark cluster on HDInsight

8/16/2017 • 15 min to read • [Edit Online](#)

In this article, you create an Apache Spark streaming sample that involves the following steps:

1. You use a standalone application to ingest messages into an Azure Event Hub.
2. With two different approaches, you retrieve the messages from Event Hub in real-time using an application running in Spark cluster on Azure HDInsight.
3. You build streaming analytic pipelines to persist data to different storage systems, or get insights from data on the fly.

Prerequisites

- An Azure subscription. See [Get Azure free trial](#).
- An Apache Spark cluster on HDInsight. For instructions, see [Create Apache Spark clusters in Azure HDInsight](#).

Spark Streaming concepts

For a detailed explanation of Spark streaming, see [Apache Spark streaming overview](#). HDInsight brings the same streaming features to a Spark cluster on Azure.

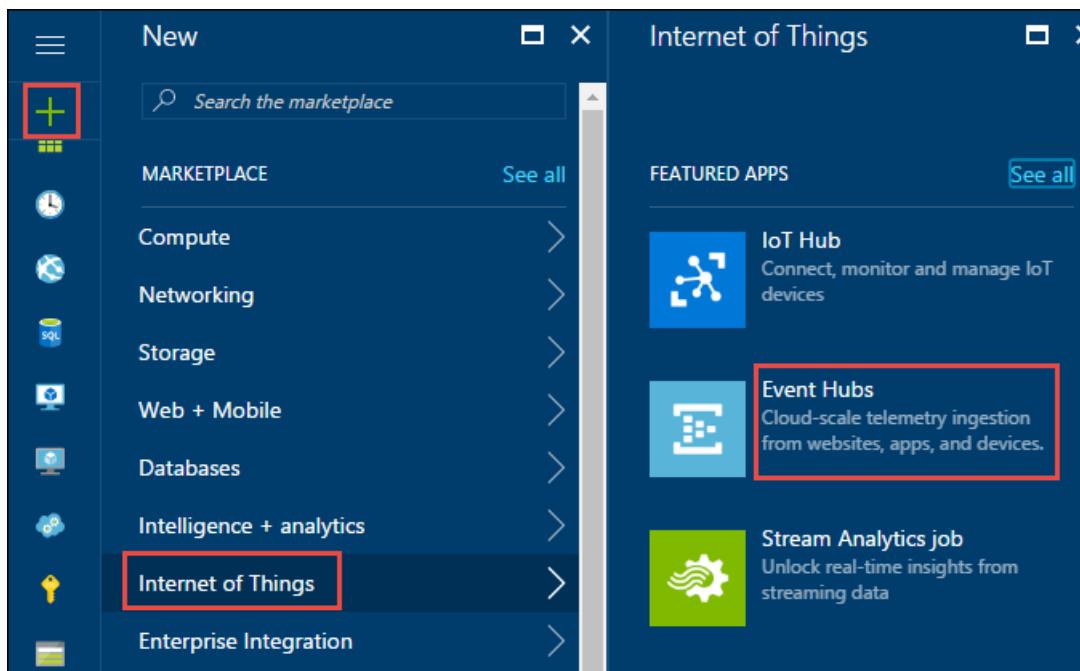
What does this solution do?

In this article, to create a Spark streaming example, perform the following steps:

1. Create an Azure Event Hub that will receive a stream of events.
2. Run a local standalone application that generates events and pushes it to the Azure Event Hub. The sample application that does this is published at <https://github.com/hdinsight/spark-streaming-data-persistence-examples>.
3. Run a streaming application remotely on a Spark cluster that reads streaming events from Azure Event Hub and perform various data processing/analysis.

Create an Azure Event Hub

1. Log on to the [Azure Portal](#), and click **New** at the top left of the screen.
2. Click **Internet of Things**, then click **Event Hubs**.



3. In the **Create namespace** blade, enter a namespace name. choose the pricing tier (Basic or Standard). Also, choose an Azure subscription, resource group, and location in which to create the resource. Click **Create** to create the namespace.

Create namespace

Event Hubs - PREVIEW

* Name	mysbnamespace ✓
	.servicebus.windows.net
* Pricing tier	Standard >
* Subscription	<input type="button" value="▼"/>
* Resource group <small>i</small>	<input type="radio"/> Create new <input checked="" type="radio"/> Use existing <input type="button" value="▼"/>
* Location	South Central US <input type="button" value="▼"/>
<input type="checkbox"/> Pin to dashboard	
<input style="background-color: #0078d4; color: white; font-weight: bold; padding: 5px; margin-right: 10px;" type="button" value="Create"/> Automation options	

NOTE

You should select the same **Location** as your Apache Spark cluster in HDInsight to reduce latency and costs.

4. In the Event Hubs namespace list, click the newly-created namespace.
5. In the namespace blade, click **Event Hubs**, and then click **+ Event Hub** to create a new Event Hub.

The screenshot shows the Azure Event Hubs management interface for a resource named 'mysbnamespace'. On the left, a navigation menu includes 'Overview', 'Access control (IAM)', 'Tags', 'Diagnose and solve problems', 'SETTINGS' (with options like 'Shared access policies', 'Scale', 'Properties', 'Locks', and 'Automation script'), and 'ENTITIES' (with a selected 'Event Hubs' item). On the right, a main panel displays a search bar ('Search to filter items...') and a table header 'NAME'. A red box highlights the '+ Event Hub' button in the top-left corner of the main area.

6. Type a name for your Event Hub, set the partition count to 10, and message retention to 1. We are not archiving the messages in this solution so you can leave the rest as default, and then click **Create**.

Create Event Hub

mysbnamespace - PREVIEW

* Name
myeventhub ✓

Partition Count 10

Message Retention 1

Archive
On Off

Time window (minutes)
5

Size window (MB)
300

* Container
Nothing selected >

Storage Account
Nothing selected

Create

7. The newly created Event Hub is listed in the Event Hub blade.

mysbnamespace - Event Hubs

Event Hub - PREVIEW

Search (Ctrl+/
)

Event Hub

Search to filter items...

NAME	STATUS	MESSAGE RETENTION	PARTITION COUNT
myeventhub	Active	1	10

OVERVIEW

Access control (IAM)

Tags

Diagnose and solve problems

SETTINGS

Shared access policies

Scale

Properties

Locks

Automation script

ENTITIES

Event Hubs

8. Back in the namespace blade (not the specific Event Hub blade), click **Shared access policies**, and then click **RootManageSharedAccessKey**.

The screenshot shows the Azure portal interface for managing shared access policies. On the left, there's a sidebar with options like Overview, Access control (IAM), Tags, Diagnose and solve problems, and SETTINGS. Under SETTINGS, 'Shared access policies' is selected and highlighted with a red box. The main pane displays a list of shared access policies, with 'RootManageSharedAccessKey' being the active one. Below the policy name, there are sections for CLAIMS, showing 'Manage', 'Send', and 'Listen'. To the right of the policy name, there are buttons for Manage, Send, and Listen.

- Click the copy button to copy the **RootManageSharedAccessKey** primary key and connection string to the clipboard. Save these to use later in the tutorial.

This screenshot shows the detailed configuration for the 'RootManageSharedAccessKey' policy. It includes fields for Policy name (set to 'RootManageSharedAccessKey'), Claim (with 'Manage', 'Send', and 'Listen' checked), and various key/connection string fields. The 'PRIMARY KEY' field contains a blurred value and features a copy icon, which is highlighted with a red box. Other fields include 'SECONDARY KEY' (blurred value, copy icon), 'CONNECTION STRING-PRIMARY KEY' ('Endpoint=sb://mysbnamespace.servicebus.windows.net', copy icon), and 'CONNECTION STRING-SECONDARY KEY' ('Endpoint=sb://mysbnamespace.servicebus.windows.net', copy icon).

Send messages to Azure Event Hub using a sample Scala application

In this section you use a standalone local Scala application that generates a stream of events and sends it to Azure Event Hub that you created earlier. This application is available on GitHub at <https://github.com/hdinsight/eventhubs-sample-event-producer>. The steps here assume that you have already forked this GitHub repository.

- Make sure you have the following installed on the computer where you run this application.
 - Oracle Java Development kit. You can install it from [here](#).
 - Apache Maven. You can download it from [here](#). Instructions to install Maven are available [here](#).
- Open a command prompt and navigate to the location you cloned the GitHub repo for the sample Scala application and run the following command to build the application.

```
mvn package
```

3. The output jar for the application, **com-microsoft-azure-eventhubs-client-example-0.2.0.jar**, is created under **/target** directory. You use this JAR later in this article to test the complete solution.

Create application to receive messages from Event Hub into a Spark cluster

We have two approaches to connect Spark Streaming and Azure Event Hubs, Receiver-based connection and Direct-DStream-based connection. Direct-DStream-based is introduced on Jan of 2017, in the 2.0.3 release. It is supposed to replace the original receiver-based connection as it is more performant and resource-efficient. More details found in <https://github.com/hdinsight/spark-eventhubs>. Direct DStream only supports Spark 2.0+.

Build applications with the dependency to spark-eventhubs connector

We will also publish the staging version of Spark-EventHubs in GitHub. To use the staging version of Spark-EventHubs, the first step is to indicate GitHub as the source repo by adding the following entry to pom.xml:

```
<repository>
    <id>spark-eventhubs</id>
    <url>https://raw.github.com/hdinsight/spark-eventhubs/maven-repo/</url>
    <snapshots>
        <enabled>true</enabled>
        <updatePolicy>always</updatePolicy>
    </snapshots>
</repository>
```

You can then add the following dependency to your project to take the pre-released version.

Maven Dependency

```
<!-- https://mvnrepository.com/artifact/com.microsoft.azure/spark-streaming-eventhubs_2.11 -->
<dependency>
    <groupId>com.microsoft.azure</groupId>
    <artifactId>spark-streaming-eventhubs_2.11</artifactId>
    <version>2.0.4</version>
</dependency>
```

SBT Dependency

```
// https://mvnrepository.com/artifact/com.microsoft.azure/spark-streaming-eventhubs_2.11
libraryDependencies += "com.microsoft.azure" % "spark-streaming-eventhubs_2.11" % "2.0.4"
```

Direct DStream Connection

A pre-built jar file containing examples using Direct DStream can be downloaded in http://central.maven.org/maven2/com/microsoft/azure/spark-streaming-eventhubs_2.11/2.0.4/spark-streaming-eventhubs_2.11-2.0.4.jar.

The jar file contains three examples whose source code are available at <https://github.com/hdinsight/spark-eventhubs/tree/master/examples/src/main/scala/com/microsoft/spark/streaming/examples/directdstream>.

Taking [WindowingWordCount](#) as an example:

```

private def createStreamingContext(
    sparkCheckpointDir: String,
    batchDuration: Int,
    namespace: String,
    eventHubName: String,
    eventHubParams: Map[String, String],
    progressDir: String) = {
  val ssc = new StreamingContext(new SparkContext(), Seconds(batchDuration))
  ssc.checkpoint(sparkCheckpointDir)
  val inputDirectStream = EventHubsUtils.createDirectStreams(
    ssc,
    namespace,
    progressDir,
    Map(eventHubName -> eventHubParams))

  inputDirectStream.map(receivedRecord => (new String(receivedRecord.getBody), 1)).
    reduceByKeyAndWindow((v1, v2) => v1 + v2, (v1, v2) => v1 - v2, Seconds(batchDuration * 3),
      Seconds(batchDuration)).print()

  ssc
}

def main(args: Array[String]): Unit = {

  if (args.length != 8) {
    println("Usage: program progressDir PolicyName PolicyKey EventHubNamespace EventHubName" +
      " BatchDuration(seconds) Spark_Checkpoint_Directory maxRate")
    sys.exit(1)
  }

  val progressDir = args(0)
  val policyName = args(1)
  val policykey = args(2)
  val namespace = args(3)
  val name = args(4)
  val batchDuration = args(5).toInt
  val sparkCheckpointDir = args(6)
  val maxRate = args(7)

  val eventhubParameters = Map[String, String] (
    "eventhubs.policyname" -> policyName,
    "eventhubs.policykey" -> policykey,
    "eventhubs.namespace" -> namespace,
    "eventhubs.name" -> name,
    "eventhubs.partition.count" -> "32",
    "eventhubs.consumergroup" -> "$Default",
    "eventhubs.maxRate" -> s"$maxRate"
  )

  val ssc = StreamingContext.getOrCreate(sparkCheckpointDir,
    () => createStreamingContext(sparkCheckpointDir, batchDuration, namespace, name,
      eventhubParameters, progressDir))

  ssc.start()
  ssc.awaitTermination()
}

```

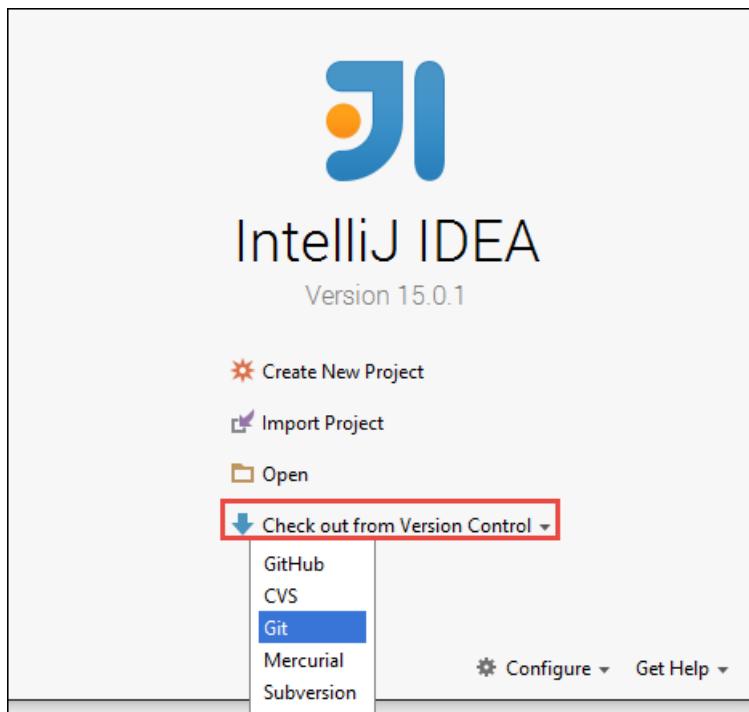
In the above example, `eventhubParameters` are the parameters specific to a single EventHubs instance and you have to pass it to the `createDirectStreams` API which constructs a Direct DStream object mapping to a Event Hubs namespace. Over the Direct DStream object, you can call any DStream API provided by Spark Streaming API framework. In this example, we calculate the frequency of each word within the last 3 micro batch intervals.

Receiver-based Connection

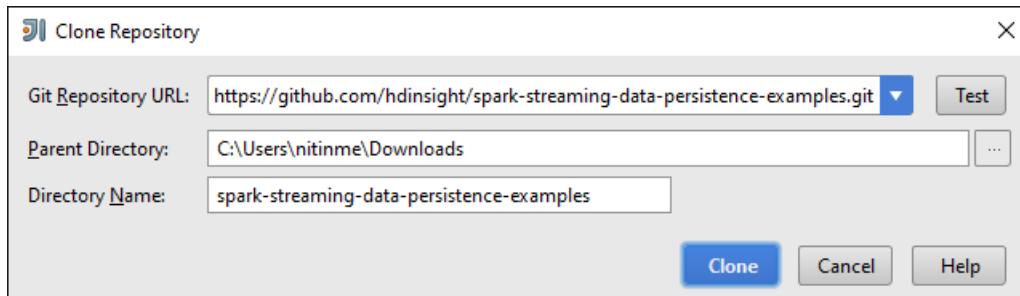
A Spark streaming example application written in Scala, which receives events and route the to different

destinations, is available at <https://github.com/hdinsight/spark-streaming-data-persistence-examples>. Follow the steps below to update the application for your Event Hub configuration and create the output jar.

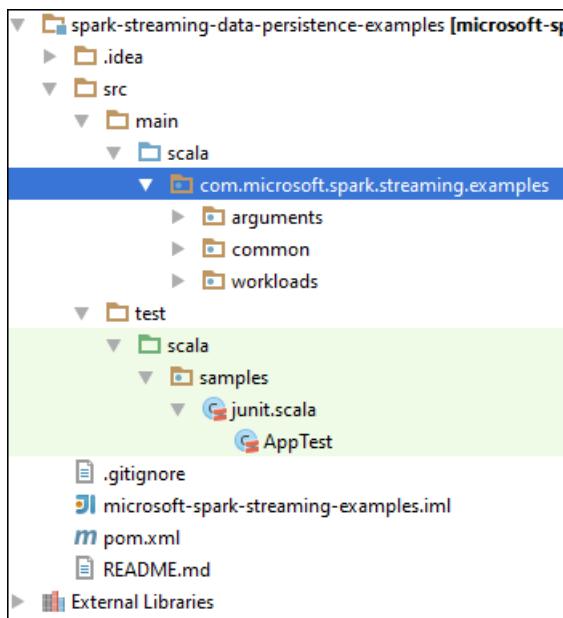
1. Launch IntelliJ IDEA and from the launch screen select **Check out from Version Control** and then click **Git**.



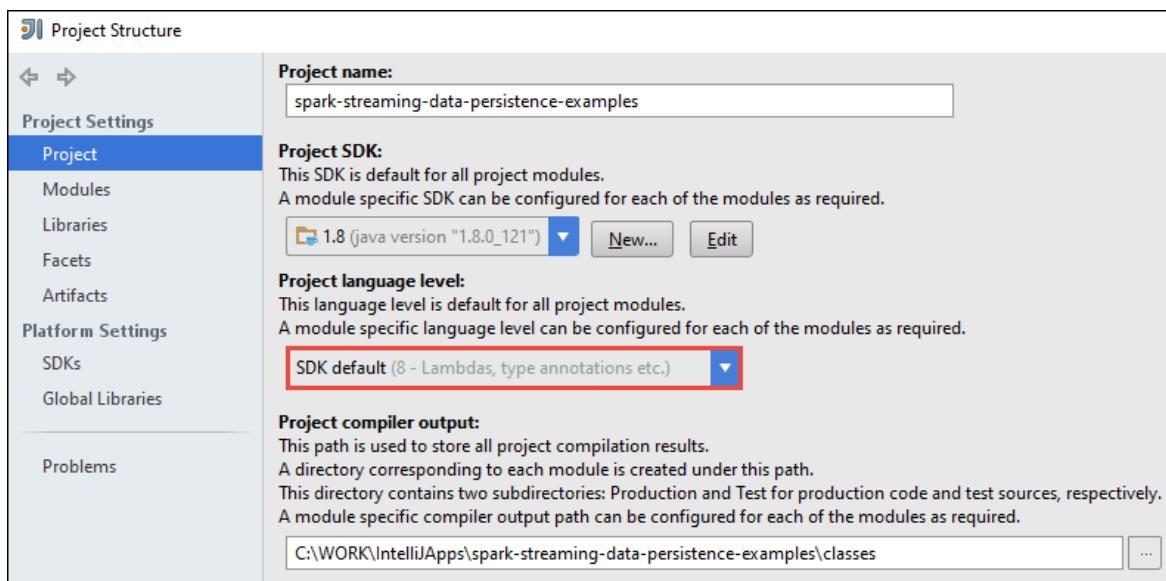
2. In the **Clone Repository** dialog box, provide the URL to the Git repository to clone from, specify the directory to clone to, and then click **Clone**.



3. Follow the prompts till the project is completely cloned. Press **Alt + 1** to open the **Project View**. It should resemble the following.



4. Make sure the application code is compiled with Java8. To ensure this, click **File**, click **Project Structure**, and on the **Project** tab, make sure Project language level is set to **8 - Lambdas, type annotations, etc..**



5. Open the **pom.xml** and make sure the Spark version is correct. Under `<properties>` node, look for the following snippet and verify the Spark version.

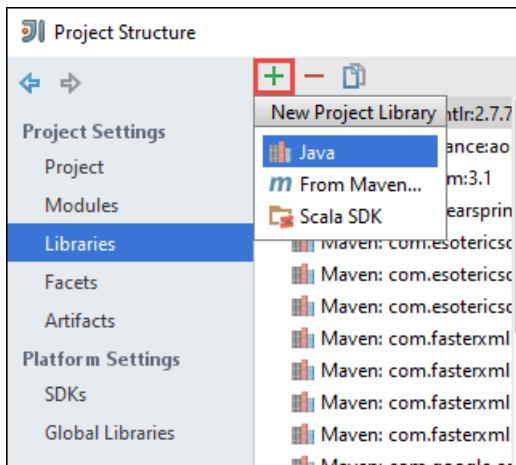
```

<scala.version>2.11.8</scala.version>
<scala.compat.version>2.11.8</scala.compat.version>
<scala.binary.version>2.11</scala.binary.version>
<spark.version>2.0.0</spark.version>

```

6. The application requires a dependency jar called **JDBC driver jar**. This is required to write the messages received from Event Hub into an Azure SQL database. You can download this jar (v4.1 or later) from [here](#). Add reference to this jar in the project library. Perform the following steps:

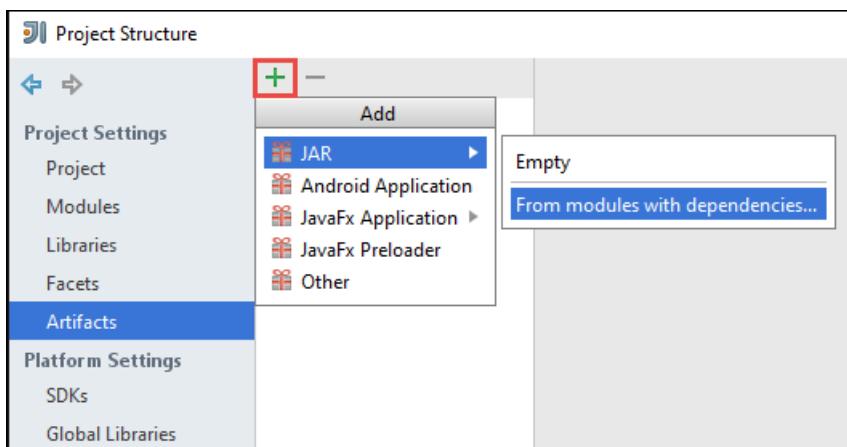
- a. From IntelliJ IDEA window where you have the application open, click **File**, click **Project Structure**, and then click **Libraries**.
- b. Click the add icon (+), click **Java**, and then navigate to the location where you downloaded the JDBC driver jar. Follow the prompts to add the jar file to the project library.



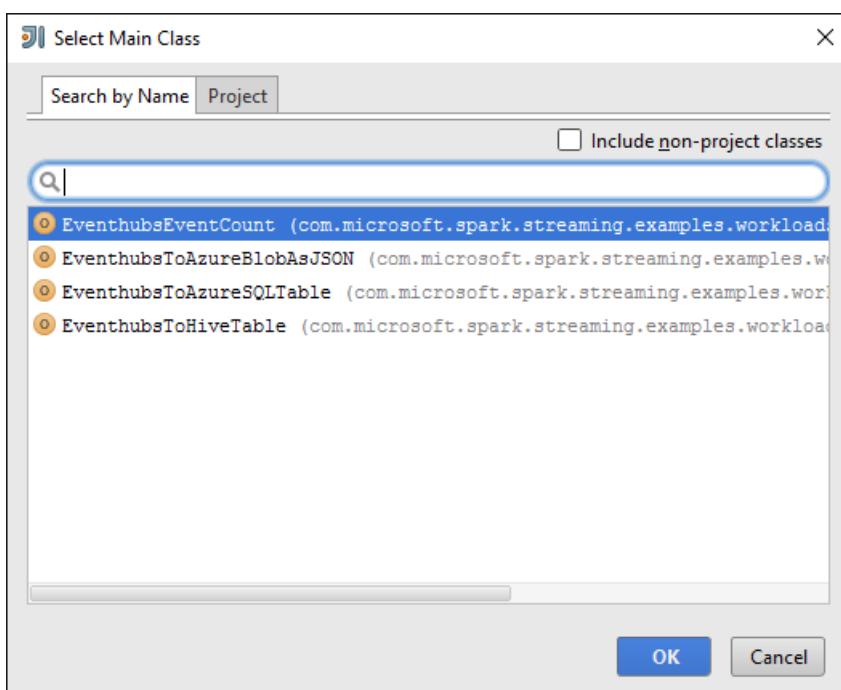
c. Click **Apply**.

7. Create the output jar file. Perform the following steps.

- In the **Project Structure** dialog box, click **Artifacts** and then click the plus symbol. From the pop-up dialog box, click **JAR**, and then click **From modules with dependencies**.

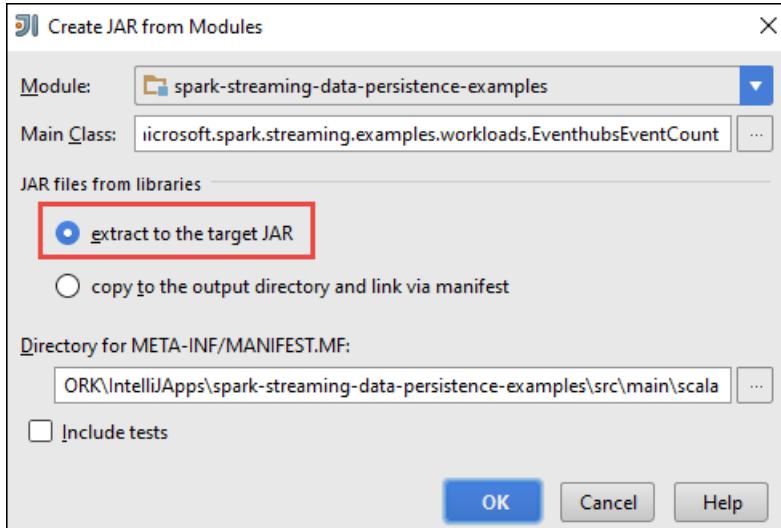


- In the **Create JAR from Modules** dialog box, click the ellipsis (...) against the **Main Class**.
- In the **Select Main Class** dialog box, select any of the available classes and then click **OK**.

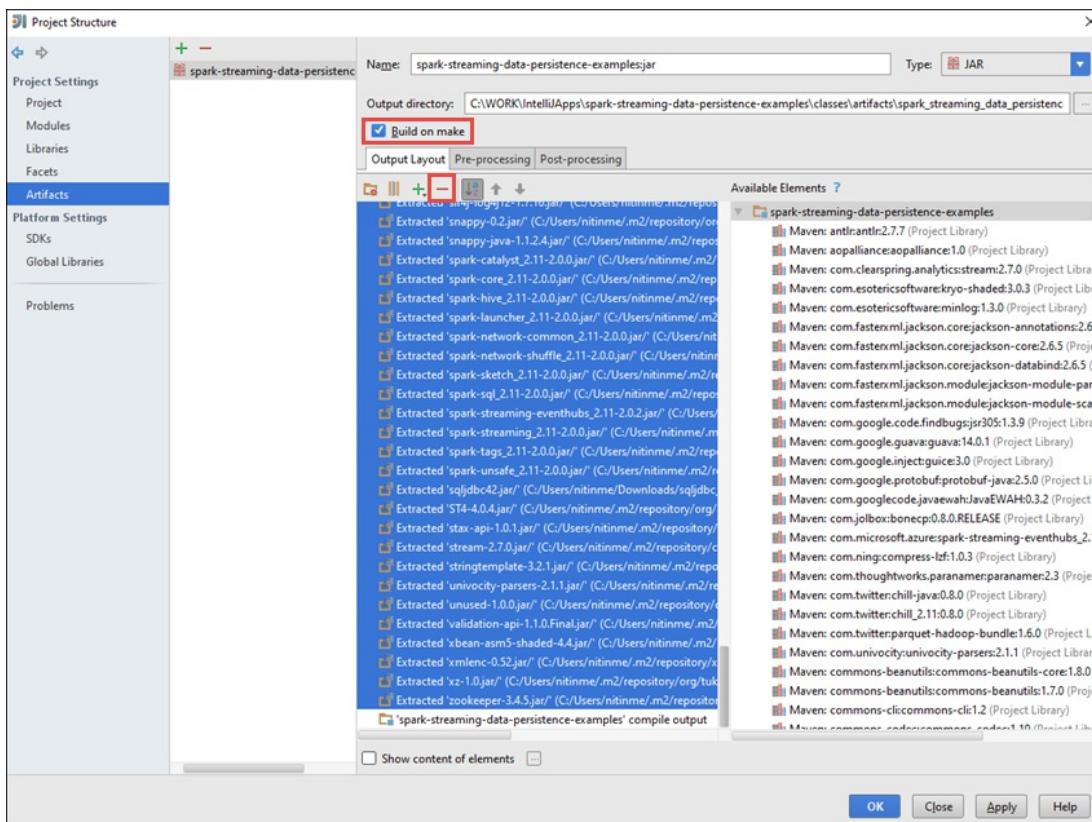


- In the **Create JAR from Modules** dialog box, make sure that the option to **extract to the target**

JAR is selected, and then click **OK**. This creates a single JAR with all dependencies.

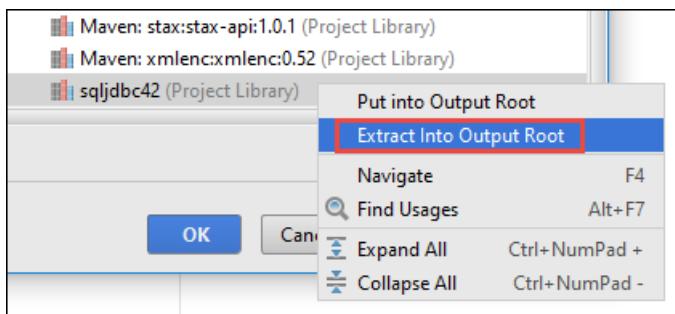


- e. The **Output Layout** tab lists all the jars that are included as part of the Maven project. You can select and delete the ones on which the Scala application has no direct dependency. For the application we are creating here, you can remove all but the last one (**spark-streaming-data-persistence-examples compile output**). Select the jars to delete and then click the **Delete** icon ().

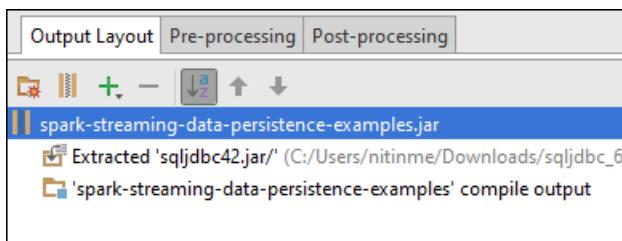


Make sure **Build on make** box is selected, which ensures that the jar is created every time the project is built or updated. Click **Apply**.

- f. In the **Output Layout** tab, right at the bottom of the **Available Elements** box, you have the SQL JDBC jar that you added earlier to the project library. You must add this to the **Output Layout** tab. Right-click the jar file, and then click **Extract Into Output Root**.

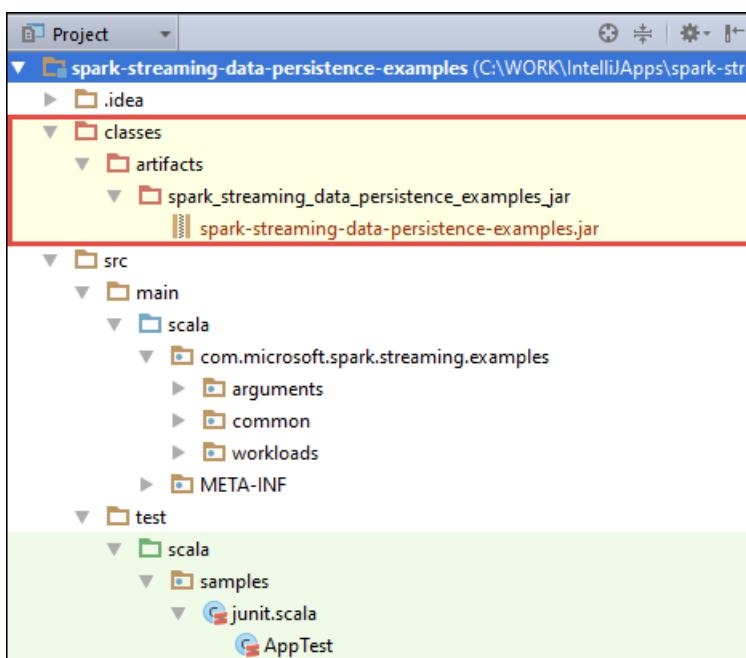


The **Output Layout** tab should now look like this.



In the **Project Structure** dialog box, click **Apply** and then click **OK**.

- From the menu bar, click **Build**, and then click **Make Project**. You can also click **Build Artifacts** to create the jar. The output jar is created under **\classes\artifacts**.



Run the application remotely on a Spark cluster using Livy

In this article you use Livy to run the Apache Spark streaming application remotely on a Spark cluster. For detailed discussion on how to use Livy with HDInsight Spark cluster, see [Submit jobs remotely to an Apache Spark cluster on Azure HDInsight](#). Before you can start running the Spark streaming application, there are a couple of things you should do:

- Start the local standalone application to generate events and sent to Event Hub. Use the following command to do so:

```
java -cp com.microsoft.azure.eventhubs-client-example-0.2.0.jar
com.microsoft.eventhubs.client.example.EventhubsClientDriver --eventhubs-namespace "mysbnamespace" --
eventhubs-name "myeventhub" --policy-name "mysendpolicy" --policy-key "<policy key>" --message-length
32 --thread-count 32 --message-count -1
```

2. Copy the streaming jar (**spark-streaming-data-persistence-examples.jar**) to the Azure Blob storage associated with the cluster. This makes the jar accessible to Livy. You can use [AzCopy](#), a command line utility, to do so. There are a lot of other clients you can use to upload data. You can find more about them at [Upload data for Hadoop jobs in HDInsight](#).
3. Install CURL on the computer where you are running these applications from. We use CURL to invoke the Livy endpoints to run the jobs remotely.

Run the Spark streaming application to receive the events into an Azure Storage Blob as text

Open a command prompt, navigate to the directory where you installed CURL, and run the following command (replace username/password and cluster name):

```
curl -k --user "admin:mypassword1!" -v -H "Content-Type: application/json" -X POST --data
@C:\Temp\inputBlob.txt "https://mysparkcluster.azurehdinsight.net/livy/batches"
```

The parameters in the file **inputBlob.txt** are defined as follows:

```
{
  "file": "wasb:///example/jars/spark-streaming-data-persistence-examples.jar",
  "className": "com.microsoft.spark.streaming.examples.workloads.EventhubsEventCount",
  "args": [
    "--eventhubs-namespace", "mysbnamespace",
    "--eventhubs-name", "myeventhub",
    "--policy-name", "myreceivepolicy",
    "--policy-key", "<put-your-key-here>",
    "--consumer-group", "$default",
    "--partition-count", 10,
    "--batch-interval-in-seconds", 20,
    "--checkpoint-directory", "/EventCheckpoint",
    "--event-count-folder", "/EventCount/EventCount10"
  ],
  "numExecutors": 20,
  "executorMemory": "1G",
  "executorCores": 1,
  "driverMemory": "2G"
}
```

Let us understand what the parameters in the input file are:

- **file** is the path to the application jar file on the Azure storage account associated with the cluster.
- **className** is the name of the class in the jar.
- **args** is the list of arguments required by the class
- **numExecutors** is the number of cores used by Spark to run the streaming application. This should always be at least twice the number of Event Hub partitions.
- **executorMemory**, **executorCores**, **driverMemory** are parameters used to assign required resources to the streaming application.

NOTE

You do not need to create the output folders (EventCheckpoint, EventCount/EventCount10) that are used as parameters. The streaming application creates them for you.

When you run the command, you should see an output like the following:

```
< HTTP/1.1 201 Created
< Content-Type: application/json; charset=UTF-8
< Location: /18
< Server: Microsoft-IIS/8.5
< X-Powered-By: ARR/2.5
< X-Powered-By: ASP.NET
< Date: Tue, 01 Dec 2015 05:39:10 GMT
< Content-Length: 37
<
{"id":1,"state":"starting","log":[]}* Connection #0 to host mysparkcluster.azurehdinsight.net left intact
```

Make a note of the batch ID in the last line of the output (in this example it is '1'). To verify that the application runs successfully, you can look at your Azure storage account associated with the cluster and you should see the

/EventCount/EventCount10 folder created there. This folder should contain blobs that captures the number of events processed within the time period specified for the parameter **batch-interval-in-seconds**.

The Spark streaming application will continue to run until you kill it. To do so, use the following command:

```
curl -k --user "admin:mypassword1!" -v -X DELETE "https://mysparkcluster.azurehdinsight.net/livy/batches/1"
```

Run the applications to receive the events into an Azure Storage Blob as JSON

Open a command prompt, navigate to the directory where you installed CURL, and run the following command (replace username/password and cluster name):

```
curl -k --user "admin:mypassword1!" -v -H "Content-Type: application/json" -X POST --data @C:\Temp\inputJSON.txt "https://mysparkcluster.azurehdinsight.net/livy/batches"
```

The parameters in the file **inputJSON.txt** are defined as follows:

```
{ "file": "wasb:///example/jars/spark-streaming-data-persistence-examples.jar",  
  "className": "com.microsoft.spark.streaming.examples.workloads.EventhubsToAzureBlobAsJSON", "args": ["--  
    eventhubs-namespace", "mysbnamespace", "--eventhubs-name", "myeventhub", "--policy-name", "myreceivepolicy",  
    "--policy-key", "<put-your-key-here>", "--consumer-group", "$default", "--partition-count", 10, "--batch-  
    interval-in-seconds", 20, "--checkpoint-directory", "/EventCheckpoint", "--event-count-folder",  
    "/EventCount/EventCount10", "--event-store-folder", "/EventStore10"], "numExecutors": 20,  
  "executorMemory": "1G", "executorCores": 1, "driverMemory": "2G" }
```

The parameters are similar to what you specified for the text output, in the previous step. Again, you do not need to create the output folders (EventCheckpoint, EventCount/EventCount10) that are used as parameters. The streaming application creates them for you.

After you run the command, you can look at your Azure storage account associated with the cluster and you should see the **/EventStore10** folder created there. Open any file prefixed with **part-** and you should see the events processed in a JSON format.

Run the applications to receive the events into a Hive table

To run the Spark streaming application that streams events into a Hive table you need some additional components. These are:

- datanucleus-api-jdo-3.2.6.jar
- datanucleus-rdbms-3.2.9.jar
- datanucleus-core-3.2.10.jar
- hive-site.xml

The **.jar** files are available on your HDInsight Spark cluster at `/usr/hdp/current/spark-client/lib`. The **hive-site.xml** is available at `/usr/hdp/current/spark-client/conf`.

You can use [WinScp](#) to copy over these files from the cluster to your local computer. You can then use tools to copy these files over to your storage account associated with the cluster. For more information on how to upload files to the storage account, see [Upload data for Hadoop jobs in HDInsight](#).

Once you have copied over the files to your Azure storage account, open a command prompt, navigate to the directory where you installed CURL, and run the following command (replace username/password and cluster name):

```
curl -k --user "admin:mypassword1!" -v -H "Content-Type: application/json" -X POST --data @C:\Temp\inputHive.txt "https://mysparkcluster.azurehdinsight.net/livy/batches"
```

The parameters in the file **inputHive.txt** are defined as follows:

```
{ "file": "wasb:///example/jars/spark-streaming-data-persistence-examples.jar",  
  "className": "com.microsoft.spark.streaming.examples.workloads.EventhubsToHiveTable", "args": [ "--eventhubs-  
    namespace", "mysbnamespace", "--eventhubs-name", "myeventhub", "--policy-name", "myreceivepolicy", "--  
    policy-key", "<put-your-key-here>", "--consumer-group", "$default", "--partition-count", 10, "--batch-  
    interval-in-seconds", 20, "--checkpoint-directory", "/EventCheckpoint", "--event-count-folder",  
    "/EventCount/EventCount10", "--event-hive-table", "EventHiveTable10" ], "jars":  
  ["wasb:///example/jars/datanucleus-api-jdo-3.2.6.jar", "wasb:///example/jars/datanucleus-rdbms-3.2.9.jar",  
   "wasb:///example/jars/datanucleus-core-3.2.10.jar"], "files": ["wasb:///example/jars/hive-site.xml"],  
  "numExecutors": 20, "executorMemory": "1G", "executorCores": 1, "driverMemory": "2G" }
```

The parameters are similar to what you specified for the text output, in the previous steps. Again, you do not need to create the output folders (EventCheckpoint, EventCount/EventCount10) or the output Hive table (EventHiveTable10) that are used as parameters. The streaming application creates them for you. Note that the **jars** and **files** option includes paths to the .jar files and the hive-site.xml that you copied over to the storage account.

To verify that the hive table was successfully created, you can SSH into the cluster and run Hive queries.

Once you are connected using SSH, you can run the following command to verify that the Hive table, **EventHiveTable10**, is created.

```
show tables;
```

You should see an output similar to the following:

```
OK  
eventhivetetable10  
hivesampletable
```

You can also run a SELECT query to view the contents of the table.

```
SELECT * FROM eventhivetetable10 LIMIT 10;
```

You should see an output like the following:

```
ZN90apUSQODDTx7n6Toh6jDbuPngqT4c  
sor2M7xsFwmaRW8W8NDwMneFNMrOVkW1  
o2HcsU735ejSi2bGEcbUSB4btCFmI1lW  
TLuibq4rbj0T9st9eEzIWJwNGtMWYoYS  
HKCpP1WFWAJILwR69MAq863nCWYzDEw6  
Mvx0GQOPYvPR7ezBEPiHYKTKiEhYammQ  
85dRppSBSbZgThLr1s0GMgKqynDUqudr  
5LAwkNqorLj3ZN9a2mfWr9rZqeXKN4pF  
ulf9wSFNjD7BZXCyunozeov9QpEIYmJ  
vWzM3nvOja8DhYcn0n5eTf0tZ966pa  
Time taken: 4.434 seconds, Fetched: 10 row(s)
```

Run the applications to receive the events into an Azure SQL database table

Before running this step, make sure you have an Azure SQL database created. For instructions, see [Create a SQL database in minutes](#). To complete this section, you need values for database name, database server name, and the database administrator credentials as parameters. You do not need to create the database table though. The Spark streaming application creates that for you.

Open a command prompt, navigate to the directory where you installed CURL, and run the following command:

```
curl -k --user "admin:mypassword1!" -v -H "Content-Type: application/json" -X POST --data @C:\Temp\inputSQL.txt "https://mysparkcluster.azurehdinsight.net/livy/batches"
```

The parameters in the file **inputSQL.txt** are defined as follows:

```
{ "file":"wasb:///example/jars/spark-streaming-data-persistence-examples.jar",  
"className":"com.microsoft.spark.streaming.examples.workloads.EventhubsToAzureSQLTable", "args":["--  
eventhubs-namespace", "mysbnamespace", "--eventhubs-name", "myeventhub", "--policy-name", "myreceivepolicy",  
"--policy-key", "<put-your-key-here>", "--consumer-group", "$default", "--partition-count", 10, "--batch-  
interval-in-seconds", 20, "--checkpoint-directory", "/EventCheckpoint", "--event-count-folder",  
"/EventCount/EventCount10", "--sql-server-fqdn", "<database-server-name>.database.windows.net", "--sql-  
database-name", "mysparkdatabase", "--database-username", "sparkdbadmin", "--database-password", "<put-  
password-here>", "--event-sql-table", "EventContent" ], "numExecutors":20, "executorMemory":1G,  
"executorCores":1, "driverMemory":2G }
```

To verify that the application runs successfully, you can connect to the Azure SQL database using SQL Server Management Studio. For instructions on how to do that, see [Connect to SQL Database with SQL Server Management Studio](#). Once you are connected to the database, you can navigate to the **EventContent** table that was created by the streaming application. You can run a quick query to get the data from the table. Run the following query:

```
SELECT * FROM EventCount
```

You should see output similar to the following:

```
00046b0f-2552-4980-9c3f-8bba5647c8ee  
000b7530-12f9-4081-8e19-90acd26f9c0c  
000bc521-9c1b-4a42-ab08-dc1893b83f3b  
00123a2a-e00d-496a-9104-108920955718  
0017c68f-7a4e-452d-97ad-5cb1fe5ba81b  
001KsmqL2gfu5ZcuQuTqTxQVvyGCqPp9  
001vIZg0Stka4DXtud0e3tX7XbfMnZrN  
00220586-3e1a-4d2d-a89b-05c5892e541a  
0029e309-9e54-4e1b-84be-cd04e6fce5ec  
003333cf-874f-4045-9da3-9f98c2b4ea49  
0043c07e-8d73-420a-9af7-1fc94575356  
004a11a9-0c2c-4bc0-a7d5-2e0ebd947ab9
```

See also

- [Overview: Apache Spark on Azure HDInsight](#)
- [Design of Receiver-based Connection and Direct DStream](#)

Scenarios

- [Spark with BI: Perform interactive data analysis using Spark in HDInsight with BI tools](#)
- [Spark with Machine Learning: Use Spark in HDInsight for analyzing building temperature using HVAC data](#)
- [Spark with Machine Learning: Use Spark in HDInsight to predict food inspection results](#)
- [Website log analysis using Spark in HDInsight](#)

Create and run applications

- [Create a standalone application using Scala](#)
- [Run jobs remotely on a Spark cluster using Livy](#)

Tools and extensions

- Use HDInsight Tools Plugin for IntelliJ IDEA to create and submit Spark Scala applications
- Use HDInsight Tools Plugin for IntelliJ IDEA to debug Spark applications remotely
- Use Zeppelin notebooks with a Spark cluster on HDInsight
- Kernels available for Jupyter notebook in Spark cluster for HDInsight
- Use external packages with Jupyter notebooks
- Install Jupyter on your computer and connect to an HDInsight Spark cluster

Manage resources

- Manage resources for the Apache Spark cluster in Azure HDInsight
- Track and debug jobs running on an Apache Spark cluster in HDInsight

Use Spark Structured Streaming with Kafka (preview) on HDInsight

8/16/2017 • 4 min to read • [Edit Online](#)

Learn how to use Spark Structured Streaming to read data from Apache Kafka on Azure HDInsight.

Spark structured streaming is a stream processing engine built on Spark SQL. It allows you to express streaming computations the same as batch computation on static data. For more information on Structured Streaming, see the [Structured Streaming Programming Guide \[Alpha\]](#) at Apache.org.

IMPORTANT

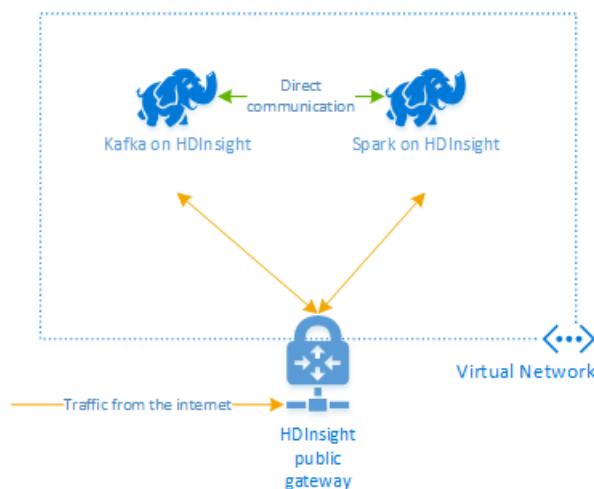
This example used Spark 2.1 on HDInsight 3.6. Structured Streaming is considered **alpha** on Spark 2.1.

The steps in this document create an Azure resource group that contains both a Spark on HDInsight and a Kafka on HDInsight cluster. These clusters are both located within an Azure Virtual Network, which allows the Spark cluster to directly communicate with the Kafka cluster.

When you are done with the steps in this document, remember to delete the clusters to avoid excess charges.

Create the clusters

Apache Kafka on HDInsight does not provide access to the Kafka brokers over the public internet. Anything that talks to Kafka must be in the same Azure virtual network as the nodes in the Kafka cluster. For this example, both the Kafka and Spark clusters are located in an Azure virtual network. The following diagram shows how communication flows between the clusters:



NOTE

The Kafka service is limited to communication within the virtual network. Other services on the cluster, such as SSH and Ambari, can be accessed over the internet. For more information on the public ports available with HDInsight, see [Ports and URIs used by HDInsight](#).

While you can create an Azure virtual network, Kafka, and Spark clusters manually, it's easier to use an Azure Resource Manager template. Use the following steps to deploy an Azure virtual network, Kafka, and Spark clusters to your Azure subscription.

1. Use the following button to sign in to Azure and open the template in the Azure portal.

[Deploy to Azure >](#)

The Azure Resource Manager template is located at

<https://hditutorialdata.blob.core.windows.net/armtemplates/create-linux-based-kafka-spark-cluster-in-vnet-v4.1.json>.

This template creates the following resources:

- A Kafka on HDInsight 3.5 cluster.
- A Spark on HDInsight 3.6 cluster.
- An Azure Virtual Network, which contains the HDInsight clusters.

IMPORTANT

The structured streaming notebook used in this example requires Spark on HDInsight 3.6. If you use an earlier version of Spark on HDInsight, you receive errors when using the notebook.

2. Use the following information to populate the entries on the **Custom deployment** blade:

TEMPLATE

 Customized template
4 resources

 Edit template

 Edit parameters

 Learn more

BASICS

* Subscription

* Resource group  Create new Use existing

* Location

SETTINGS

* Base Cluster Name  

Cluster Login User Name 

* Cluster Login Password  

Ssh User Name 

* Ssh Password  

TERMS AND CONDITIONS

[Azure Marketplace Terms](#) | [Azure Marketplace](#)

By clicking "Purchase," I (a) agree to the applicable legal terms associated with the offering; (b) authorize Microsoft to charge or bill my current payment method for the fees associated the offering(s), including applicable taxes, with the same billing frequency as my Azure subscription, until I discontinue use of the offering(s); and (c) agree that, if the deployment involves 3rd party offerings, Microsoft may share my contact information and other details of such deployment with the publisher of that offering.

Microsoft assumes no responsibility for any actions performed by third-party templates and does not provide rights for third-

I agree to the terms and conditions stated above

Pin to dashboard

Purchase

- **Resource group:** Create a group or select an existing one. This group contains the HDInsight cluster.
- **Location:** Select a location geographically close to you.
- **Base Cluster Name:** This value is used as the base name for the Spark and Kafka clusters. For example, entering **hdi** creates a Spark cluster named **spark-hdi_** and a Kafka cluster named **kafka-hdi**.
- **Cluster Login User Name:** The admin user name for the Spark and Kafka clusters.
- **Cluster Login Password:** The admin user password for the Spark and Kafka clusters.
- **SSH User Name:** The SSH user to create for the Spark and Kafka clusters.
- **SSH Password:** The password for the SSH user for the Spark and Kafka clusters.

3. Read the **Terms and Conditions**, and then select **I agree to the terms and conditions stated above**.
4. Finally, check **Pin to dashboard** and then select **Purchase**. It takes about 20 minutes to create the clusters.

Once the resources have been created, you are redirected to the resource group blade.

NAME	TYPE	LOCATION
kafka-hdi	HDInsight Clu...	North Central US
spark-hdi	HDInsight Clu...	North Central US
gateway-199eb825822c4b6d83bfee52fa1	Load balancer	North Central US
gateway-b546fc43f4d54289a39728c2398	Load balancer	North Central US
headnode-199eb825822c4b6d83bfee52fa	Load balancer	North Central US
headnode-b546fc43f4d54289a39728c239	Load balancer	North Central US
nic-gateway-0-b546fc43f4d54289a39728	Network inter...	North Central US
nic-gateway-1-199eb825822c4b6d83bfee	Network inter...	North Central US
nic-gateway-2-b546fc43f4d54289a39728	Network inter...	North Central US

IMPORTANT

Notice that the names of the HDInsight clusters are **spark-BASENAME** and **kafka-BASENAME**, where BASENAME is the name you provided to the template. You use these names in later steps when connecting to the clusters.

Get the Kafka brokers

The code in this example connects to the Kafka broker hosts in the Kafka cluster. To find the Kafka broker hosts, use the following PowerShell or Bash example:

```
$creds = Get-Credential -UserName "admin" -Message "Enter the HDInsight login"
$clusterName = Read-Host -Prompt "Enter the Kafka cluster name"
$resp = Invoke-WebRequest -Uri
"https://$clusterName.azurehdinsight.net/api/v1/clusters/$clusterName/services/KAFKA/components/KAFKA_BROKER" ` 
    -Credential $creds
$respObj = ConvertFrom-Json $resp.Content
$brokerHosts = $respObj.host_components.HostRoles.host_name
($brokerHosts -join ":9092,") + ":9092"
```

```
curl -u admin:$PASSWORD -G
"https://$CLUSTERNAME.azurehdinsight.net/api/v1/clusters/$CLUSTERNAME/services/KAFKA/components/KAFKA_BROKER" | 
jq -r '["\(.host_components[].HostRoles.host_name):9092"] | join(",")'
```

NOTE

This example expects `$PASSWORD` to contain the password for the cluster login, and `$CLUSTERNAME` to contain the name of the Kafka cluster.

This example uses the `jq` utility to parse data out of the JSON document.

The output is similar to the following text:

```
wn0-kafka.0owcb1lr5hze3hxdja3mq1rhhe.ex.internal.cloudapp.net:9092,wn1-
kafka.0owcb1lr5hze3hxdja3mq1rhhe.ex.internal.cloudapp.net:9092,wn2-
kafka.0owcb1lr5hze3hxdja3mq1rhhe.ex.internal.cloudapp.net:9092,wn3-
kafka.0owcb1lr5hze3hxdja3mq1rhhe.ex.internal.cloudapp.net:9092
```

Save this information, as it is used in the following sections of this document.

Get the notebooks

The code for the example described in this document is available at <https://github.com/Azure-Samples/hdinsight-spark-kafka-structured-streaming>.

Upload the notebooks

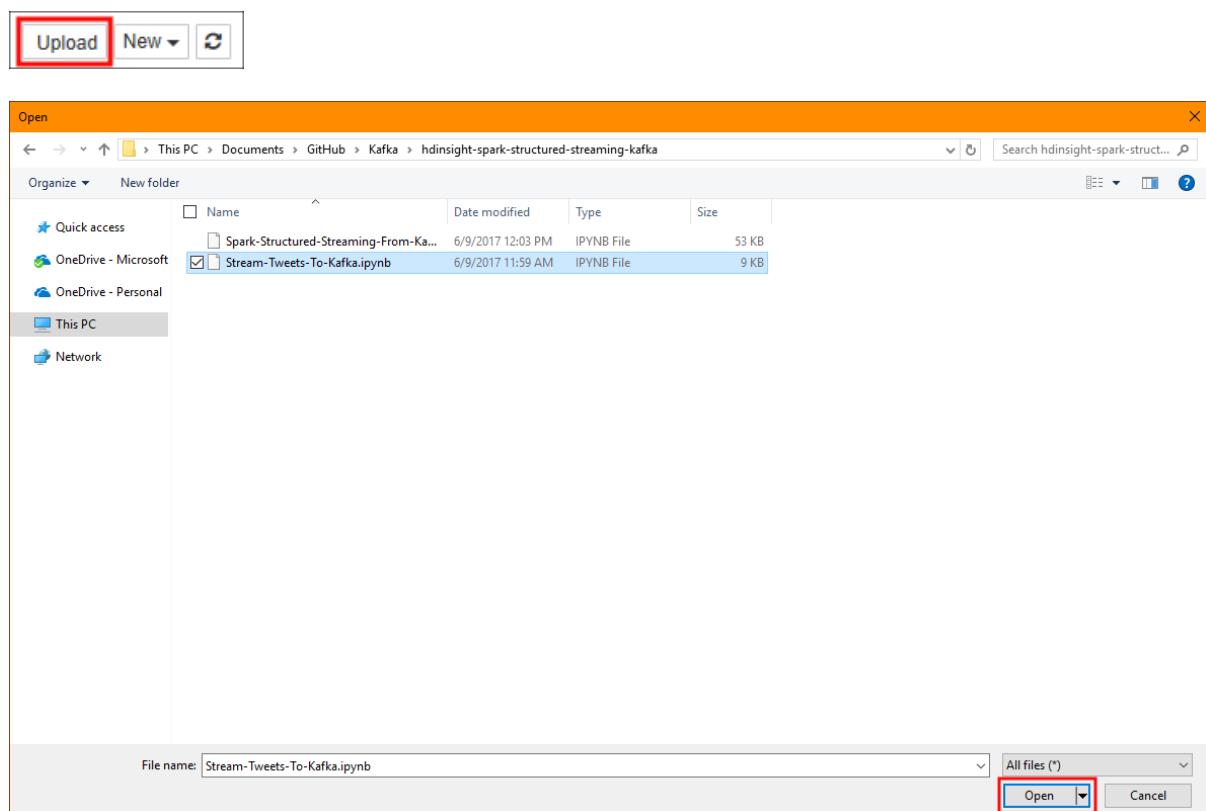
Use the following steps to upload the notebooks from the project to your Spark on HDInsight cluster:

1. In your web browser, connect to the Jupyter notebook on your Spark cluster. In the following URL, replace `CLUSTERNAME` with the name of your Kafka cluster:

```
https://CLUSTERNAME.azurehdinsight.net/jupyter
```

When prompted, enter the cluster login (admin) and password used when you created the cluster.

2. From the upper right side of the page, use the **Upload** button to upload the **Stream-Tweets-To-Kafka.ipynb** file to the cluster. Select **Open** to start the upload.



3. Find the **Stream-Tweets-To_Kafka.ipynb** entry in the list of notebooks, and select **Upload** button beside it.



4. Repeat steps 1-3 to load the **Spark-Structured-Streaming-From-Kafka.ipynb** notebook.

Load tweets into Kafka

Once the files have been uploaded, select the **Stream-Tweets-To_Kafka.ipynb** entry to open the notebook. Follow the steps in the notebook to load tweets into Kafka.

Process tweets using Spark Structured Streaming

From the Jupyter Notebook home page, select the **Spark-Structured-Streaming-From-Kafka.ipynb** entry. Follow the steps in the notebook to load tweets from Kafka using Spark Structured Streaming.

Next steps

Now that you have learned how to use Spark Structured Streaming, see the following documents to learn more about working with Spark and Kafka:

- [How to use Spark streaming \(DStream\) with Kafka.](#)
- [Start with Jupyter Notebook and Spark on HDInsight](#)

Using Spark Structured Streaming on HDInsight to process events from Event Hubs

8/15/2017 • 9 min to read • [Edit Online](#)

In this article you will learn to process real-time telemetry using Spark Structured Streaming. To accomplish this you will perform the following high-level steps:

1. Provision an HDInsight cluster with Spark 2.1.
2. Provision an Event Hubs instance.
3. Compile and run on your local workstation a sample Event Producer application that generates events to send to Event Hubs.
4. Use the [Spark Shell](#) to define and run a simple Spark Structured Streaming application.

Prerequisites

- An Azure subscription. See [Get Azure free trial](#).

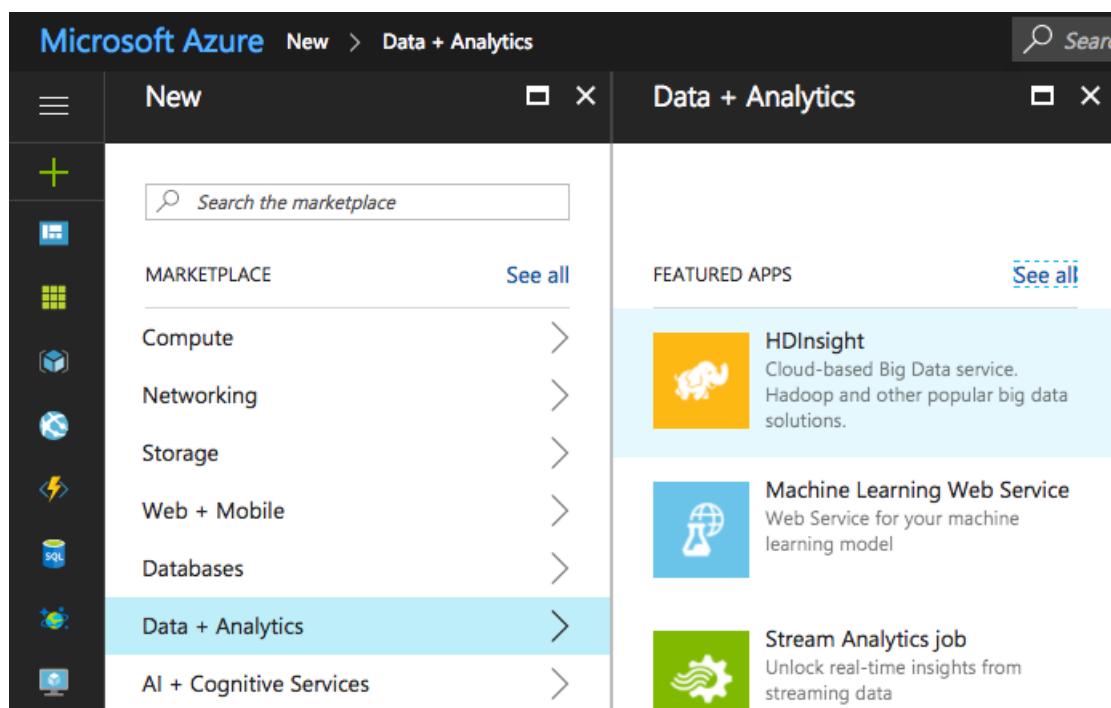
Make sure you have the following installed on the computer where you run Event Producer application:

- Oracle Java Development kit. You can install it from [here](#).
- Apache Maven. You can download it from [here](#). Instructions to install Maven are available [here](#).

Provision an HDInsight cluster with Spark 2.1

To provision your HDInsight Cluster, follow these steps:

1. Sign in to the [Azure Portal](#).
2. Select + New, Data + Analytics, HDInsight.



3. On the basics blade, provide a unique name for your cluster.

4. Select the Subscription in which to create the cluster.
5. Select Cluster type.
6. On the Cluster configuration blade, select Cluster type of Spark and Version Spark 2.1.0 (HDI 3.6). Select select to apply the cluster type.

Cluster configuration

* Cluster type <small>i</small>	* Operating system	* Version
Spark	Linux	Spark 2.1.0 (HDI 3.6)
* Cluster tier <small>i</small>		
STANDARD PREMIUM		

Spark : Fast data analytics and cluster computing using in-memory processing.

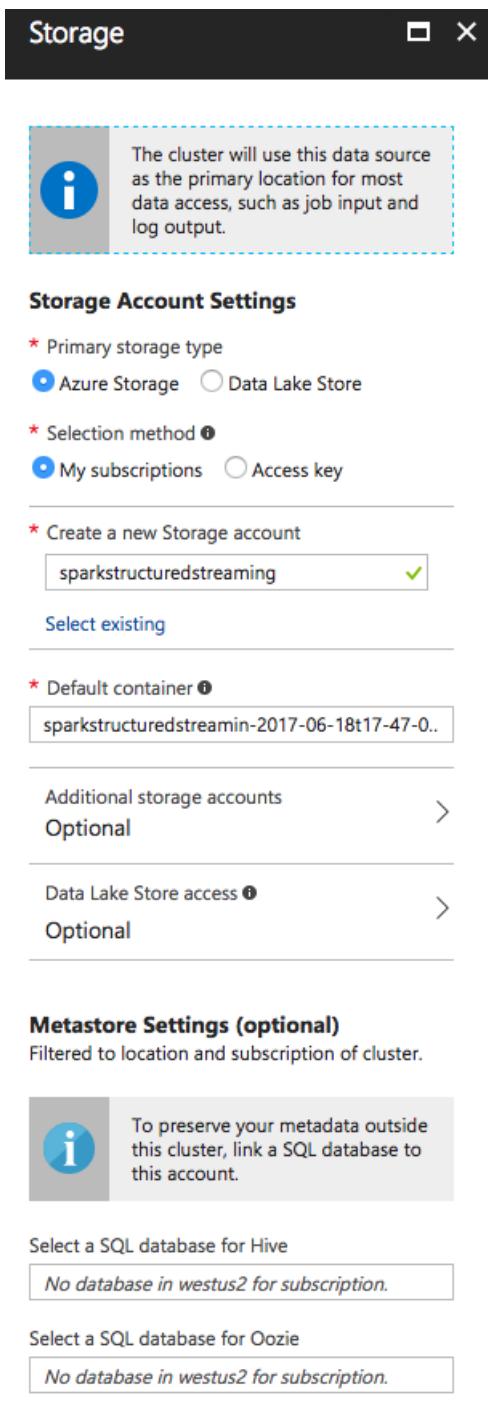
7. Leave the Cluster login username as "admin".
8. Provide a password for the Cluster login password. This will be the password used for both admin and the sshuser accounts.
9. Leave the Secure Shell (SSH) username as "sshuser".
10. Create a new Resource Group or select an existing one as your prefer.
11. Choose a Location in which to deploy your cluster.

Basics □ X

* Cluster name	<input type="text" value=""/>	✓
	.azurehdinsight.net	
* Subscription	<input type="text"/>	
<hr/>		
* Cluster type <small>i</small>	>	
Spark 2.1 on Linux (HDI 3.6)		
<hr/>		
* Cluster login username <small>i</small>	<input type="text" value="admin"/>	
* Cluster login password <small>i</small>	<input type="password" value="*****"/>	
<hr/>		
Secure Shell (SSH) username <small>i</small>	<input type="text" value="sshuser"/>	
<hr/>		
<input checked="" type="checkbox"/> Use same password as cluster login <small>i</small>		
<hr/>		
* Resource group <small>i</small>		
<input checked="" type="radio"/> Create new	<input type="radio"/> Use existing	
<hr/>		
* Location	<input type="text" value="West US 2"/>	
<hr/>		
	Click here to view cores usage.	

12. Select Next.
13. On the Storage blade, leave Primary Storage type set to Azure Storage and Selection method set to My subscriptions.

14. Under Select a storage account, select Create new and provide a name for the new Storage Account. This account will act as the default storage for your cluster.
15. Enter a unique name for the new Storage Account.
16. Leave the remaining settings at their defaults and select Next.



17. On the Cluster summary blade, select Create.

It will take about 20 minutes to provision your cluster. Continue on to the next section while the cluster provisions.

Provision an Event Hubs namespace

In this task you will provision the Event Hubs namespace that will ultimately contain your Event Hubs instance.

1. Continue in the [Azure Portal](#).
2. Select + New, Internet of Things, Event Hubs.

New X

Internet of Things X

Search the marketplace

MARKETPLACE	See all	FEATURED APPS	See all
Compute	>	 IoT Hub Connect, monitor and manage IoT devices	
Networking	>	 Event Hubs Cloud-scale telemetry ingestion from websites, apps, and devices.	
Storage	>		
Web + Mobile	>		
Databases	>		
Data + Analytics	>		
AI + Cognitive Services	>		
Internet of Things	>	 Time Series Insights (preview) Azure Time Series Insights is a fully managed analytics, storage, and visualization service that makes it	
Enterprise Integration	>	 Stream Analytics job Unlock real-time insights from	

3. On the Create namespace blade, provide a unique name for your Event Hubs namespace.
4. Leave the Pricing tier at Standard.
5. Choose a Subscription and Resource Group as appropriate.
6. For the Location, choose the same Location as you used for your HDInsight cluster.

Create namespace X

Event Hubs - PREVIEW

* Name
 ✓
.servicebus.windows.net

* Pricing tier
 Standard >

* Subscription
 ▼

* Resource group ?
 Create new Use existing
Spark ▼

* Location
West US 2 ▼

Throughput Units ?
 1

Enable auto-inflate?

7. Select Create.

Provision an Event Hub

In this task you will provision the Event Hub instance that will receive events from a sample application that generates random events, and that you will use as the source for events to process using Spark Structured Streaming.

1. Once your Event Hubs namespace has provisioned, navigate to it in the [Azure Portal](#).
2. At the top of the blade, select + Event Hub.



3. In the Create Event Hub blade, enter the name "hub1" for your Event Hub.
4. Leave the remaining settings at their defaults. Note that your Event Hub will have 2 partitions (as set in Partition Count).

Create Event Hub
hdiz-docs-eventhubs - PREVIEW

* Name
hub1

Partition Count ⓘ
2

Message Retention ⓘ
1

Archive
On Off

Time window (minutes)
5

Size window (MB)
300

* Container
Nothing selected >

Storage Account
Nothing selected

Sample Archive file name formats
[Namespace]/[EventHub]/[PartitionId]/[Year]/[Month]/[Day]/[Hour]/[Minute]/[Second]

Archive file name format ⓘ
[Empty input field]

5. Select Create.
6. On your Event Hub namespace blade, select Shared access policies from the side menu.

SETTINGS

Shared access policies

7. In the list of Shared Access Policies, select the RootManageSharedAccessKey.

POLICY	CLAIMS
RootManageSharedAccessKey	Manage, Send, Listen

8. Copy the value under Primary Key and paste it into a temporary text file, this value is your Policy Key. Also, take note that the Policy Name is "RootManageSharedAccessKey".

PRIMARY KEY 0P1Q17Wd1rdLP1OZAYn6dD2S13Bb3nF3h2XZD9hvyy...

9. Close the Policy blade and select Properties from the side menu.

NAME
hdiz-docs-eventhubs

LOCATION
West US

SUBSCRIPTION ID
[REDACTED]

CREATED AT
6/17/2017, 2:06:20 PM

UPDATED AT
6/17/2017, 2:06:43 PM

STATUS
Active

PROVISIONING STATE
Succeeded

10. The value under Name is the your namespace, take note of this value along with your Policy Key and Name. Also, take note that the name of your Event Hub itself is "hub1".

Build, Configure and Run the Event Producer

In this task you will clone a sample application that creates random events and sends them to a configured Event Hub. This sample application is available in GitHub at <https://github.com/hdinsight/eventhubs-sample-event-producer>.

1. Make sure you have the git tools installed. You can download from [GIT Downloads](#).
2. Open a command prompt or terminal shell and run the following command from the directory of your choice to clone the project.

```
git clone https://github.com/hdinsight/eventhubs-sample-event-producer.git eventhubs-client-sample
```

- This will create a new folder called eventhubs-client-sample. Within the shell or command prompt navigate into this folder.
- Run Maven to build the application by using the following command.

```
mvn package
```

- Within the shell or command prompt, navigate into the target directory that is created and contains the file `com-microsoft-azure-eventhubs-client-example-0.2.0.jar`.
- Next, you will need to build up the command line to run the Event Producer against your Event Hub. Do this by replacing the values in the command as follows:

```
java -cp com-microsoft-azure-eventhubs-client-example-0.2.0.jar
com.microsoft.azure.eventhubs.client.example.EventhubsClientDriver --eventhubs-namespace "<namespace>" -
--eventhubs-name "hub1" --policy-name "RootManageSharedAccessKey" --policy-key "<policyKey>" --message-
length 32 --thread-count 1 --message-count -1
```

- For example, a complete command would look similar to the following:

```
java -cp com-microsoft-azure-eventhubs-client-example-0.2.0.jar
com.microsoft.azure.eventhubs.client.example.EventhubsClientDriver --eventhubs-namespace
"sparkstreaming" --eventhubs-name "hub1" --policy-name "RootManageSharedAccessKey" --policy-key
"2P1Q17Wd1rdLP10ZQYn6dD2S13Bb3nF3h2XZD9hvyyU" --message-length 32 --thread-count 1 --message-count -1
```

- The command will start up and if your configuration is correct in a few moments you will see output related to the events it is sending to your Event Hub, similar to the following:

```
Map('policyKey' -> 2P1Q17Wd1rdLP10ZQYn6dD2S13Bb3nF3h2XZD9hvyyU, 'eventhubsName' -> hub1, 'policyName' ->
RootManageSharedAccessKey, 'eventhubsNamespace' -> sparkstreaming, 'messageCount' -> -1, 'messageLength' ->
32, 'threadCount' -> 1)
Events per thread: -1 (-1 for unlimited)
10 > Sun Jun 18 11:32:58 PDT 2017 > 1024 > 1024 > Sending event: ZZ93958saG5BUKbvUI9wHVmpuA2TrebS
10 > Sun Jun 18 11:33:46 PDT 2017 > 2048 > 2048 > Sending event: RQorGRbTPp6U2wYzRSnZU1WEltRvTZ7R
10 > Sun Jun 18 11:34:33 PDT 2017 > 3072 > 3072 > Sending event: 36Eoy2r8ptqibdlfCYSGgXe6ct4Ay0X3
10 > Sun Jun 18 11:35:19 PDT 2017 > 4096 > 4096 > Sending event: bPZma9V0CqOn6Hj9bhrrJT0bX2rbPSn3
10 > Sun Jun 18 11:36:06 PDT 2017 > 5120 > 5120 > Sending event: H2TVD77HNTVyGsVcj76g0daVnYxN4Sqs
```

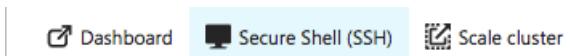
- Leave the Event Producer running while you continue on to the steps.

Run Spark Shell on your HDInsight cluster

In this task, you will SSH into the head node of your HDInsight cluster, launch the Spark Shell and run a Spark Streaming application that will retrieve and process the events from Event Hubs.

By this point your HDInsight cluster should be ready. If not, you will need to wait until it finishes provisioning. Once it is ready, proceed with the following steps.

- Navigate to your deployed HDInsight cluster in the [Azure Portal](#).
- Select Secure Shell.



- Follow the instructions displayed for connecting to your cluster via SSH from your local environment. In general, this will mean running SSH as follows:

```
ssh sshuser@<yourclustername>-ssh.azurehdinsight.net
```

4. Complete the login by providing the password you supplied when provisioning the cluster.
5. The application you will build requires the Spark Streaming Event Hubs package. To run the Spark Shell so that it automatically retrieves this dependency from [Maven Central](#), be sure to supply the packages switch with the Maven coordinates as follows:

```
spark-shell --packages "com.microsoft.azure:spark-streaming-eventhubs_2.11:2.1.0"
```

6. Once the Spark Shell is finished loading, you should see:

```
Welcome to  
____/_____\ \ /____/\ /  
_ \ \ / _ \ \ /` / \ / ' /  
/___/ .__/\_,/_/ /_/ \_\ version 2.1.0.2.6.0.10-29  
/_/  
  
Using Scala version 2.11.8 (OpenJDK 64-Bit Server VM, Java 1.8.0_131)  
Type in expressions to have them evaluated.  
Type :help for more information.  
  
scala>
```

7. Copy the following code snippet into a text editor and modify it so it has the Policy Key and Namespace set as appropriate for your Event Hub.

```
val eventhubParameters = Map[String, String] (  
    "eventhubs.policyname" -> "RootManageSharedAccessKey",  
    "eventhubs.policykey" -> "<policyKey>",  
    "eventhubs.namespace" -> "<namespace>",  
    "eventhubs.name" -> "hub1",  
    "eventhubs.partition.count" -> "2",  
    "eventhubs.consumergroup" -> "$Default",  
    "eventhubs.progressTrackingDir" -> "/eventhubs/progress",  
    "eventhubs.sql.containsProperties" -> "true"  
)
```

8. Paste the modified snippet into the waiting scala> prompt and press return. You should see output similar to:

```
scala> val eventhubParameters = Map[String, String] (  
|     "eventhubs.policyname" -> "RootManageSharedAccessKey",  
|     "eventhubs.policykey" -> "2P1Q17Wd1rdLP10ZQYn6dD2S13Bb3nF3h2XZD9hvyyU",  
|     "eventhubs.namespace" -> "sparkstreaming",  
|     "eventhubs.name" -> "hub1",  
|     "eventhubs.partition.count" -> "2",  
|     "eventhubs.consumergroup" -> "$Default",  
|     "eventhubs.progressTrackingDir" -> "/eventhubs/progress",  
|     "eventhubs.sql.containsProperties" -> "true"  
| )  
eventhubParameters: scala.collection.immutable.Map[String, String] =  
Map(eventhubs.sql.containsProperties -> true, eventhubs.name -> hub1, eventhubs.consumergroup ->  
$Default, eventhubs.partition.count -> 2, eventhubs.progressTrackingDir -> /eventhubs/progress,  
eventhubs.policykey -> 2P1Q17Wd1rdLP10ZQYn6dD2S13Bb3nF3h2XZD9hvyyU, eventhubs.namespace -> hdiz-docs-  
eventhubs, eventhubs.policyname -> RootManageSharedAccessKey)
```

9. Next, you will begin to author a Spark Structured Streaming query by specifying the source. Paste the following into Spark Shell and press return.

```
val inputStream = spark.readStream.  
format("eventhubs").  
options(eventhubParameters).  
load()
```

10. You should see output similar to:

```
inputStream: org.apache.spark.sql.DataFrame = [body: binary, offset: bigint ... 5 more fields]
```

11. Next, author the query so that it writes its output to the Console. Do this by pasting the following into Spark Shell and pressing return.

```
val streamingQuery1 = inputStream.writeStream.  
outputMode("append").  
format("console").start().awaitTermination()
```

12. You should see some batches start with output similar to the following

```
-----  
Batch: 0  
-----  
[Stage 0:> (0 + 2) / 2]
```

13. This will be followed by the output results of the processing of each microbatch of events.

```
-----  
Batch: 0  
-----  
17/06/18 18:57:39 WARN TaskSetManager: Stage 1 contains a task of very large size (419 KB). The maximum  
recommended task size is 100 KB.  
+-----+-----+-----+-----+-----+-----+  
| body|offset|seqNumber|enqueuedTime|publisher|partitionKey|properties|  
+-----+-----+-----+-----+-----+-----+  
|[7B 22 74 65 6D 7...| 0| 0| 1497734887| null| null| Map()  
|[7B 22 74 65 6D 7...| 112| 1| 1497734887| null| null| Map()  
|[7B 22 74 65 6D 7...| 224| 2| 1497734887| null| null| Map()  
|[7B 22 74 65 6D 7...| 336| 3| 1497734887| null| null| Map()  
|[7B 22 74 65 6D 7...| 448| 4| 1497734887| null| null| Map()  
|[7B 22 74 65 6D 7...| 560| 5| 1497734887| null| null| Map()  
|[7B 22 74 65 6D 7...| 672| 6| 1497734887| null| null| Map()  
|[7B 22 74 65 6D 7...| 784| 7| 1497734888| null| null| Map()  
|[7B 22 74 65 6D 7...| 896| 8| 1497734888| null| null| Map()  
|[7B 22 74 65 6D 7...| 1008| 9| 1497734888| null| null| Map()  
|[7B 22 74 65 6D 7...| 1120| 10| 1497734888| null| null| Map()  
|[7B 22 74 65 6D 7...| 1232| 11| 1497734888| null| null| Map()  
|[7B 22 74 65 6D 7...| 1344| 12| 1497734888| null| null| Map()  
|[7B 22 74 65 6D 7...| 1456| 13| 1497734888| null| null| Map()  
|[7B 22 74 65 6D 7...| 1568| 14| 1497734888| null| null| Map()  
|[7B 22 74 65 6D 7...| 1680| 15| 1497734888| null| null| Map()  
|[7B 22 74 65 6D 7...| 1792| 16| 1497734888| null| null| Map()  
|[7B 22 74 65 6D 7...| 1904| 17| 1497734888| null| null| Map()  
|[7B 22 74 65 6D 7...| 2016| 18| 1497734889| null| null| Map()  
|[7B 22 74 65 6D 7...| 2128| 19| 1497734889| null| null| Map()  
+-----+-----+-----+-----+-----+  
only showing top 20 rows
```

14. As new events arrive from the Event Producer, they will be processed by this Structured Streaming query.

15. Be sure to delete your HDInsight cluster when you are finished running this sample.

See also

This article showed how to author a Spark Streaming application that processed events from Event Hubs.

- For more information on Spark Structured Streaming in HDInsight, see [Spark Structured Streaming Overview](#)

Creating highly available Spark Streaming jobs in YARN

8/16/2017 • 12 min to read • [Edit Online](#)

Understanding Spark Streaming Jobs



Spark Streaming enables you to implement scalable, high-throughput, fault-tolerant applications for the processing of data streams. Running Spark Streaming applications on a HDInsight Spark cluster, you can connect them to process data from a variety of sources such as Azure Event Hubs, Azure IoT Hub, Kafka, Flume, Twitter, ZeroMQ, raw TCP sockets or even by monitoring the HDFS filesystem for changes.

Spark Streaming creates long-running jobs during which you are able to apply transformations to the data and then push out the results to filesystems, databases, dashboards and the console. It uses micro-batches to processes data, by waiting to collect a time-defined batch of events (usually configred in the range of less than a second to a few seconds), before it sends the batch of events on for processing. It supports fault tolerance with the guarantee that any given event would be processed exactly once, even in the face of a node failure.

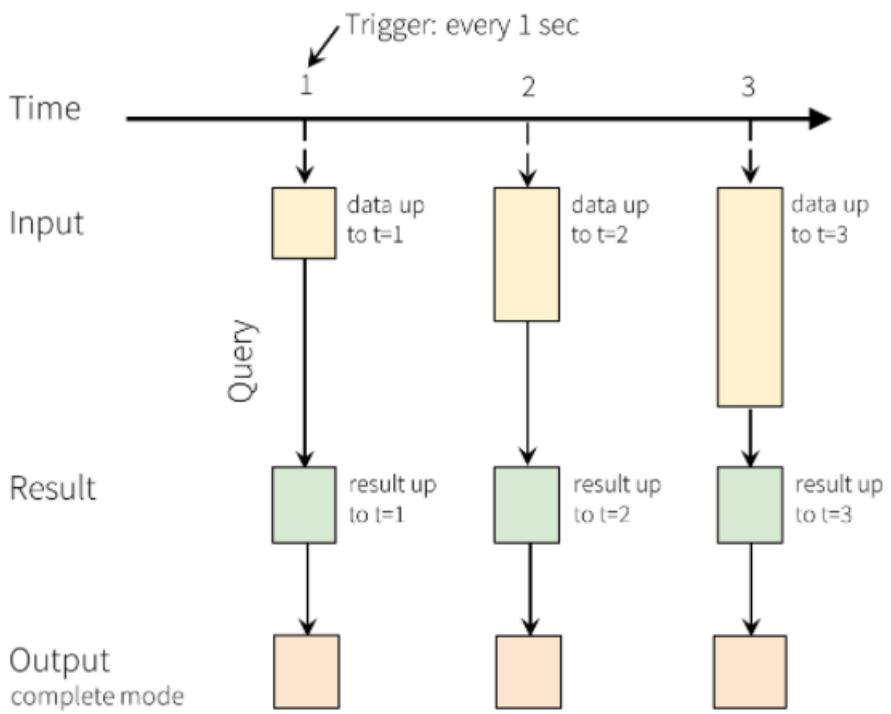
Understanding DStreams



Spark Streaming represents a continuous stream of data using a discretized stream or DStream. This DStream can be created from input sources like Event Hubs or Kafka, or by applying transformation on another DStream. When an event arrives at your Spark Streaming application, this event is stored in a reliable way- it is replicated so that multiple nodes have a copy of it. This ensures that the failure of any single node will not result in the loss of your event.

Spark core uses RDDs (resilient distributed datasets). RDDs distribute data across multiple nodes in the cluster, where each node generally maintains its data completely in-memory for best performance. Each RDD represents events collected over some user defined timeframe called the batch interval. Every time the batch interval elapses a new RDD is produced that contains all the data in the interval of time that just completed. This continuous set of RDDs are collected into a DStream. A Spark Streaming application processes micro-batches containing events, and ultimately acts on the data stored in the RDD each batch contains.

Understanding Spark Structured Streaming Jobs



Spark Structured Streaming was introduced in Spark 2.0. It is a an analytic engine for streaming structured data. Spark Structured Streaming shares the same set of APIs with the SparkSQL batching engine. As with Spark Streaming, Spark Structured Streaming provides the ability to run computation over continuously arriving data and uses a micro-batches streaming model. Spark Structured Streaming represents a stream of data as an unbounded Input Table (heightwise) - in other words the table continues to grow as new data arrives. This Input Table is continuously processed by a long running query, and the results flushed out to an Output Table.

In Structured Streaming data arrives at the system and is immediately ingested into the Input Table. You write queries that perform operations against this Input Table. The query output yields another table, called the Results Table. The Results Table contains results of your query from which you draw any data you would send to an external datastore (such a relational database). The timing of when data is processed from the Input Table is controlled by the trigger interval. By default Structured Streaming tries to process the data as soon as it arrives. However, you can also configure the trigger to run on a longer interval, so that the streaming data is processed according to time-based batches. The output mode controls the data in the Results Tables. This data may be completely refreshed everytime there is new data so that it includes all of the output data since the streaming query began (called `complete mode`), or it may only contain just the data that is new since the last time the query was processed (called `append mode`).

Creating Fault Tolerant Spark Streaming Jobs

In order to be able to create a highly-available environment for your Spark Streaming Jobs, you must first start by architecting your individual jobs for recovery in the event of failure. This is also called creating fault-tolerant jobs. This section covers relavent techniques. RDDs have a number of properties that aid the ability to create highly-available and fault tolerant Spark Streaming Jobs. These properties are listed below:

- Batches of input data stored in RDDs as a DStream are autpmatically replicated in memory for fault-tolerance
- Data lost due to worker failure can be recomputed from replicated input data on different workers, as long as those worker nodes are available
- Fast Fault Recovery can occur within 1 second, as recovery from faults/stragglers happens via computation in memory

Achieving Exactly Once Semantics with Spark Streaming

To achieve exactly once processing of an event by a Spark Streaming application, you need to consider how all of the points of failure restart after having an issue and how you can avoid data loss. Consider that in a Spark Streaming application you have an input source, a driver process that manages the long running job, one or more receiver processes that pull data from the input source, and tasks that apply the processing and push the results to an output sink. To achieve exactly once semantics means ensuring that no data is lost at any point and that the processing is restartable, regardless of where the failure occurs. The following describes each of the key components of achieving exactly once semantics.

Replayable Sources & Reliable Receivers

The source your Spark Streaming application is reading your events from must be replayable. This means that it should be possible to ask the source to provide the message again in cases where the message was retrieved but then the system failed before it could be persisted or processed. In Azure both Azure Event Hubs and Kafka on HDInsight provide replayable sources. An even simpler example of a replayable source is a fault-tolerant file system like HDFS, Azure Storage blobs or Azure Data Lake Store where all the data can be kept in perpetuity (and you can re-read it in its entirety at any point). In Spark Streaming, sources like Event Hubs and Kafka have reliable receivers, meaning they keep track of their progress reading through the source and persist it to fault-tolerant storage (either within ZooKeeper or in Spark Streaming checkpoints written to HDFS). That way, if a receiver fails and is later restarted it can pick up where it left off.

Use the Write Ahead Log

Spark Streaming supports the use of a Write Ahead Log, where any event received from a source is first written to Spark's checkpoint directory in fault-tolerant storage (in Azure this is HDFS backed by either Azure Storage or Azure Data Lake Store) before being stored in an RDD by the receiver running within a worker. In your Spark Streaming application, the Write Ahead Log is enabled for all receivers by setting the

```
spark.streaming.receiver.writeAheadLog.enable
```

 configuration to `true`. The Write Ahead Log provides fault-tolerance for failures of both the driver and the executors.

In the case of workers running tasks against the event data, consider that following the insertion into the Write Ahead Log, the event is inserted by the receiver into an RDD, which by definition is both replicated and distributed across multiple workers. Should the task fail because the worker running it crashed, the task will simply be restarted on another worker that has a replica of the event data, so the event is not lost.

Use Checkpoints

The drivers need to be restartable. If the driver running your Spark Streaming application crashes, it takes down with it all executors running receivers, tasks and any RDD's storing event data. This surfaces two considerations-first you need to be able to save the progress of the job so you can resume it later. This is accomplished by checkpointing the Directed Acyclic Graph (DAG) of the DStream periodically to fault-tolerant storage. This metadata includes the configuration used to create the streaming application, the operations that define the application, and any batches that are queued but not yet completed. This will enable a failed driver to be restarted from the checkpoint information. When it restarts, it will launch new receivers that themselves recover the event data back into RDD's from the Write Ahead Log. Checkpoints are enabled in Spark Streaming in two steps. On the StreamingContext object you configure the path in storage to where the checkpoints are stored:

```
val ssc = new StreamingContext(spark, Seconds(1))
ssc.checkpoint("/path/to/checkpoints")
```

In HDInsight, these checkpoints should be saved to your default storage attached to your cluster (either Azure Storage or Azure Data Lake Store). Next, you need to specify a checkpoint interval (in seconds) on the DStream that controls how often any state data (e.g., state derived from the input event) is persisted to storage. Persisting state data this way can reduce the computation needed when rebuilding the state from the source event.

```

val lines = ssc.socketTextStream("hostname", 9999)
lines.checkpoint(30)
ssc.start()
ssc.awaitTermination()

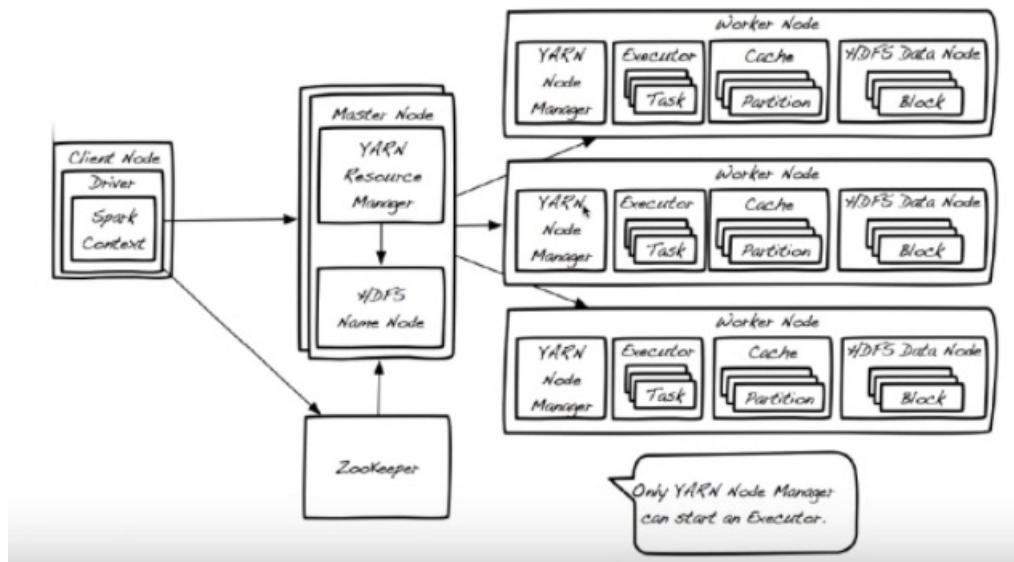
```

Use Idempotent Sinks

The destination to which your job writes results must be able to handle the situation where it is given the same result more than once. It must be able to detect such duplicate results and ignore them. You can achieve idempotent sinks by implementing logic that first checks for the existence of the result in the datastore.

- If the result already exists, then the write should appear to succeed from the perspective of your Spark job, but in reality your data store ignored the duplicate data.
- If the result does not exist, then it should insert the new result into its storage. One example of this is to use a stored procedure with Azure SQL Database that is used to insert events into a table.
- When the stored procedure is invoked, it first looks up the event by key fields and only if none are found is the record inserted into the table. Another example is to use a partitioned file system, like Azure Storage blobs or Azure Data Lake store. In this case your sink logic does not need to check for the existence of a file. If the file representing the event exists, it is simply overwritten with the same data. Otherwise, a new file is created at the computed path. In the end idempotent sinks should support being called multiple times with the same data and no change of state should result.

About Spark Streaming and YARN



In HDInsight cluster work is coordinated by YARN. Architecting for high availability for Spark Streaming has to include not only techniques for Spark Streaming, but also for YARN components. An example configuration using YARN is shown above. There are a number of considerations for this configuration.

Planning for failure

Considering which components could fail, namely an executor or a driver, is a first step in augmenting your HDInsight cluster YARN configuration for high-availability. Also, certain Spark Streaming job failures may include data guarantee requirements which needs additional configurations and setups. For example, a streaming application may have the business requirement for ZERO data loss guarantees in spite of any type of error that could occur in the hosting streaming system or HDInsight cluster.

If an **executor** fails, then tasks and receivers are restarted by Spark automatically, there is no configuration change needed. Importantly, if a **driver** fails, then all of its associated executors fail. Also all computation and received blocks are lost. Use DStream Checkpointing (discussed in an earlier section of this article) to recover from driver

failure. As mentioned previously, DStream Checkpointing periodically saves the DAG of DStreams to fault-tolerant storage (such as Azure BLOBs). Checkpointing allows for the failed driver to be restarted using checkpoint information. This driver restart will launch new executors and will also restart receivers.

To recover drivers w/DStream Checkpointing

- Configure automatic driver restart on the YARN by updating the configuration `yarn.resourcemanager.am.max-attempts` setting
- Set a checkpoint directory in a HDFS-compatible file system by `streamingContext.checkpoint(hdfsDirectory)`
- Restructure source code to use checkpoints for recovery (example function shown below for setup code)
- Configure to recover lost data by enabling WAL via `sparkConf.set("spark.streaming.receiver.writeAheadLog.enable", "true")` and disabling in-memory replication via `StorageLevel.MEMORY_AND_DISK_SER` for input DStreams

```
def creatingFunc() : StreamingContext = {
    val context = new StreamingContext(...)
    val lines = KafkaUtils.createStream(...)
    val words = lines.flatMap(...)
    ...
    context.checkpoint(hdfsDir)
}

val context = StreamingContext.getOrCreate(hdfsDir, creatingFund)
context.start()
```

To summarize, using checkpointing + WAL + reliable receivers, you will be able to deliver "at least once" data recovery

- Exactly once, as long as received data is not lost and if outputs are idempotent or transactional
- Exactly once (alternative) via new Kafka Direct approach - uses Kafka as replicated log, does not use receivers or WALS

Common Mistakes in High Availability and Mitigation Steps

- Not monitoring and managing your streaming jobs - It's much more difficult to monitor streaming jobs than batch jobs. Spark streaming jobs are typically long-running. Also YARN doesn't aggregate logs until a job finishes. Spark checkpoints can't survive app or Spark upgrades. Need to clear checkpoint directory during upgrade. Configure your YARN Cluster mode to run drivers even if client fails. Set up automatic restart on driver, update spark configuration parameters to do this as shown below.

```
spark.yarn.maxAppAttempts = 2
spark.yarn.am.attemptFailuresValidityInterval=1h
```

- In order to work with enhanced metrics for monitoring, here are some considerations. Spark Streaming UI AND Spark have a configurable metrics system, or you can also use additional libraries, such as Graphite/Grafana to download dashboard metrics such as 'num records processed', 'memory/GC usage on driver & executors', 'total delay', 'utilization of the cluster' and others... In Structured Streaming (2.1 or greater only) you can use `StreamingQueryListener` to gather additional metrics.
- Not segmenting long-running jobs. When a Spark Streaming application is submitted to the cluster, the YARN queue where the job runs must be defined. You can use a [YARN Capacity Scheduler](#) to submitting long-running jobs to separate queue.
- Not shutting down your streaming application gracefully - If your offsets are known and state stored is externally then you can programmatically stop your streaming application at the appropriate place. One

technique is to use 'thread hooks' in Spark, by checking for external flag every n seconds -or- use by using a marker file. You can 'touch' this file when starting the application on HDFS and then remove the file when you want to stop. If using this method, use a separate thread in Spark application, which calls code similar to that shown below.

```
streamingContext.stop(stopSparkContext = true, stopGracefully = true)  
//store offsets externally in an external database, to be able to recover on restart
```

Conclusion

This article provided a series of actions that you can take to run Spark Streaming jobs in a fault-tolerant and highly-available way on a YARN cluster on HDInsight. These includes making cluster configuration changes for both YARN and Spark Streaming, for example to enable checkpointing and use the write-ahead log, and detailed in which scenarios the various activities should be used.

- [Spark Streaming Overview](#)
- [Long-running Spark Streaming Jobs on YARN](#)
- [Structured Streaming: Fault Tolerant Semantics](#)
- [Discretized Streams: A Fault-Tolerant Model for Scalable Stream Processing](#)

Creating Spark Streaming jobs with exactly once event processing semantics

8/16/2017 • 5 min to read • [Edit Online](#)

Stream processing applications may take different approaches to how they handle the re-processing of messages in the face of some failure in the system, these approaches yield different event processing semantics:

- At least once: In some cases, it is acceptable for a given event to be processed multiple times. If the system guarantees that the message will always get processed, but it may get processed more than once, than it provides an at least once semantics.
- At most once: In some cases, it is acceptable to lose messages, but it is critical that messages not be processed more than once. A system to supports this is providing an at most once processing semantics.
- Exactly once: In other cases, a message must processed once and only once irrespective of failures in components of the system. Systems that support this provide an exactly once semantics.

This article focuses on how you configure Spark Streaming to achieve exactly once processing semantics.

Achieving Exactly Once Semantics with Spark Streaming

To achieve exactly once processing of an event by a Spark Streaming application, you need to consider how all of the points of failure restart after having an issue and how you can avoid data loss. Consider that in a Spark Streaming application you have an input source, a driver process that manages the long running job, one or more receiver processes that pull data from the input source, and tasks that apply the processing and push the results to an output sink. To achieve exactly once semantics means ensuring that no data is lost at any point and that the processing is restartable, regardless of where the failure occurs. The following describes each of the key components of achieving exactly once semantics.

Replayable Sources & Reliable Receivers

The source your Spark Streaming application is reading your events from must be replayable. This means that it should be possible to ask the source to provide the message again in cases where the message was retrieved but then the system failed before it could be persisted or processed. In Azure, both Azure Event Hubs and Kafka on HDInsight provide replayable sources. An even simpler example of a replayable source is a fault-tolerant file system like HDFS, Azure Storage blobs or Azure Data Lake Store where all the data can be kept in perpetuity (and you can re-read it in its entirety at any point). In Spark Streaming, sources like Event Hubs and Kafka have reliable receivers, meaning they keep track of their progress reading thru the source and persist it to fault-tolerant storage (either within ZooKeeper or in Spark Streaming checkpoints written to HDFS). That way, if a receiver fails and is later restarted it can pick up where it left off.

Use the Write Ahead Log

Spark Streaming supports the use of a Write Ahead Log, where any event received from a source is first written to Spark's checkpoint directory in fault-tolerant storage (in Azure this is HDFS backed by either Azure Storage or Azure Data Lake Store) before being stored in an RDD by the receiver running within a worker. In your Spark Streaming application, the Write Ahead Log is enabled for all receivers by setting the `spark.streaming.receiver.writeAheadLog.enable` configuration to `true`. The Write Ahead Log provides fault-tolerance for failures of both the driver and the executors.

In the case of workers running tasks against the event data, consider that following the insertion into the Write Ahead Log, the event is inserted by the receiver into an RDD, which by definition is both replicated and distributed across multiple workers. Should the task fail because the worker running it crashed, the task will simply be

restarted on another worker that has a replica of the event data, so the event is not lost.

Use Checkpoints

The drivers need to be restartable. If the driver running your Spark Streaming application crashes, it takes down with it all executors running receivers, tasks and any RDD's storing event data. This surfaces two considerations—first you need to be able to save the progress of the job so you can resume it later. This is accomplished by checkpointing the Directed Acyclic Graph (DAG) of the DStream periodically to fault-tolerant storage. This metadata includes the configuration used to create the streaming application, the operations that define the application, and any batches that are queued but not yet completed. This will enable a failed driver to be restarted from the checkpoint information. When it restarts, it will launch new receivers that themselves recover the event data back into RDD's from the Write Ahead Log. Checkpoints are enabled in Spark Streaming in two steps. On the StreamingContext object you configure the path in storage to where the checkpoints are stored:

```
val ssc = new StreamingContext(spark, Seconds(1))
ssc.checkpoint("/path/to/checkpoints")
```

In HDInsight, these checkpoints should be saved to your default storage attached to your cluster (either Azure Storage or Azure Data Lake Store). Next, you need to specify a checkpoint interval (in seconds) on the DStream that controls how often any state data (e.g., state derived from the input event) is persisted to storage. Persisting state data this way can reduce the computation needed when rebuilding the state from the source event.

```
val lines = ssc.socketTextStream("hostname", 9999)
lines.checkpoint(30)
ssc.start()
ssc.awaitTermination()
```

Use Idempotent Sinks

The destination to which your job writes results must be able to handle the situation where it is given the same result more than once. It must be able to detect such duplicate results and ignore them. You can achieve idempotent sinks by implementing logic that first checks for the existence of the result in the datastore. If the result already exists, the write should appear to succeed from the perspective of your Spark job, but in reality your data store ignored the duplicate data. If the result does not exist, then it should insert the new result into its storage. One example of this is to use a stored procedure with Azure SQL Database that is used to insert events into a table. When the stored procedure is invoked, it first looks up the event by key fields and only if none are found is the record inserted into the table. Another example is to use a partitioned file system, like Azure Storage blobs or Azure Data Lake store. In this case your sink logic does not need to check for the existence of a file. If the file representing the event exists, it is simply overwritten with the same data. Otherwise, a new file is created at the computed path. In the end idempotent sinks should support being called multiple times with the same data and no change of state should result.

Next steps

This article covered the key components of achieving exactly once semantics with your Spark Streaming applications.

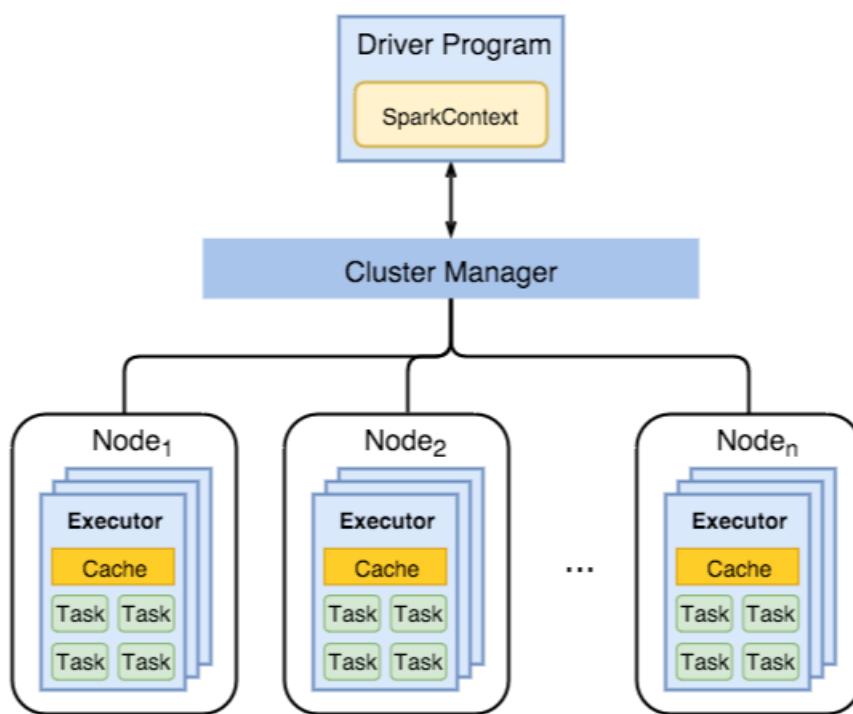
- Review [Spark Streaming Overview](#) for an architectural overview of a Spark Streaming application.
- See [Creating highly available Spark Streaming jobs in YARN](#) for guidance on how to enable high availability for your Spark Streaming applications in HDInsight.

Optimizing and configuring Spark Jobs for Performance

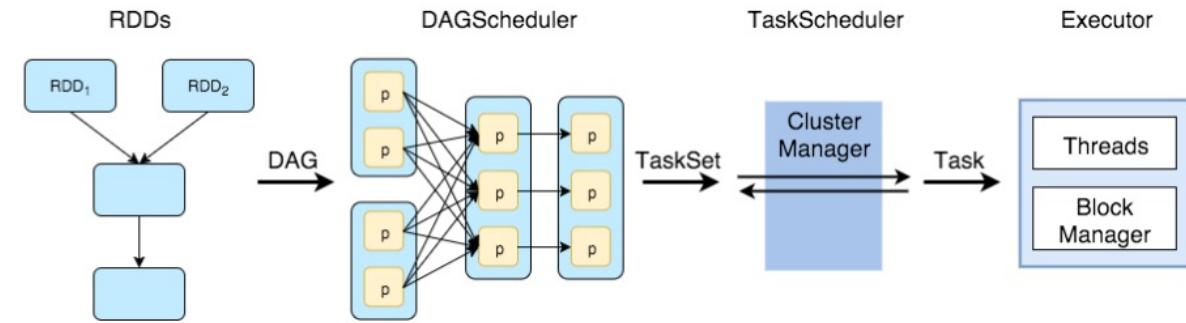
8/16/2017 • 16 min to read • [Edit Online](#)

A Spark cluster is an installation of the Apache Spark library onto a HDInsight Hadoop cluster. Each HDInsight cluster includes default configuration parameters for your Spark cluster at the top level and also at the level of Spark services and service instances in your Spark cluster. A Spark Job is a set of multiple tasks executed via parallel computation. Spark Jobs are generated in response to a Spark actions (such as 'collect' or 'save'). Spark uses a threadpool of tasks for parallel execution rather than a pool of JVM resources (used by MapReduce).

A key aspect of managing a HDInsight Hadoop cluster is monitoring all jobs on the cluster to make sure they are running in a predictable manner. This application monitoring includes Apache Spark job monitoring and optimization. The diagram below shows the core Spark Architecture. It is important to consider the execution objects when determining how to optimize Spark Jobs. The objects shown in the diagram are the Driver Program and its associated Spark Context, the Cluster Manager, and the n-number of Worker Nodes. Each Worker Node includes its own Executor, Cache and n-number of Task instances.



The diagram below shows the core Spark Job workflow stages. As above, it's important to review Job workflow objects when optimizing Spark Jobs. In the diagram the data is represented by the low-level RDD Objects. The next step is the DAG Scheduler. The DAG Scheduler interacts with the Task Scheduler to schedule, submit, launch and retry tasks. These two schedulers interact with the worker instances. Worker instances host threads and also make use of a Block Manager. The Block Manager stores and serves blocks of data to the workflow.

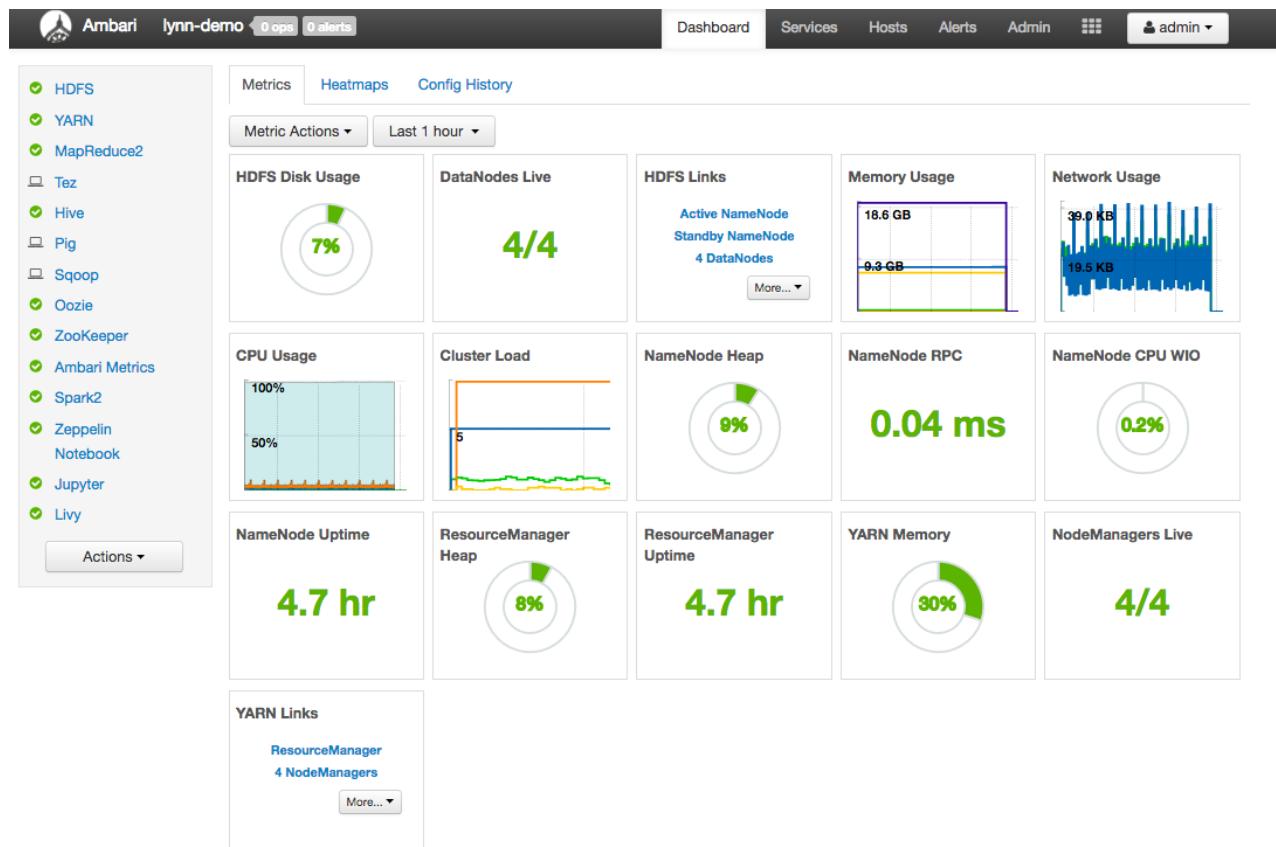


Viewing Cluster Configuration Settings

It is common practice when performing performance optimization on an HDInsight cluster to begin by verifying cluster configuration settings. To do this for not only your particular Apache Spark configuration, but also other services that you may have installed, you launch the HDInsight Dashboard from the Azure Portal by clicking the 'Dashboard' link on the Spark cluster blade.

You will be prompted to login with the username and password that you specified when you setup the HDInsight cluster. After you enter your administrator cluster credentials, then you'll be presented with the Ambari Web UI. This UI provides a dashboard view of the key cluster resource utilization metrics.

An example Ambari HDInsight Dashboard is shown below.



In addition to viewing the top level cluster metrics, you can also view cluster configuration information. To see configuration values for Apache Spark, click on the 'Configs' tab, then click on the 'Spark2' (or 'Spark' depending on the version of Spark that you've installed on your cluster) service link in the service list.

You will be presented with a list of configuration values for your cluster as shown below.

To view Spark configuration values, click the one of the links labeled with the word 'spark' in the link title. Configurations for Spark include the following both custom and advanced configuration values include these configuration categories:

- Custom Spark2-defaults
- Custom Spark2-metrics-properties
- Advanced Spark2-defaults
- Advanced Spark2-env
- Advanced spark2-hive-site-override

After you click one of these links you can view and also update configuration values. If you create a non-default set of configuration values, then you can see a history of any configuration updates you've performed in this UI as well. This configuration history can be helpful if you wish to verify a non-default configuration is in use for performance optimization.

Note: If you only wish to verify common Spark configuration settings, you can also click on the 'Environment' tab on the top level Spark Job UI interface. In this view, you can view, but not change, cluster configuration values. The Spark Job UI is described in the next section of this article.

Track an application in the Spark UI

When working on Spark Job performance, it's important to start by understanding how to get visibility into Job performance via the various HDInsight job monitoring tools. To understand how these tools work, it's best to generate a sample workload. A simple way to generate sample Spark Jobs is by running one or more of the included Jupyter demo notebook(s). Click on the Jupyter blade on your HDInsight instance in the portal and then continue clicking to open and run all cells from one or more sample notebooks.

After your sample workload has completed, from the cluster blade, click 'Cluster Dashboard', and then click 'YARN' to launch the YARN UI. Because you started the Spark job using Jupyter notebooks, the application in the log has the name **remotesparkmagics**. Click the application ID against the application name to get more information about the job. This launches the application view.

For such applications that are launched from the Jupyter notebooks, the status is always RUNNING until you exit the notebook. From the application view, you can drill down further to find out the containers associated with the application and the logs (stdout/stderr). You can also launch the Spark UI by clicking the link next to the Tracking URL (in this case 'Application Master'), as shown below.



The screenshot shows the Hadoop Application Overview page. At the top, there is a logo for 'hadoop' and a link 'Logged in as: dr.who'. Below the header, the application ID is displayed as 'application_1498242895398_0006'. The main content is a table with the following data:

Kill Application		Application Overview
User:	livy	
Name:	remotesparkmagics	
Application Type:	SPARK	
Application Tags:	livy-session-2-9h4j1hj	
Application Priority:	0 (Higher Integer value indicates higher priority)	
YarnApplicationState:	RUNNING: AM has registered with RM and started running.	
Queue:	default	
FinalStatus Reported by AM:	Application has not completed yet.	
Started:	Fri Jun 23 20:13:03 +0000 2017	
Elapsed:	1hrs, 2mins, 52sec	
Tracking URL:	ApplicationMaster	
Log Aggregation Status:	NOT_START	
Diagnostics:		
Unmanaged Application:	false	
Application Node Label expression:	<Not set>	
AM container Node Label expression:	<DEFAULT_PARTITION>	

Clicking the Tracking URL link will open the Spark Jobs UI. After the Spark Jobs UI renders, you can drill down to view specific implementation details for the Spark jobs that have been spawned by the application workload(s) that you started prior to navigating here. You can review detailed information about jobs, stages, storage, environment, executors and Spark SQL via this UI.

The default view is open to the **Jobs** tab. The Jobs tab lists recently run Spark jobs, ordered by Job Id. It provides a high-level view of the status of Job workflow execution outcome by displaying the Job Id, job description, data and time that the job was submitted, job execution duration, job stage and task status. An example of the Spark Jobs UI is shown below.

Spark Jobs [\(?\)](#)

User: yarn
 Total Uptime: 1.0 h
 Scheduling Mode: FIFO
 Completed Jobs: 6

[Event Timeline](#)

Completed Jobs (6)

Job Id (Job Group) ▾	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
5 (4)	Job group for statement 4 runJob at PythonRDD.scala:441	2017/06/23 20:13:32	2 s	2/2	2/2
4 (3)	Job group for statement 3 runJob at PythonRDD.scala:441	2017/06/23 20:13:30	0.4 s	1/1	1/1
3 (2)	Job group for statement 2 saveAsTable at NativeMethodAccessImpl.java:0	2017/06/23 20:13:25	2 s	1/1	1/1
2 (2)	Job group for statement 2 csv at NativeMethodAccessImpl.java:0	2017/06/23 20:13:20	2 s	1/1	2/2
1 (2)	Job group for statement 2 csv at NativeMethodAccessImpl.java:0	2017/06/23 20:13:19	1 s	1/1	1/1
0 (2)	Job group for statement 2 csv at NativeMethodAccessImpl.java:0	2017/06/23 20:13:18	1 s	1/1	1/1

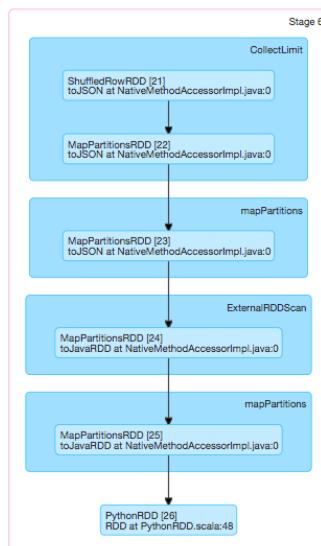
As mentioned, there are a number of views available in the Spark Jobs UI. They allow you to review detailed execution information about the Spark Jobs that have been run on your cluster. This information is key when monitoring and optimizing Spark Job executions. It is especially important that you review the job stages and tasks using the DAG view and understand the overhead of each job stage so that you can verify that your Spark Job is performing as expected.

- Click the **Executors** tab to see processing and storage information for each executor. In this tab, you can also retrieve the call stack by clicking on the Thread Dump link.
- Click the **Stages** tab to see the stages associated with your Spark Job
 - Each job stage can have multiple job tasks for which you can view detailed execution statistics
 - From the stage details page, you can launch the DAG Visualization by clicking the link at the top of the page to expand the DAG visualization view. The DAG (Direct Aclyic Graph) represents the different stages in the application. Each blue box in the graph represents a Spark operation invoked from the application.
 - From the stage details page, you can also launch the application timeline view. Expand the Event Timeline link at the top of the page to view a visualization of the job event execution details.

Details for Stage 6 (Attempt 0)

Total Time Across All Tasks: 71 ms
 Locality Level Summary: Node local: 1

[DAG Visualization](#)



[Show Additional Metrics](#)

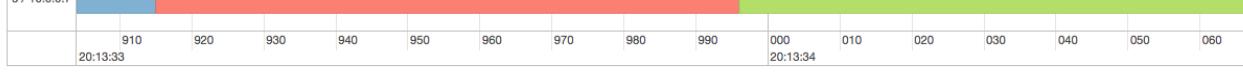
[Event Timeline](#)

Enable zooming

Scheduler Delay Executor Computing Time Getting Result Time

Task Deserialization Time Shuffle Write Time

Shuffle Read Time Result Serialization Time



Summary Metrics for 1 Completed Tasks

Metric	Min	25th percentile	Median	75th percentile	Max
Duration	71 ms	71 ms	71 ms	71 ms	71 ms
GC Time	0 ms	0 ms	0 ms	0 ms	0 ms

Aggregated Metrics by Executor

Executor ID ▲	Address	Task Time	Total Tasks	Failed Tasks	Killed Tasks	Succeeded Tasks
5	stdout stderr	10.0.0.7:37315	0.2 s	1	0	1

Tasks (1)

Index ▲	ID	Attempt	Status	Locality Level	Executor ID / Host	Launch Time	Duration	GC Time	Errors
0	7	0	SUCCESS	NODE_LOCAL	5 / 10.0.0.7	stdout stderr	2017/06/23 20:13:33	71 ms	

TIP: You can also use the Spark History Server to access details about Spark jobs executions that have previously completed.

Logging

Spark uses [log4j](#) for logging. You can configure it by adding a `log4j.properties` file in the `conf` directory, or through the Ambari interface. One of the challenges of having various logs generated during job execution, is being able to correlate information stored in those logs with one another. To address this issue, make the following configuration changes:

- Add thread-id to executor logs: `+[%t]`
- Add precise time stamp in executor logs: `%d{ISO8601}`

Spark Job Optimization Techniques

Listed below are a set of common Spark Job optimization challenges, considerations and recommended actions to improve results.

1. Choose the correct data abstraction

Spark started out using RDDs to abstract data, then introduced DataFrames and DataSets in more recent versions. When deciding which to use, consider the following:

- **DataFrames**

- Best choice in most situations
- Provides query optimization through Catalyst
- Whole stage code generation
- Direct memory access
- Low GC (garbage collection) overhead
- Not as developer-friendly as DataSets due to no compile-time checks or domain object programming

- **DataSets**

- Good in complex ETL pipelines where the performance impact is acceptable
- Not good in aggregations where the impact on performance can be devastating
- Provides query optimization through Catalyst
- Developer-friendly by providing domain object programming and compile-time checks
- Adds serialization/deserialization overhead
- High GC overhead
- Breaks whole stage code generation

- **RDDs**

- If you are using Spark 2.x, there should be no reason to use RDDs, except when you need to build a new custom RDD
- No query optimization through Catalyst
- No whole stage code generation
- High GC overhead
- Legacy APIs that put you in the Spark 1.x world

2. Use optimal data format

Spark supports many formats out of the box, such as csv, json, xml, parquet, orc, and avro. It can be extended to support many more formats with external data sources: <https://spark-packages.org>.

The most recommended format for performance is parquet with snappy compression (default in Spark 2.x). Parquet stores data in columnar format, and is highly optimized in Spark.

3. Storage selection impacts performance

When you create a new Spark cluster, you have the option to select Azure Blob Storage or Azure Data Lake Store as your cluster's default storage. Both options give you the benefit of transient storage, meaning, your data does not get automatically deleted when you delete your cluster. You can recreate your cluster and still access your data. Refer to the table below for speed considerations when selecting your default cluster:

STORE TYPE	FILE SYSTEM	SPEED	TRANSIENT	USE CASES
Azure Blob Storage	<code>wasb://url/</code>	Standard	Yes	Transient cluster
Azure Data Lake Store	<code>adl://url/</code>	Faster	Yes	Transient cluster
Local HDFS	<code>hdfs://url/</code>	Fastest	No	Interactive 24/7 cluster

4. Caching

Spark provides its own native caching mechanisms, and can be used through different methods such as `.persist()`, `.cache()`, and `CACHE TABLE`. This native caching is effective with small data sets as well as in ETL pipelines where you need to cache intermediate results. However, it currently does **not** work well with partitioning, since a cached table does not retain the partitioning data. Therefore, a more generic and reliable caching technique

is storage layer caching.

- Native Spark caching (not recommended)
 - Good for small datasets
 - Does not work with partitioning (will be fixed in the future)
- Storage level caching (recommended)
 - Can be implemented using [Alluxio](#)
 - Uses in-memory + SSD caching
- Local HDFS (recommended)
 - `hdfs://mycluster`
 - SSD caching
 - Cached data will be lost when you delete the cluster, requiring cache rebuild

5. Use Memory Efficiently

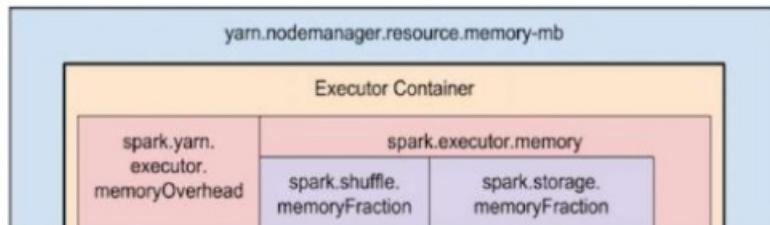
Because Spark operates by placing data in memory, appropriately managing memory resources is a key aspect of optimizing the execution of Spark Jobs. There are several techniques that you can use to use your cluster's memory efficiently. These include the following:

- Prefer smaller data partitions, account for data size, types and distribution in your partitioning strategy
- Consider the newer, more efficient Kryo data Serialization, rather than the default Java Serialization
- Prefer to use YARN, as it separates spark-submit per batch
- Monitor and tune Spark configuration settings

For reference, the Spark memory structure and some key executor memory parameters are shown below.

Spark Memory

If you are using YARN, then YARN controls the maximum sum of memory used by the containers on each Spark node. The graphic below shows the key objects and relationship between them.



Here are set of common practices you can try if you are addressing 'out of memory' messages:

- Review DAG Management Shuffles -> reduce by map-side reducing, pre-partition (or bucketize) source data, maximize single shuffle, reduce the amount of data sent
- Prefer 'ReduceByKey'(has fixed memory limit) to 'GroupByKey'(more powerful, i.e. aggregations, windowing, etc.. but, has unbounded memory limit)
- Prefer 'TreeReduce'(does more work on the executors or partitions) to 'Reduce'(does all work on the driver)
- Leverage DataFrame rather than the lower-level RDD object
- Create ComplexTypes which encapsulate actions, such as 'Top N', various aggregations or windowing ops

6. Optimize Data Serialization

Because Spark Job are distributed, appropriate data serialization is key to best Spark Job performance. There are two serialization options for Spark. Java serialization is the default. There is a new serialization library, Kryo, available. Kryo serialization can result in faster and more compact serialization than that of Java. Kryo serialization is a newer format and it does not yet support all Serializable types. Also it requires that you register the classes in your program.

7. Use bucketing

Bucketing is similar to data partitioning, but each bucket can hold a set of column values (bucket), instead of just one. This is great for partitioning on large (in the millions +) number of values, like product IDs. A bucket is determined by hashing the bucket key of the row. Bucketed tables offer unique optimizations because they store metadata about how they were bucketed and sorted.

Some advanced bucketing features are:

- Query optimization based on bucketing meta-information
- Optimized aggregations
- Optimized joins

You can use partitioning and bucketing at the same time.

8. Fix slow Joins/Shuffles

If you have slow jobs on Join/Shuffle, for example it may take 20 seconds to run a map job, but 4 hrs when running a job where the data is joined or shuffled, the cause is probably data skew. Data skew is defined as asymmetry in your job data. To fix data skew, you should salt the entire key, or perform an isolated salt (meaning apply the salt to only some subset of keys). If you are using the 'isolated salt' technique, you should further filter to isolate your subset of salted keys in map joins. Another option is to introduce a bucket column and pre-aggregate in buckets first.

Another factor causing slow joins could be the join type. By default, Spark uses the `SortMerge` join type. This type of join is best suited for large data sets, but is otherwise computationally expensive (slow) because it must first sort the left and right sides of data before merging them. A `Broadcast` join, on the other hand, is best suited for smaller data sets, or where one side of the join is significantly smaller than the other side. It broadcasts one side to all executors, so requires more memory for broadcasts in general.

You can change the join type in your configuration by setting `spark.sql.autoBroadcastJoinThreshold`, or you can change the type with a join hint using the DataFrame APIs (`dataframe.join(broadcast(df2))`).

Example:

```
// Option 1
spark.conf.set("spark.sql.autoBroadcastJoinThreshold", 1*1024*1024*1024)

// Option 2
val df1 = spark.table("FactTableA")
val df2 = spark.table("dimMP")
df1.join(broadcast(df2), Seq("PK")).
  createOrReplaceTempView("V_JOIN")
sql("SELECT col1, col2 FROM V_JOIN")
```

If you are using bucketed tables, you have a third join type: `Merge` join. A correctly pre-partitioned and pre-sorted dataset will skip the expensive sort phase in a `SortMerge` join.

The order of joins matters, particularly in more complex queries. Make sure you start with the most selective joins. Also, move joins that increase the number of rows after aggregations when possible.

Additionally, to manage parallelism, specifically to fight Cartesian Joins, you can add nested structures, windowing and/or skip step(s) in your Spark Job.

9. Spark Cluster Custom Configuration

Depending on your Spark workload, you may determine that a non-default Spark configuration would result in more optimized Spark Job executions. You should perform benchmark testing with key workloads to validate any non-default cluster configurations. Some of the common parameters that you may consider adjusting are listed below with associated parameter notes.

- Executors (--num-executors)
 - set the appropriate number of executors
- Cores for each executor (--executor-cores)
 - have middle-sized executors, as other processes will consume some portion of the available memory
- Memory for each executor (--executor-memory)
 - controls heap size on YARN, you'll need to leave some memory for execution overhead

Selecting the correct executor size

When deciding your executor configuration, you need to consider what the Java Garbage Collection (GC) overhead will be.

- Factors to reduce size
 1. Reduce heap size below 32GB to keep GC-overhead < 10%
 2. Reduce cores to keep GC-overhead < 10%
- Factors to increase size
 1. Reduce communication overhead between executors
 2. Reduce number of open connections between executors (N²) on larger clusters (> 100 executors)
 3. Increase heap size to accommodate for memory demanding tasks
 4. Optional: Reduce per executor memory overhead
 5. Optional: Increase utilization and concurrency by oversubscribing CPU

As a general rule of thumb when selecting the executor size:

1. Start with 30GB per executor and distribute available machine cores
2. Increase number of executor-cores for larger clusters (> 100 executors)
3. Increase/decrease size based on trial runs and factors from previous slide (like observed GC-overhead)

When running concurrent queries, consider the following:

1. Start with 30GB per executor and all machine cores
2. Create multiple parallel spark applications by oversubscribing CPU (around 30% latency improvement)
3. Distribute queries across parallel applications
4. Increase/decrease size based on trial runs and factors from previous slide (like observed GC-overhead)

Always remember to monitor your query performance for outliers or other performance issues, by looking at the time line view, SQL graph, job statistics, etc. Sometimes one or a few of the executors are slower than the others, and tasks take much longer to execute. This frequently happens on larger clusters (> 30 nodes). To mitigate, divide work into a larger number of tasks so the scheduler can compensate for slow tasks. For example, have at least 2x as many tasks as the number of executor cores in the application. Also, enable speculative execution of tasks:

```
conf: spark.speculation = true .
```

10. Optimize to Speed Up Job execution

- Cache, but do it wisely. If you use the data twice, then cache it.
- Broadcast variables, then they are visible to all executors. They are only serialized once. Results in very fast lookups.
- Threading - use thread pool on driver. Results in fast operation, with many tasks.

As always, monitor your running jobs regularly for performance issues. If you need more insight into certain issues, consider one of the following performance profiling tools:

- [Intel PAL Tool](#) - CPU, Storage, network bandwidth utilization
- [Oracle Java 8 Mission Control](#) – Spark/executor code profiling

Key to Spark 2.x query performance is the Tungsten engine, which depends on whole stage code generation. In

some cases, whole stage code generation may be disabled.

For example, if you use a non-mutable type (`string`) in the aggregation expression, `SortAggregate` will appear instead of `HashAggregate`.

```
MAX(AMOUNT) -> MAX(cast(AMOUNT as DOUBLE))
```

When we discovered this issue with one customer's dataset, fixing `string` in an aggregation expression and re-enabling code generation improved performance by 3x.

Conclusion

There are a number of core considerations you need to pay attention to make sure your Spark Jobs run in a predictable and performant way. It's key for you to focus on using the best Spark cluster configuration for your particular workload. Along with that, you'll need to monitor the execution of long-running and/or high resource consuming Spark Job executions. The most common challenges center around memory pressure due to improper configurations (particularly wrong-sized executors), long-running operations and tasks which result in cartesian operations. Using caching judiciously can significantly speed up jobs. Finally, it's important to adjust for data skew in your job tasks.

See also

- [Debug Spark jobs running on Azure HDInsight](#)
- [Manage resources for a Spark cluster on HDInsight](#)
- [Use the Spark REST API to submit remote jobs to a Spark cluster](#)
- [Tuning Spark](#)
- [How to Actually Tune Your Spark Jobs So They Work](#)
- [Kryo Serialization](#)

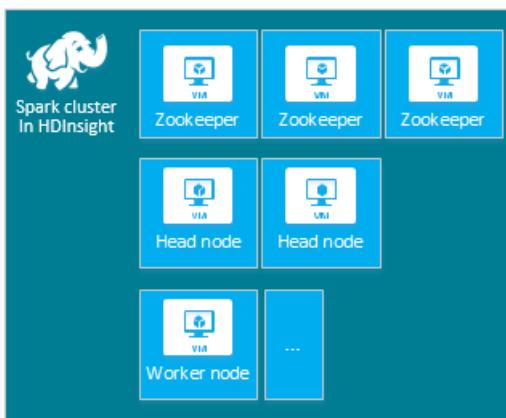
Configuring Spark Settings

8/16/2017 • 8 min to read • [Edit Online](#)

Understanding Default Cluster Nodes

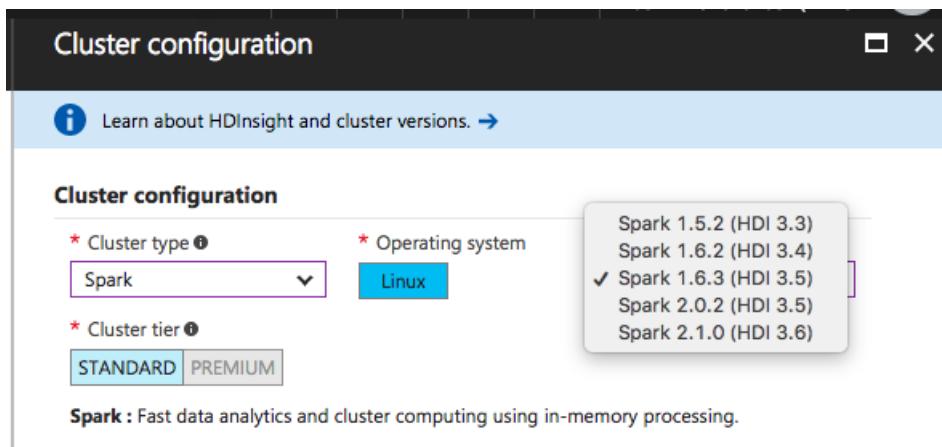
A HDInsight Spark cluster includes an installation of the Apache Spark library. Each HDInsight cluster includes default configuration service parameters for all of its installed services - including Spark. A key aspect of managing a HDInsight Hadoop cluster is monitoring workload, including Spark Jobs, to make sure they are running in a predictable manner. It is important to consider the physical cluster configuration when determining how to optimize cluster configuration to best run Spark Jobs.

The default HDInsight Apache Spark cluster includes the following nodes: two head nodes, one or more worker nodes, three ZooKeeper nodes. Below is a diagram of the default configuration. Determining the correct number and VM sizes for the nodes in your HDInsight cluster is one of many considerations about cluster configuration which can also affect Spark configuration. Non-default HDInsight configuration values often go hand-in-hand with non-default Spark configuration values. When you create an HDInsight Spark cluster, you will be presented with suggested VM sizes for each of the components (D12 v2 or greater as of this writing, which are [Memory-optimized Linux VM sizes](#) for Azure).



Understanding Spark Versions

Selecting the best version of Spark for your cluster is another important consideration. Spark 2.x (plus configuration) has the potential to run much better than Spark 1.x. This is because 2.x has a number of performance optimizations, such as Tungsten, Catalyst Query Optimization and more. HDInsight includes multiple versions of both Spark and HDInsight itself. It's important to select the version of Spark that best suits your workloads. Each version of Spark includes a set of default cluster settings. Shown below are the Spark versions that you can select from when deploying a new cluster.



TIP: The default version of Apache Spark from the HDInsight service may change without notice. If you have a version dependency, Microsoft recommends that you specify the particular version when you create clusters using .NET SDK/Azure PowerShell and Azure CLI.

Apache Spark provides three locations to configure the system:

- Spark properties control most application parameters and can be set by using a `SparkConf` object, or through Java system properties.
- Environment variables can be used to set per-machine settings, such as the IP address, through the `conf/spark-env.sh` script on each node.
- Logging can be configured through `log4j.properties`.

When you select a particular version of Spark, your cluster includes a number of default configuration settings. For whichever version of Spark that you choose, you can change the default Spark configuration values by providing a custom Spark configuration file. An example is shown below.

```
spark.hadoop.io.compression.codecs org.apache.hadoop.io.compress.GzipCodec
spark.sql.files.openCostInBytes 1099511627776
spark.sql.files.maxPartitionBytes 1099511627776
spark.hadoop.mapreduce.input.fileinputformat.split.minsize 1099511627776
spark.hadoop.parquet.block.size 1099511627776
```

In the example shown above (taken from a bioinformatics use case) default values for a number of Spark configuration parameters are overridden. These are the compression codec, hadoop mapreduce split minimum size and parquet block sizes, as well as the spar sql partition and open file sizes default values. These configuration changes were made because the associated data and jobs (i.e. genomic data) have particular characteristics which will perform better using these custom configuration settings.

Viewing Cluster Configuration Settings

It is common practice when performing performance optimization on an HDInsight cluster to begin by verifying current cluster configuration settings. To do this for not only your particular Apache Spark configuration, but also other services that you may have installed, you launch the HDInsight Dashboard from the Azure Portal by clicking the 'Dashboard' link on the Spark cluster blade.

You will be prompted to login with the username and password that you specified when you setup the HDInsight cluster. After you enter your administrator cluster credentials, then you'll be presented with the Ambari Web UI. This UI provides a dashboard view of the key cluster resource utilization metrics. It also includes a tab for `Config History`. Here you can quickly jump to configuration information about any installed service, including Spark.

After you click on `Config History`, then click on `Spark2` to see configuration values for Apache Spark. Next click

on the **Configs** tab, then click on the **Spark2** (or **Spark** depending on the version of Spark that you've installed on your cluster) service link in the service list. You will be presented with a list of configuration values for your cluster as shown below.

To view Spark configuration values, click the one of the links labeled with the word 'spark' in the link title. Configurations for Spark include the following both custom and advanced configuration values include these configuration categories:

- Custom Spark2-defaults
- Custom Spark2-metrics-properties
- Advanced Spark2-defaults
- Advanced Spark2-env
- Advanced spark2-hive-site-override

After you click one of these links you can view and also update configuration values. If you create a non-default set of configuration values, then you can see a history of any configuration updates you've performed in this UI as well. This configuration history can be helpful if you wish to verify a non-default configuration is in use for performance optimization.

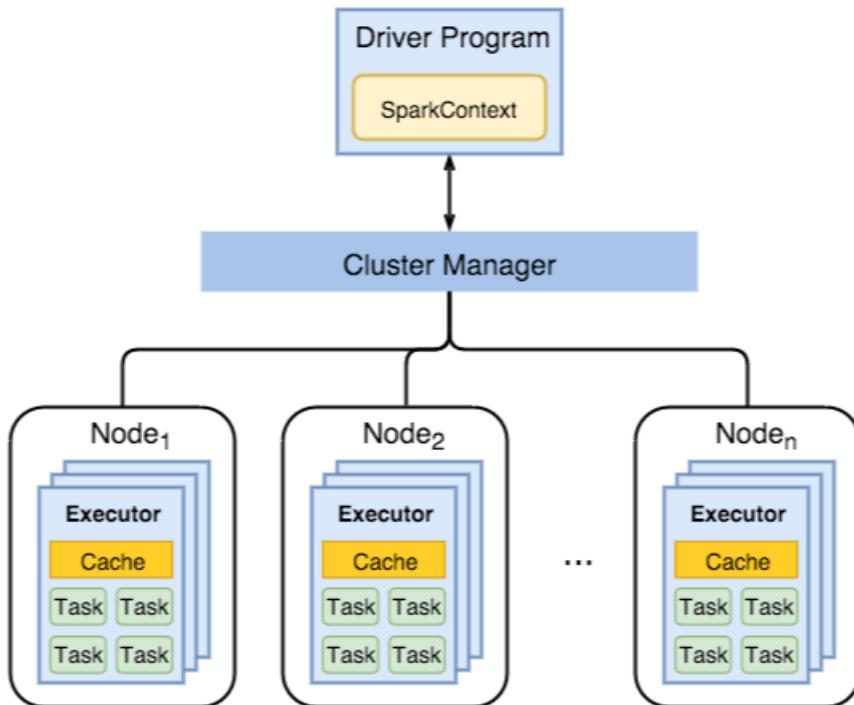
The screenshot shows the Ambari UI interface for managing configurations. On the left, there's a sidebar with a tree view of services: HDFS, YARN, MapReduce2, Tez, Hive, Pig, Sqoop, Oozie, ZooKeeper, Ambari Metrics, and **Spark2**. The **Spark2** service is currently selected. The main content area has tabs for **Summary** and **Configs**, with **Configs** being active. It displays a configuration group named **Default (9)**. There are two versions listed: **V2** (internal, 5 hours ago, HDP-2.6) and **V1** (hdinsightwatchd..., 5 hours ago, HDP-2.6). Below this, a message says "Internal authored on Fri, Jun 23, 2017 11:34". A "Save" button is visible. The main list contains various configuration items, each with a disclosure triangle:

- Advanced livy2-conf
- Advanced livy2-env
- Advanced livy2-spark-blacklist
- Advanced spark2-defaults
- Advanced spark2-env
- Advanced spark2-hive-site-override
- Advanced spark2-log4j-properties
- Advanced spark2-logsearch-conf
- Advanced spark2-metrics-properties
- Advanced spark2-thrift-fairscheduler
- Advanced spark2-thrift-sparkconf
- Custom livy2-conf
- Custom spark2-defaults
- Custom spark2-hive-site-override
- Custom spark2-metrics-properties
- Custom spark2-thrift-fairscheduler
- Custom spark2-thrift-sparkconf

Note: If you only wish to verify common Spark configuration settings, you can also click on the [Environment](#) tab on the top level [Spark Job UI](#) interface. From this page, you can view, but not change, running Spark cluster configuration values.

Configuring Spark Executors

Because Spark Jobs use worker resources, particularly memory, it's common to adjust Spark configuration values for worker node processes or Spark Executors. The diagram below shows key Spark objects such as the Driver Program and its associated Spark Context, the Cluster Manager, and the n-number of Worker Nodes. Each Worker Node includes its own Executor, Cache and n-number of Task instances.



Three key parameters that are often adjusted to tune Spark configurations to improve application requirements are `spark.executor.instances`, `spark.executor.cores`, and `spark.executor.memory`. An Executor is a process launched for a Spark application. It runs on the worker node and is responsible for carrying out the tasks for the application. The default number of executors and the executor sizes for each cluster is calculated based on the number of worker nodes and the worker node size. These are stored in `spark-defaults.conf` on the cluster head nodes. You can edit these values in a running cluster by clicking the link [Custom spark-defaults](#) in the Ambari web UI (shown earlier in this article). After you make changes, then you'll be prompted in the UI to [Restart](#) all the affected services.

TIP: The three configuration parameters can be configured at the cluster level (for all applications that run on the cluster) or can be specified for each individual application as well.

Another source of information about the resources being used by the Spark Executors is the Spark Application UI. In the Spark UI, click the [Executors](#) tab to display a Summary and Detail view of the configuration and resources consumed by the executors as shown below. This can help you to understand when you might want to change default values for Spark executors for the entire cluster or a particular set of job executions.

Executors

Summary

RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time (GC Time)	Input	Shuffle Read	Shuffle Write
Active(6) 0	0.0 B / 12.3 GB	0.0 B	15	0	0	727	727	1.8 min (2 s)	388.9 MB	5.1 KB	5.4 KB
Dead(0) 0	0.0 B / 0.0 B	0.0 B	0	0	0	0	0	0 ms (0 ms)	0.0 B	0.0 B	0.0 B
Total(6) 0	0.0 B / 12.3 GB	0.0 B	15	0	0	727	727	1.8 min (2 s)	388.9 MB	5.1 KB	5.4 KB

Executors

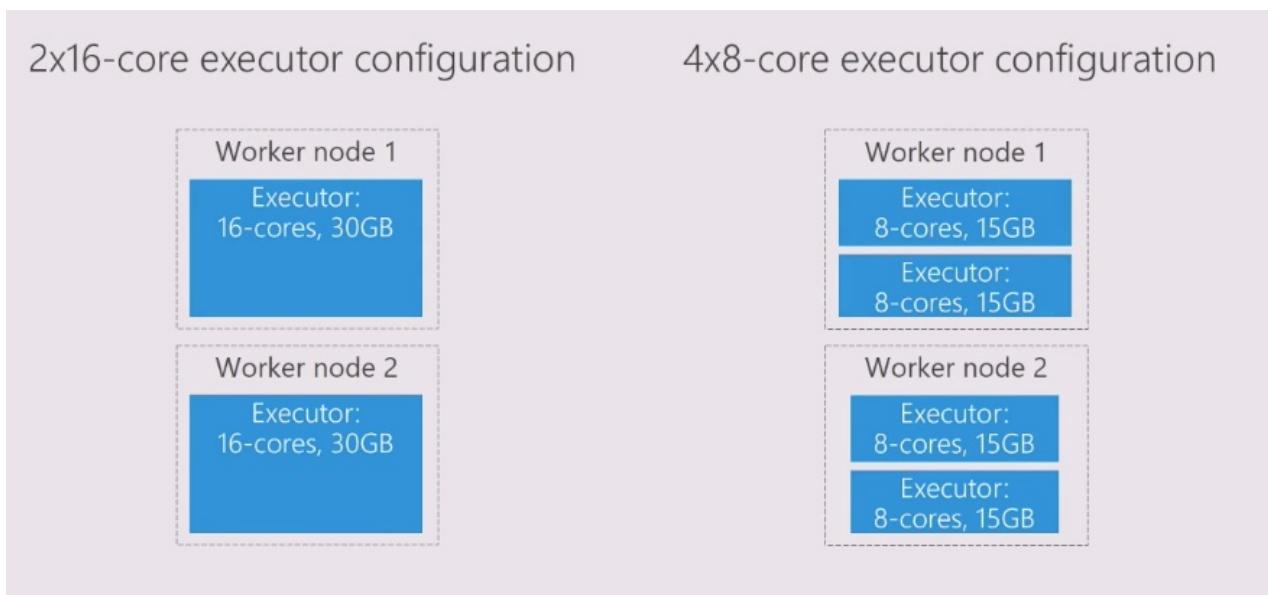
Show 20 entries											Search:				
Executor ID	Address	Status	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time (GC Time)	Input	Shuffle Read	Shuffle Write	Logs
driver	10.0.0.9:40768	Active	0	0.0 B / 384.1 MB	0.0 B	0	0	0	0	0	0 ms (0 ms)	0.0 B	0.0 B	0.0 B	stdout stderr
1	10.0.0.15:44619	Active	0	0.0 B / 2.4 GB	0.0 B	3	0	0	160	160	24 s (0.3 s)	63.2 MB	976 B	1.1 KB	stdout stderr
2	10.0.0.10:34315	Active	0	0.0 B / 2.4 GB	0.0 B	3	0	0	163	163	23 s (0.3 s)	76.2 MB	531 B	1.1 KB	stdout stderr
3	10.0.0.6:35216	Active	0	0.0 B / 2.4 GB	0.0 B	3	0	0	140	140	26 s (0.4 s)	90.7 MB	1.2 KB	1 KB	stdout stderr
4	10.0.0.9:39280	Active	0	0.0 B / 2.4 GB	0.0 B	3	0	0	125	125	14 s (0.4 s)	77.2 MB	1.3 KB	1.1 KB	stdout stderr
5	10.0.0.4:42183	Active	0	0.0 B / 2.4 GB	0.0 B	3	0	0	139	139	24 s (0.4 s)	81.6 MB	1.1 KB	1.1 KB	stdout stderr

Showing 1 to 6 of 6 entries

Previous 1 Next

Alternatively, you can use the Ambari REST API to programmatically verify HDInsight and Spark cluster configuration settings. More information is available via the [GitHub repository Ambari API reference](#).

Depending on your Spark workload, you may determine that a non-default Spark configuration would result in more optimized Spark Job executions. You should perform benchmark testing with key workloads to validate any non-default cluster configurations. Some of the common parameters that you may consider adjusting are listed below with associated parameter notes. Also an example of how you might configure two worker nodes with different node configuration values is shown in the graphic below.

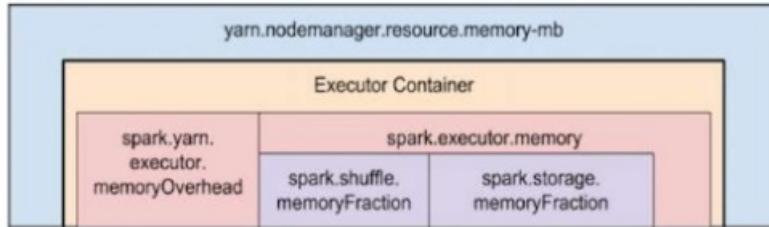


- Executors (--num-executors)
 - set the appropriate number of executors
- Cores for each executor (--executor-cores)
 - have middle-sized executors, as other processes will consume some portion of the available memory
- Memory for each executor (--executor-memory)
 - controls heap size on YARN, you'll need to leave some memory for execution overhead

For reference key Spark executor memory parameters are shown below.

- `spark.executor.memory` defines the TOTAL amount of memory available for the executor
- `spark.storage.memoryFraction` (default ~ 60%) defines the amount available for storing persisted RDDs
- `spark.shuffle.memoryFraction` (default ~ 20%) defines the amount reserved for shuffle
- `spark.storage.unrollFraction/safetyFraction` (~30% of total memory) - avoid using, this is used internally by Spark

Because YARN controls the maximum sum of memory used by the containers on each Spark node it is useful to understand the relationship between YARN configuration objects and Spark objects per node. The graphic below shows the key objects and relationship between them.



Change the parameters for an application running in Jupyter notebook

Spark clusters in HDInsight include a number of components that are available on the clusters by default. Each of these components includes default configuration values which can be overridden as business needs dictate.

- Spark Core - Spark Core, Spark SQL, Spark streaming APIs, GraphX, and MLlib
- Anaconda - a python package manager
- Livy - the Apache Spark REST API (used to submit remote jobs to a HDInsight Spark cluster)
- Jupyter and Zeppelin notebooks - interactive browser-based UI for interacting with your Spark cluster
- ODBC driver -- connects Spark clusters in HDInsight from BI tools such as Microsoft Power BI and Tableau

For applications running in the Jupyter notebook, you can use the `%%configure` command to make configuration changes from within the notebook itself. These configuration changes will be applied to the Spark Jobs run from your notebook instance. Ideally, you must make such changes at the beginning of the application, before you run your first code cell. This ensures that the configuration is applied to the Livy session, when it gets created. If you want to change the configuration at a later stage in the application, you must use the `-f` parameter. However, by doing so all progress in the application will be lost.

The code below shows how to change the configuration for an application running in a Jupyter notebook.

```
%%configure
{"executorMemory": "3072M", "executorCores": 4, "numExecutors":10}
```

Conclusion

There are a number of core configuration settings that you need to monitor and adjust to make sure your Spark Jobs run in a predictable and performant way. It's key for you to focus on using the best Spark cluster configuration for your particular workloads. Along with that, you'll need to monitor the execution of long-running and/or high resource consuming Spark Job executions. The most common challenges center around memory pressure due to improper configurations (particularly wrong-sized executors), long-running operations and tasks which result in cartesian operations.

See also

- What are the Hadoop components and versions available with HDInsight?
- Manage resources for a Spark cluster on HDInsight
- Set up clusters in HDInsight with Hadoop, Spark, Kafka, and more
- Apache Spark Configuration
- Running Spark on YARN

Choosing between RDD, DataFrame, and Dataset for Spark

8/16/2017 • 9 min to read • [Edit Online](#)

What is an RDD?

Big Data applications rely on iterative, distributed computing for faster processing of large data sets. To distribute data processing over multiple jobs, the data is typically reused or shared across jobs. To share data between existing distributed computing systems you need to store data in some intermediate stable distributed store such as HDFS. This makes the overall computations of jobs slower.

Resilient Distributed Datasets (RDDs) address this by enabling fault-tolerant, distributed, in-memory computations. (RDDs) are the baseline data abstractions in Spark. They are fault-tolerant collections of elements stored in-memory or on-disk that can be operated on in parallel. An RDD can hold many kinds of source data. RDDs are created by loading an external dataset or distributing a collection from the Spark driver program.

An RDD can be persisted in-memory across operations. When an RDD is persisted, each node stores any partitions of it that it computes in-memory and then reuses them in other actions on the data set. You can mark an RDD as persistent just by calling the `persist()` or `cache()` method. You can also specify the storage level: on-disk or in-memory as a serialized Java object. Cached, or persistent, RDDs are fault-tolerant without replication.

Each RDD maintains its lineage (for example, the sequence of transformations that resulted in the RDD). If an RDD is lost because a node crashed, it can be reconstructed by replaying the sequence of operations.

What are RDD operations?

RDDs support two types of operations: transformations and actions.

- **Transformations** create a new dataset from an existing one. Transformations are lazy by default, meaning that no transformation is executed until you execute an action. This does not apply to persistent RDDs. Examples of transformations include: map, filter, sample, union, and more.
- **Actions** return a value to the driver program after running a computation on the dataset. Examples include: reduce, collect, count, first, foreach, etc.

The code sample below contains both transitions and actions. It shows how to search through error messages in a log file that is stored in HDFS using Scala. A source file is loaded, then filtered, cached and the rows in cache are counted. Next an additional filter is applied to the cached rows. Row are again counted and finally returned via the collect method.

```
val file = spark.textFile("hdfs://...")  
val errors = file.filter(line => line.contains("ERROR"))  
errors.cache()  
errors.count()  
errors.filter(line => line.contains("Web")).count()  
errors.filter(line => line.contains("Error")).collect()
```

An Evolving API

The Apache Spark API as a whole is evolving at a rapid pace, including changes and additions to its core APIs. One of the most disruptive areas of change is around the representation of data sets. Although Spark 1.0 used the RDD

API, two new alternative and incompatible APIs have been introduced. Spark 1.3 introduced the radically different DataFrame API and the Spark 1.6 release introduced a preview of the new Dataset API.

You may be wondering whether to jump from RDDs directly to the Dataset API, or whether to first move to the DataFrame API.

Currently, the DataFrame APIs offer the most performance. RDD APIs still exist in Spark 2.x for backwards compatibility, and should not be used. Going forward, only the DataFrame and Dataset APIs will be developed.

The main disadvantage to RDDs is that they don't perform particularly well. Whenever Spark needs to distribute the data within the cluster, or write the data to disk, it does so using Java serialization by default (although it is possible to use Kryo as a faster alternative in most cases). The overhead of serializing individual Java and Scala objects is expensive and requires sending both data and structure between nodes (each serialized object contains the class structure as well as the values). There is also the overhead of garbage collection that results from creating and destroying individual objects.

DataFrame API

Spark 1.3 introduced a new DataFrame API to improve the performance and scalability of Spark. The DataFrame API introduced the concept of a schema to describe the data, allowing Spark to manage the schema and only pass data between nodes, in a much more efficient way than using Java serialization. There are also advantages when performing computations in a single process as Spark can serialize the data into off-heap storage in a binary format and then perform many transformations directly on this off-heap memory, avoiding the garbage-collection costs associated with constructing individual objects for each row in the data set. Because Spark understands the schema, there is no need to use Java serialization to encode the data.

The DataFrame API is radically different from the RDD API because it is an API for building a relational query plan that Spark's Catalyst optimizer can then execute. The API is natural for developers who are familiar with building query plans, but not natural for the majority of developers. The query plan can be built from SQL expressions in strings or from a more functional approach using a fluent-style API.

Example: Filter by attribute with DataFrame in Scala

```
//SQL Style  
df.filter("age > 21");  
  
//Expression builder style:  
df.filter(df.col("age").gt(21));
```

Because the code is referring to data attributes by name, it is not possible for the compiler to catch any errors. If attribute names are incorrect then the error will only detected at runtime, when the query plan is created.

How do I make a dataframe?

You can load a dataframe directly from an input data source. See the following notebooks included with your HDInsight Spark cluster for more information.

- Read and write data from Azure Storage Blobs (WASB)
- Read and write data from Hive tables

Spark SQL and dataframes

You can run SQL queries over dataframes once you register them as temporary tables within the SQL context.

The HDInsight Spark kernel supports easy inline SQL queries. Simply type `%sql` followed by a SQL query to run a SQL query on a dataframe.

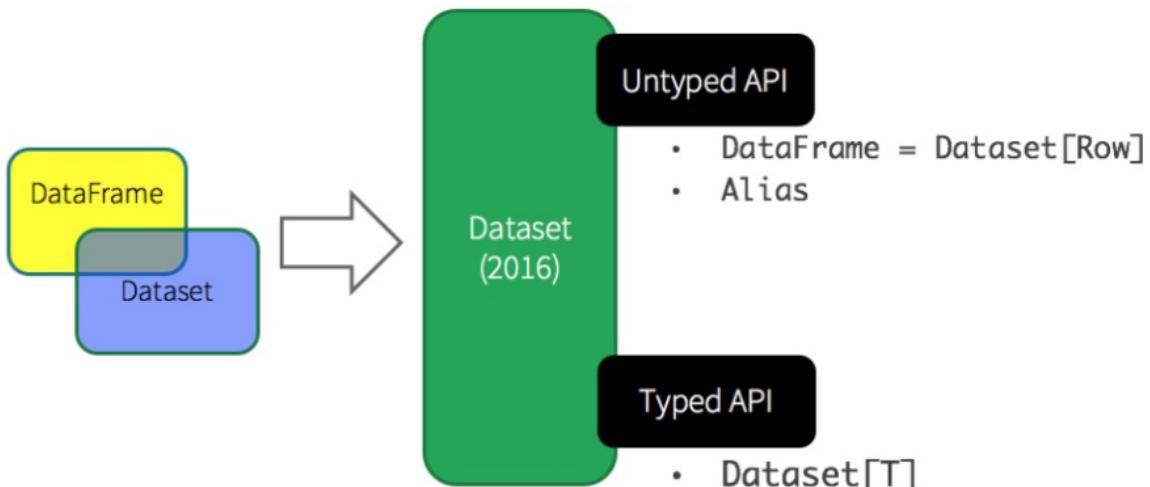
See [Spark SQL and DataFrame Guide](#) for more information.

Dataset API

The Dataset API, released as an API preview in Spark 1.6 (and as generally available in Spark 2.x), aims to provide the best of both worlds; the familiar object-oriented programming style and compile-time type-safety of the RDD API but with the performance benefits of the Catalyst query optimizer. Datasets also use the same efficient off-heap storage mechanism as the DataFrame API.

When it comes to serializing data, the Dataset API has the concept of encoders which translate between JVM representations (objects) and Spark's internal binary format. Spark has built-in encoders which are very advanced in that they generate byte code to interact with off-heap data and provide on-demand access to individual attributes without having to de-serialize an entire object. Spark does not yet provide an API for implementing custom encoders, but that is planned for a future release.

Unified Apache Spark 2.0 API



Example: Creating Dataset from a list of objects

```
val sc = new SparkContext(conf)
val sqlContext = new SQLContext(sc)
import sqlContext.implicits._
val sampleData: Seq[ScalaPerson] = ScalaData.sampleData()
val dataset = sqlContext.createDataset(sampleData)
```

Transformations with the Dataset API look very much like the RDD API and deal with the Person class rather than an abstraction of a row.

Example: Filter by attribute with Dataset

```
dataset.filter(_.age < 21);
```

Despite the similarity with RDD code, this code is building a query plan, rather than dealing with individual objects, and if age is the only attribute accessed, then the rest of the object's data will not be read from off-heap storage.

With the release of Spark 2.x, there are really only two programmatic APIs now; RDD and Dataset. For backwards compatibility, DataFrame still exists but is just a synonym for a Dataset.

CSV support is now built-in and based on the DataBricks spark-csv project, making it easy to create Datasets from

CSV data with little coding.

Spark 2.0 is a major release, and there are some breaking changes that mean you may need to rewrite some of your code. Here are some things we ran into when updating our apache-spark-examples.

For Scala users, SparkSession replaces SparkContext and SQLContext as the top-level context, but still provides access to SQLContext and SQLContext for backwards compatibility

DataFrame is now a synonym for Dataset[Row] and you can use these two types interchangeably, although we recommend using the latter. Performing a map() operation on a Dataset now returns a Dataset rather than an RDD, reducing the need to keep switching between the two APIs, and improving performance.

Some Java functional interfaces, such as FlatMapFunction, have been updated to return Iterator rather than Iterable.

RDD vs. Dataset 2.0

Both the RDD API and the Dataset API represent data sets of a specific class. For instance, you can create an RDD[Person] as well as a Dataset[Person] so both can provide compile-time type-safety. Both can also be used with the generic Row structure provided in Spark for cases where classes might not exist that represent the data being manipulated, such as when reading CSV files.

RDDs can be used with any Java or Scala class and operate by manipulating those objects directly with all of the associated costs of object creation, serialization and garbage collection.

Datasets are limited to classes that implement the Scala Product trait, such as case classes. There is a very good reason for this limitation. Datasets store data in an optimized binary format, often in off-heap memory, to avoid the costs of deserialization and garbage collection. Even though it feels like you are coding against regular objects, Spark is really generating its own optimized byte-code for accessing the data directly.

The Scala code examples below show working with RDDs, Dataframes and Datasets.

```

// RDD raw object manipulation
val rdd: RDD[Person] = ...
val rdd2: RDD[String] = rdd.map(person => person.lastName)
// Dataset optimized direct access to memory without deserializing objects
val ds: Dataset[Person] = ...
val ds2: Dataset[String] = ds.map(person => person.lastName)

// Create SparkSession (start point for a Spark driver program in 2x)
val spark = SparkSession.builder
    .master("local[*]")
    .appName("Example")
    .getOrCreate()
// Access (legacy) SparkContext and SQLContext
spark.sparkContext
spark.sqlContext

// Create a Dataset from a collection
var ds: Dataset[String] = spark.createDataset(List("one","two","three"))

// Convert an RDD to a Dataset
// Use SparkSession createDataset method to convert an RDD to a Dataset
// Import spark.implicits_ (spark is the name of the SparkSession variable)
// Import implicits so that Spark can infer types when creating Datasets
import spark.implicits._

val rdd: RDD[Person] = ??? // assume this exists
val dataset: Dataset[Person] = spark.createDataset[Person](rdd)

// Convert a DataFrame (which is really a Dataset[Row]) to a Dataset
// Convert to a Dataset of a specific class by performing a map() operation
// Read a text file into a DataFrame a.k.a. Dataset[Row]
var df: Dataset[Row] = spark.read.text("people.txt")

// Use map() to convert to a Dataset of a specific class
var ds: Dataset[Person] = spark.read.text("people.txt")
    .map(row => parsePerson(row))
def parsePerson(row: Row) : Person = ??? // fill in parsing logic here

// Read a CSV directly as a Dataset (CSV must contain a header row)
val ds: Dataset[Person] = spark.read
    .option("header","true")
    .csv("people.csv")
    .as[Person]

```

Conclusion

Currently, the DataFrame APIs offer the most performance. RDD APIs still exist in Spark 2.x for backwards compatibility, and should not be used. Going forward, only the DataFrame and Dataset APIs will be developed.

The Spark Dataset API is very performant and provides a more natural way to code than using the more low-level RDD abstraction. Given the rapid evolution of Spark it is likely that this API will mature and become the de-facto API for developing new applications.

The Spark API is moving from unstructured computation (RDDs) towards structured computation (Datasets) because of the many performance optimizations that the latter allows. DataFrames were a step in direction of structured computation but lacked developer-desired features such as compile time safety and lambda functions. The Spark Dataset API unifies both Dataframes and RDDs.

Datasets

RDDs

- Functional Programming
- Type-safe

Dataframes

- Relational
- Catalyst query optimization
- Tungsten direct/packed RAM
- JIT code generation
- Sorting/shuffling without deserializing



See also

- [Get started: Create an Apache Spark cluster in HDInsight and run interactive Spark SQL queries](#)
- [Spark SQL, DataFrames and Datasets Guide](#)
- [RDDs, DataFrames and Datasets video](#)

What is HBase in HDInsight: A NoSQL database that provides BigTable-like capabilities for Hadoop

8/16/2017 • 3 min to read • [Edit Online](#)

Apache HBase is an open-source, NoSQL database that is built on Hadoop and modeled after Google BigTable. HBase provides random access and strong consistency for large amounts of unstructured and semistructured data in a schemaless database organized by column families.

Data is stored in the rows of a table, and data within a row is grouped by column family. HBase is a schemaless database in the sense that neither the columns nor the type of data stored in them need to be defined before using them. The open-source code scales linearly to handle petabytes of data on thousands of nodes. It can rely on data redundancy, batch processing, and other features that are provided by distributed applications in the Hadoop ecosystem.

How is HBase implemented in Azure HDInsight?

HDInsight HBase is offered as a managed cluster that is integrated into the Azure environment. The clusters are configured to store data directly in [Azure Storage](#) or [Azure Data Lake Store](#), which provides low latency and increased elasticity in performance and cost choices. This enables customers to build interactive websites that work with large datasets, to build services that store sensor and telemetry data from millions of end points, and to analyze this data with Hadoop jobs. HBase and Hadoop are good starting points for big data project in Azure; in particular, they can enable real-time applications to work with large datasets.

The HDInsight implementation leverages the scale-out architecture of HBase to provide automatic sharding of tables, strong consistency for reads and writes, and automatic failover. Performance is enhanced by in-memory caching for reads and high-throughput streaming for writes. HBase cluster can be created inside virtual network. For details, see [Create HDInsight clusters on Azure Virtual Network](#).

How is data managed in HDInsight HBase?

Data can be managed in HBase by using the `create`, `get`, `put`, and `scan` commands from the HBase shell. Data is written to the database by using `put` and read by using `get`. The `scan` command is used to obtain data from multiple rows in a table. Data can also be managed using the HBase C# API, which provides a client library on top of the HBase REST API. An HBase database can also be queried by using Hive. For an introduction to these programming models, see [Get started using HBase with Hadoop in HDInsight](#). Co-processors are also available, which allow data processing in the nodes that host the database.

NOTE

Thrift is not supported by HBase in HDInsight.

Scenarios: Use cases for HBase

The canonical use case for which BigTable (and by extension, HBase) was created was web search. Search engines build indexes that map terms to the web pages that contain them. But there are many other use cases that HBase is suitable for—several of which are itemized in this section.

- Key-value store

HBase can be used as a key-value store, and it is suitable for managing message systems. Facebook uses HBase for their messaging system, and it is ideal for storing and managing Internet communications.

WebTable uses HBase to search for and manage tables that are extracted from webpages.

- Sensor data

HBase is useful for capturing data that is collected incrementally from various sources. This includes social analytics, time series, keeping interactive dashboards up-to-date with trends and counters, and managing audit log systems. Examples include Bloomberg trader terminal and the Open Time Series Database (OpenTSDB), which stores and provides access to metrics collected about the health of server systems.

- Real-time query

[Phoenix](#) is a SQL query engine for Apache HBase. It is accessed as a JDBC driver, and it enables querying and managing HBase tables by using SQL.

- HBase as a platform

Applications can run on top of HBase by using it as a datastore. Examples include Phoenix, OpenTSDB, Kiji, and Titan. Applications can also integrate with HBase. Examples include Hive, Pig, Solr, Storm, Flume, Impala, Spark, Ganglia, and Drill.

Next steps

- [Get started using HBase with Hadoop in HDInsight](#)
- [Create HDInsight clusters on Azure Virtual Network](#)
- [Configure HBase replication in HDInsight](#)
- [Analyze Twitter sentiment with HBase in HDInsight](#)
- [Use Maven to build Java applications that use HBase with HDInsight \(Hadoop\)](#)

See also

- [Apache HBase](#)
- [Bigtable: A Distributed Storage System for Structured Data](#)

Choosing storage option for HBase on HDInsight

8/15/2017 • 2 min to read • [Edit Online](#)

HBase on HDInsight can be configured to use either Azure Storage blobs or Azure Data Lake Store as the location where it stores its data (e.g., Storefiles) and metadata. When you provision an HDInsight cluster, either option can be used as the default storage location. Additionally, either can be attached to the HDInsight cluster as additional storage.

Azure Storage

When using Azure Storage blobs, the files managed by HBase will be stored as block blobs in the configured Storage Account. There are some limitations you should be aware of when choosing Azure Storage that may affect the performance and scalability of your HDInsight HBase cluster:

- **Total Storage Account Capacity:** Azure Storage has an upper limit of 500 TB per Storage Account. This means that if you only have one Storage Account attached to your HDInsight HBase cluster, then the maximum size it can support will be close to 500 TB.
- **Maximum Blob Size:** Each individual blob in Azure Storage blobs has a maximum size of approximately 4.75 TB. This dictates maximum size of your individual Storefiles.
- **Maximum Throughput for Storage Account:** The Storage Account has an all-up limit of 20,000 requests per second. This limit represents the combined requests per second against all blobs in the Storage Account. This means there is an upper limit of how many "hot" Storefiles your Storage Account will support. For example, if you had 41 Storefiles that were hot, each using 500 requests per second, you would hit this upper limit because your Storage Account would be experiencing 20,500 requests per second. When you hit the limit, your requests will be throttled, slowing down the execution of your HBase operations.
- **Maximum Throughput for a Single Blob:** Each blob in Azure Storage supports a maximum throughput of up to 60 MB/s or 500 requests per second. This limit effectively caps the throughput available for serving requests against your Storefiles, since each Storefile is represented by a blob. When you exceed these limits, your requests will be throttled, slowing down the execution of your HBase operations against that Storefile.

For the latest on Azure Storage limits, see [Azure subscription limits and quotas](#).

Data Lake Store

When using Azure Data Lake Store, the files managed by HBase are stored by Data Lake Store without any of the aforementioned limitations that apply to Azure Storage Blobs. This provides many benefits:

- No limits on the all up size of your HBase storage.
- No limits on the size of your Storefiles.
- High throughput and IOPs.

Additionally, because Azure Data Lake Store is designed to support write-once, read many analytic workloads, it automatically replicates files to multiple serving nodes. For example, in read scenarios the Storefile can be read from any of the replicas. This means that multiple HBase operations against the same Storefile may be served from different nodes, giving your queries a greater maximum throughput than if the data was served from one node only.

Region availability limits

When utilizing Azure Data Lake Store, consider that it is not available in all regions. In order to use Data Lake Store with your HDInsight HBase cluster, you will need to provision your HDInsight cluster in the same location in which

you have provisioned an instance of Data Lake Store. For the latest list of regions supporting Azure Data Lake Store, see [Azure Products by Region](#)

See also

- For more details on HDInsight storage architecture, see [HDInsight Architecture](#)

Get started with an Apache HBase example in HDInsight

8/16/2017 • 7 min to read • [Edit Online](#)

Learn how to create an HBase cluster in HDInsight, create HBase tables, and query tables by using Hive. For general HBase information, see [HDInsight HBase overview](#).

WARNING

Billing for HDInsight clusters is prorated per minute, whether you are using them or not. Be sure to delete your cluster after you have finished using it. For more information, see [How to delete an HDInsight cluster](#).

Prerequisites

Before you begin trying this HBase example, you must have the following items:

- **An Azure subscription.** See [Get Azure free trial](#).
- [Secure Shell\(SSH\)](#).
- [curl](#).

Create HBase cluster

The following procedure uses an Azure Resource Manager template to create a version 3.4 Linux-based HBase cluster and the dependent default Azure Storage account. To understand the parameters used in the procedure and other cluster creation methods, see [Create Linux-based Hadoop clusters in HDInsight](#).

1. Click the following image to open the template in the Azure portal. The template is located in a public blob container.

[Deploy to Azure >](#)

2. From the **Custom deployment** blade, enter the following values:

- **Subscription:** Select your Azure subscription that is used to create the cluster.
- **Resource group:** Create an Azure Resource Management group or use an existing one.
- **Location:** Specify the location of the resource group.
- **ClusterName:** Enter a name for the HBase cluster.
- **Cluster login name and password:** The default login name is **admin**.
- **SSH username and password:** The default username is **sshuser**. You can rename it.

Other parameters are optional.

Each cluster has an Azure Storage account dependency. After you delete a cluster, the data retains in the storage account. The cluster default storage account name is the cluster name with "store" appended. It is hardcoded in the template variables section.

3. Select **I agree to the terms and conditions stated above**, and then click **Purchase**. It takes about 20 minutes to create a cluster.

NOTE

After an HBase cluster is deleted, you can create another HBase cluster by using the same default blob container. The new cluster picks up the HBase tables you created in the original cluster. To avoid inconsistencies, we recommend that you disable the HBase tables before you delete the cluster.

Create tables and insert data

You can use SSH to connect to HBase clusters and then use HBase Shell to create HBase tables, insert data, and query data. For more information, see [Use SSH with HDInsight](#).

For most people, data appears in the tabular format:

ID	Name	Home phone	Office phone	Office address
1000	John Dole	1-425-000-0001	1-425-000-0002	1111 San Gabriel Dr

In HBase (an implementation of BigTable), the same data looks like:

	Column family: Personal		Column family: Office	
ID	Name	Phone	Phone	Address
1000	John Dole	1-425-000-0001	1-425-000-0002	1111 San Gabriel Dr

To use the HBase shell

1. From SSH, run the following HBase command:

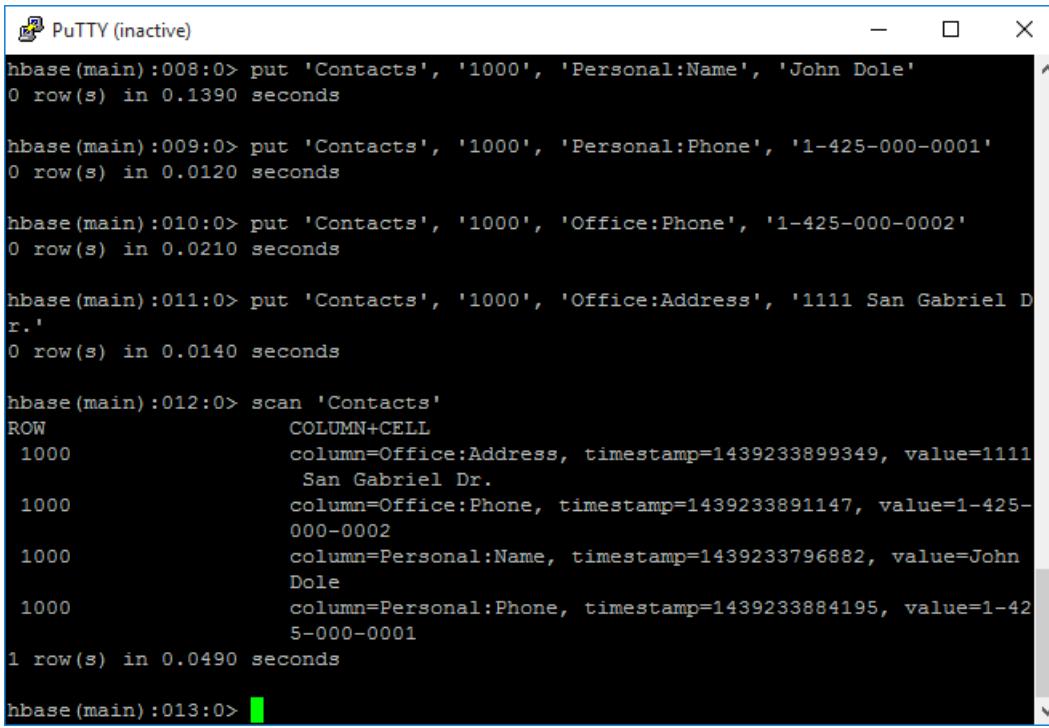
```
hbase shell
```

2. Create an HBase with two-column families:

```
create 'Contacts', 'Personal', 'Office'  
list
```

3. Insert some data:

```
put 'Contacts', '1000', 'Personal:Name', 'John Dole'  
put 'Contacts', '1000', 'Personal:Phone', '1-425-000-0001'  
put 'Contacts', '1000', 'Office:Phone', '1-425-000-0002'  
put 'Contacts', '1000', 'Office:Address', '1111 San Gabriel Dr.'  
scan 'Contacts'
```



```
hbase(main):008:0> put 'Contacts', '1000', 'Personal:Name', 'John Dole'
0 row(s) in 0.1390 seconds

hbase(main):009:0> put 'Contacts', '1000', 'Personal:Phone', '1-425-000-0001'
0 row(s) in 0.0120 seconds

hbase(main):010:0> put 'Contacts', '1000', 'Office:Phone', '1-425-000-0002'
0 row(s) in 0.0210 seconds

hbase(main):011:0> put 'Contacts', '1000', 'Office:Address', '1111 San Gabriel Dr.'
0 row(s) in 0.0140 seconds

hbase(main):012:0> scan 'Contacts'
ROW                                COLUMN+CELL
 1000      column=Office:Address, timestamp=1439233899349, value=1111
            San Gabriel Dr.
 1000      column=Office:Phone, timestamp=1439233891147, value=1-425-
            000-0002
 1000      column=Personal:Name, timestamp=1439233796882, value=John
            Dole
 1000      column=Personal:Phone, timestamp=1439233884195, value=1-42-
            5-000-0001
1 row(s) in 0.0490 seconds

hbase(main):013:0>
```

4. Get a single row

```
get 'Contacts', '1000'
```

You shall see the same results as using the scan command because there is only one row.

For more information about the HBase table schema, see [Introduction to HBase Schema Design](#). For more HBase commands, see [Apache HBase reference guide](#).

5. Exit the shell

```
exit
```

To bulk load data into the contacts HBase table

HBase includes several methods of loading data into tables. For more information, see [Bulk loading](#).

A sample data file can be found in a public blob container, <wasb://hbasecontacts@hditutorialdata.blob.core.windows.net/contacts.txt>. The content of the data file is:

8396	Calvin Raji	230-555-0191	230-555-0191	5415 San Gabriel Dr.
16600	Karen Wu	646-555-0113	230-555-0192	9265 La Paz
4324	Karl Xie	508-555-0163	230-555-0193	4912 La Vuelta
16891	Jonn Jackson	674-555-0110	230-555-0194	40 Ellis St.
3273	Miguel Miller	397-555-0155	230-555-0195	6696 Anchor Drive
3588	Osa Agbonile	592-555-0152	230-555-0196	1873 Lion Circle
10272	Julia Lee	870-555-0110	230-555-0197	3148 Rose Street
4868	Jose Hayes	599-555-0171	230-555-0198	793 Crawford Street
4761	Caleb Alexander	670-555-0141	230-555-0199	4775 Kentucky Dr.
16443	Terry Chander	998-555-0171	230-555-0200	771 Northridge Drive

You can optionally create a text file and upload the file to your own storage account. For the instructions, see [Upload data for Hadoop jobs in HDInsight](#).

NOTE

This procedure uses the Contacts HBase table you have created in the last procedure.

1. From SSH, run the following command to transform the data file to StoreFiles and store at a relative path specified by Dimporttsv.bulk.output. If you are in HBase Shell, use the exit command to exit.

```
hbase org.apache.hadoop.hbase.mapreduce.ImportTsv -  
Dimporttsv.columns="HBASE_ROW_KEY,Personal:Name,Personal:Phone,Office:Phone,Office:Address" -  
Dimporttsv.bulk.output="/example/data/storeDataFileOutput" Contacts  
wasb://hbasecontacts@hditutorialdata.blob.core.windows.net/contacts.txt
```

2. Run the following command to upload the data from /example/data/storeDataFileOutput to the HBase table:

```
hbase org.apache.hadoop.hbase.mapreduce.LoadIncrementalHFiles /example/data/storeDataFileOutput  
Contacts
```

3. You can open the HBase shell, and use the scan command to list the table content.

Use Hive to query HBase

You can query data in HBase tables by using Hive. In this section, you create a Hive table that maps to the HBase table and uses it to query the data in your HBase table.

1. Open **PuTTY**, and connect to the cluster. See the instructions in the previous procedure.
2. From the SSH session, use the following command to start Beeline:

```
beeline -u 'jdbc:hive2://localhost:10001;transportMode=http' -n admin
```

For more information about Beeline, see [Use Hive with Hadoop in HDInsight with Beeline](#).

3. Run the following HiveQL script to create a Hive table that maps to the HBase table. Make sure that you have created the sample table referenced earlier in this tutorial by using the HBase shell before you run this statement.

```
CREATE EXTERNAL TABLE hbasecontacts(rowkey STRING, name STRING, homephone STRING, officephone STRING,  
officeaddress STRING)  
STORED BY 'org.apache.hadoop.hive.hbase.HBaseStorageHandler'  
WITH SERDEPROPERTIES ('hbase.columns.mapping' =  
':key,Personal:Name,Personal:Phone,Office:Phone,Office:Address')  
TBLPROPERTIES ('hbase.table.name' = 'Contacts');
```

4. Run the following HiveQL script to query the data in the HBase table:

```
SELECT count(rowkey) FROM hbasecontacts;
```

Use HBase REST APIs using Curl

The REST API is secured via [basic authentication](#). You shall always make requests by using Secure HTTP (HTTPS) to help ensure that your credentials are securely sent to the server.

1. Use the following command to list the existing HBase tables:

```
curl -u <UserName>:<Password> \
-G https://<ClusterName>.azurehdinsight.net/hbaserest/
```

2. Use the following command to create a new HBase table with two-column families:

```
curl -u <UserName>:<Password> \
-X PUT "https://<ClusterName>.azurehdinsight.net/hbaserest/Contacts1/schema" \
-H "Accept: application/json" \
-H "Content-Type: application/json" \
-d "{\"@name\":\"Contact1\", \"ColumnSchema\":[{\"name\":\"Personal\"}, {"name\":\"Office\"}]}\" \
-v
```

The schema is provided in the JSON format.

3. Use the following command to insert some data:

```
curl -u <UserName>:<Password> \
-X PUT "https://<ClusterName>.azurehdinsight.net/hbaserest/Contacts1/false-row-key" \
-H "Accept: application/json" \
-H "Content-Type: application/json" \
-d "{\"Row\": [{\"key\": \"MTAwMA==\", \"Cell\": [ {\"column\": \"UGVyc29uYWw6TmFtZQ==\", \
\"$\": \"Sm9obiBEb2xl\"}]}]}\" \
-v
```

You must base64 encode the values specified in the -d switch. In the example:

- MTAwMA==: 1000
- UGVyc29uYWw6TmFtZQ==: Personal:Name
- Sm9obiBEb2xl: John Doe

[false-row-key](#) allows you to insert multiple (batched) values.

4. Use the following command to get a row:

```
curl -u <UserName>:<Password> \
-X GET "https://<ClusterName>.azurehdinsight.net/hbaserest/Contacts1/1000" \
-H "Accept: application/json" \
-v
```

For more information about HBase Rest, see [Apache HBase Reference Guide](#).

NOTE

Thrift is not supported by HBase in HDInsight.

When using Curl or any other REST communication with WebHCat, you must authenticate the requests by providing the user name and password for the HDInsight cluster administrator. You must also use the cluster name as part of the Uniform Resource Identifier (URI) used to send the requests to the server:

```
curl -u <UserName>:<Password> \
-G https://<ClusterName>.azurehdinsight.net/templeton/v1/status
```

You should receive a response similar to the following response:

```
{"status": "ok", "version": "v1"}
```

Check cluster status

HBase in HDInsight ships with a Web UI for monitoring clusters. Using the Web UI, you can request statistics or information about regions.

To access the HBase Master UI

1. Sign into the Ambari Web UI at <https://<Clustername>.azurehdinsight.net>.
2. Click **HBase** from the left menu.
3. Click **Quick links** on the top of the page, point to the active Zookeeper node link, and then click **HBase Master UI**. The UI is opened in another browser tab:

The screenshot shows the Apache HBase Master UI interface. At the top, there's a navigation bar with tabs for Home, Table Details, Procedures, Local Logs, Log Level, Debug Dump, Metrics Dump, and HBase Configuration. Below the navigation bar, the title 'Region Servers' is displayed, followed by a table with columns: ServerName, Start time, Requests Per Second, and Num. Regions. Two rows are listed: one for IP 10.0.0.10 with start time Thu Nov 17 20:57:00 UTC 2016 and 2 regions; and another for IP 10.0.0.12 with start time Thu Nov 17 20:57:08 UTC 2016 and 1 region. A summary row at the bottom shows Total: 2 regions. Below the Region Servers section, there's a 'Backup Masters' section which is currently empty.

The HBase Master UI contains the following sections:

- region servers
- backup masters
- tables
- tasks
- software attributes

Delete the cluster

To avoid inconsistencies, we recommend that you disable the HBase tables before you delete the cluster.

WARNING

Billing for HDInsight clusters is prorated per minute, whether you are using them or not. Be sure to delete your cluster after you have finished using it. For more information, see [How to delete an HDInsight cluster](#).

Troubleshoot

If you run into issues with creating HDInsight clusters, see [access control requirements](#).

Next steps

In this article, you learned how to create an HBase cluster and how to create tables and view the data in those tables from the HBase shell. You also learned how to use a Hive query on data in HBase tables and how to use the HBase C# REST APIs to create an HBase table and retrieve data from the table.

To learn more, see:

- [HDInsight HBase overview](#): HBase is an Apache, open-source, NoSQL database built on Hadoop that provides random access and strong consistency for large amounts of unstructured and semistructured data.

Using the HBase REST SDK

8/16/2017 • 5 min to read • [Edit Online](#)

When you use [HBase](#) as your massively scalable NoSQL database, you are given two primary choices to work with your data: [Hive queries and calls to HBase's RESTful API](#). A common way to work with the REST API is through the use of `curl`, or similar.

If your developers are more familiar with `C#`, with .NET as the platform of choice, the [Microsoft HBase REST Client Library for .NET](#) provides a client library on top of the HBase REST API, and is available as a NuGet package to quickly get started.

Installing the SDK

The HBase .NET SDK is provided as a NuGet package, which can be installed from the Visual Studio [NuGet Package Manager Console](#) with the following command:

```
Install-Package Microsoft.HBase.Client
```

Instantiating a new HBaseClient object

To begin using the library, you must instantiate a new `HBaseClient` object, passing in `ClusterCredentials` composed of the `Uri` to your cluster, the Hadoop user name and password.

```
var credentials = new ClusterCredentials(new Uri("https://CLUSTERNAME.azurehdinsight.net"), "USERNAME",  
    "PASSWORD");  
client = new HBaseClient(credentials);
```

Replace CLUSTERNAME with your HDInsight HBase cluster name, and USERNAME and PASSWORD with the Hadoop credentials specified on cluster creation. The default Hadoop user name is **admin**.

Create a new table

HBase, like any other RDBMS, stores data in tables. A table consists of a `Rowkey`, the primary key and one or more group of columns known as **Column families**. The data in table is horizontally distributed by `Rowkey` range into **Regions**. Each region has a start and end key. A table can have one or more regions. As the data in table grows, HBase splits large regions into smaller regions. Regions are stored in **Region Servers**. One region server can store multiple regions.

The data is physically stored in **HFiles**. A single HFile contains data for one table, one region and one column family. Rows in HFile are stored sorted on Rowkey. HFile has a B+ Tree index for speedy retrieval of the rows.

To create a new table, you specify a `TableSchema` and columns. In the code below, we are first checking for the existence of the table prior to creating it.

```
if (!client.ListTablesAsync().Result.name.Contains("RestSDKTable"))
{
    // Create the table
    var newTableSchema = new TableSchema {name = "RestSDKTable" };
    newTableSchema.columns.Add(new ColumnSchema() {name = "t1"});
    newTableSchema.columns.Add(new ColumnSchema() {name = "t2"});

    await client.CreateTableAsync(newTableSchema);
}
```

We created a new table named "RestSDKTable" with two column families, t1 and t2. As discussed above, column families are stored separately in different HFiles, thus it makes sense to have a separate column family for data which is queried often. For example, we'll add columns to the t1 column family that we'll query often.

Delete a table

Deleting a table is as simple as this:

```
await client.DeleteTableAsync("RestSDKTable");
```

Inserting data

When you insert data, you must specify a unique row key. This serves as the Id for the row. All of the data is stored in a `byte[]` array. You'll notice that we are storing the `title`, `director`, and `release_date` columns within the t1 column family, and `description` and `tagline` within the t2 column family. This is because we tend to query the t1 column family more often. You may partition your data into column families however you see fit.

```

var key = "fifth_element";
var row = new CellSet.Row { key = Encoding.UTF8.GetBytes(key) };
var value = new Cell
{
    column = Encoding.UTF8.GetBytes("t1:title"),
    data = Encoding.UTF8.GetBytes("The Fifth Element")
};
row.values.Add(value);
value = new Cell
{
    column = Encoding.UTF8.GetBytes("t1:director"),
    data = Encoding.UTF8.GetBytes("Luc Besson")
};
row.values.Add(value);
value = new Cell
{
    column = Encoding.UTF8.GetBytes("t1:release_date"),
    data = Encoding.UTF8.GetBytes("1997")
};
row.values.Add(value);
value = new Cell
{
    column = Encoding.UTF8.GetBytes("t2:description"),
    data = Encoding.UTF8.GetBytes("In the colorful future, a cab driver unwittingly becomes the central figure in the search for a legendary cosmic weapon to keep Evil and Mr Zorg at bay.")
};
row.values.Add(value);
value = new Cell
{
    column = Encoding.UTF8.GetBytes("t2:tagline"),
    data = Encoding.UTF8.GetBytes("The Fifth is life")
};
row.values.Add(value);

var set = new CellSet();
set.rows.Add(row);

await client.StoreCellsAsync("RestSDKTable", set);

```

HBase implements BigTable. Thus, the format from our data above will look like:

Column family: t1			Column family: t2		
id	title	director	release_date	description	tagline
fifth_element	The Fifth Element	Luc Besson	1997	In the colorful future...	The Fifth is life

Selecting data

To read data from the HBase table, pass the table name and row key to the `GetCellsAsync` method to return the `CellSet`.

```

var key = "fifth_element";

var cells = await client.GetCellsAsync("RestSDKTable", key);
// Get the first value from the row.
Console.WriteLine(Encoding.UTF8.GetString(cells.rows[0].values[0].data));
// Get the value for t1:title
Console.WriteLine(Encoding.UTF8.GetString(cells.rows[0].values
    .Find(c => Encoding.UTF8.GetString(c.column) == "t1:title").data));
// With the previous insert, it should yield: "The Fifth Element"

```

In this case, we're just returning the first matching row (there should only be one when using a unique key), then

converting the returned values into `string` format from the `byte[]` array. You may also convert the values to other types, such as an integer for our movie's release date:

```
var releaseDateField = cells.rows[0].values
    .Find(c => Encoding.UTF8.GetString(c.column) == "t1:release_date");
int releaseDate = 0;

if (releaseDateField != null)
{
    releaseDate = Convert.ToInt32(Encoding.UTF8.GetString(releaseDateField.data));
}
Console.WriteLine(releaseDate);
// Should return 1997
```

Scanning over rows

HBase uses `scan` to retrieve one or more rows. In this example, we're requesting multiple rows in batches of 10, and retrieving data whose key values are between 25 and 35. After retrieving our rows, we delete the scanner to clean up resources.

```
var tableName = "mytablename";

// Assume the table has integer keys and we want data between keys 25 and 35
var scanSettings = new Scanner()
{
    batch = 10,
    startRow = BitConverter.GetBytes(25),
    endRow = BitConverter.GetBytes(35)
};
RequestOptions scanOptions = RequestOptions.GetDefaultOptions();
scanOptions.AlternativeEndpoint = "hbaserest0/";
ScannerInformation scannerInfo = null;
try
{
    scannerInfo = await client.CreateScannerAsync(tableName, scanSettings, scanOptions);
    CellSet next = null;
    while ((next = client.ScannerGetNextAsync(scannerInfo, scanOptions).Result) != null)
    {
        foreach (var row in next.rows)
        {
            // ... read the rows
        }
    }
}
finally
{
    if (scannerInfo != null)
    {
        await client.DeleteScannerAsync(tableName, scannerInfo, scanOptions);
    }
}
```

Next steps

In this article, we learned how to use the HBase .NET SDK to work with the HBase REST API. Learn more about HBase and other tools to work with its data by following the links below.

- [Get started with an Apache HBase example in HDInsight](#)
- [Build an end-to-end application with Analyze real-time Twitter sentiment with HBase](#)

Setting up Backup and Replication for HBase and Phoenix on HDInsight

8/16/2017 • 9 min to read • [Edit Online](#)

HBase supports a few different approaches for guarding against data loss. These approaches are:

- Copying the hbase folder
- Export then Import
- Copy table
- Snapshots
- Replication

As Apache Phoenix stores all of its metadata in HBase tables, any option you take that backs up the HBase system catalog tables applies to the backup of Phoenix metadata.

This article covers each of these approaches, providing guidance one when to use which and how to setup that form of backup.

Copying the hbase folder

HDInsight on HBase stores its data in the default storage selected when provisioning the cluster, which can be either Azure Storage blobs or Azure Data Lake store. In either case, hbase stores all of its data and metadata files under the following path:

```
/hbase
```

If you are using Azure Storage blobs, the external view of this same path is as follows, where the hbase folder sits at the root of the blob container in your Azure Storage account:

```
wasbs://<containername>@<accountname>.blob.core.windows.net/hbase
```

In Azure Data Lakes Store, the hbase folder simple sits under the root path you specified during cluster provisioning, typically underneath a clusters folder, with a subfolder named after your HDInsight cluster:

```
/clusters/<clusterName>/hbase
```

This folder contains all of the data that hbase has flushed to disk, but it may not contain all off the data HBase is managing in-memory. Therefore, it is important to shut down your cluster first before relying on this folder as an accurate representation of your HBase data. Once you have done so, however, you can use this approach to restore HBase in two different ways.

1. Without moving the data at all, you can create a new HDInsight instance and have it point to this same storage location. The new instance will therefore be provisioned with all of the existing data.
2. You can use [AzCopy](#) (for Azure Storage) or [AdlCopy](#) (for Data Lake Store) to copy the hbase folder to another Azure Storage blobs container or Data Lake Store location, and then start a new cluster with that data.

Note that this approach is very course-grained, you have to copy all of the HBase data and have no mechanisms for selecting a subset of Tables or Column Families to copy.

Export then Import

In this approach, from the source HDInsight cluster, you use the Export utility (included with HBase) to export data from a source Table that you indicate and the data is written to the default attached storage. You can then copy the export folder to the destination storage location, and run the Import utility in the context of the destination HDInsight cluster.

To export a table, you need to first SSH into the head node of your source HDInsight cluster and then run the following hbase command providing the name of your table and the location to export to in the default storage.

```
hbase org.apache.hadoop.hbase.mapreduce.Export "<tableName>" "<path>/<to>/<export>"
```

To import the table, you need to SSH into the head node of your destination HDInsight cluster and then run the following bbase command.

```
hbase org.apache.hadoop.hbase.mapreduce.Import "<tableName>" "<path>/<to>/<export>"
```

When specifying the export path, you supply paths that refer to the default storage or to any of the attached storage options, so long as you adjust to use the full path syntax. For example, in Azure Storage, this has the following form:

```
wasbs://<containername>@<accountname>.blob.core.windows.net/<path>
```

In Azure Data Lake Store, the expanded syntax has the form:

```
adl://<accountName>.azuredatalakestore.net:443/<path>
```

As illustrated, this approach offers table level granularity. You can get even more granular with the process by specifying a date range (in the form of start and end times in milliseconds since the Unix epoch) for the rows to include, which allows you to perform the process incrementally:

```
hbase org.apache.hadoop.hbase.mapreduce.Export "<tableName>" "<path>/<to>/<export>" <numberOfVersions>
<startTimeInMS> <endTimeInMS>
```

Note that you have to specify the number of versions of each row to include, so if you want all version in the data range, for specify an arbitrarily large number like 1000 that your data is not likely to exceed.

Copy Table

The CopyTable utility copies data directly from a source Table in a row by row fashion to an existing destination table with the same schema as the source, where the destination table can be on the same cluster or a different HBase cluster.

To use CopyTable, you need to SSH into the head node of your HDInsight cluster which will act as the source. Then you run the hbase command with the following syntax:

```
hbase org.apache.hadoop.hbase.mapreduce.CopyTable --new.name=<destTableName> --peer.adr=<destinationAddress>
<srcTableName>
```

If you are using CopyTable to copy to a table on the same cluster, then you can omit the peer switch. Otherwise, you need to provide the destinationAddress, which has the following form:

```
<destAddress> = <ZooKeeperQuorum>:<Port>:<ZnodeParent>
```

The `<ZooKeeperQuorum>` needs to be the comma separated list of ZooKeeper nodes, for example:

```
zk0-hdizc2.54o2oqawzlwev1fxgay2500xtg.dx.internal.cloudapp.net,zk4-
hdizc2.54o2oqawzlwev1fxgay2500xtg.dx.internal.cloudapp.net,zk3-
hdizc2.54o2oqawzlwev1fxgay2500xtg.dx.internal.cloudapp.net
```

The `<Port>` on HDInsight defaults to 2181, and the `<ZnodeParent>` is `/hbase-unsecure`. So the complete using our example quorum would be

```
zk0-hdizc2.54o2oqawzlwev1fxgay2500xtg.dx.internal.cloudapp.net,zk4-
hdizc2.54o2oqawzlwev1fxgay2500xtg.dx.internal.cloudapp.net,zk3-
hdizc2.54o2oqawzlwev1fxgay2500xtg.dx.internal.cloudapp.net:2181:/hbase-unsecure
```

See the section Manually Collecting the ZooKeeper Quorum List in this article for details on how to retrieve these values for your HDInsight cluster.

The CopyTable utility supports additional parameters that let you specify the timerange of rows to copy, as well as the subset of column families in a table to copy. To view the complete list of parameters supported by CopyTable, run CopyTable without any parameters:

```
hbase org.apache.hadoop.hbase.mapreduce.CopyTable
```

When using CopyTable, it is important to recognize that it places a processing burden on the source table as it does a scan of the content to copy over to the destination table. This may reduce your HBase cluster's performance as it executes.

[!NOTE]

For a robust script you can use to automate the copying of data between tables, see `hdi_copy_table.sh` in the [Azure HBase Utils](<https://github.com/Azure/hbase-utils/tree/master/replication>) repository on GitHub.

Manually Collecting the ZooKeeper Quorum List

When both HDInsight clusters are in the same Virtual Network the above resolution using the internal host names just works. If you are trying to use CopyTable for HDInsight clusters in two separate Virtual Networks (that are connected by a VPN Gateway), then you will need to instead provide the host IP addresses of the Zookeeper nodes in the quorum. The following section describes how to build the ZooKeeper quorum list for either situation.

To acquire the quorum host names you can run the following curl command:

```
curl -u admin:<password> -X GET -H "X-Requested-By: ambari"
"https://<clusterName>.azurehdinsight.net/api/v1/clusters/<clusterName>/configurations?type=hbase-
site&tag=TOPOLOGY_RESOLVED" | grep "hbase.zookeeper.quorum"
```

The curl command retrieves a JSON document with hbase configuration and the grep call filters the listing to just the line referring to the "hbase.zookeeper.quorum" key and value pair. The output of this command looks similar to the following:

```
"hbase.zookeeper.quorum" : "zk0-hdizc2.54o2oqawzlwev1fxgay2500xtg.dx.internal.cloudapp.net,zk4-
hdizc2.54o2oqawzlwev1fxgay2500xtg.dx.internal.cloudapp.net,zk3-
hdizc2.54o2oqawzlwev1fxgay2500xtg.dx.internal.cloudapp.net"
```

The value you need is the entire string on the right of the colon.

If you need to retrieve the IP addresses for these hosts, you can use the following curl command against each host in the previous list.

```
curl -u admin:<password> -X GET -H "X-Requested-By: ambari"  
"https://<clusterName>.azurehdinsight.net/api/v1/clusters/<clusterName>/hosts/<zookeeperHostFullName>" | grep  
"ip"
```

Where `<zookeeperHostFullName>` is the full DNS name of the ZooKeeper host, such as `zk0-hdizc2.54o2oqawzlwevlfxgay2500xtg.dx.internal.cloudapp.net`.

The output of the above command contains the IP address for the specified host and looks similar to:

```
100    "ip" : "10.0.0.9",
```

Remember that you will need to collect the IP addresses for all ZooKeeper nodes in your quorum, and then rebuild destAddress as follows:

```
<destAddress> = <Host_1_IP>,<Host_2_IP>,<Host_3_IP>:<Port>:<ZnodeParent>
```

For example:

```
<destAddress> = 10.0.0.9,10.0.0.8,10.0.0.12:2181:/hbase-unsecure
```

Snapshots

Snapshots enable you to take a point-in-time backup of data in your HBase datastore. They have minimal overhead and complete within seconds because a snapshot operation is effectively a metadata operation that captures the names of the files in storage relevant to that point in time. At the time of a snapshot, no actual data is copied. Snapshots take advantage of the immutable nature of the data stored in HDFS (e.g., where updates, deletes and inserts are actually represented as new data) to provide this point in time capability. You can restore the snapshot (a process referred to as cloning) on the same cluster. You can also export a snapshot to another cluster.

To create a snapshot, SSH in to the head node of your HDInsight HBase cluster and run the hbase shell.

```
hbase shell
```

Within the hbase shell, run the snapshot command providing the of the table to snapshot and a name for the snapshot:

```
snapshot '<tableName>', '<snapshotName>'
```

You can restore a snapshot by name within the hbase shell by first disabling the table, restoring the snapshot and then re-enabling the table:

```
disable '<tableName>'  
restore_snapshot '<snapshotName>'  
enable '<tableName>'
```

If you wish to restore a snapshot to a new table, you can do with clone_snapshot:

```
clone_snapshot '<snapshotName>', '<newTableName>'
```

To export a snapshot to HDFS for use by another cluster, first make sure you have created the snapshot as described previously. Then you will need to use the ExportSnapshot utility. This is run from within the SSH session to the head node, but not within the hbase shell:

```
hbase org.apache.hadoop.hbase.snapshot.ExportSnapshot -snapshot <snapshotName> -copy-to <hdfsHBaseLocation>
```

In the command the specified must refer to any of the storage locations accessible to your source cluster, and should point to the hbase folder used by your destination cluster. For example, if you had a secondary Azure Storage account attached to your source cluster that provides access to the container used by the default storage of the destination cluster, you could use a command similar to the following:

```
hbase org.apache.hadoop.hbase.snapshot.ExportSnapshot -snapshot 'Snapshot1' -copy-to  
'wasbs://secondcluster@myaccount.blob.core.windows.net/hbase'
```

Once the snapshot has been exported in this way, you should SSH into the head node of the destination cluster and restore the snapshot using the restore_snapshot command within the hbase shell as previously described.

Note that snapshots provide a complete backup of a table at the time the snapshot command is taken. They do not provide the ability to perform incremental snapshots by windows of time, nor to specify subsets of columns families to include in the snapshot.

Replication

HBase replication enables you to automatically push transactions from a source cluster to a destination cluster, using an asynchronous mechanism that has minimal overhead on the source cluster. In HDInsight, you can setup replication between clusters where:

- The source and destination clusters are in the same virtual network
- The source and destinations clusters are in different virtual networks connected by a VPN gateway, but both clusters exist in the same geographic location
- The source cluster and destinations clusters are in different virtual networks connected by a VPN gateway and each cluster exists in a different geographic location

Irrespective of the deployment topology, the general setup for replication is as follows:

1. Create the tables and populate data in the source cluster.
2. Create empty destination tables in the destination cluster that follow the same schema as the tables used in the source.
3. Register the destination cluster as a peer to the source cluster.
4. Enable replication on the desired source tables.
5. Copy existing data from the source tables to the destination tables.
6. Replication will automatically copy new data modifications as they happen on the source tables to the destination tables.

Enabling replication in this way on HDInsight is accomplished by applying a Script Action to your running source HDInsight cluster.

For a step by step walkthru of enabling replication in your cluster, or to experiment with replication on sample clusters provisioned in Virtual Networks using ARM templates, see [Configure HBase replication](#). This guide also includes instructions for enabling replication of Phoenix metadata, which is still experimental.

See also

- [Configure HBase replication](#)

Using Spark to read and write HBase data

8/16/2017 • 6 min to read • [Edit Online](#)

Apache HBase is typically queried either with its low level API of scans, gets and puts or with a SQL syntax using Phoenix. Apache Spark can be used as a convenient and performant alternative way to query and modify data stored by HBase. This is enabled by the use of the Spark HBase Connector.

This article covers how to setup your HDInsight Spark cluster so that it can query and modify data in your HDInsight HBase cluster using the Spark HBase Connector.

Deployment Environment

To begin, you will need two separate HDInsight clusters- one of the HBase cluster type and one of the Spark cluster type with Spark 2.1 (HDI 3.6) installed. The Spark cluster will need to be able to communicate directly with the HBase cluster with minimal latency, so deploying both clusters within the same Virtual Network is the recommended configuration. For instructions on how to deploy an HDInsight cluster into a Virtual Network, see [Create Linux based clusters in HDInsight using the Azure Portal](#).

This article assumes you have deployed your Spark and HDInsight cluster into one Virtual Network and that you have SSH access to both. You will also need to have access to the default storage attached to each cluster.

Overall Process

The high-level process for enabling your Spark cluster to query your HDInsight cluster is as follows:

1. Acquire the hbase-site.xml file from your HBase cluster configuration folder (/etc/hbase/conf).
2. Place a copy of hbase-site.xml in your Spark 2 configuration folder (/etc/spark2/conf).
3. Run spark-shell referencing the Spark Hbase Connector by its Maven coordinates in the packages switch.
4. Define a catalog that maps the schema from Spark to Hbase
5. Interact with the HBase data via either the RDD or DataFrame APIs.

The following sections walk thru each section in detail.

Prepare sample data in HBase

In this step, you will create a simple table in HBase with some basic content you can query using Spark.

1. Connect to the head node of your HBase cluster via SSH. For instructions on how to connect via SSH, see [Connect to HDInsight using SSH](#).
2. Run the hbase shell by executing the following command:

```
hbase shell
```

3. Create a Contacts table with the column families "Personal" and "Office":

```
create 'Contacts', 'Personal', 'Office'
```

4. Load a few sample rows of data by executing the following:

```

put 'Contacts', '1000', 'Personal:Name', 'John Dole'
put 'Contacts', '1000', 'Personal:Phone', '1-425-000-0001'
put 'Contacts', '1000', 'Office:Phone', '1-425-000-0002'
put 'Contacts', '1000', 'Office:Address', '1111 San Gabriel Dr.'
put 'Contacts', '8396', 'Personal:Name', 'Calvin Raji'
put 'Contacts', '8396', 'Personal:Phone', '230-555-0191'
put 'Contacts', '8396', 'Office:Phone', '230-555-0191'
put 'Contacts', '8396', 'Office:Address', '5415 San Gabriel Dr.'

```

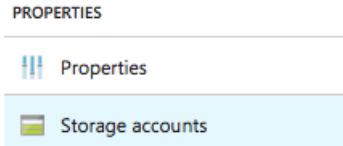
Acquire hbase-site.xml from your HBase cluster

In this step, you SSH into the head node of your HBase cluster and copy the hbase-site.xml file from local storage to your clusters default storage and download it from default storage, upload it to the default storage within your Spark cluster and copy it to the correct location in your Spark clusters local storage.

1. Connect to the head node of your HBase cluster via SSH.
2. Run the following command to copy the hbase-site.xml from local storage to the root of your HBase cluster's default storage:

```
hdfs dfs -copyFromLocal /etc/hbase/conf/hbase-site.xml /
```

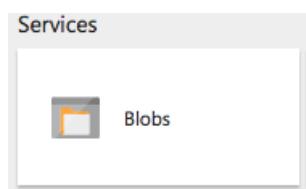
3. Navigate to your HBase cluster using the [Azure Portal](#).
4. Select Storage accounts.



5. Select the Storage account in the list that has a checkmark under the Default column.

STORAGE	CONTAINER / DIRECTORY	DEFAULT
hdizdocshbase	hdizdocshbase	✓

6. On the Storage account blade, select the Blobs tile.



7. In the list of containers, select the container that is used by your HBase cluster.

8. In the file list, select hbase-site.xml.

example	6/22/2017, 1:43:28 PM	Block blob	0 B
hbase	6/22/2017, 1:41:59 PM	Block blob	0 B
hbase-site.xml	6/22/2017, 5:59:55 PM	Block blob	7.31 KiB
HdiSamples	6/22/2017, 1:43:43 PM	Block blob	0 B
hdp	6/22/2017, 1:36:58 PM	Block blob	0 B

9. On the Blob properties panel, select Download and save it hbase-site.xml to a location on your local

machine.

A screenshot of a web browser showing the 'Blob properties' page for a file named 'hbase-site.xml'. At the top, there are 'Download' and 'Delete' buttons. Below that, the word 'NAME' is followed by 'hbase-site.xml'. There is also a small 'X' button in the top right corner of the window.

Place hbase-site.xml on your Spark cluster

In this step you upload the hbase-site.xml to the default storage within your Spark cluster and copy it to the correct location in your Spark clusters local storage.

1. Navigate to your Spark cluster using the [Azure Portal](#).
2. Select Storage accounts.

A screenshot of the 'Storage accounts' section in the Azure portal. It shows a list of storage accounts. The account 'hdizdocshbase' is selected, indicated by a blue background and a checkmark in the 'Default' column.

3. Select the Storage account in the list that has a checkmark under the Default column.

STORAGE	CONTAINER / DIRECTORY	DEFAULT
hdizdocshbase	hdizdocshbase	✓

4. On the Storage account blade, select the Blobs tile.

A screenshot of the 'Blobs' section in the Azure portal. It shows a list of containers. The container 'spark-hdiz-docs-2017-06-22t20-41-27-790z' is selected, indicated by a blue background.

5. In the list of containers, select the container that is used by your Spark cluster.

6. Select upload.

A screenshot of the 'Upload' interface for a container. The container name 'spark-hdiz-docs-2017-06-22t20-41-27-790z' is shown at the top. Below it are buttons for 'Upload', 'Refresh', 'Delete container', 'Properties', and 'Access policy'. The 'Upload' button is highlighted with a blue background.

7. Choose the hbase-site.xml file you previously downloaded to your local machine.

A screenshot of the 'Upload blob' interface. At the top, it says 'Upload blob'. Below that is a 'Files' section with a single file 'hbase-site.xml'. Underneath, there are dropdown menus for 'Blob type' (set to 'Block blob') and 'Block size' (set to '100 MB').

8. Select Upload.
9. Connect to the head node of your Spark cluster via SSH.
10. Run the following command to copy hbase-site.xml from your Spark cluster's default storage to the Spark 2 conf folder on the cluster's local storage:

```
sudo hdfs dfs -copyToLocal /hbase-site.xml /etc/spark2/conf
```

Run Spark Shell referencing the Spark HBase Connector

In this step you launch an instance of Spark Shell that references the Spark HBase Connector.

1. Connect to the head node of your Spark cluster via SSH.
2. Run the following command, note the packages switch which references the Spark HBase Connector.

```
spark-shell --packages com.hortonworks:shc-core:1.1.0-2.1-s_2.11
```

3. Keep this Spark Shell instance open and continue to the next step.

Define a Catalog and Query

In this step you define a catalog object that maps the schema from Spark to HBase.

1. Within your open Spark Shell, run the following import statements

```
import org.apache.spark.sql.{SQLContext, _}
import org.apache.spark.sql.execution.datasources.hbase._
import org.apache.spark.{SparkConf, SparkContext}
import spark.sqlContext.implicits._
```

2. Run the following to define a catalog for the Contacts table you create in HBase. In the belowm you defines a schema for the HBase table with name Contacts, identify the row key as key, and map the column names as they will be used in Spark to the column family, column name and column type as they appear in HBase. Note that the rowkey also has to be defined in details as a named column (rowkey), which has a specific column family, cf, of rowkey.

```
def catalog = s"""
  | "table": {"namespace": "default", "name": "Contacts"},
  | "rowkey": "key",
  | "columns": {
  |   "rowkey": {"cf": "rowkey", "col": "key", "type": "string"},
  |   "officeAddress": {"cf": "Office", "col": "Address", "type": "string"},
  |   "officePhone": {"cf": "Office", "col": "Phone", "type": "string"},
  |   "personalName": {"cf": "Personal", "col": "Name", "type": "string"},
  |   "personalPhone": {"cf": "Personal", "col": "Phone", "type": "string"}
  | }
  | """ .stripMargin
```

3. Run the following to define a method that will provide a DataFrame around your Contacts table in HBase:

```

def withCatalog(cat: String): DataFrame = {
    spark.sqlContext
    .read
    .options(Map(HBaseTableCatalog.tableCatalog->cat))
    .format("org.apache.spark.sql.execution.datasources.hbase")
    .load()
}

```

4. Create an instance of the DataFrame by running:

```
val df = withCatalog(catalog)
```

5. Query the DataFrame by running:

```
df.show()
```

6. You should see your two rows of data in output similar to the following:

rowkey	officeAddress	officePhone	personalName	personalPhone
1000	1111 San Gabriel Dr.	1-425-000-0002	John Dole	1-425-000-0001
8396	5415 San Gabriel Dr.	230-555-0191	Calvin Raji	230-555-0191

7. Next, register a temp table so you can query the HBase table use Spark SQL:

```
df.registerTempTable("contacts")
```

8. Issue a SQL query against the contacts table:

```
val query = spark.sqlContext.sql("select personalName, officeAddress from contacts")
query.show()
```

9. You should see results similar to the following:

personalName	officeAddress
John Dole	1111 San Gabriel Dr.
Calvin Raji	5415 San Gabriel Dr.

Insert new data

1. Next, insert a new Contact record. To do so, define the ContactRecord class:

```
case class ContactRecord(  
    rowkey: String,  
    officeAddress: String,  
    officePhone: String,  
    personalName: String,  
    personalPhone: String  
)
```

2. Create an instance of ContactRecord and save it within an array:

```
val newContact = ContactRecord("16891", "40 Ellis St.", "674-555-0110", "John Jackson", "230-555-0194")  
  
var newData = new Array[ContactRecord](1)  
newData(0) = newContact
```

3. Save the array of new data to HBase:

```
sc.parallelize(newData).toDF.write  
.options(Map(HBaseTableCatalog.tableCatalog -> catalog))  
.format("org.apache.spark.sql.execution.datasources.hbase").save()
```

4. Examine the results:

```
df.show()
```

5. You should have output similar to the following:

```
+-----+-----+-----+-----+  
|rowkey| officeAddress| officePhone|personalName| personalPhone|  
+-----+-----+-----+-----+  
| 1000|1111 San Gabriel Dr.|1-425-000-0002| John Dole|1-425-000-0001|  
| 16891|        40 Ellis St.| 674-555-0110|John Jackson| 230-555-0194|  
| 8396|5415 San Gabriel Dr.| 230-555-0191| Calvin Raji| 230-555-0191|  
+-----+-----+-----+-----+
```

Next Steps:

- Read more about the [Spark HBase Connector](#) and view the source on GitHub.

Monitoring HBase with Operations Management Suite (OMS)

8/15/2017 • 5 min to read • [Edit Online](#)

The HDInsight HBase Monitoring solution is a management solution for Azure Log Analytics that collects important HDInsight HBase performance metrics from your HDInsight Cluster nodes and provides the tools to search the metrics, as well HBase-specific visualizations and dashboards. By using the metrics that you collect with the solution, you can create custom monitoring rules and alerts. You can monitor the metrics for multiple HDInsight HBase clusters across multiple Azure subscriptions.

Log Analytics is a service in [Operations Management Suite \(OMS\)](#) that monitors your cloud and on-premises environments to maintain their availability and performance. It collects data generated by resources in your cloud and on-premises environments and from other monitoring tools to provide analysis across multiple sources.

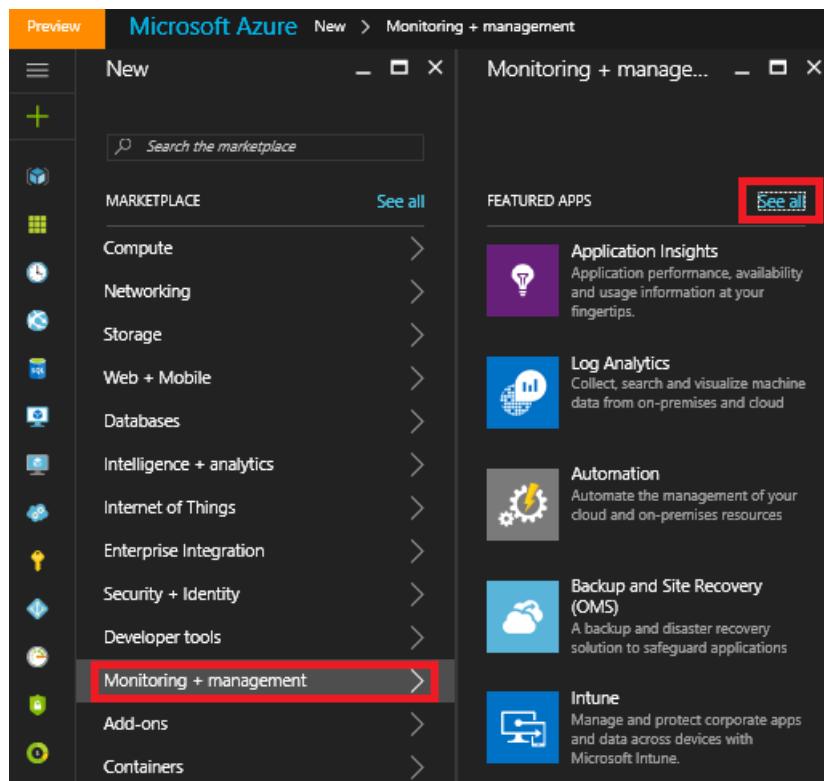
[Management solutions](#) add functionality to OMS, providing additional data and analysis tools to Log Analytics. Log Analytics management solutions are a collection of logic, visualization, and data acquisition rules that provide metrics pivoted around a particular area. They may also define new record types to be collected that can be analyzed with Log Searches or by additional user interface provided by the solution in the dashboard.

[Insight & Analytics](#) is built on the underlying Log Analytics platform. You can choose to use the Log Analytics capabilities and pay per GB ingested into the service or switch your workspace to the Insight & Analytics tier and pay per node managed by the service. Insight & Analytics offers a superset of the capabilities offered under Log Analytics. The HBase Monitoring solution is available to either Log Analytics or Insight & Analytics.

When you provision the HDInsight HBase Monitoring solution, you will create an OMS workspace. You can think of the workspace as a unique Log Analytics environment with its own data repository, data sources, and solutions. You may create multiple workspaces in your subscription to support multiple environments such as production and test.

Provision the HDInsight HBase Monitoring management solution

1. Sign in to the [Azure portal](#) using your Azure subscription.
2. In the **New** blade under **Marketplace**, select **Monitoring + management**.
3. In the **Monitoring + management** blade, click **See all**.

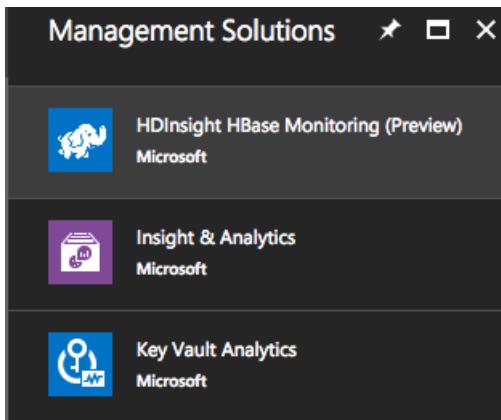


4. In the listing, look for the **Management Solutions** band. To the right of **Management Solutions**, click **More**.

This screenshot shows the 'Monitoring + management' blade. At the top, there's a search bar labeled 'Search Monitoring + management'. Below it is a large promotional card for 'Log Analytics' by Microsoft, which collects, searches, and visualizes machine data from on-premises and cloud. A 'Create' button is visible at the bottom of this card. Under the heading 'Recommended', there are six icons: Insight & Analytics (Microsoft), Automation & Control (Microsoft), Security & Compliance (Microsoft), Automation (Microsoft), Scheduler (Microsoft), and Intune (Microsoft). At the bottom, under 'Management Solutions', there are six more icons: Activity Log Analytics (Microsoft), Update Management (Microsoft), Azure Networking Analytics (Preview) (Microsoft), Application Insights (Microsoft), Key Vault Analytics (Preview) (Microsoft), and Change Tracking (Microsoft). A 'More' button is located at the far right of this section, highlighted with a red box.

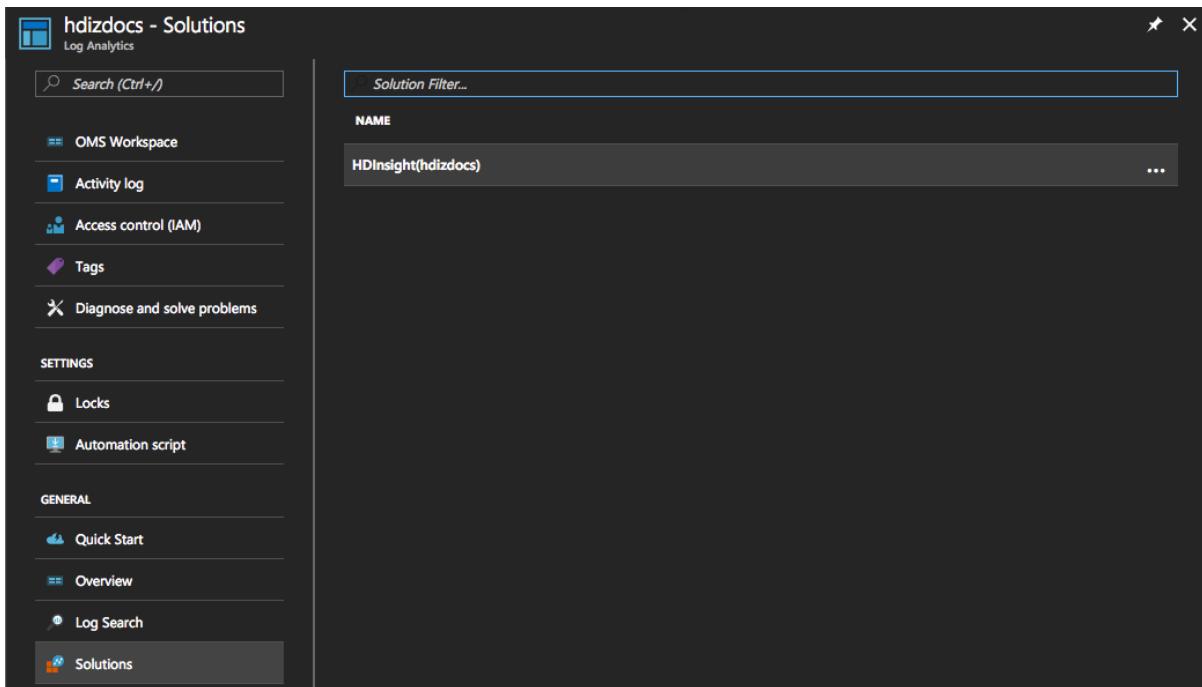
5. In the **Management Solutions** blade, select the HDInsight HBase Monitoring management solution that

you want to add to a workspace.

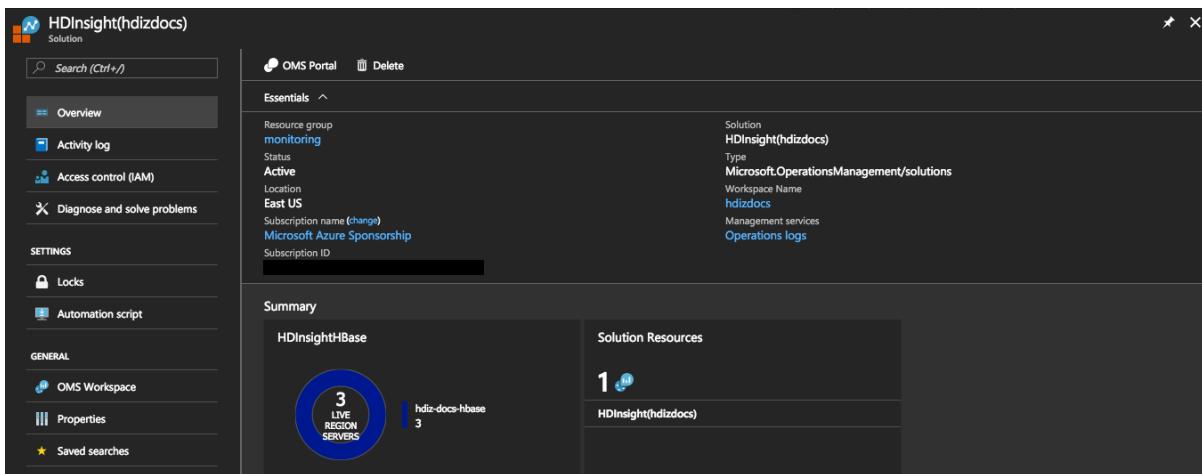


6. In the management solution blade, review information about the management solution, and then click **Create**.
7. In the *management solution name* blade, select an existing workspace that you want to associate with the management solution or create a new OMS workspace and then select it.
8. Change workspace settings for the Azure subscription, resource group, and location as appropriate.

9. Select **Create**.
10. To start using the management solution that you've added to your workspace, navigate to **Log Analytics** > **workspace name** > **Solutions**. An entry for your management solution is displayed in the list. Click the entry to navigate to the solution.



11. The blade for your HDInsight HBase monitoring solution should appear.



12. At this point, your Summary tiles will not show any data because you have yet to configure your HDInsight HBase cluster to send data to Log Analytics.

Connecting an HDInsight HBase cluster to Log Analytics

Before you can use the tools provided by the HDInsight HBase Monitoring solution, you need to configure your cluster so that it transmits the metrics from its region server, head nodes and ZooKeeper nodes to Log Analytics. This is accomplished by running a Script Action against your HDInsight HBase cluster.

Acquire Your OMS Workspace ID and Workspace Key

You will need your OMS Workspace ID and Workspace Key to enable the nodes in your cluster to authenticate with Log Analytics. To acquire these values, follow these steps:

- From your HBase Monitoring blade in the Azure Portal, select Overview.

The screenshot shows the Azure portal interface for the 'HDInsight(hdizdocs)' solution. On the left, there's a navigation sidebar with sections like Overview, Activity log, Access control (IAM), Diagnose and solve problems, SETTINGS (Locks, Automation script), GENERAL (OMS Workspace, Properties, Saved searches), and more. The main content area displays the 'Essentials' tab for the 'monitoring' resource group. It shows the solution name 'HDInsight(hdizdocs)', type 'Microsoft.OperationsManagement/solutions', workspace name 'hdizdocs', management services 'Operations logs', and subscription information. Below this is a 'Summary' section for 'HDInsightHBase', which lists '3 LIVE REGION SERVERS' and '1 HDInsight(hdizdocs)'.

2. Select OMS Portal. This will launch the OMS Portal in a new browser tab or window.

The screenshot shows the Azure portal interface with the 'HDInsight(hdizdocs)' solution selected. The left sidebar shows the 'Overview' section is currently active. The main content area displays the 'Essentials' tab for the 'Resource group'. It includes a search bar, an 'OMS Portal' button, a 'Delete' button, and a summary card for the 'Resource group'.

3. On the OMS Portal Home, select Settings.

The screenshot shows the Microsoft Operations Management Suite (OMS) portal home screen. The top navigation bar says 'Microsoft Operations Management Suite'. The left sidebar has icons for Home, Add, Find, Log Search, My Dashboard, Solutions Gallery, Usage, and Settings. The main content area features a 'Find...' search bar, a 'Log Search' section with a magnifying glass icon, a 'My Dashboard' section with a grid icon, a 'Solutions' section with a gear icon, and a 'Usage' section with a bar chart icon. A large blue callout on the right says 'Settings' with '8' data sources connected and '1 NEW' solutions. A banner at the top says 'Try the free Microsoft OMS mobile app for managing your data'.

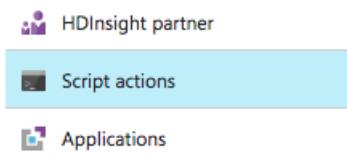
4. Select Connected Source, Linux Servers.

- Take note of the Workspace ID and Primary Key values displayed, these are the Workspace ID and Workspace Key values you will need for the Script Action.

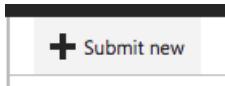
Run the Script Action

To enable data collection from your HDInsight HBase cluster, you need to run a Script Action against all the nodes in the cluster. Follow these steps to do so:

1. Navigate to the blade for your HDInsight HBase cluster in the Azure Portal.
2. Select Script Actions.



3. Select Submit New.



4. In the Submit script action, set the Script type to "- Custom".
5. Provide a name for this script.
6. For the Bash Script URI, paste in the following URI:

```
https://raw.githubusercontent.com/hdinsight/HDIInsightOMS/master/monitoring/script2.sh
```

7. For the Node types, select all three (Head, Region, ZooKeeper).
8. In the Parameters text box, enter your Workspace ID and your Workspace Key, enclosing each value in quotes and separating the two quoted values with a space.

```
"WorkspaceID" "WorkspaceKey"
```

For example:

```
"481506d5-f04e-4901-afa6-0a688232a1c1"
"bQCW1P27febK2k/S/+70jxgap2A2HTUU9V1YHE7nfW8uR31XZx30EzJvnVNvOBo7pe+W5+ahn/my6JDtIufcg=="
```

9. Select Persist this script action to rerun when new nodes are added to the cluster.

Submit script action X

Script type ([learn more](#))

- Custom ▼

* Name

Enable HBase OMS Monitoring ✓

* Bash script URI

<https://raw.githubusercontent.com/hdinsight...> ✓

Node type(s):

Head

Region

Zookeeper

Parameters

```
"481506d5-f04e-4907-afa6-0a688232a1c5" ✓
"bQCW1P27febK2k/S/+70jxgap2A2HTUU9V
1YHE7nfW8uR31XZx3OEzJvnVNvOBo7pe+
W5+ahn/mv6DtTlufcg=="
```

Persist this script action to rerun when new nodes are added to the cluster.

10. Select Create.

11. The Script Action will take a few minutes to run. You can monitor its status from Script Actions blade.

SCRIPT ACTION HISTORY

NAME	DATE	CONTEXT
Enable HBase OMS Monitoring		Submitted by user

12. When the Script Action completes, you should see a green checkmark next to the script name in the listing.

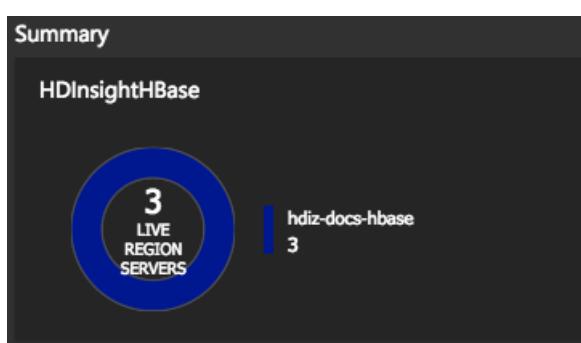
SCRIPT ACTION HISTORY

NAME	DATE	CONTEXT
Enable HBase OMS Monitoring	6/25/2017	Submitted by user

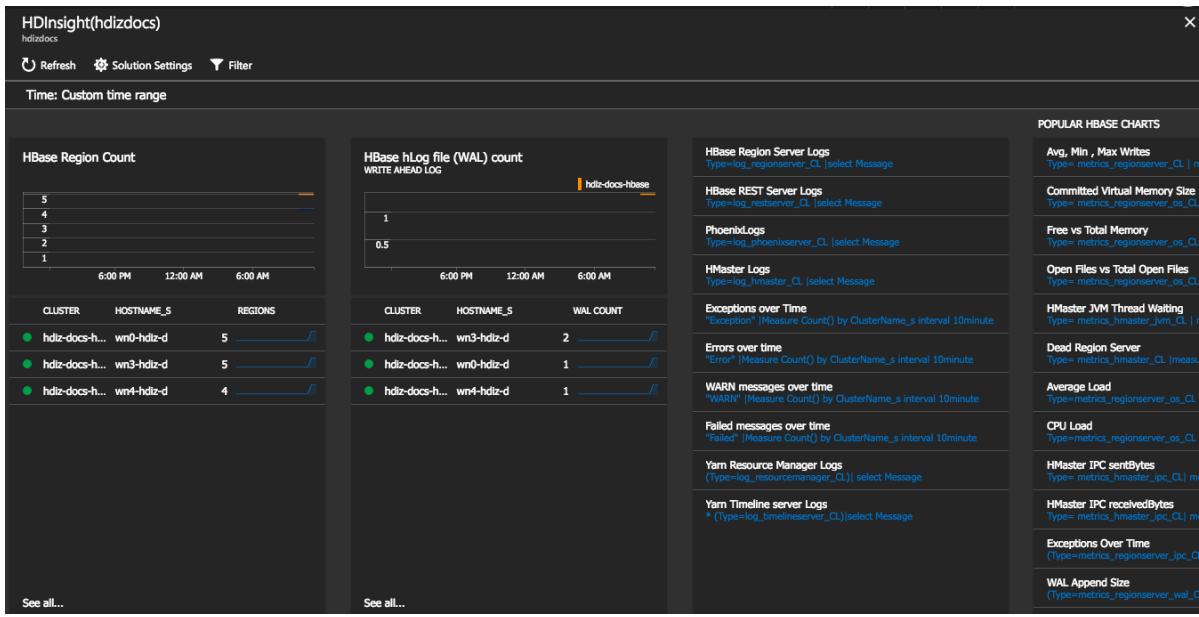
View the HDInsight HBase Monitoring Solution

After the Script Action completes, you should start to see data in the Monitoring Solution within a few minutes.

1. Within the Azure Portal, navigate to blade for your HDInsight HBase solution.
2. Select the Overview tab.
3. Under Summary, you should see a tile indicating the count of Region Servers that are being monitored.



- Select the tile with the region server count. The main dashboard will be displayed.
- This Dashboard provides access to statscs about the Regions, the Write-Ahead-Log (WAL) count in use, a collection of Log Analytics searches (e.g., for Region Server Logs, Phoenix logs, Exceptions, etc.) and a collection of Popular charts that provide visualization of relevant metrics.



- Selecting any one of these will drill down into the Log Search view where you can refine the query and explore the data in more detail.

See Also

- You can create alerts against metrics collected by the HDInsight HBase Monitoring management solution, see [Creating Alerts](#) for step by step instructions.
- Learn more about how to conduct [Log Searches](#).

HBase - Migrating to a New Version

8/16/2017 • 5 min to read • [Edit Online](#)

The process of upgrading HDInsight clusters is [straightforward for most cluster types](#), such as Spark and Hadoop. Since these are job-based clusters, the steps to conduct are to back up transient (locally stored) data, delete the existing cluster, create a new cluster in the same VNET subnet, import transient data, and start jobs/continue processing on the new cluster.

Since HBase is a database, there are additional steps one must take in order to upgrade. The general steps leading up to the actual upgrade workflow remain the same, such as planning and testing. This article covers the additional steps required for a successful HBase upgrade with minimal downtime.

Review HBase compatibility

One crucial step to take before upgrading HBase is to review any incompatibilities between major and minor versions of HBase. The steps following this section only work if there are no version compatibility issues between the source and destination clusters. We **highly recommend that you review the [HBase book](#) before undertaking an upgrade**, specifically the compatibility matrix contained within.

Here is an example of the compatibility matrix:

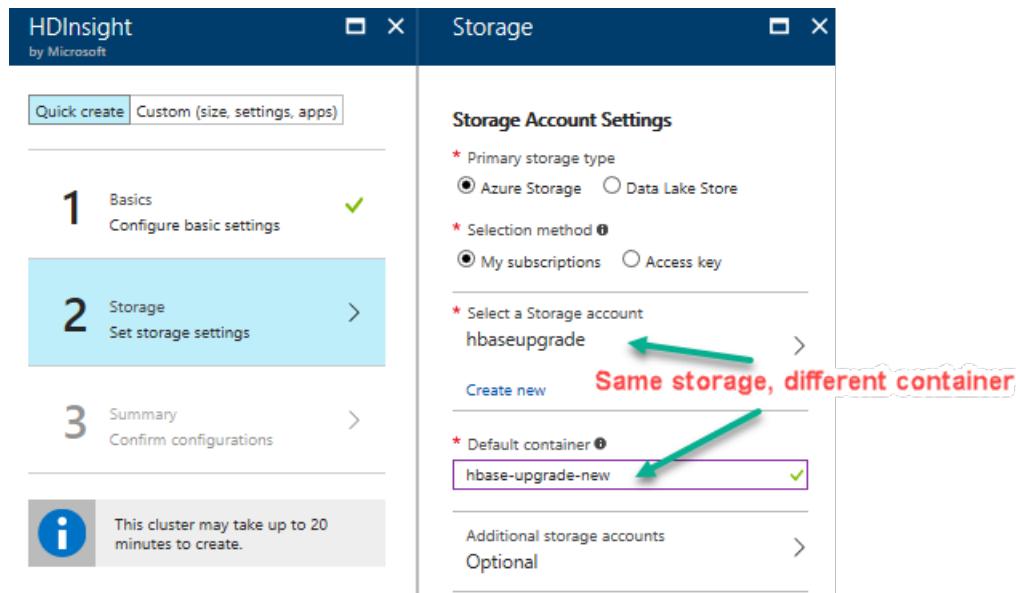
COMPATIBILITY TYPE	MAJOR	MINOR	PATCH
Client-Server wire Compatibility	N	Y	Y
Server-Server Compatibility	N	Y	Y
File Format Compatibility	N	Y	Y
Client API Compatibility	N	Y	Y
Client Binary Compatibility	N	N	Y
Server-Side Limited API Compatibility			
Stable	N	Y	Y
Evolving	N	N	Y
Unstable	N	N	N
Dependency Compatibility	N	Y	Y
Operational Compatibility	N	N	Y

Please note that the above indicates what *could* break, not necessarily what *will* break. Specific breaking changes should be outlined within the version release notes.

HDInsight upgrade with same HBase major version

The following scenario is for upgrading from HDInsight 3.4 to 3.6 with the same HBase major version. The same general steps can be followed when upgrading other version numbers, provided no compatibility issues between versions.

1. Make sure that your application works with the new version. This can be done by first checking the compatibility matrix and release notes, as outlined in the previous section. You may also test your application in a cluster running the target version of HDInsight and HBase.
2. Create a [new HDInsight cluster](#), using the same storage account, but with a different container name.



3. Flush your source HBase cluster. This is the cluster from which you are upgrading. Run the following script, the latest version of which can be found on [GitHub](#):

```
#!/bin/bash

#-----
# SCRIPT TO FLUSH ALL HBASE TABLES.
#-----#
```

LIST_OF_TABLES=/tmp/tables.txt
HBASE_SCRIPT=/tmp/hbase_script.txt
TARGET_HOST=\$1

usage ()
{
 if [["\$1" == "-h"]] || [["\$1" == "--help"]]
 then
 cat << ...

Usage:
\$0 [hostname]

Note: Providing hostname is optional and not required when script
is executed within HDInsight cluster with access to 'hbase shell'.

However host name should be provided when executing the script as
script-action from HDInsight portal.

For Example:

1. Executing script inside HDInsight cluster (where 'hbase shell' is accessible):

```

$0

[No need to provide hostname]

2. Executing script from HDinsight Azure portal:

Provide Script URL.

Provide hostname as a parameter (i.e. hn0, hn1 or wn2 etc.)..

...
exit
fi
}

validate_machine ()
{
THIS_HOST=`hostname` 

if [[ ! -z "$TARGET_HOST" ]] && [[ $THIS_HOST != $TARGET_HOST* ]]
then
    echo "[INFO] This machine '$THIS_HOST' is not the right machine ($TARGET_HOST) to execute the script."
    exit 0
fi
}

get_tables_list ()
{
hbase shell << ... > $LIST_OF_TABLES 2> /dev/null
list
exit
...
}

add_table_for_flush ()
{
TABLE_NAME=$1
echo "[INFO] Adding table '$TABLE_NAME' to flush list..."
cat << ... >> $HBASE_SCRIPT
    flush '$TABLE_NAME'
...
}

clean_up ()
{
rm -f $LIST_OF_TABLES
rm -f $HBASE_SCRIPT
}

#####
# MAIN #
#####

usage $1

validate_machine

clean_up

get_tables_list

START=false

while read LINE
do
    if [[ $LINE == TABLE ]]
    then
        START=true
        continue
    fi
    if $START
    then
        echo $LINE
    fi
done
```

```

elif [[ $LINE == *row*in*seconds ]]
then
    break
elif [[ $START == true ]]
then
    add_table_for_flush $LINE
fi

done < $LIST_OF_TABLES

cat $HBASE_SCRIPT

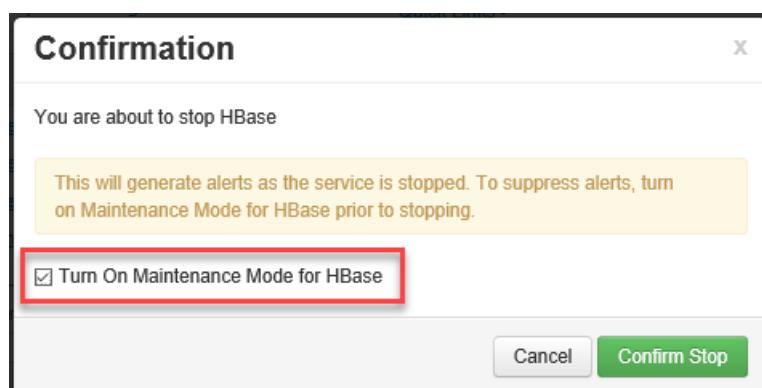
hbase shell $HBASE_SCRIPT << ... 2> /dev/null
exit
...

```

When you write data to HBase, it is first written to an in-memory store, called a *memstore*. Once the memstore reaches a certain size, it is flushed to disk for durability. This long-term storage is the cluster's storage account. Since we will be deleting the old cluster, the memstores will go away, potentially losing data. The above script manually flushes the memstore for each table for long-term retention.

1. Stop ingestion to the old HBase cluster.
2. Flush the cluster again with the above script. This will ensure that any remaining data in the memstore is flushed. The flushing process will be faster this time around, because most of the memstore has already been flushed in the previous step.
3. Log on to Ambari ([HTTPS://CLUSTERNAME.azurehdinsight.net](https://CLUSTERNAME.azurehdinsight.net), where CLUSTERNAME is the name of your old cluster) and stop the HBase services. When prompted to confirm that you'd like to stop the services, check the box to turn on maintenance mode for HBase. See [Manage HDInsight clusters by using the Ambari Web UI](#) for more information on connecting to and using Ambari.

The screenshot shows the Ambari web interface for an HBase cluster. The top navigation bar includes links for Ambari, hbase-upgr..., 0 ops, 0 alerts, Dashboard, Services (with a red circle '1'), Hosts, Alerts, Admin, and a user icon. The sidebar on the left lists various services: HDFS, MapReduce2, YARN, Tez, Hive, HBase (with a red circle '2'), Pig, Snoop, Oozie, ZooKeeper, Ambari Metrics, and SmartSense. The main content area displays the 'Summary' and 'Metrics' tabs for the HBase service. The 'Summary' tab shows Active HBase Master (Started), Standby HBase Master (Started), and RegionServers (4/4 Live). The 'Metrics' tab shows Read and Write latencies and Open Connections. On the right, a 'Service Actions' dropdown is open, with a red circle '3' highlighting the 'Stop' option.



4. Log in to Ambari for the **new HDInsight cluster**. We need to change the `fs.defaultFs` HDFS setting to

point to the container name used by the original cluster. This setting can be found under HDFS -> Configs -> Advanced -> Advanced core-site.

The screenshot shows the Ambari web interface for managing a Hbase-upgrade-new cluster. The top navigation bar includes links for Ambari, the cluster name, and tabs for Dashboard, Services (with a red notification badge), Hosts, and Alerts. The main content area is titled 'Configs' (also with a red badge). On the left, a sidebar lists various services: HDFS (selected), YARN, MapReduce2, Tez, Hive, HBase (highlighted with a green checkmark), Pig, Sqoop, Oozie, ZooKeeper, and Ambari Metrics. Below the sidebar is an 'Actions' button. The central panel displays configuration groups V2 and V1, both updated 6 minutes ago for HDP-2.6. A dropdown menu allows switching between groups. The 'Advanced' tab is selected under the 'NameNode' section, which lists 'NameNode hosts' as hn0-hbase.j40lt5354owehbgbyrlkjabrka.bx.internal.cloudapp.net and 1 other'. A slider for 'NameNode new generation size' is set to 200 MB. At the bottom, a section for 'Advanced core-site' configurations shows 'fs.defaultFS' set to wasb://hbase-upgrade-new@hbaseupgrade.blob.core.windows.net. A red box highlights the 'wasb://' part of the URL.

5. Save your changes and restart all required services (Ambari will indicate which services require restart)

6. Point your application to the new cluster.

Avoid having your applications rely on a static DNS name for your cluster by hard-coding it, as it will change when upgrading. There are two options we recommend to mitigate this issue: either configure a CNAME in your domain name's DNS settings that points to the cluster's name, or use a configuration file for your application that you can update without redeploying.

1. Start the ingestion to see if everything is functioning as normal.
2. Delete the original cluster after you verify everything is working as expected.

The time savings for this process can be significant versus not following this approach. In our tests, the downtime for this approach has been minimal, bringing it down from hours to minutes. This downtime is caused by the steps to flush the memstore, then the time to configure and restart the services on the new cluster. Your results will vary, depending on the number of nodes, amount of data, and other variables.

Next steps

In this article, we covered the steps necessary to upgrade an HBase cluster. Learn more about HBase and upgrading HDInsight clusters by following the links below:

- Learn how to [upgrade other HDInsight cluster types](#)
- Learn more about [connecting to and using Ambari](#) to manage your clusters
- Read in-depth information about [changing Ambari configs](#), including settings to optimize your HBase and other HDInsight clusters
- Learn about the [various Hadoop components available with HDInsight](#)

title: Introduction: Phoenix in HDInsight - Azure HDInsight | Microsoft Docs description: " services: hdinsight documentationcenter: "

tags: azure-portal keywords: HBase,phoenix,sql

Introduction: Phoenix in HDInsight

Apache Phoenix is an open source, massively parallel relational database layer over [HBase](#). It allows you to use SQL-like queries over HBase. It uses JDBC drivers underneath to enable users to create, delete, alter SQL tables, indexes, views and sequences, upsert rows individually and in bulk. It uses noSQL native compilation instead of using MapReduce to compile queries enabling the creation of low latency applications on top of HBase. To do this, Phoenix adds coprocessors to support running client-supplied code in the address space of the server, executing the code colocated with the data. This minimizes client/server data transfer.

Like other tools used to query big data using SQL-like syntax, such as [Hive](#) and [Spark SQL](#), Apache Phoenix opens up big data queries to non-developers who do not want to learn a new programming language just to work with data. The benefit to developers is writing highly performant queries with much less code. Unlike Hive and Spark SQL, Phoenix is highly optimized for HBase, providing powerful features and ease-of-use over MapReduce.

When you submit a SQL query, Phoenix compiles the query to HBase native calls and runs the scan or plan in parallel for optimization. This layer of abstraction takes away the focus (and requirement) of the developer from writing MapReduce jobs, bearing in mind big data fundamentals, and allows them instead to focus on the business logic and the workflow of their application around the big data storage Phoenix gives them access to.

Query performance optimization and other features

Apache Phoenix adds several performance enhancements, as well as some powerful features, to your HBase queries.

Secondary indexes

HBase has a single index which is lexicographically sorted on the primary row key. As such, records can only be accessed through the row key. Accessing records through any column other than the row key, requires scanning all of the data and applying the required filter. In a secondary index, the column or expressions indexed form an alternate row key, allowing lookups and range scans on the index.

You can create a secondary index using the `CREATE INDEX` command:

```
CREATE INDEX ix_purchasetype on SALTEDWEBLOGS (purchasetype, transactiondate)INCLUDE(bookname,quantity);
```

This typically yields a significant performance increase over executing queries without an index. Such secondary indexes are known as a **covering index**, wherein the index contains all of the columns included in the query. Therefore, the table lookup is not required and the index satisfies the entire query.

Views

Phoenix views provide a nice way to overcome one of HBase's limitations; the performance degradation experienced when you create more than around 100 physical tables. Views help in this regard by enabling multiple virtual tables to share the same underlying physical HBase table.

Creating views is very similar to using standard SQL view syntax. One primary difference is that you can define additional columns for your view in addition to the columns inherited from its base table. You may also optionally add new `KeyValue` columns.

For example, let's suppose we have a physical table named `product_metrics` with the following definition:

```
CREATE TABLE product_metrics (
    metric_type CHAR(1),
    created_by VARCHAR,
    created_date DATE,
    metric_id INTEGER
    CONSTRAINT pk PRIMARY KEY (metric_type, created_by, created_date, metric_id));
```

We can define a view over top of this table, that adds additional columns:

```
CREATE VIEW mobile_product_metrics (carrier VARCHAR, dropped_calls BIGINT) AS
SELECT * FROM product_metrics
WHERE metric_type = 'm';
```

Additional columns may be added post-creation with an `ALTER VIEW` statement.

Skip Scan

Skip scan uses one or more columns of a composite index to find distinct values. This implements intra-row scanning as opposed to Range Scan, yielding [improved performance](#). While scanning, the first matched value is skipped along with the index until it finds the next value.

The skip scan leverages the `SEEK_NEXT_USING_HINT` enum of the HBase filter. Using this, the skip scan keeps track of which set of keys or ranges of keys are being searched for in each column. It then takes a key that was passed to it during filter evaluation, and figures out whether it is one of the combinations. If not, it evaluates the next highest key to which to jump.

Transactions

While HBase provides row-level transactions, Phoenix integrates with [Tephra](#) to add cross-row and cross-table transaction support with full [ACID](#) semantics.

As with traditional SQL transactions, transactions provided through the Phoenix transaction manager allow you to ensure an atomic unit of data is successfully upserted, rolling back the transaction if the upsert operation fails on any transaction-enabled table.

To create a new table with transactions enabled, use the `TRANSACTIONAL=true` property in your `CREATE` statement:

```
CREATE TABLE my_table (k BIGINT PRIMARY KEY, v VARCHAR) TRANSACTIONAL=true;
```

If you wish to alter an existing table to be transactional, use the same property in an `ALTER` statement:

```
ALTER TABLE my_other_table SET TRANSACTIONAL=true;
```

Be aware that you cannot switch a transactional table back to being non-transactional.

Follow [these steps](#) to enable Phoenix transactions.

Salted Tables

A common issue when writing records with sequential keys to HBase is known as RegionServer Hotspotting. Though you may have multiple region servers in your cluster, your writes are all occurring on just one. This creates

the hotspotting issue where, instead of your write workload being distributed across all of the available region servers, just one is handling the load. Since each region has a pre-defined maximum size, when a region reaches that size limit, it is split into two small regions. When that happens, one of these new regions takes all new records, becoming the new hotspot victim.

To mitigate this problem, achieving better performance, we can pre-split tables in a way that all of the region servers are equally used. Phoenix provides us with Salted tables, where it transparently adds the salting byte to the row key for a particular table. The table is pre-split on the salt byte boundaries to ensure equal load distribution among region servers during the initial phase of the table. This distributes the write workload across all of the available region servers, thus improving the write and read performance. This is achieved by specifying the `SALT_BUCKETS` table property at table creation time.

Example:

```
CREATE TABLE Saltedweblogs (
    transactionid varchar(500) Primary Key,
    transactiondate Date NULL,
    customerid varchar(50) NULL,
    bookid varchar(50) NULL,
    purchasetype varchar(50) NULL,
    orderid varchar(50) NULL,
    bookname varchar(50) NULL,
    categoryname varchar(50) NULL,
    invoice number varchar(50) NULL,
    invoicestatus varchar(50) NULL,
    city varchar(50) NULL,
    state varchar(50) NULL,
    paymentamount DOUBLE NULL,
    quantity INTEGER NULL,
    shippingamount DOUBLE NULL) SALT_BUCKETS=4;
```

Enabling and tuning Phoenix through Ambari

When you provision an HDInsight HBase cluster, you gain access to [Ambari](#) to easily make configuration changes.

To enable or disable Phoenix, and to control Phoenix's query timeout settings, log in to Ambari Web UI (https://YOUR_CLUSTER_NAME.azurehdinsight.net) using your Hadoop user credentials specified during cluster creation.

Once logged in, select **HBase** from the list of services in the left-hand menu, then select the **Configs** tab.

The screenshot shows the Ambari Web UI interface. On the left, there is a sidebar with several service icons: HDFS, YARN, MapReduce2, Tez, Hive, HBase, Pig, and Sqoop. The 'HBase' icon is highlighted with a red box. In the main content area, there are three tabs: 'Summary', 'Heatmaps', and 'Configs'. The 'Configs' tab is also highlighted with a red box. Below the tabs, there are buttons for 'Group', 'Default (9)', and 'Manage Co...'. There is a comparison section showing two configurations: 'V2' (internal, 2 days ago, HDP-2.6) and 'V1'. At the bottom, there are buttons for 'X', 'V2', and a checked 'internal authored on T...' button.

Find the **Phoenix SQL** configuration section to enable/disable phoenix, as well as set the query timeout in minutes and seconds.

Phoenix SQL

Enable Phoenix

Enabled

Phoenix Query Timeout

<input type="button" value="<"/> 1 <input type="button" value=">"/>	<input type="button" value="<"/> 0 <input type="button" value=">"/>
Minutes	Seconds

Next steps

In this article, we introduced Phoenix, and how it can help improve both development and query execution efficiency when using an HBase HDInsight cluster.

- Get some hands-on experience with Phoenix: [Use Apache Phoenix with Linux-based HBase clusters in HDInsight](#)

Use Apache Phoenix with HBase clusters in HDInsight

8/16/2017 • 5 min to read • [Edit Online](#)

If you want to query HBase using SQL instead of scans, you can by using [Apache Phoenix](#) in HDInsight. You use the SQLLine utility to submit SQL to Phoenix and view the results.

For more information about Phoenix, see [Phoenix in 15 minutes or less](#) and [Phoenix in HDInsight](#) for an overview.

For information on the Phoenix grammar, see [Phoenix Grammar](#).

NOTE

For the Phoenix version information in HDInsight, see [What's new in the Hadoop cluster versions provided by HDInsight?](#).

Use SQLLine to create a new HBase table

[SQLLine](#) is a command-line utility to execute SQL.

Prerequisites

Before you can use SQLLine, you must have the following:

- **An HBase cluster in HDInsight.** For information on provisioning an HBase cluster, see [Get started with Apache HBase in HDInsight](#).
- **Connect to the HBase cluster via SSH.** For instructions, see [Connect to HDInsight using SSH](#).

To query HBase, you SSH into the head node and then run queries. These queries need to communicate with the ZooKeeper quorum.

Locating your ZooKeeper host names

1. Open Ambari by browsing to <https://<ClusterName>.azurehdinsight.net>.
2. Enter the HTTP (cluster) username and password to login.
3. Click **ZooKeeper** from the left menu. You see three **ZooKeeper Servers** listed.
4. Click one of the **ZooKeeper Servers** listed. On the Summary pane, find the **Hostname**. It is similar to <zk1-jdolehb.3lnng4rcvp5uzokyktxs4a5dhd.bx.internal.cloudapp.net>.

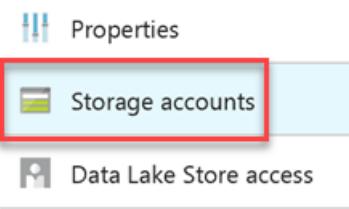
To highlight how you can efficiently leverage HDInsight HBase and Phoenix to analyze big data stores, this walkthrough shows you how to use HBase Phoenix to do sales analysis of an imaginary online book store.

Upload sample data to HDInsight cluster storage account

To begin, we need to upload the sample data we'll use for the remaining exercises. Read more about various ways to [upload data for Hadoop jobs in HDInsight](#).

1. Browse to the HBase HDInsight cluster on the Azure portal.
2. On the left-hand menu, select **Storage accounts**.

PROPERTIES



3. Select your Azure Blob Storage or Azure Data Lake Store account listed in the Storage accounts pane.

4. Browse to the `/example/data` folder.

- If using Azure Blob Storage, the `/example/data` folder will be located within your cluster's container.
- In Azure Data Lake Store, use the data explorer to navigate to `/clusters/hbase/example/data`.

5. Upload the sample `weblogs.csv` file to this location.

Bulk Load data into HBase using Phoenix

Phoenix provides two ways to bulk load into HBase. The `psql` command line utility, which is a single-threaded client loading tool, and a MapReduce bulk loading utility. PSQL is suited for gigabytes of data, whereas MapReduce is used for much larger data volumes.

To bulk upload data into HBase using the Phoenix PSQL command line tool, perform the following actions:

1. SSH into your HBase cluster. For more information, see [Use SSH with HDInsight](#).

2. From SSH, run the following commands to run **SQLLine**:

```
cd /usr/hdp/current/phoenix-client/bin  
.sqlline.py <ZooKeeperHostname>:2181:/hbase-unsecure
```

3. Run the following command to create the weblogs table.

```
CREATE TABLE weblogs (  
    transactionid varchar(500) Primary Key,  
    "t1".transactiondate Date NULL,  
    "t1".customerid varchar(50) NULL,  
    "t1".bookid varchar(50) NULL,  
    "t1".purchasetype varchar(50) NULL,  
    "t1".orderid varchar(50) NULL,  
    "t1".bookname varchar(50) NULL,  
    "t1".categoryname varchar(50) NULL,  
    "t1".invoicenumber varchar(50) NULL,  
    "t1".invoicestatus varchar(50) NULL,  
    "t1".city varchar(50) NULL,  
    "t1".state varchar(50) NULL,  
    "t2".paymentamount DOUBLE NULL,  
    "t2".quantity INTEGER NULL,  
    "t2".shippingamount DOUBLE NULL);
```

Note: The above query creates a weblogs table with two column families, t1 and t2. Column families are stored separately in different HFiles, thus it makes sense to have a separate column family for data which is queried often. The paymentamount, quantity, and shippingamount columns will be queried often, so they are in a different column family.

4. Run the following command in a **new** Hadoop command line window.

```
cd /usr/hdp/current/phoenix-client/bin
```

5. Now, let's copy the weblogs.csv file from our storage account via `hdfs` to our local temp directory.

```
hdfs dfs -copyToLocal /example/data/weblogs.csv /tmp/
```

6. Finally, we'll use PSQL to bulk insert the rows into our new HBase table.

```
./psql.py -t WEBLOGS [The FQDN of one of the Zookeepers] /tmp/weblogs.csv
```

The above code executes the PSQL client utility to bulk upload data into the weblogs table. It takes three parameters: table name (-t), zookeeper fqdn, and the path of the csv file to bulk load from.

Once the PSQL operation is complete, you should have an output on your command window similar to the following:

```
CSV Upsert complete. 278972 rows upserted
Time: 64.549 sec(s)
```

Perform data retrieval queries with SQLLine

Close the extra Hadoop command line window you opened to run the PSQL script, and switch back over to the one running SQLLine. If you closed it, please follow the steps to once again open SQLLine.

Execute the following query to select the book name and quantity sold of books purchased between January and March:

```
SELECT bookname, sum(quantity) AS QuantitySold from WEBLOGS
WHERE Purchasetype='Purchased' and Month(transactiondate)>=1 and Month(transactiondate)<=3
GROUP BY bookname;
```

You should see an output similar to the following:

```
+-----+-----+
|      BOOKNAME      | QUANTITY SOLD |
+-----+-----+
| Advances in school psychology | 90233   |
| History of political economy | 93772   |
| New Christian poetry       | 90467   |
| Science in Dispute         | 92221   |
| Space fact and fiction    | 95544   |
| THE BOOK OF WITNESSES     | 93753   |
| The adventures of Arthur Conan Doyle | 93710   |
| The voyages of Captain Cook | 94207   |
| Understanding American politics | 91119   |
+-----+-----+
9 rows selected (21.942 seconds)
```

For more information, see [SQLLine manual](#) and [Phoenix Grammar](#).

Next steps

In this article, you have learned how to use Apache Phoenix in HDInsight. To learn more, see:

- [HDInsight HBase overview](#): HBase is an Apache, open-source, NoSQL database built on Hadoop that provides random access and strong consistency for large amounts of unstructured and semistructured data.
- [Provision HBase clusters on Azure Virtual Network](#): With virtual network integration, HBase clusters can be deployed to the same virtual network as your applications so that applications can communicate with HBase directly.
- [Configure HBase replication in HDInsight](#): Learn how to configure HBase replication across two Azure datacenters.
- [Analyze Twitter sentiment with HBase in HDInsight](#): Learn how to do real-time [sentiment analysis](#) of big data by using HBase in a Hadoop cluster in HDInsight.

title: Introduction: Bulk Loading with Phoenix via psql in HDInsight - Azure HDInsight | Microsoft Docs
description: "
services: hdinsight documentationcenter: "

tags: azure-portal keywords: HBase,phoenix,sql

Bulk Loading with Phoenix via psql in HDInsight

Apache Phoenix is an open source, massively parallel relational database layer over HBase. It allows you to use SQL-like queries over HBase. It uses JDBC drivers underneath to enable users to create, delete, alter SQL tables, indexes, views and sequences, upsert rows individually and in bulk. It uses noSQL native compilation instead of using MapReduce to compile queries enabling the creation of low latency applications on top of HBase. To do this, Phoenix adds co-processors to support running client-supplied code in the address space of the server, executing the code colocated with the data. This minimizes client/server data transfer. In order to work with data using Phoenix in HDInsight, you must first create tables and load data into them. The next section addresses some of the methods available for you to do this.

Methods available for performing Bulk Load with Phoenix

There are multiple ways to get data into HBase including using client API's, a MapReduce job with TableOutputFormat or inputting the data manually via the HBase shell. Many customers are interested in using Apache Phoenix – a SQL layer over HBase for its ease of use. Phoenix provides two methods for loading CSV data into Phoenix tables – a client loading tool via the `psql` command line utility, and a MapReduce-based bulk load tool.

The `pq1` tool is a single-threaded client loading tool and it is best suited for loading megabytes or, possibly, gigabytes of data. Note that all CSV files to be loaded must have the '.csv' file extension (this is because arbitrary SQL scripts with the '.sql' file extension can also be supplied on the PSQL command line).

Bulk loading via MapReduce is used for much larger data volumes because it does have the restrictions of `psql` (i.e. single-threaded execution). Bulk loading via Map Reduce is preferred for production scenarios.

Before you start loading data using methods supported by Apache Phoenix you may want to verify that Phoenix is enabled and query timeout settings are as expected in your HDInsight cluster instance. A quick way to do this is to access your HDInsight cluster Ambari dashboard, click on HBase and then the Configuration tab. Scroll down on the page to verify that Apache Phoenix is set to `enabled` as in the example screen shot below.

Phoenix SQL

Enable Phoenix

Enabled

Phoenix Query Timeout

1	0
Minutes	Seconds

Using psql to bulk load tables

1. Create a new table using standard SQL syntax. Save your query as a `createCustomersTable.sql` file. For example:

```
CREATE TABLE Customers (
    ID varchar NOT NULL PRIMARY KEY,
    Name varchar,
    Income decimal,
    Age INTEGER,
    Country varchar);
```

2. Copy your CSV file (such as `customers.csv` shown below) into a `/tmp/` directory for loading into your newly-created table. Use the `hdfs` command (shown below) to copy your CSV file to your desired source location.

```
1,Samantha,260000.0,18,US
2,Sam,10000.5,56,US
3,Anton,550150.0,Norway
... 4997 more rows
```

```
hdfs dfs -copyToLocal /example/data/customers.csv /tmp/
```

1. Create a SQL SELECT query to verify the input data loaded properly using standard SQL syntax. Save your query as a `listCustomers.sql` file. Query to execute on the data: You can put any SQL query which you would like to run on the data (let's say `Query.sql`). A Sample query:

```
SELECT Name, Income from Customers group by Country;
```

2. Bulk Load the data by opening a *new** Hadoop command line window and running the `cd...` and Bulk Load `psql.py` commands shown below, to first change to the execution directory location and to execute the bulk load using the `psql` tool. Note: The example shown below assumes that you have copied the `customers.csv` file from our storage account via `hdfs` to your local temp directory.

```
cd /usr/hdp/current/phoenix-client/bin  
python psql.py ZookeeperQuorum createCustomersTable.sql /tmp/customers.csv listCustomers.sql
```

Tip: To determine the `ZookeeperQuorum` name you'll need to locate the zookeeper quorum string. The zookeeper string is present in file `/etc/hbase/conf/hbase-site.xml`. The name of property is `hbase.zookeeper.quorum`.

After the `psql` operation has completed, you should see an output on your command window similar to the one below:

```
CSV Upsert complete. 5000 rows upserted  
Time: 4.548 sec(s)
```

Using MapReduce to bulk load tables

For higher-throughput loading distributed over the cluster, you can use the MapReduce load tool. This loader first converts all data into HFiles, and then provides the created HFiles to HBase after the HFile creation is complete.

Launch the CSV MapReduce loader by using the hadoop command with the Phoenix client jar, as shown below:

```
hadoop jar phoenix-<version>-client.jar org.apache.phoenix.mapreduce.CsvBulkLoadTool --table CUSTOMERS --  
input /data/customers.csv
```

Follow these steps to use the MapReduce bulk load command

- Locate your `ZookeeperQuorum` string value. See the previous section (Step 4) to get the location of the value for your cluster instance.
- Create a new table: Write a SQL statement to create table in a file (as with `CreateCustomersTable.sql`) in Step 1 above.
- Verify the schema of your new table, go to `/usr/hdp/current/phoenix-client/bin` and run the command below:

```
python psql.py ZookeeperQuorum
```

- Verify the schema of your table, run `!describe inputTable`
- Get the path (location) of your input data (or `customers.csv` file): Figure out the path of your input data. The input files may be in your WASB/ADLS storage account. Let's say the input files are present in `inputFolderBulkLoad` under the parent directory of your storage account.
- Change to the execution directory for the MapReduce bulk load command, which is at `/usr/hdp/current/phoenix-client/bin`, using the command below:

```
cd /usr/hdp/current/phoenix-client/bin
```

- Bulk Load via the command shown below

```
/usr/hdp/current/phoenix-client$ HADOOP_CLASSPATH=/usr/hdp/current/hbase-client/lib/hbase-  
protocol.jar:/etc/hbase/conf hadoop jar \  
/usr/hdp/2.4.2.0-258/phoenix/phoenix-4.4.0.2.4.2.0-258-client.jar  
org.apache.phoenix.mapreduce.CsvBulkLoadTool --table Customers --input \  
/inputFolderBulkLoad/customers.csv -zookeeper ZookeeperQuorum:2181:/hbase-unsecure
```

Note: If you are bulk-loading via MapReduce from ADLS Storage, then you need to locate the root directory for ADLS. In order to find root directory, locate the entry for `hbase.rootdir` in `hbase-site.xml`.

In the command below, `adl://hdinsightconf1.azuredatalakestore.net:443/hbase1` is the ADLS root directory. In order to run bulk load command, cd to `/usr/hdp/current/phoenix-client` and pass ADLS input and output folders as parameters as shown below:

```
$ HADOOP_CLASSPATH=$(hbase mapredcp):/etc/hbase/conf hadoop jar /usr/hdp/2.4.2.0-258/phoenix/phoenix-4.4.0.2.4.2.0-258-client.jar \
org.apache.phoenix.mapreduce.CsvBulkLoadTool --table Customers --input \
adl://hdinsightconf1.azuredatalakestore.net:443/hbase1/data/hbase/temp/input/customers.csv \
-zookeeper ZookeeperQuorum:2181:/hbase-unsecure --output \
adl://hdinsightconf1.azuredatalakestore.net:443/hbase1/data/hbase/output1
```

Recommendations

1. Use the same storage medium: Use same storage medium for both input and output folders. This means that both the input and output folders should be either in WASB or in ADLS. If you want to transfer data from WASB to ADLS, you can use the `distcp` command. An example command is shown below.

```
hadoop distcp wasb://@.blob.core.windows.net/example/data/gutenberg
adl://.azuredatalakestore.net:443/myfolder
```

1. Use larger-size worker nodes: The map processes of the MapReduce bulk copy produce large amounts of temporary output which fill up the available non-DFS space. Use a larger sized worker node VM if you intend to perform a large amount of bulk loading. The number of worker nodes you allocate to your cluster will directly affect the processing speed of the bulk load activity.
2. Split input files: Because the bulk load is a storage-intensive operation, splitting your input files into multiple chunks(~10GB each) and then perform bulk load on them will result in better performance.
3. Avoid Region Server hotspots: HBase sequential write may suffer from region server hotspotting if your row key is monotonically increasing. Salting the row key provides a way to mitigate this problem. Phoenix provides a way to transparently salt the row key with a salting byte for a particular table. See link in the 'next steps' section for more detail.

Next steps

In this article, you have learned how to use bulk load data using psql and the MapReduce command from Apache Phoenix in HDInsight. To learn more, see:

- [Bulk Data Loading with Apache Phoenix](#)
- [Use Apache Phoenix with Linux-based HBase clusters in HDInsight](#)
- [Salted Tables](#)
- [Phoenix Grammar](#)

Read and Write Phoenix Data from a Spark cluster

8/15/2017 • 1 min to read • [Edit Online](#)

Apache HBase data can be queried either with its low level API of scans, gets and puts or with a SQL syntax using Apache Phoenix. Phoenix is an API for HBase which uses a JDBC driver (rather than Hadoop MapReduce) to extend the HBase key-value store to enable features that make it similar to a relational database. These features include adding a SQL query engine, metadata repository and an embedded JDBC driver. Phoenix was originally developed at Salesforce, and it was subsequently open-sourced as an Apache project. It is important to note that Phoenix is designed to work only with HBase data.

Apache Spark can be used as a convenient and performant alternative way to query and modify data stored by HBase. This method of cross-cluster access is enabled by the use of the Spark-HBase Connector (also called the [SHC](#)). See [Using Spark to Query HBase](#) for details on this approach.

- **IMPORTANT** As of this writing (June 2017) HDInsight does not support the open source [Apache Spark plugin for Phoenix](#). You are advised to use the Spark-HBase connector to support querying HBase from Spark at this time.

See Also

- [Using Spark to Query HBase](#)
 - [Spark HBase Connector](#)
 - [Phoenix Spark dependency list](#)
 - [Apache Phoenix and HBase Past Present and Future of SQL over HBase](#)
 - [New Features in Apache Phoenix](#)
 - [Apache Spark Plugin for Apache Phoenix](#)
-

Using the Phoenix Query Server REST SDK

8/16/2017 • 11 min to read • [Edit Online](#)

Apache Phoenix is an open source, massively parallel relational database layer over HBase. It allows you to use SQL-like queries over HBase through tools like [SQLLine](#) using SSH. Phoenix also provides an HTTP server called Phoenix Query Server (PQS) that exposes a thin client that supports two transport mechanisms, JSON and Protocol Buffers, for client communication. Protocol Buffers is the default mechanism, and offers more efficient communication than JSON.

In this article, we'll show you how to use the PQS REST SDK to create tables, upsert rows individually and in bulk, and select data using SQL statements. We'll be using the [Microsoft .NET driver for Apache Phoenix Query Server](#) in our examples. This SDK is built on [Apache Calcite's Avatica APIs](#), which exclusively uses Protocol Buffers as the serialization format.

Refer to the [Apache Calcite Avatica Protocol Buffers Reference](#) for more information.

Installing the SDK

Microsoft .NET driver for Apache Phoenix Query Server is provided as a NuGet package, which can be installed from the Visual Studio **NuGet Package Manager Console** with the following command:

```
Install-Package Microsoft.Phoenix.Client
```

Instantiating a new PhoenixClient object

To begin using the library, you must instantiate a new `PhoenixClient` object, passing in `ClusterCredentials` composed of the `Uri` to your cluster, the Hadoop user name and password.

```
var credentials = new ClusterCredentials(new Uri("https://CLUSTERNAME.azurehdinsight.net/"), "USERNAME",  
    "PASSWORD");  
client = new PhoenixClient(credentials);
```

Replace CLUSTERNAME with your HDInsight HBase cluster name, and USERNAME and PASSWORD with the Hadoop credentials specified on cluster creation. The default Hadoop user name is **admin**.

Generating a unique connection Id

When we prepare to send one or more requests to PQS, we need to include a unique connection identifier to associate one or more requests with the connection.

```
string connId = Guid.NewGuid().ToString();
```

In each of our samples, you will see that we initially make a call to the `OpenConnectionRequestAsync` method, passing in the unique connection id. We subsequently define `ConnectionProperties` and `RequestOptions`, passing those objects as well as the generated connection id to the `ConnectionSyncRequestAsync` method. PQS's `ConnectionSyncRequest` object helps ensure that the client and server have a consistent view of the database properties.

The ConnectionSyncRequest ConnectionProperties

When you make a call to `ConnectionSyncRequestAsync`, you pass a `ConnectionProperties` object:

```
ConnectionProperties connProperties = new ConnectionProperties
{
    HasAutoCommit = true,
    AutoCommit = true,
    HasReadOnly = true,
    ReadOnly = false,
    TransactionIsolation = 0,
    Catalog = "",
    Schema = "",
    IsDirty = true
};
await client.ConnectionSyncRequestAsync(connId, connProperties, options);
```

Here is a breakdown of the properties of interest:

PROPERTY	DESCRIPTION
AutoCommit	A boolean denoting if <code>autoCommit</code> is enabled for Phoenix transactions.
ReadOnly	A boolean denoting whether the connection is read-only.
TransactionIsolation	An integer which denotes the level of transaction isolation per the JDBC specification.
Catalog	The name of the catalog to use when fetching connection properties
Schema	The name of the schema to use when fetching connection properties.
IsDirty	A boolean denoting if the properties have been altered.

Here are the `TransactionIsolation` possible values:

ISOLATION VALUE	DESCRIPTION
0	Transactions are not supported.
1	Dirty reads, non-repeatable reads and phantom reads may occur.
2	Dirty reads are prevented, but non-repeatable reads and phantom reads may occur.
4	Dirty reads and non-repeatable reads are prevented, but phantom reads may occur.
8	Dirty reads, non-repeatable reads, and phantom reads are all prevented.

Create a new table

HBase, like any other RDBMS, stores data in tables. Using Phoenix, we are able to use standard SQL queries to create new tables, defining the primary key and column types as well.

For this and the remaining examples, we'll be using the instantiated `PhoenixClient` object as defined above.

```

string connId = Guid.NewGuid().ToString();
RequestOptions options = RequestOptions.GetGatewayDefaultOptions();

// You can set certain request options, such as timeout in milliseconds:
options.TimeoutMillis = 300000;

// In gateway mode, PQS requests will be https://<cluster dns name>.azurehdinsight.net/hbasephoenix<N>/
// Requests sent to hbasephoenix0/ will be forwarded to PQS on workernode0
options.AlternativeEndpoint = "hbasephoenix0/";
CreateStatementResponse createStatementResponse = null;
OpenConnectionResponse openConnResponse = null;

try
{
    // Opening connection
    var info = new pbc::MapField<string, string>();
    openConnResponse = await client.OpenConnectionRequestAsync(connId, info, options);

    // Syncing connection
    ConnectionProperties connProperties = new ConnectionProperties
    {
        HasAutoCommit = true,
        AutoCommit = true,
        HasReadOnly = true,
        ReadOnly = false,
        TransactionIsolation = 0,
        Catalog = "",
        Schema = "",
        IsDirty = true
    };
    await client.ConnectionSyncRequestAsync(connId, connProperties, options);

    // Create the statement
    createStatementResponse = client.CreateStatementRequestAsync(connId, options).Result;

    // Create the table if it does not exist
    string sql = "CREATE TABLE IF NOT EXISTS Customers (Id varchar(20) PRIMARY KEY, FirstName varchar(50), " +
                 "LastName varchar(100), StateProvince char(2), Email varchar(255), Phone varchar(15))";
    await client.PrepareAndExecuteRequestAsync(connId, sql, createStatementResponse.StatementId, long.MaxValue,
                                                int.MaxValue, options);

    Console.WriteLine($"Table \"Customers\" created.");
}
catch (Exception e)
{
    Console.WriteLine(e);
    throw;
}
finally
{
    if (createStatementResponse != null)
    {
        client.CloseStatementRequestAsync(connId, createStatementResponse.StatementId, options).Wait();
        createStatementResponse = null;
    }

    if (openConnResponse != null)
    {
        client.CloseConnectionRequestAsync(connId, options).Wait();
        openConnResponse = null;
    }
}
}

```

We created a new table named "Customers" using the `IF NOT EXISTS` option. The `CreateStatementRequestAsync` call is used to create a new Statement in the Avitica (PQS) server. In the `finally` block, we ensure that the `CreateStatementResponse` that this method returns, as well as the `OpenConnectionResponse` object are properly

closed.

Inserting data individually

This individual data insert example, as well as the bulk insert example, references a `List<string>` collection of States and territories that you can use:

```
var states = new List<string> { "AL", "AK", "AS", "AZ", "AR", "CA", "CO", "CT", "DE", "DC", "FM", "FL", "GA",  
    "GU", "HI", "ID", "IL", "IN", "IA", "KS", "KY", "LA", "ME", "MH", "MD", "MA", "MI", "MN", "MS", "MO", "MT",  
    "NE", "NV", "NH", "NJ", "NM", "NY", "NC", "ND", "MP", "OH", "OK", "OR", "PW", "PA", "PR", "RI", "SC", "SD",  
    "TN", "TX", "UT", "VT", "VI", "VA", "WA", "WV", "WI", "WY" };
```

We'll use the table's `StateProvince` column value in a select operation later on.

```
string connId = Guid.NewGuid().ToString();  
RequestOptions options = RequestOptions.GetGatewayDefaultOptions();  
options.TimeoutMillis = 300000;  
  
// In gateway mode, PQS requests will be https://<cluster dns name>.azurehdinsight.net/hbasephoenix<N>/  
// Requests sent to hbasephoenix0/ will be forwarded to PQS on workernode0  
options.AlternativeEndpoint = "hbasephoenix0/";  
OpenConnectionResponse openConnResponse = null;  
StatementHandle statementHandle = null;  
try  
{  
    // Opening connection  
    pbc::MapField<string, string> info = new pbc::MapField<string, string>();  
    openConnResponse = await client.OpenConnectionRequestAsync(connId, info, options);  
    // Syncing connection  
    ConnectionProperties connProperties = new ConnectionProperties  
    {  
        HasAutoCommit = true,  
        AutoCommit = true,  
        HasReadOnly = true,  
        ReadOnly = false,  
        TransactionIsolation = 0,  
        Catalog = "",  
        Schema = "",  
        IsDirty = true  
    };  
    await client.ConnectionSyncRequestAsync(connId, connProperties, options);  
  
    string sql = "UPSERT INTO Customers VALUES (?,?,?,?,?,?)";  
    PrepareResponse prepareResponse = await client.PrepareRequestAsync(connId, sql, 100, options);  
    statementHandle = prepareResponse.Statement;  
  
    var r = new Random();  
  
    // Insert 300 rows  
    for (int i = 0; i < 300; i++)  
    {  
        var list = new pbc.RepeatedField<TypedValue>();  
        var id = new TypedValue  
        {  
            StringValue = "id" + i,  
            Type = Rep.String  
        };  
        var firstName = new TypedValue  
        {  
            StringValue = "first" + i,  
            Type = Rep.String  
        };  
        var lastName = new TypedValue  
        {  
            StringValue = "last" + i,  
            Type = Rep.String  
        };  
    }  
}
```

```

        stringValue = last + ,
        Type = Rep.String
    );
    var state = new TypedValue
    {
        StringValue = states.ElementAt(r.Next(0, 49)),
        Type = Rep.String
    );
    var email = new TypedValue
    {
        StringValue = $"email{1}@junkemail.com",
        Type = Rep.String
    );
    var phone = new TypedValue
    {
        StringValue = $"555-229-341{i.ToString().Substring(0,1)}",
        Type = Rep.String
    );
    list.Add(id);
    list.Add(firstName);
    list.Add(lastName);
    list.Add(state);
    list.Add(email);
    list.Add(phone);

    Console.WriteLine("Inserting customer " + i);

    await client.ExecuteRequestAsync(statementHandle, list, long.MaxValue, true, options);
}

await client.CommitRequestAsync(connId, options);

Console.WriteLine("Upserted customer data");

}
catch (Exception ex)
{
}

finally
{
    if (statementHandle != null)
    {
        await client.CloseStatementRequestAsync(connId, statementHandle.Id, options);
        statementHandle = null;
    }
    if (openConnResponse != null)
    {
        await client.CloseConnectionRequestAsync(connId, options);
        openConnResponse = null;
    }
}
}

```

The overall structure of executing an insert statement is very similar to how we created a new table. One thing of note is at the end of the `try` block, we explicitly commit the transaction. Also notice that we are executing an insert request 300 times since it's in a loop. This makes for a lengthy process due to excessive requests over a thin client. A more efficient execution plan is to insert our records within a batch process.

Batch inserting data

The following code is nearly identical to the code we used for inserting data individually. In this case, however, we're using the `updateBatch` object in a call to `ExecuteBatchRequestAsync`, as opposed to repeatedly calling `ExecuteRequestAsync` with our prepared statement.

```
string connId = Guid.NewGuid().ToString();
```

```

    RequestOptions options = RequestOptions.GetGatewayDefaultOptions();
    options.TimeoutMillis = 300000;

    // In gateway mode, PQS requests will be https://<cluster dns name>.azurehdinsight.net/hbasephoenix<N>/
    // Requests sent to hbasephoenix0/ will be forwarded to PQS on workernode0
    options.AlternativeEndpoint = "hbasephoenix0/";
    OpenConnectionResponse openConnResponse = null;
    CreateStatementResponse createStatementResponse = null;
    try
    {
        // Opening connection
        pbc::MapField<string, string> info = new pbc::MapField<string, string>();
        openConnResponse = await client.OpenConnectionRequestAsync(connId, info, options);
        // Syncing connection
        ConnectionProperties connProperties = new ConnectionProperties
        {
            HasAutoCommit = true,
            AutoCommit = true,
            HasReadOnly = true,
            ReadOnly = false,
            TransactionIsolation = 0,
            Catalog = "",
            Schema = "",
            IsDirty = true
        };
        await client.ConnectionSyncRequestAsync(connId, connProperties, options);

        // Creating statement
        createStatementResponse = await client.CreateStatementRequestAsync(connId, options);

        string sql = "UPSERT INTO Customers VALUES (?, ?, ?, ?, ?, ?)";
        PrepareResponse prepareResponse = client.PrepareRequestAsync(connId, sql, long.MaxValue, options).Result;
        var statementHandle = prepareResponse.Statement;
        var updates = new pbc.RepeatedField<UpdateBatch>();

        var r = new Random();

        // Insert 300 rows
        for (int i = 300; i < 600; i++)
        {
            var list = new pbc.RepeatedField<TypedValue>();
            var id = new TypedValue
            {
                StringValue = "id" + i,
                Type = Rep.String
            };
            var firstName = new TypedValue
            {
                StringValue = "first" + i,
                Type = Rep.String
            };
            var lastName = new TypedValue
            {
                StringValue = "last" + i,
                Type = Rep.String
            };
            var state = new TypedValue
            {
                StringValue = states.ElementAt(r.Next(0, 49)),
                Type = Rep.String
            };
            var email = new TypedValue
            {
                StringValue = $"email{1}@junkemail.com",
                Type = Rep.String
            };
            var phone = new TypedValue
            {
                StringValue = $"555-229-341{i.ToString().Substring(0, 1)}".

```

```

        Type = Rep.String
    };
    list.Add(id);
    list.Add(firstName);
    list.Add(lastName);
    list.Add(state);
    list.Add(email);
    list.Add(phone);

    var batch = new UpdateBatch
    {
        ParameterValues = list
    };
    updates.Add(batch);

    Console.WriteLine($"Added customer {i} to batch");
}

var executeBatchResponse = await client.ExecuteBatchRequestAsync(connId, statementHandle.Id, updates, options);

Console.WriteLine("Batch upserted customer data");

}
catch (Exception ex)
{
}

finally
{
    if (openConnResponse != null)
    {
        await client.CloseConnectionRequestAsync(connId, options);
        openConnResponse = null;
    }
}
}

```

In our environment, individually inserting 300 new records took almost 2 minutes. Inserting 300 records as a batch, however, took only about 6 seconds!

Selecting data

In this example, we'll show how you can reuse the same connection to execute multiple queries. First we'll demonstrate selecting all, and fetching remaining records once the default maximum of 100 have been returned. Then we'll show using a total row count select statement, retrieving the single scalar result. Finally, we'll execute a select statement that returns the total number of customers per State.

```

string connId = Guid.NewGuid().ToString();
RequestOptions options = RequestOptions.GetGatewayDefaultOptions();

// In gateway mode, PQS requests will be https://<cluster dns name>.azurehdinsight.net/hbasephoenix<N>/
// Requests sent to hbasephoenix0/ will be forwarded to PQS on workernode0
options.AlternativeEndpoint = "hbasephoenix0/";
OpenConnectionResponse openConnResponse = null;
StatementHandle statementHandle = null;

try
{
    // Opening connection
    pbc::MapField<string, string> info = new pbc::MapField<string, string>();
    openConnResponse = await client.OpenConnectionRequestAsync(connId, info, options);
    // Syncing connection
    ConnectionProperties connProperties = new ConnectionProperties
    {
        UseAutoCommit = true
    }
}

```

```

    HasAutoCommit = true,
    AutoCommit = true,
    HasReadOnly = true,
    ReadOnly = false,
    TransactionIsolation = 0,
    Catalog = "",
    Schema = "",
    IsDirty = true
};

await client.ConnectionSyncRequestAsync(connId, connProperties, options);
var createStatementResponse = await client.CreateStatementRequestAsync(connId, options);

string sql = "SELECT * FROM Customers";
ExecuteResponse executeResponse = await client.PrepareAndExecuteRequestAsync(connId, sql,
createStatementResponse.StatementId, long.MaxValue, int.MaxValue, options);

pb::RepeatedField<Row> rows = executeResponse.Results[0].FirstFrame.Rows;
// Loop through all of the returned rows and display the first two columns
for (int i = 0; i < rows.Count; i++)
{
    Row row = rows[i];
    Console.WriteLine(row.Value[0].ScalarValue.StringValue + " " + row.Value[1].ScalarValue.StringValue);
}

// 100 is hard coded in server side as the default firstframe size
// In order to get remaining rows, FetchRequestAsync is used
Console.WriteLine("");
Console.WriteLine($"Number of rows: {rows.Count}");

// Fetch remaining rows, offset is not used, simply set to 0
// if FetchResponse.Frame.Done = true, that means all the rows fetched
FetchResponse fetchResponse = await client.FetchRequestAsync(connId, createStatementResponse.StatementId,
0, int.MaxValue, options);
Console.WriteLine($"Frame row count: {fetchResponse.Frame.Rows.Count}");
Console.WriteLine($"Fetch response is done: {fetchResponse.Frame.Done}");
Console.WriteLine("");

// Running query 2
string sql2 = "select count(*) from Customers";
ExecuteResponse countResponse = await client.PrepareAndExecuteRequestAsync(connId, sql2,
createStatementResponse.StatementId, long.MaxValue, int.MaxValue, options);
long count = countResponse.Results[0].FirstFrame.Rows[0].Value[0].ScalarValue.NumberValue;

Console.WriteLine($"Total customer records: {count}");
Console.WriteLine("");

// Running query 3
string sql3 = "select StateProvince, count(*) as Number from Customers group by StateProvince order by
Number desc";
ExecuteResponse groupByResponse = await client.PrepareAndExecuteRequestAsync(connId, sql3,
createStatementResponse.StatementId, long.MaxValue, int.MaxValue, options);

pb::RepeatedField<Row> stateRows = groupByResponse.Results[0].FirstFrame.Rows;
for (int i = 0; i < stateRows.Count; i++)
{
    Row row = stateRows[i];
    Console.WriteLine(row.Value[0].ScalarValue.StringValue + ":" + row.Value[1].ScalarValue.NumberValue);
}
}

catch (Exception ex)
{

}

finally
{
    if (statementHandle != null)
    {
        await client.CloseStatementRequestAsync(connId, statementHandle.Id, options);
        statementHandle = null;
    }
}

```

```

        }
        if (openConnResponse != null)
        {
            await client.CloseConnectionRequestAsync(connId, options);
            openConnResponse = null;
        }
    }
}

```

The output of the executed select statements should yield the following result:

```

id0 first0
id1 first1
id10 first10
id100 first100
id101 first101
id102 first102
...
// TRUNCATED FOR BREVITY

...
id185 first185
id186 first186
id187 first187
id188 first188

Number of rows: 100
Frame row count: 500
Fetch response is done: True

Total customer records: 600

NJ: 21
CA: 19
GU: 17
NC: 16
IN: 16
MA: 16
AZ: 16
ME: 16
IL: 15
OR: 15
...
// TRUNCATED FOR BREVITY

...
MO: 10
HI: 10
GA: 10
DC: 9
NM: 9
MD: 9
MP: 9
SC: 7
AR: 7
MH: 6
FM: 5

```

Next steps

- Learn more about [Phoenix in HDInsight](#)
- Read [Using the HBase REST SDK](#) for information on another HBase-related SDK you can use.

Phoenix Performance Best Practices

8/15/2017 • 11 min to read • [Edit Online](#)

This article provides the fundamental techniques you should consider when optimizing the performance of your Phoenix deployment on HDInsight.

Phoenix and HBase

One of the most important considerations when optimizing the performance of Phoenix really boils down to making sure HBase is well optimized. Recall that Phoenix creates a relational data model atop HBase that converts SQL queries into HBase operations like scans. This means that the design of your table schema, the selection and ordering of the fields in your primary key, and your use of indexes all affect how performant Phoenix can be in querying HBase.

Table Schema Design

When you create a table in Phoenix, that table is stored in an HBase table. The HBase table contains column families, which are groups of columns that are accessed together. A row in the Phoenix table is a row in the HBase table, where each row consists of versioned cells associated with one or more columns. A single HBase row is logically a collection of key-value pairs each having the same rowkey value. In other words where for each of the key-value pairs they each have a rowkey attribute, and the value of that rowkey attribute is the same when they are part of the same row.

When it comes to the schema design of your table in Phoenix it is important to consider the primary key design, column family design, column design and how the data is partitioned.

Primary Key Design

The primary key that you define on a table in Phoenix actually dictates the semantics for how data is stored within the rowkey of the underlying HBase table. In HBase, the only way to access a particular row is via the rowkey. In addition, data stored in an HBase table is sorted by the rowkey. The way Phoenix builds the rowkey value is by concatenating the values of each of the columns in the row, in the order they are defined in the primary key.

For example, say you had a table for contacts that had the first name, last name, phone number and address (all in the same column family). You could define a Primary Key based on an increasing sequence number, so your rowkeys would look like:

ROWKEY	ADDRESS	PHONE	FIRSTNAME	LASTNAME
1000	1111 San Gabriel Dr.	1-425-000-0002	John	Dole
8396	5415 San Gabriel Dr.	1-230-555-0191	Calvin	Raji

However, if you frequently query by lastName this would not perform well because you would need to do a table scan and read the value of every lastname, so you might define a primary key on the lastName, firstName and a social security number (to disambiguate two residents at the same address with the same name, like a father and son).

ROWKEY	ADDRESS	PHONE	FIRSTNAME	LASTNAME	SOCIALSECURITYNUM
1000	1111 San Gabriel Dr.	1-425-000-0002	John	Dole	111
8396	5415 San Gabriel Dr.	1-230-555-0191	Calvin	Raji	222

If you defined a primary key on the table for lastName, firstName and socialSecurityNum your row keys as generated by Phoenix might look as follows:

ROWKEY	ADDRESS	PHONE	FIRSTNAME	LASTNAME	SOCIALSECURITYNUM
Dole-John-111	1111 San Gabriel Dr.	1-425-000-0002	John	Dole	111
Raji-Calvin-222	5415 San Gabriel Dr.	1-230-555-0191	Calvin	Raji	222

By way of example, for the fist row in the example above the data stored actually looks like:

ROWKEY	KEY	VALUE
Dole-John-111	address	1111 San Gabriel Dr.
Dole-John-111	phone	1-425-000-0002
Dole-John-111	firstName	John
Dole-John-111	lastName	Dole
Dole-John-111	socialSecurityNum	111

As you can see, the rowkey now stores a duplicate copy of the data. It is important to be aware of the size and number of columns you include in your Primary Key, because this value is included with every cell in the underlying HBase table.

Also, if the primary key you have selected has values that are monotonically increasing, you should create the table with salt buckets (see the Partition Data section below) to help avoid creating write hotspots.

Column Family Design

Choosing which columns to group into the same column family is another important consideration. If some columns are accessed more frequently than others, you should create multiple column families to separate the frequently-accessed columns from rarely-accessed columns.

Another consideration, is to identify which columns tend to be accessed together and put those in the same column family.

Column Design

There are a few considerations to what you store within a column as well:

- Keep VARCHAR columns under 1MB or so due to the I/O costs of large columns. When processing queries, HBase materializes cells in full before sending them over to the client, and the client receives them in full before handing them off to the application code.

- Store column values using a compact format such as protobuf, Avro, msgpack or BSON. Avoid JSON if you can as it is significantly less compact.
- Consider compressing data before storage to cut latency and I/O costs.

Partition Data

Phoenix enables you to control the number of regions on which your data is distributed, which can significantly increase read/write performance. You can accomplish this either by salting or pre-splitting your data. Both are tasks you perform when creating your Phoenix table.

To salt a table during creation, you specify the number of salt buckets, for example:

```
CREATE TABLE CONTACTS (...) SALT_BUCKETS = 16
```

Salting does the splitting of the table along the value of primary key lines, choosing the values automatically. If you want to control where the table splits occur, you can pre-split the table by providing the actual values that define the ranges along which the splitting occurs. For example, the following creates a table that will be split along five regions:

```
CREATE TABLE CONTACTS (...) SPLIT ON ('CS', 'EU', 'NA')
```

Index Design

A Phoenix index is an HBase table that stores a copy of some or all of the data from the table that it indexes. You apply an index to benefit specific kinds of queries.

When you have an index in place and you query a table, Phoenix selects the best index for the query automatically. The primary index is created automatically based on the primary keys you select. You can also create secondary indexes by specifying which columns are included based on the anticipated queries the index will support.

When designing your indexes, keep the following points in mind:

- Only create the indexes you need.
- Limit the number of indexes on frequently updated tables (since updates to a table translates into writes to both the main table and the tables containing the indexes).

Create secondary indexes

Secondary indexes can improve read performance by turning what would normally be a full table scan into a point lookup (at the cost of storage space and write speed). Secondary indexes can be added or removed after table creation and don't require changes to existing queries – queries simply run faster. Depending on your needs, consider creating covered indexes, functional indexes, or both.

Use covered indexes

Covered indexes are indexes that include additional data from the row in addition to the values that are indexed, eliminating the need to go back to the primary table once the desired index entry has been found.

For example, in our contact table we could create a secondary index on the socialSecurityNum column alone. This would speedup queries that filter by socialSecurityNum values, but to pull back other field values like firstName and lastName would require another read against the main table.

ROWKEY	ADDRESS	PHONE	FIRSTNAME	LASTNAME	SOCIALSECURITYNUM

ROWKEY	ADDRESS	PHONE	FIRSTNAME	LASTNAME	SOCIALSECURITYNUM
Dole-John-111	1111 San Gabriel Dr.	1-425-000-0002	John	Dole	111
Raji-Calvin-222	5415 San Gabriel Dr.	1-230-555-0191	Calvin	Raji	222

However, if we frequently want to look up the firstName and lastName given the socialSecurityNum, we could create a covered index as follows that includes the firstName and lastName as actual data in the index table:

```
CREATE INDEX ssn_idx ON CONTACTS (socialSecurityNum) INCLUDE(firstName, lastName);
```

This would enable the following query to acquire all data just by reading from the table containing the secondary index:

```
SELECT socialSecurityNum, firstName, lastName FROM CONTACTS WHERE socialSecurityNum > 100;
```

Use functional indexes

Functional indexes allow you to create an index on an arbitrary expressions that you expect to be used in queries. Once you have a functional index in place and a query uses that expression, the index may be used to retrieve the results instead of the data table.

For example, you could create an index to allow you to do case insensitive searches on the combined first name and last name of a person:

```
CREATE INDEX FULLNAME_UPPER_IDX ON "Contacts" (UPPER("firstName" || ' ' || "lastName"));
```

Query Design

Naturally, an important aspect of performant Phoenix queries is the actual design of your query.

The main considerations in query design are:

- Understand the query plan and make sure it is as expected
- Join efficiently

Understand the query plan

In SQLLine, use EXPLAIN followed by your SQL query to view the plan of operations that Phoenix will perform when executing the query.

The key things to look for are:

- Plan is using your primary key when appropriate.
- Plan uses the secondary indexes you intend, as opposed to the data table.
- Plan uses RANGE SCAN or SKIP SCAN whenever possible rather than TABLE SCAN.

Plan examples

Assume you have a table called FLIGHTS that stores flight delay information.

Say you wanted to select all the flights with the AIRLINEID of "19805" which is not a field that is in the primary key nor in any index:

```
select * from "FLIGHTS" where airlineid = '19805';
```

You would run the explain command as follows:

```
explain select * from "FLIGHTS" where airlineid = '19805';
```

A plan for this query would look like this:

```
CLIENT 1-CHUNK PARALLEL 1-WAY ROUND ROBIN FULL SCAN OVER FLIGHTS  
SERVER FILTER BY AIRLINEID = '19805'
```

In the above plan, notice the phrase FULL SCAN OVER FLIGHTS. This indicates the execution will do a TABLE SCAN over all rows in the table, instead of using the more efficient RANGE SCAN or SKIP scan option.

Now, say you want to query for flights in January 1st, 2014 for the carrier "AA" where its flightnum was greater than 1. Let's assume that the columns year, month, dayofmonth, carrier and flightnum exist in our table, and are all part of the composite primary key. Our query would look as follows:

```
select * from "FLIGHTS" where year = 2014 and month = 1 and dayofmonth = 2 and carrier = 'AA' and flightnum > 1;
```

Let's examine the plan for this query with:

```
explain select * from "FLIGHTS" where year = 2014 and month = 1 and dayofmonth = 2 and carrier = 'AA' and  
flightnum = 1;
```

The resulting plan looks as follows:

```
CLIENT 1-CHUNK PARALLEL 1-WAY ROUND ROBIN RANGE SCAN OVER FLIGHTS [2014,1,2,'AA',2] - [2014,1,2,'AA',*]
```

Notice in the above the values in square brackets after the table name show the range of values for the primary keys having values that are fixed with year of 2014, month of 1, dayofmonth of 2 but allow values for flightnum starting (and including) 2 and upwards. This query plan confirms our primary key is being used as we would expect.

Next, say we created an index on the flights table called carrier2_idx that is on the carrier field only, but includes flightdate, tailnum, origin and flightnum as covered columns whose data is also stored in the index.

```
CREATE INDEX carrier2_idx ON FLIGHTS (carrier) INCLUDE(FLIGHTDATE, TAILNUM, ORIGIN, FLIGHTNUM);
```

Say we want to get the carrier along with the flightdate and tailnum, as in the following query:

```
select carrier, flightdate, tailnum from "FLIGHTS" where carrier = 'AA';
```

We should see this index used, as the explain command would show:

```
CLIENT 1-CHUNK PARALLEL 1-WAY ROUND ROBIN RANGE SCAN OVER CARRIER2_IDX ['AA']
```

In the above notice it used our CARRIER2_IDX with the key value of 'AA', as was expected.

For a complete listing of the items that can appear in explain plan results, see the Explain Plans section in the

Join efficiently

Generally, you will want to avoid joins unless one side is small, especially on frequent queries.

If necessary, you can do large joins with the `/+ USE_SORT_MERGE_JOIN */hint`, but a big join will be an expensive operation over huge numbers of rows. If the overall size of all right-hand-side tables would exceed the available memory, use the `/+ NO_STAR_JOIN */hint`.

Scenarios

There are some common patterns we can use to apply the aforementioned guidance. The following provides guidance on each.

Read-heavy workloads

For read-heavy use cases, make sure you are using indexes. Additionally, to save read-time overhead, consider creating covered indexes.

Write-heavy workloads

For write-heavy workloads, if the primary key is monotonically increasing, create salt buckets to help avoid write hotspots at the expense of overall read throughput due to the additional scans needed. Also, when using UPSERT to write a large number of records, turn off autocommit and batch records (instead of writing them one by one).

Bulk deletes

When deleting a large data set, turn on autoCommit before issuing the DELETE query so that the client does not need to remember the row keys of all the keys as they are deleted. This prevents the client from buffering the rows affected by the DELETE so that Phoenix can delete them directly on the region servers without the expense of returning them to the client.

Immutable and Append-Only

If your scenario favors write-speed over data integrity, you can consider disabling the write ahead log. This is an option specified when you are creating your table and takes effect when the DISABLE_WAL option is set to true. For example:

```
CREATE TABLE CONTACTS (...) DISABLE_WAL=true;
```

For details on this and other options, see [Phoenix Grammer](#)

See Also:

- [Phoenix Tuning Guide](#)
- [Secondary Indexes](#)

Install third-party Hadoop applications on Azure HDInsight

8/16/2017 • 8 min to read • [Edit Online](#)

In this article, you will learn how to install an already published third-party Hadoop application on Azure HDInsight. For instructions on installing your own application, see [Install custom HDInsight applications](#).

The HDInsight application platform

HDInsight is the only fully-managed cloud Hadoop offering that provides optimized open source analytical clusters for Spark, Hive, Interactive Hive, MapReduce, HBase, Storm, Kafka, and R Server backed by a 99.9% SLA. Each of these big data technologies are easily deployable as managed clusters with enterprise-level security and monitoring.

The ecosystem of applications in Big data has grown with the goal of making it easier for customers to solve their big data and analytical problems faster. Today, customers often find it challenging to discover these productivity applications and then in turn, struggle to install and configure these apps.

To address this gap, the HDInsight Application Platform provides an experience unique to HDInsight where Independent Software Vendors (ISVs) can directly offer their applications to customers - and customers can easily discover, install and use these applications built for the Big data ecosystem.

Installing HDInsight applications helps customers be more productive with the [range of available Hadoop cluster types](#). These solutions can span a variety of scenarios from data ingestion, data wrangling, monitoring, visualization, optimizing performance, security, analyzing, visualization, reporting, and many more. Applications can be developed by Microsoft, ISVs, or by yourself.

Most of the HDInsight applications are installed on an empty edge node. An empty edge node is a Linux virtual machine with the same client tools installed and configured as in the head node, but with no Hadoop services running. You can use the edge node for accessing the cluster, testing your client applications, and hosting your client applications. For more information, see [Use empty edge nodes in HDInsight](#).

HDInsight applications offer the following benefits:

Ease of authoring, deploying, and management

You can install these apps on an existing HDInsight cluster as well as during new cluster creation.

Native access to cluster

The apps are installed on the Edge node and have access to the entire cluster.

Ease of configuring the app on the cluster

End users of these applications do not have to install or manage packages on each and every node and configure the application.

Install solutions on existing HDInsight clusters

Solution providers can make their solutions available to users who already have an HDInsight cluster running. This allows them to use these solutions easily and increase their productivity.

About ISVs

ISVs, or Independent Software Vendors, are organizations who build products, provide custom development services, and serve as a valuable resource to Microsoft and our customers. We invite ISVs to leverage the capability to make it easier for customers to discover and use your solution through the [Azure Marketplace](#). Please reach out to hdipartners@microsoft.com if you would like to participate.

Published ISV applications

There are seven currently seven published applications. Click each for more information:

- **DATAIKU DDS on HDInsight**: Dataiku DSS (Data Science Studio) is a software that allows data professionals (data scientists, business analysts, developers...) to prototype, build, and deploy highly specific services that transform raw data into impactful business predictions.
- **Datameer**: Datameer offers analysts an interactive way to discover, analyze, and visualize the results on Big Data. Pull in additional data sources easily to discover new relationships and get the answers you need quickly.
- **Streamsets Data Collector for HDInsight** provides a full-featured integrated development environment (IDE) that lets you design, test, deploy, and manage any-to-any ingest pipelines that mesh stream and batch data, and include a variety of in-stream transformations—all without having to write custom code.
- **Cask CDAP 4.1/4.2 for HDInsight** provides the first unified integration platform for big data that cuts down the time to production for data applications and data lakes by 80%. This application only supports Standard HBase 3.4 and 3.5 clusters.
- **H2O Artificial Intelligence for HDInsight (Beta)** H2O Sparkling Water supports the following distributed algorithms: GLM, Naïve Bayes, Distributed Random Forest, Gradient Boosting Machine, Deep Neural Networks , Deep learning, K-means , PCA, Generalized Low Rank Models, Anomaly Detection, and Autoencoders.
- **AtScale** makes BI work on Hadoop. With AtScale, business users get interactive and multi-dimensional analysis capabilities, directly on Hadoop, at maximum speed, using the tools they already know, own and love – from Microsoft Excel to Tableau Software to QlikView.
- **Talena** ensures data resiliency in the event of disasters or corruption, enabling companies to get back online faster, by leveraging the power of machine learning. Talena backs up and recovers terabyte and petabyte-sized data sets and beyond 5-10 times faster than any other solution on the market, minimizing the impact of data loss associated with human and application errors and reducing downtime to minutes and hours, as opposed to days and weeks.

How to install a published application

The instructions provided in this article use Azure portal. You can also export the Azure Resource Manager template from the portal or obtain a copy of the Resource Manager template from vendors, and use Azure PowerShell and Azure CLI to deploy the template. See [Create Linux-based Hadoop clusters in HDInsight using Resource Manager templates](#).

Prerequisites

All that's needed is an existing HDInsight cluster, or you can follow steps to [create an HDInsight cluster](#).

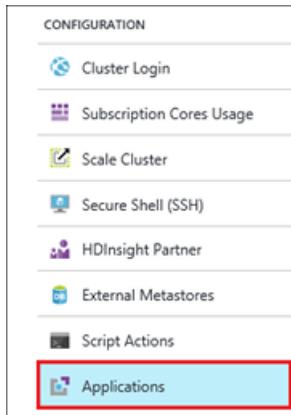
Install applications to existing clusters

The following procedure shows you how to install HDInsight applications to an existing HDInsight cluster.

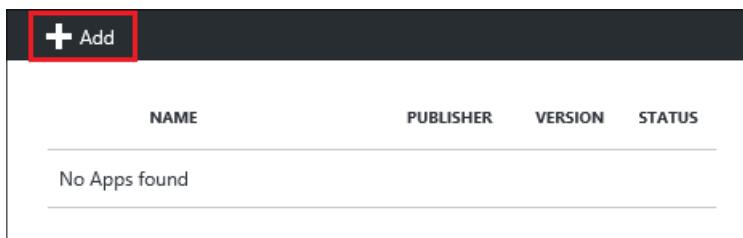
To install an HDInsight application

1. Sign in to the [Azure portal](#).
2. Click **HDInsight Clusters** in the left menu. If you don't see it, click **More Services**, and then click **HDInsight Clusters**.
3. Click an HDInsight cluster. If you don't have one, you must create one first. see [Create clusters](#).

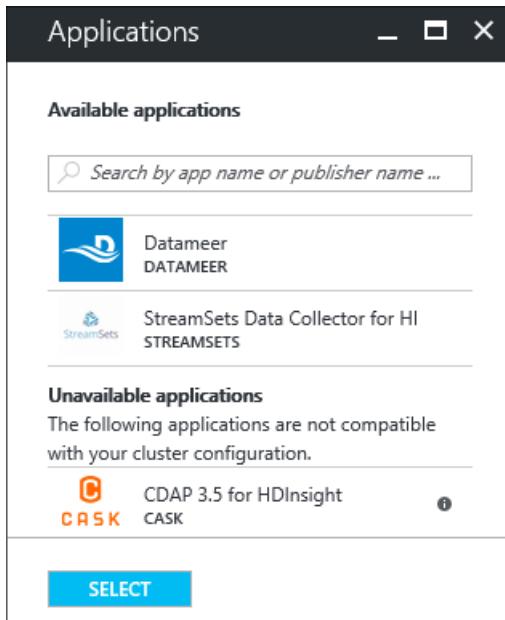
4. Click **Applications** under the **Configurations** category. You can see a list of installed applications if there are any. If you cannot find Applications, that means there is no applications for this version of the HDInsight cluster.



5. Click **Add** from the blade menu.



You can see a list of existing HDInsight applications.



6. Click one of the applications, accept the legal terms, and then click **Select**.

You can see the installation status from the portal notifications (click the bell icon on the top of the portal). After the application is installed, the application will appear on the Installed Apps blade.

Install applications during cluster creation

You have the option to install HDInsight applications when you create a cluster. During the process, HDInsight applications are installed after the cluster is created and is in the running state. The following procedure shows you how to install HDInsight applications when you create a cluster.

To install an HDInsight application

1. Sign in to the [Azure portal](#).
2. Click **NEW**, Click **Data + Analytics**, and then click **HDInsight**.
3. Select **Custom (size, settings, apps)**.
4. Enter **Cluster Name**: This name must be globally unique.
5. Click **Subscription** to select the Azure subscription that will be used for the cluster.
6. Click **Select cluster Type**, and then select:
 - **Cluster Type**: If you don't know what to choose, select **Hadoop**. It is the most popular cluster type.
 - **Operating System**: Select **Linux**.
 - **Version**: Use the default version if you don't know what to choose. For more information, see [HDInsight cluster versions](#).
 - **Cluster Tier**: Azure HDInsight provides the big data cloud offerings in two categories: Standard tier and Premium tier. For more information, see [Cluster tiers](#).
7. Click **Credentials** and then enter a password for the admin user. You must also enter an **SSH Username** and either a **PASSWORD** or **PUBLIC KEY**, which will be used to authenticate the SSH user. Using a public key is the recommended approach. Click **Select** at the bottom to save the credentials configuration.
8. Click **Resource Group** to select an existing resource group, or click **New** to create a new resource group.
9. Select the cluster **Location**.
10. Click **Next**.
11. Select your **Storage Account Settings**, selecting one of the existing storage accounts or creating a new one to be used as the default storage account for the cluster. In this blade, you may also select additional storage accounts as well as configure metastore settings to preserve your metadata outside of the cluster.
12. Click **Next**.
13. Search or browse from available applications in the **Applications** blade. Select one, accept the license agreement, then click **Next**.
14. Complete the remaining steps, such as cluster size and advanced settings. On the **Summary** blade, confirm your settings, and then click **Create**.

List installed HDInsight apps and properties

The portal shows a list of the installed HDInsight applications for a cluster, and the properties of each installed application.

To list HDInsight application and display properties

1. Sign in to the [Azure portal](#).
2. Click **HDInsight Clusters** in the left menu. If you don't see it, click **Browse**, and then click **HDInsight Clusters**.
3. Click an HDInsight cluster.
4. From the **Settings** blade, click **Applications** under the **General** category. The Installed Apps blade lists all the installed applications.

NAME	PUBLISHER	VERSION	STATUS
Datameer	Datameer	1.0.2	Installed

5. Click one of the installed applications to show the property. The property blade lists:

- App name: application name.

- Status: application status.
- Webpage: The URL of the web application that you have deployed to the edge node if there is any. The credential is the same as the HTTP user credentials that you have configured for the cluster.
- HTTP endpoint: The credential is the same as the HTTP user credentials that you have configured for the cluster.
- SSH endpoint: You can use SSH to connect to the edge node. The SSH credentials are the same as the SSH user credentials that you have configured for the cluster. For information, see [Use SSH with HDInsight](#).

6. To delete a application, right-click the application, and then click **Delete** from the context menu.

Connect to the edge node

You can connect to the edge node using HTTP and SSH. The endpoint information can be found from the [portal](#). For information, see [Use SSH with HDInsight](#).

The HTTP endpoint credentials are the HTTP user credentials that you have configured for the HDInsight cluster; the SSH endpoint credentials are the SSH credentials that you have configured for the HDInsight cluster.

Troubleshoot

See [Troubleshoot the installation](#).

Next steps

- [Install custom HDInsight applications](#): learn how to deploy an un-published HDInsight application to HDInsight.
- [Publish HDInsight applications](#): Learn how to publish your custom HDInsight applications to Azure Marketplace.
- [MSDN: Install an HDInsight application](#): Learn how to define HDInsight applications.
- [Customize Linux-based HDInsight clusters using Script Action](#): learn how to use Script Action to install additional applications.
- [Create Linux-based Hadoop clusters in HDInsight using Resource Manager templates](#): learn how to call Resource Manager templates to create HDInsight clusters.
- [Use empty edge nodes in HDInsight](#): learn how to use an empty edge node for accessing HDInsight cluster, testing HDInsight applications, and hosting HDInsight applications.

Install published application - Dataiku DDS on Azure HDInsight

8/16/2017 • 5 min to read • [Edit Online](#)

In this article, you will learn how to install the [Dataiku DDS](#) published Hadoop application on Azure HDInsight. Read [Install third-party Hadoop applications](#) for a list of available Independent Software Vendor (ISV) applications, as well as an overview of the HDInsight application platform. For instructions on installing your own application, see [Install custom HDInsight applications](#).

About Dataiku DSS

Dataiku develops [Data Science Studio \(DSS\)](#), a collaborative data science platform that enables professionals (data scientists, data engineers etc.) to build and deliver their analytical solutions more efficiently. Offering DSS as an HDInsight application enables customers to easily use data science to build big data solutions and run them at enterprise grade and scale.

A user can use DSS to implement a complete analytical solution - which could range from data ingestion (all data types, sizes, format etc.), data preparation, data processing, training and applying machine learning models, visualization and operationalizing the solution.

A customer can install DSS on HDInsight using Hadoop or Spark clusters. They can install DSS on existing clusters which are running, or while creating new clusters. DSS 4.0 also added support for using Azure Blob Storage as a connector for reading data.

When a user installs DSS on HDInsight, the user can make use of the benefits of Hadoop or Spark on HDInsight. Users can utilize DSS to build projects; the projects can generate MapReduce or Spark jobs, which makes DSS a great compliment to your HDInsight cluster. These jobs are executed as regular MapReduce or Spark jobs, and hence they get all the benefits of running these jobs on an enterprise grade platform. Since these jobs are running on HDInsight, customers can scale the cluster on demand, which allows a customer to run DSS at scale on HDInsight.

Installing the Dataiku DSS published application

For step-by-step instructions on installing this and other available ISV applications, please read [Install third-party Hadoop applications](#).

Prerequisites

When creating a new HDInsight cluster, or to install on an existing one, you must have the following configuration to install this app:

- Cluster tier(s): Standard, Premium
- Cluster type(s): Hadoop, Spark
- Cluster version(s): 3.4, 3.5

Launching Dataiku DSS for the first time

After installation, you can launch DSS from your cluster in Azure Portal by going to the the **Settings** blade, then clicking **Applications** under the **General** category. The Installed Apps blade lists all the installed applications.

The screenshot shows two windows side-by-side. The left window is titled 'Installed apps' and lists a single application: 'DSS on HDInsight' by Dataiku, version 1.0.3, status 'Installed'. The right window is titled 'Properties' for the selected app. It shows the 'APP NAME' as 'DSS on HDInsight' and the 'STATUS' as 'Installed'. A red box highlights the 'WEBPAGE' field, which contains a link: [https://joelspark-dss.apps.azurehdinsight...](https://joelspark-dss.apps.azurehdinsight.net). Below it is the 'SSH ENDPOINT' field, which contains the value 'dss.joelspark-ssh.azurehdinsight.net:22'.

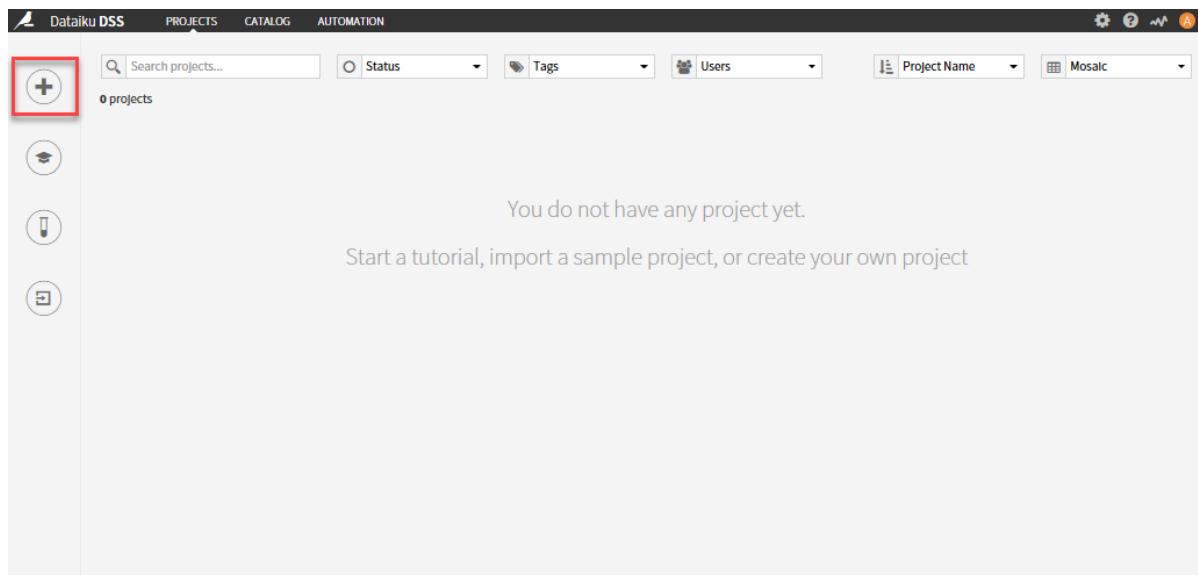
When you select DSS on HDInsight, you'll see a link to the web page, as well as the SSH endpoint path. Select the WEBPAGE link.

On first launch, you'll be presented with a form to create a new Dataiku account for free, or to sign in to an existing account. You'll also have the option to start a free 2-week trial of [Enterprise Edition](#). From this point, you have the option of continuing with entering a license key for Enterprise Edition, or using the Community Edition.

After completing your selected license option, you'll be presented with a login form. Enter the default credentials displayed prior to the login form.

Now that you've logged into Dataiku DSS, please follow these steps for a simple demonstration:

- [Download the sample orders CSV](#).
- From the DSS dashboard, click the + (New project) link on the left-hand menu to create a new project.



- In the New project form, type in a **Name**. The **Project Key** will be automatically filled with a suggested value. In this case, enter "Orders". Click **CREATE**.

+ New project

Name: Orders

Project Key: ORDERS

The project key is used to reference datasets between projects. It cannot be changed once the project is created.

CANCEL CREATE

- Click + IMPORT YOUR FIRST DATASET in your new project page.

Orders

The project *Orders* was created by Joel Hulen on Jun 30th 2017

A + add tags

B Sandbox

Flow Lab Dashboards Tasks

0 DATASETS 0 RECIPES 0 NOTEBOOKS 1 DASHBOARD 4 TASKS

0 MODELS 0 ANALYSES

+ IMPORT YOUR FIRST DATASET

- Select **Upload your files** under the **Files** dataset list. You are presented with the Upload dialog. Click on Add a file, select the `haiku_shirt_sales.csv` file you downloaded, and validate.
- The file is uploaded to DSS. Let's now check if DSS detected our CSV format correctly by clicking on the Preview button:

New Uploaded Files dataset

New dataset name: haiku_shirt_sales

CREATE

Connection Preview Schema Advanced

/haiku_shirt_sales.csv 2.28 MB

Drag and drop your files here or ADD A FILE

Creates a single dataset: multiple files must have the same schema.

✓ Connection to data source succeeded PREVIEW

- The import is almost perfect. The CSV has been detected using a Tab separator. You can see the data is in a tabular format, with columns called features and lines which represent observations. One thing is wrong with our dataset though... Apparently the file contained a blank line between the header and the data. Let's just

input 1 in the Skip next lines to sort this out:

- We can now give our new dataset a name. Enter **haiku_shirt_sales** in the field on top of the screen. Finally, we need to save our work by either hitting the **Create** button or using the shortcut Ctrl-S.
- Your dataset has been created, and you are now taken to a tabular view of your data where you can start exploring it. For each column, you should see that Dataiku Science Studio has detected a meaning, in *blue* (in our case Text, Number or Date (unparsed)). A gauge indicates the ratio of the column for which the values do not seem to match the meaning (in red) or are completely missing (blank). In our dataset, for example, the department has empty values, as well as invalid data.

Data manipulation

The Data Scientists' Murphy's law states that real world data is never in the right format. Cleaning it up usually consists of a chain of scripts with a lot of business logic, that are always difficult to maintain. Sadly, a large part of the job of the Data Scientist is to clean up data. Dataiku DSS has a dedicated tool to make this task more user-friendly.

Let's get started with data manipulation:

- Click on **Lab** in the upper-right corner.



- The Lab window opens. The lab is where you will iteratively work on your dataset to get further into it. In this tutorial, we are going to use the Visual analysis part of the Lab. Click on the **New** button below Visual analysis. You will be prompted to specify a name for your analysis. Let's leave the default name for now, then click **CREATE**.

Lab for **haiku_shirt_sales**

Visual analysis

Workspace for interactive preparation and machine learning

Back

Analysis name CREATE

- Now let's quickly make sense of our data by clicking on the **Quick columns stats** button on the upper-right corner of the page as shown:

ANALYSES

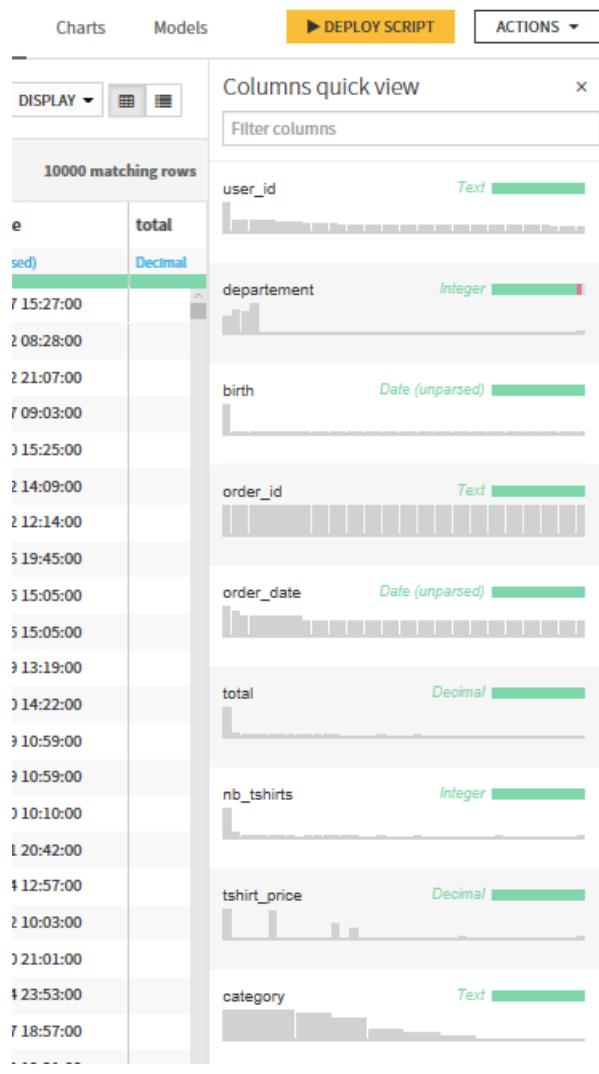
Viewing design sample

10000 rows, 9 cols

Quick columns stats

user_id	departement	birth	order_id	order_date	total	nb_tshirts	tshirt_price	category
Text	Integer	Date (unparsed)	Text	Date (unparsed)	Decimal	Integer	Decimal	Text
a4dc1548af		11/9/1970	TSG-1200493	2013-12-07 15:27:00	71.7	3	23.9	White
e8f6f7853c	91	1/31/1983	TSG-1200500	2014-01-22 08:28:00	19.9	1	19.9	Black
5802162c20	46	6/5/1958	TSG-1200502	2014-03-22 21:07:00	19.9	1	19.9	Black

- You will see statistics based on data type and values displayed in timeline-based graphs under the **Columns quick view** pane.



That's it for now. DSS is such a powerful tool with an intuitive interface, it won't take you very long to create visualizations for this data, figure out ways to clean up the data, and work with it in a multitude of ways.

To view the complete tutorial, along with others that go into greater detail, visit the [Learn Dataiku DSS](#) page next.

Next steps

- Read the Dataiku DSS [reference docs](#).
- [Install custom HDInsight applications](#): learn how to deploy an un-published HDInsight application to HDInsight.
- [Publish HDInsight applications](#): Learn how to publish your custom HDInsight applications to Azure Marketplace.
- [MSDN: Install an HDInsight application](#): Learn how to define HDInsight applications.
- [Customize Linux-based HDInsight clusters using Script Action](#): learn how to use Script Action to install additional applications.
- [Use empty edge nodes in HDInsight](#): learn how to use an empty edge node for accessing HDInsight cluster, testing HDInsight applications, and hosting HDInsight applications.

Install published application - Datameer on Azure HDInsight

8/16/2017 • 4 min to read • [Edit Online](#)

In this article, you will learn how to install the [Datameer](#) published Hadoop application on Azure HDInsight. Read [Install third-party Hadoop applications](#) for a list of available Independent Software Vendor (ISV) applications, as well as an overview of the HDInsight application platform. For instructions on installing your own application, see [Install custom HDInsight applications](#).

About Datameer

Sitting natively on the powerful Hadoop platform, Datameer extends existing Azure HDInsight capabilities by facilitating quick integration, preparation and analysis of all structured and unstructured data. Datameer makes it easy to ingest and integrate data with more than 70 sources and formats: structured, semi-structured, and unstructured. You can directly upload data, or use their unique data links to pull data on demand. Datameer's self-service functionality and familiar spreadsheet interface reduces the complexity of big data technology and dramatically accelerates time to insight. The spreadsheet interface provides a simple mechanism for entering declarative spreadsheet formulas that are translated to fully optimized Hadoop jobs. If you have BI or Excel skills, you can use Hadoop in the cloud quickly.

[Datameer documentation](#)

Installing the Datameer published application

For step-by-step instructions on installing this and other available ISV applications, please read [Install third-party Hadoop applications](#).

Prerequisites

When creating a new HDInsight cluster, or to install on an existing one, you must have the following configuration to install this app:

- Cluster tier: Standard
- Cluster type: Hadoop
- Cluster version: 3.4

Launching Datameer for the first time

After installation, you can launch Datameer from your cluster in Azure Portal by going to the the **Settings** blade, then clicking **Applications** under the **General** category. The Installed Apps blade lists all the installed applications.

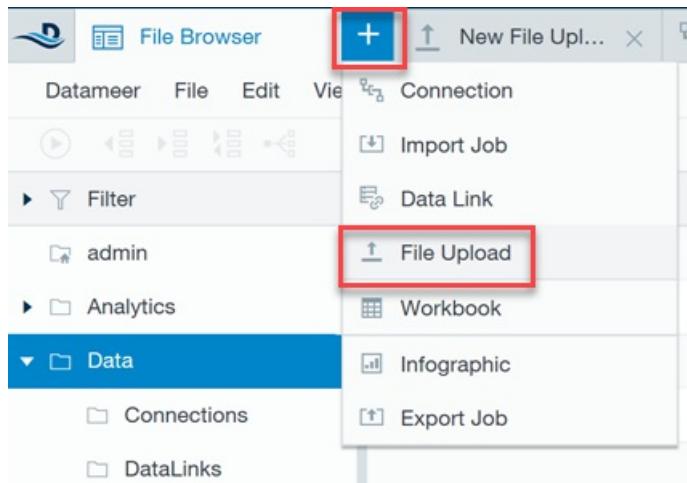
When you select Datameer, you'll see a link to the web page, as well as the SSH endpoint path. Select the WEBPAGE link.

On first launch, you'll be presented with two license options: a free 14 day trial, or the ability to activate an existing license.

After completing your selected license option, you'll be presented with a login form. Enter the default credentials displayed prior to the login form. After logging in, accept the software agreement to continue.

Now that you've logged into Datameer, please follow these steps for a "Hello World" demonstration:

- [Download the sample CSV](#).
- Click the + sign on top of the Datameer dashboard, and click **File Upload**.



- In the upload dialog, browse and select the **Hello World.csv** file you just downloaded. Make sure the **File Type** is set to CSV / TSV. Click **Next**. Keep clicking **Next** until you reach the end of the wizard.

New File Upload

File > Data Details > Define Fields > Configure > Save

File

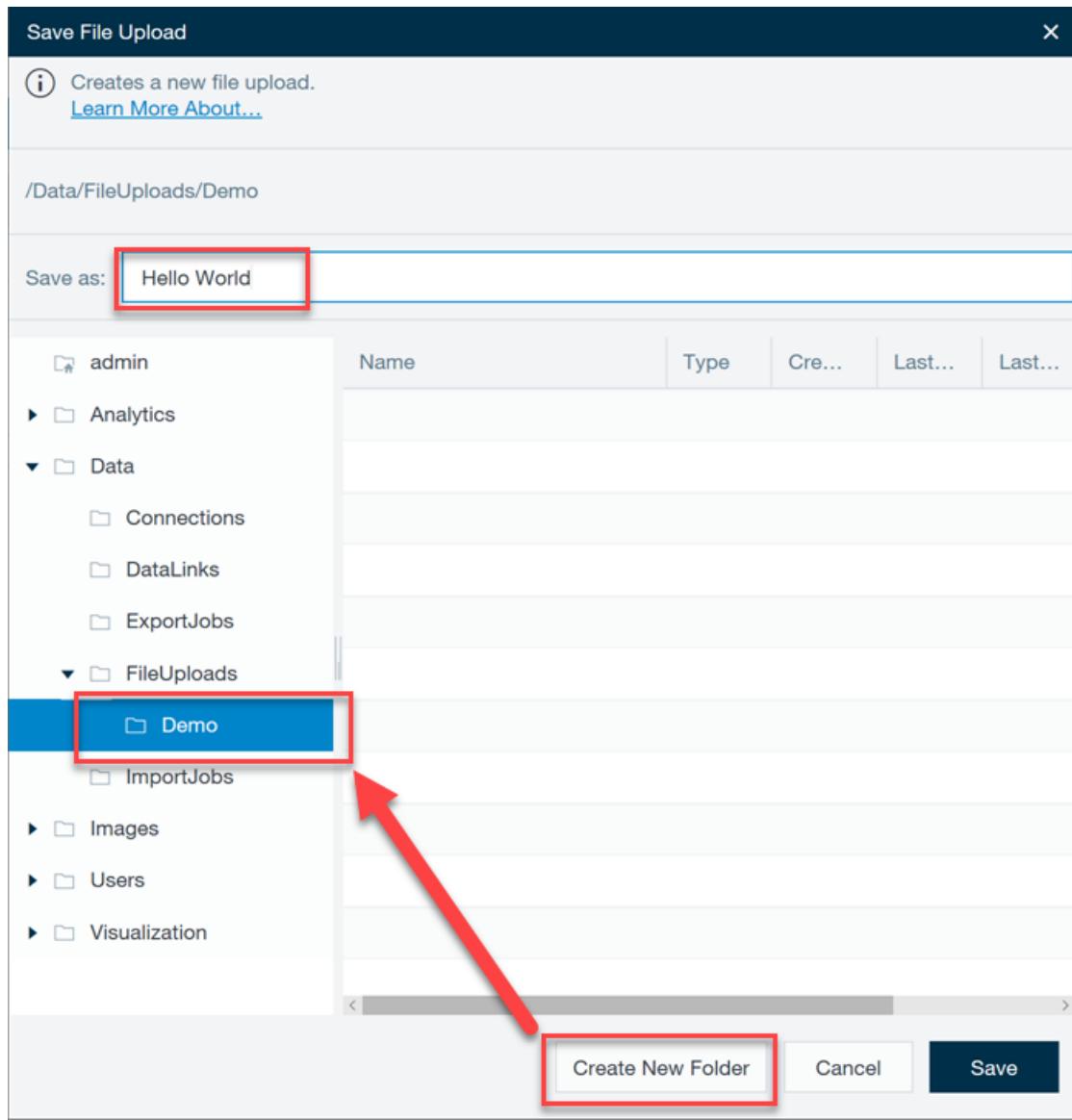
Upload File: *
C:\fakepath\Hello world.csv

File Type

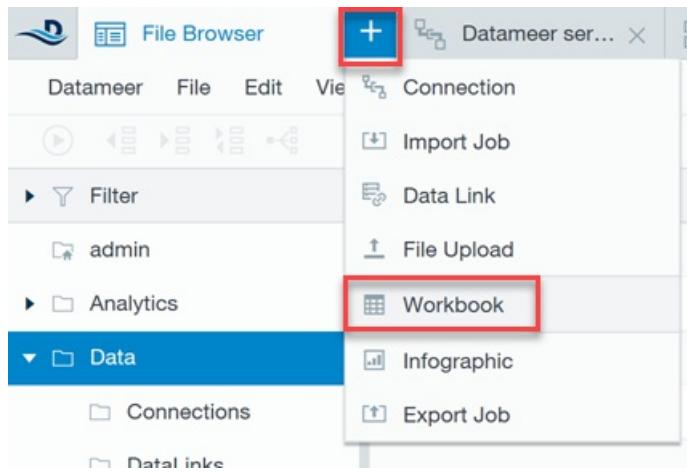
File Type:
CSV / TSV

* required

- Name the file **Hello World** underneath a New Folder. You may rename the new folder as "Demo", or similar. Click **Save**.



- Click the + sign once more and select **Workbook** to create a new Workbook for our data.



- Expand the **Data** folder, **FileUploads**, then the **Demo** folder you created when saving the "Hello World" file. Select **Hello World** from the list of files, then click **Add Data**.

Add Data

This will add a preview of the data to the workbook.
[Learn More About...](#)

/Data/FileUploads/Demo/Hello World

Name	Type	Cre...	Last...	Last...
Hello World	upl	2017-...	2017-...	2017-...

File Tree:

- admin
- Analytics
- Workbooks
- Data
 - Connections
 - DataLinks
 - ExportJobs
 - FileUploads
 - Demo
 - ImportJobs
- Images
- Users
- Visualization

Create New Folder Cancel Add Data

- You will now see the data loaded in a familiar spreadsheet interface. Click the **Filter** button in the toolbar to select a subset of the data.

Datameer File Edit S Filter... (Alt F) Analytic

Toolbar Buttons:

- +
-
- File
- Save
- Print
- Filter (highlighted with a red box)
- Link
- Sort Ascending
- Sort Descending

Table Data:

	Name	Age	City
=	Jacob	21	San Franci

- In the Apply Filter dialog, select the **City** column, **equals** operator, and type **Chicago** in the filter text box. Check the **Create filter in new sheet** checkbox, then click **Create Filter**.

Apply Filter

Filters the current sheet based on the set conditions. The result contains only the records matching the conditions. Multiple conditions can be connected using AND or OR.
[Learn More About...](#)

Apply filter to sheet: Hello_World

Create filter in new sheet

Simple Advanced

Filter column, expression and value

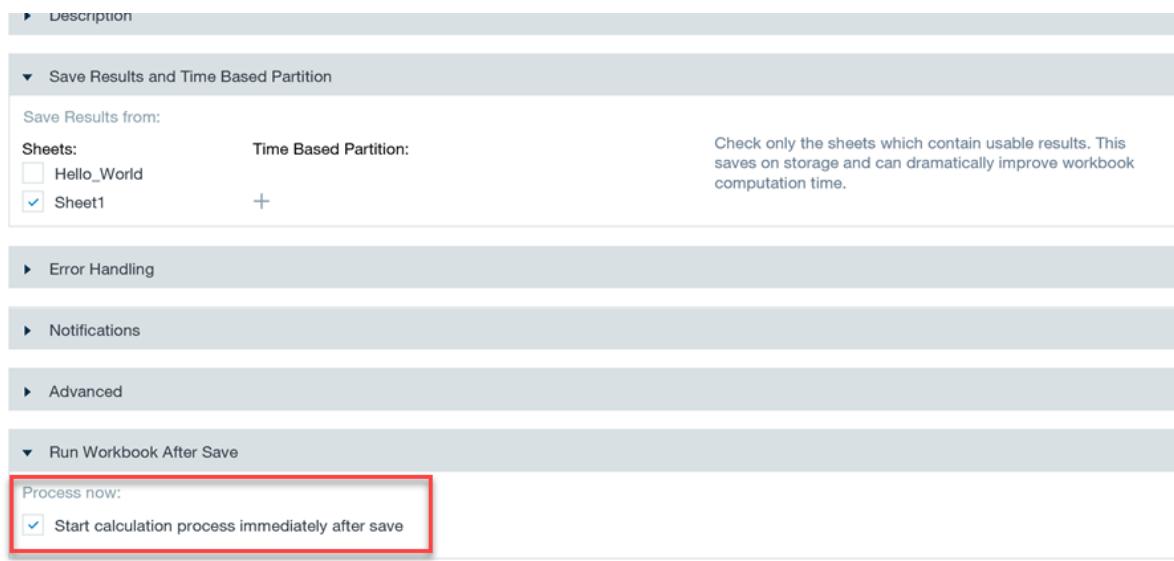


Multiple filters are connected with

AND (all criteria must match)
 OR (one criteria must match)

Cancel Create Filter

- Save the Workbook by clicking **File**, then **Save**. Supply a name, such as "Hello World Workbook".
- After entering your file name and clicking Save, you'll be presented with options for how and when to run the Workbook. For now, leave all of the options at their default values, then check the **Start calculation process immediately after save**, and click **Save**.

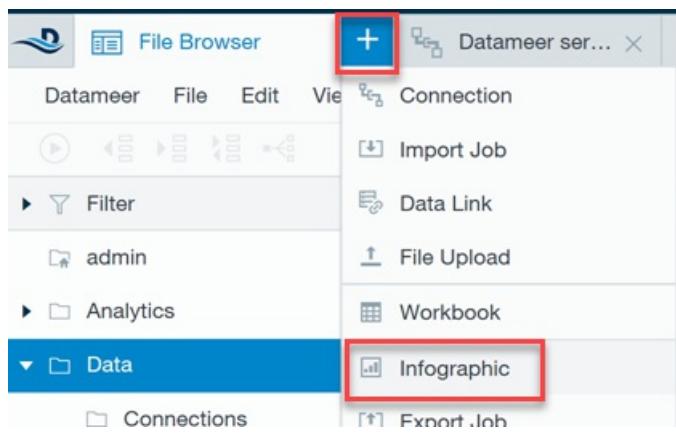


Process now:

Start calculation process immediately after save

Close Save

- Datameer provides powerful visualization tools. To display our data, we'll create an Infographic. Once again, click the + sign on top of the dashboard, then select **Infographic**.



- Drag a Bar Chart widget from the list of widgets on the left (step 1). Next, navigate through the Data folder under the data browser on the right, expand your Workbook, then your worksheet you added with the filter (step 2). Drag the **Name** column over top of the bar chart. Drop it into the **Label** target to set the Workbook's Name column as the chart's label field.

- Now drag the **Age** column into the chart's **Data* field, thus setting Age as the chart's Y axis.

Congratulations! At this point, you've created a nice visualization of your data without writing any code. Feel free to add text over top of your chart, change colors, and adding additional visualizations to discover the great options at your disposal.

Next steps

- Read the Datameer [documentation](#)
- [Install custom HDInsight applications](#): learn how to deploy an un-published HDInsight application to HDInsight.
- [Publish HDInsight applications](#): Learn how to publish your custom HDInsight applications to Azure Marketplace.
- [MSDN: Install an HDInsight application](#): Learn how to define HDInsight applications.
- [Customize Linux-based HDInsight clusters using Script Action](#): learn how to use Script Action to install additional applications.
- [Use empty edge nodes in HDInsight](#): learn how to use an empty edge node for accessing HDInsight cluster, testing HDInsight applications, and hosting HDInsight applications.

Install published application - H2O Sparkling Water on Azure HDInsight

8/16/2017 • 3 min to read • [Edit Online](#)

In this article, you will learn how to install the [H2O Sparkling Water](#) published Hadoop application on Azure HDInsight. Read [Install third-party Hadoop applications](#) for a list of available Independent Software Vendor (ISV) applications, as well as an overview of the HDInsight application platform. For instructions on installing your own application, see [Install custom HDInsight applications](#).

About H2O Sparkling Water

H2O Sparkling Water is a 100% open source, fully distributed in-memory machine learning platform with linear scalability. It allows users to combine the fast, scalable machine learning algorithms of H2O with the capabilities of Spark. With Sparkling Water, users can drive computation from Scala/R/Python and utilize the H2O Flow UI, providing an ideal machine learning platform for application developers.

H2O Sparkling Water intelligently combines the following features:

- **Best of Breed Open Source Technology** – Enjoy the freedom that comes with big data science powered by open source technology.
- **Easy-to-use WebUI and Familiar Interfaces** – Set up and get started quickly using either H2O's intuitive web-based Flow GUI or familiar programming environments like R, Python, Java, Scala, JSON, and through our powerful APIs.
- **Data Agnostic Support for all Common Database and File Types** – Easily explore and model big data from within Microsoft Excel, R Studio, Tableau and more. Connect to data from HDFS, S3, SQL and NoSQL data sources.
- **Massively Scalable Big Data Munging and Analysis** – H2O Big Joins performs 7x faster than R data.table in a benchmark, and linearly scales to 10 billion x 10 billion row joins.
- **Real-time Data Scoring** – Rapidly deploy models to production via plain-old Java objects (POJO), model-optimized Java objects (MOJO) or REST API

Resource links

- [H2O.ai Engineering Roadmap](#)
- [H2O.ai Home](#)
- [H2O.ai Documentation](#)
- [H2O.ai Support](#)
- [H2O.ai Open Source Codebase](#)

Installing the H2O Sparkling Water published application

For step-by-step instructions on installing this and other available ISV applications, please read [Install third-party Hadoop applications](#).

Prerequisites

When creating a new HDInsight cluster, or to install on an existing one, you must have the following configuration to install this app:

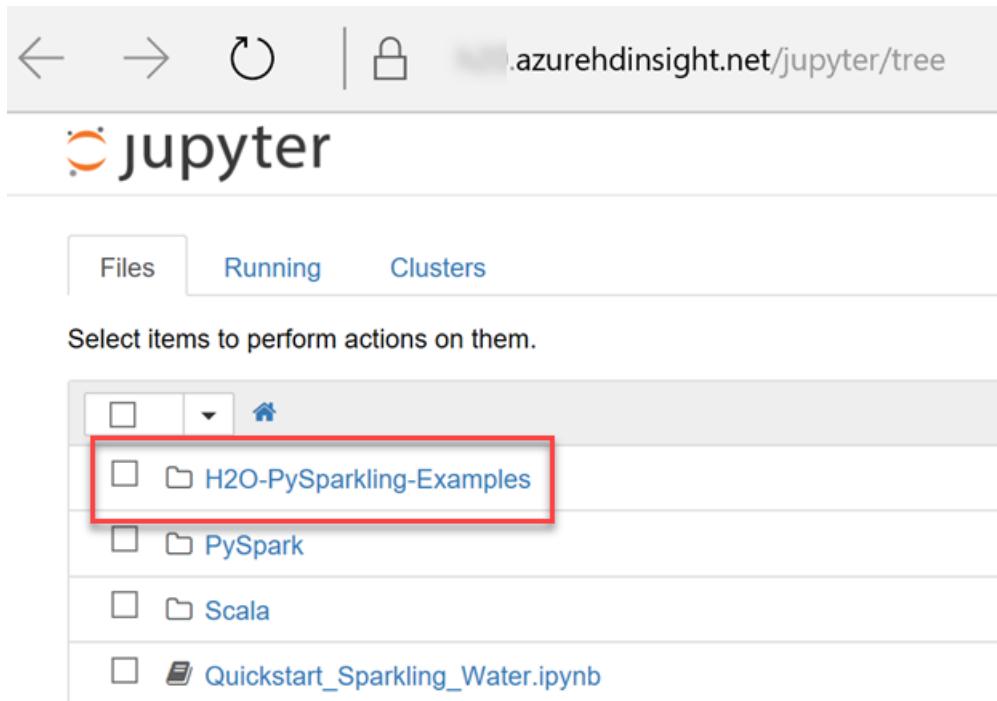
- Cluster tier(s): Standard or Premium

- Cluster type: Spark
- Cluster version(s): 3.5 or 3.6

Launching H2O Sparkling Water for the first time

After installation, you can start using H2O Sparkling Water (h2o-sparklingwater) from your cluster in Azure Portal by opening Jupyter Notebooks (<https://<ClusterName>.azurehdinsight.net/jupyter>). An alternate way to get to Jupyter is by selecting **Cluster dashboard** from your cluster blade in the portal, then selecting **Jupyter Notebook**. You will be prompted to enter your credentials. Enter the cluster's hadoop credentials you specified on cluster creation.

In Jupyter, you will see 3 folders: H2O-PySparkling-Examples, PySpark Examples, and Scala Examples. Select the **H2O-PySparkling-Examples** folder.



Now that you've logged into Jupyter Notebook and selected the H2O folder, please follow these steps for a "Hello World" demonstration:

- The first step when creating a new notebook is to configure the Spark environment. This information is included in the **Sentiment_analysis_with_Sparkling_Water** example. When configuring the Spark environment, be sure to use the correct jar, and specify the IP address provided by the output of the first cell.

```
In [1]: %%configure -f
{
  "conf": {
    "spark.ext.h2o.announce.rest.url": "http://ed10-h20.stmzo4fs3jyufpodoxbmrnqzg.dx.internal.cloudapp.net:5000/flows",
    "spark.jars": "wasb://H2O-Sparkling-Water-files/sparkling-water-assembly-all.jar",
    "spark.submit.pyFiles": "wasb://H2O-Sparkling-Water-files/pySparkling.egg",
    "spark.locality.wait": "3000",
    "spark.scheduler.minRegisteredResourcesRatio": "1",
    "spark.task.maxFailures": "1",
    "spark.yarn.am.extraJavaOption": "-XX:MaxPermSize=384m",
    "spark.yarn.max.executor.failures": "1",
    "maximizeResourceAllocation": "true"
  },
  "driverMemory": "21G",
  "executorMemory": "21G",
  "numExecutors": 3
}
```

- Start the H2O Cluster.

```
In [ ]: import pyspark
import pysparkling, h2o
import os
os.environ["PYTHON_EGG_CACHE"] = "~/"

h2o_context = pysparkling.H2OContext.getOrCreate(sc)
```

- Once the H2O Cluster is up and running, open H2O Flow by going to

<https://<ClusterName>-h2o.apps.azurehdinsight.net:443>.

Note: If you are unable to open H2O Flow (it just redirects you to a help page), try clearing your browser cache. If still unable to reach it, you likely do not have enough resources on your cluster. Try increasing the number of Worker nodes under the **Scale cluster** option in your cluster blade.

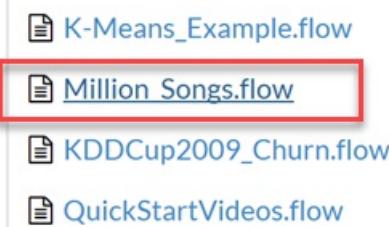
The screenshot shows the H2O Flow application. At the top, there's a navigation bar with 'H2O FLOW' and various dropdown menus like 'Flow', 'Cell', 'Data', 'Model', 'Score', 'Admin', and 'Help'. Below the navigation is a toolbar with icons for file operations. The main area has a search bar with 'assist' typed in. To the left, there's a 'Assistance' panel listing various H2O routines with their descriptions. On the right, there's a 'Help' sidebar with tabs for 'OUTLINE', 'FLOWS', 'CLIPS', and 'HELP' (which is selected). The 'HELP' tab shows a list of example flows.

Routine	Description
importFiles	Import file(s) into H ₂ O
getFrames	Get a list of frames in H ₂ O
splitFrame	Split a frame into two or more frames
mergeFrames	Merge two frames into one
getModels	Get a list of models in H ₂ O
getGrids	Get a list of grid search results in H ₂ O
getPredictions	Get a list of predictions in H ₂ O
getJobs	Get a list of jobs running in H ₂ O
buildModel	Build a model
importModel	Import a saved model
predict	Make a prediction
getRDDs	Get a list of Spark's RDDs
getDataFrames	Get a list of Spark's data frames

HELP

- examples
- 2016_H2O_Tour_Chicago.flow
- Amazon_Fine_Food_Sentiment_Analysis.flow
- Lending_Club.flow
- Strata_SJ_2016.flow
- Spark_SVM_Model.flow
- DataSourceAPI_Example.flow
- Ham_Or_Spam_Pipeline.flow
- GBM_Example.flow
- DeepLearning_MNIST.flow
- GLM_Example.flow
- DRF_Example.flow
- K-Means_Example.flow
- Million_Songs.flow

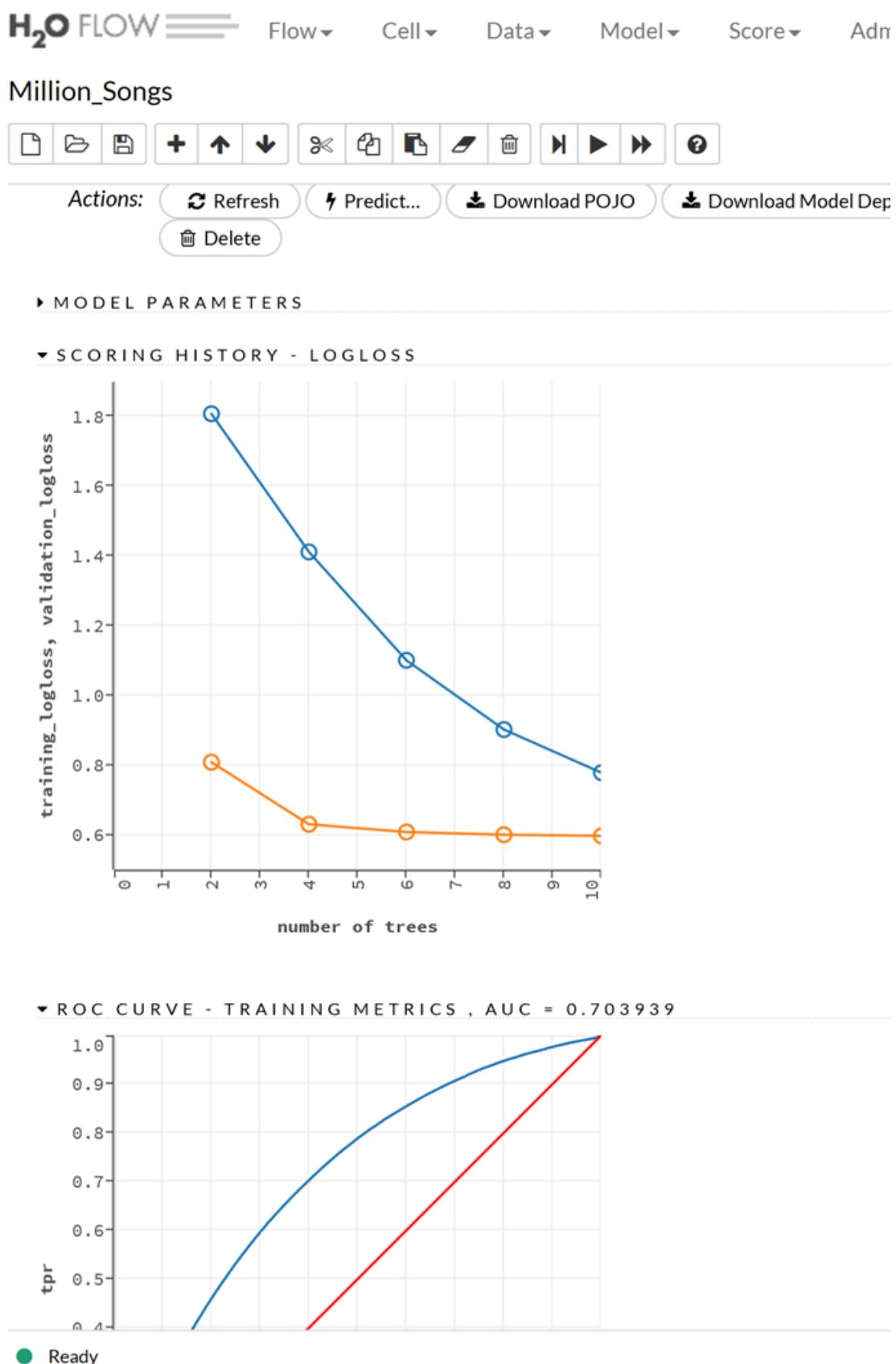
- Select the **Million_Songs.flow** example from the menu on the right. When prompted with a warning, click **Load Notebook**. This demo is designed to run in a few minutes using real data. The goal is to predict from the data whether the song was released before or after 2004 using binary classification.



- Find the path containing **milsongs-cls-train.csv.gz**, and replace the entire path with <https://h2o-public-test-data.s3.amazonaws.com/bigdata/laptop/milsongs/milsongs-cls-train.csv.gz>.
- Find the path containing **milsongs-cls-test.csv.gz** and replace with <https://h2o-public-test-data.s3.amazonaws.com/bigdata/laptop/milsongs/milsongs-cls-test.csv.gz>.
- Click the **Run All** button on the toolbar to execute all statements within the notebook cells.



- After several minutes, you should see an output similar to the following:



That's it! You've harnessed artificial intelligence in Spark within a matter of minutes. Feel free to explore more examples in H2O Flow to get a feel for the different types of machine learning algorithms you can use.

Next steps

- Read the H2O documentation

- [Install custom HDInsight applications](#): learn how to deploy an un-published HDInsight application to HDInsight.
- [Publish HDInsight applications](#): Learn how to publish your custom HDInsight applications to Azure Marketplace.
- [MSDN: Install an HDInsight application](#): Learn how to define HDInsight applications.
- [Customize Linux-based HDInsight clusters using Script Action](#): learn how to use Script Action to install additional applications.
- [Use empty edge nodes in HDInsight](#): learn how to use an empty edge node for accessing HDInsight cluster, testing HDInsight applications, and hosting HDInsight applications.

Install published application - StreamSets Data Collector on Azure HDInsight

8/16/2017 • 2 min to read • [Edit Online](#)

In this article, you will learn how to install the [StreamSets Data Collector for HDInsight](#) published Hadoop application on Azure HDInsight. Read [Install third-party Hadoop applications](#) for a list of available Independent Software Vendor (ISV) applications, as well as an overview of the HDInsight application platform. For instructions on installing your own application, see [Install custom HDInsight applications](#).

About StreamSets Data Collector

StreamSets Data Collector deploys on top of Azure HDInsight application. It provides a full-featured integrated development environment (IDE) that lets you design, test, deploy, and manage any-to-any ingest pipelines that mesh stream and batch data, and include a variety of in-stream transformations—all without having to write custom code.

StreamSets Data Collector lets you build data flows, including numerous Big Data components such as HDFS, Kafka, Solr, Hive, HBASE, and Kudu. Once StreamSets Data Collector is running on edge or in your Hadoop cluster, you get real-time monitoring for both data anomalies and data flow operations, including threshold-based alerting, anomaly detection, and automatic remediation of error records.

Because it is architected to logically isolate each stage in a pipeline, you can meet new business requirements by dropping in new processors and connectors without code and with minimal downtime.

Resource links

- [StreamSets Docs](#)
- [Blog](#)
- [Tutorials](#)
- [Developer Support Forum](#)
- [Slack Public StreamSets Channel](#)
- [Source Code](#)

Installing the StreamSets Data Collector published application

For step-by-step instructions on installing this and other available ISV applications, please read [Install third-party Hadoop applications](#).

Prerequisites

When creating a new HDInsight cluster, or to install on an existing one, you must have the following configuration to install this app:

- Cluster tier(s): Standard or Premium
- Cluster version(s): 3.5 and above

Launching StreamSets Data Collector for the first time

After installation, you can launch StreamSets from your cluster in Azure Portal by going to the the **Settings** blade, then clicking **Applications** under the **General** category. The Installed Apps blade lists all the installed applications.

Name	Publisher	Version	Status
StreamSets Data Collector...	StreamSets	1.0.12	Installed

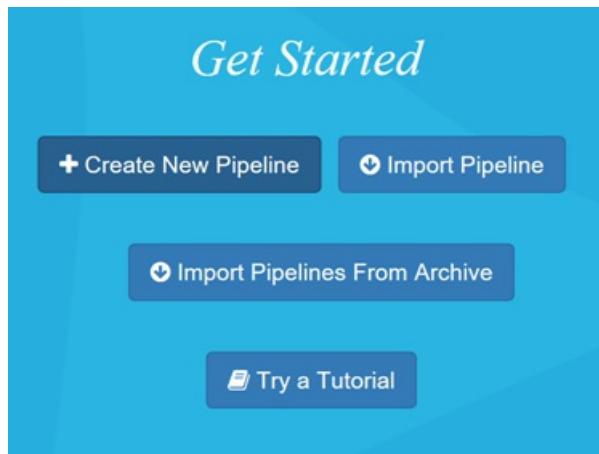
WEBPAGE
<https://spark-hdiz-docs-sdc.apps.azurehdinsigh...>

SSH ENDPOINT
<sdc.spark-hdiz-docs-ssh.azurehdinsigh...>

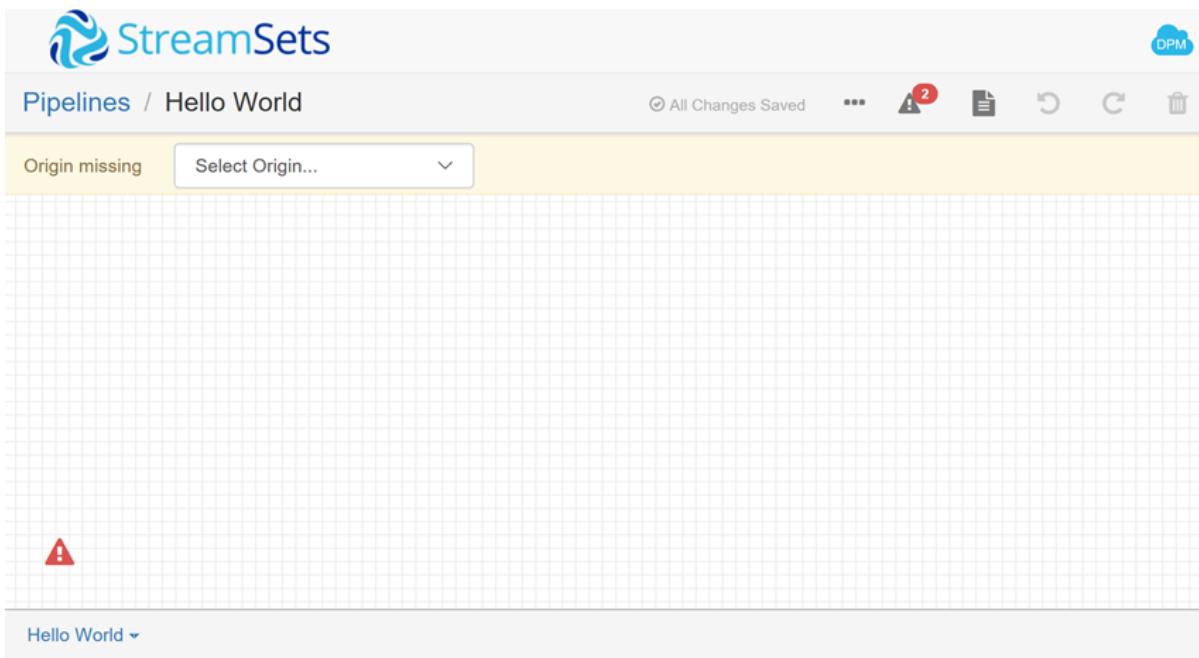
When you select StreamSets Data Collector, you'll see a link to the web page, as well as the SSH endpoint path. Select the WEBPAGE link.

In the Login dialog box, use the following credentials to log in: admin / admin.

- On the Get Started page, click **Create New Pipeline**.



- In the New Pipeline window, enter a name for the pipeline ("Hello World"), optionally enter a description, and click **Save**.
- The Data Collector console will appear. The Properties panel displays pipeline properties.



- You are now ready to [follow the official StreamSets tutorial](#). This will provide you with detailed step-by-step directions to create your first pipeline.

Next steps

- Read the StreamSets Data Collector [documentation](#)
- [Install custom HDInsight applications](#): learn how to deploy an un-published HDInsight application to HDInsight.
- [Publish HDInsight applications](#): Learn how to publish your custom HDInsight applications to Azure Marketplace.
- [MSDN: Install an HDInsight application](#): Learn how to define HDInsight applications.
- [Customize Linux-based HDInsight clusters using Script Action](#): learn how to use Script Action to install additional applications.
- [Use empty edge nodes in HDInsight](#): learn how to use an empty edge node for accessing HDInsight cluster, testing HDInsight applications, and hosting HDInsight applications.

Install published application - Cask Data Application Platform (CDAP) on Azure HDInsight

8/16/2017 • 5 min to read • [Edit Online](#)

In this article, you will learn how to install the [CDAP](#) published Hadoop application on Azure HDInsight. Read [Install third-party Hadoop applications](#) for a list of available Independent Software Vendor (ISV) applications, as well as an overview of the HDInsight application platform. For instructions on installing your own application, see [Install custom HDInsight applications](#).

About CDAP

Developing applications in the traditional Hadoop world is not an easy task. Listed below are some key aspects that add to the challenges faced by a Hadoop developer:-

- Over the past few years, the increased interest in the Big Data space has resulted in a technology explosion in the Hadoop ecosystem. It has become progressively difficult to keep track of all the existing technologies as well as new ones come up.
- Simple processes like data ingestion and ETL require a complicated setup which is not generally extensible or reusable.
- Apart from the significant learning curve involved in using each of the different Hadoop technologies, there is a substantial amount of time spent in integrating all of them to form a data processing solution.
- Moving from a proof-of-concept solution to a production-ready one is far from a trivial step involving multiple iterations and can lead to an increased unpredictability in delivery times.
- It is hard to locate data and trace its flow in an application. Collecting metrics and auditing is generally a challenge and often requires building a separate solution.

How does CDAP help?

CDAP (Cask Data Application Platform) is a unified integration platform for big data. The highlight of CDAP is that a user can focus on building applications rather than its underlying infrastructure and integration.

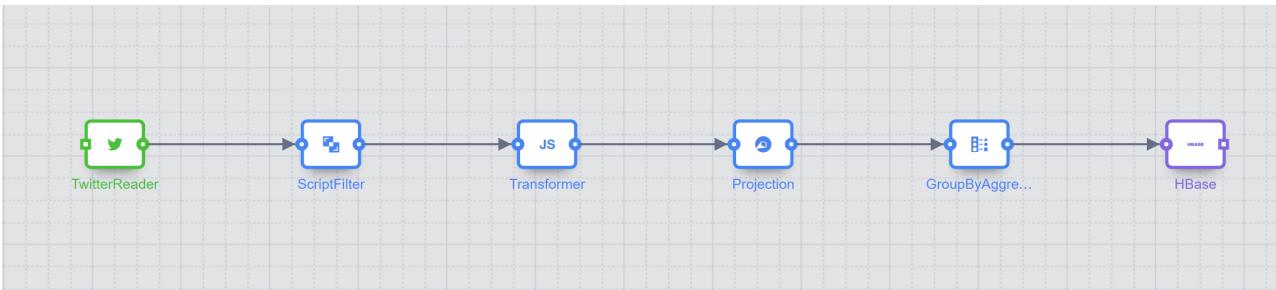
CDAP works using high-level concepts and abstractions which are familiar to developers and empowers them to use their existing skills to build new solutions. These abstractions hide the complexities of internal systems and encourage re-usability of solutions.

An extension called [Cask Hydrator](#) is available in CDAP, which provides a rich user interface to develop and manage data pipelines. A data pipeline is composed of various plugins which perform several tasks like data acquisition, transformation, analysis, and post-run operations.

Each CDAP **plugin** has well-defined interfaces which essentially means that evaluating different technologies would just be a matter of replacing a plugin with another one – there is no need to touch the rest of the application.

CDAP **pipelines** provide a high-level pictorial flow of the data in your application which enables developers to easily visualize the end-to-end flow of the data and all the steps involved in the processing starting from its ingestion, to the various transformations and analyses performed on the data followed by the eventual writing into an external data store.

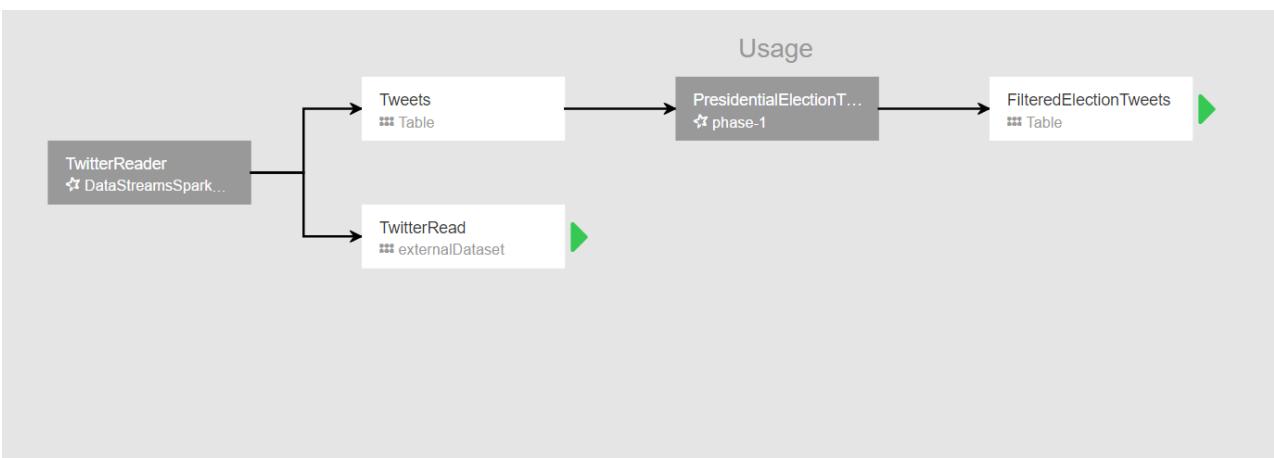
Here is an example of a data pipeline which ingests twitter data in real time, filters out some tweets based on some pre-defined criteria, transforms, and projects the data into a more readable format, groups them according to a set of values and writes the results into an HBase store.



The end-to-end pipeline was completely built using the **Cask Hydrator UI**, utilizing its plugin interface and drag-and-drop functionality to form connections between each stage. It is easy to isolate and modify the functionality of each plugin independent of the rest of the pipeline. Using CDAP, similar pipelines can be built and validated in less than a couple of hours. In the traditional Hadoop world, constructing such solutions could easily take a few days.

Additionally, CDAP provides an extension called **Cask Tracker** where you can visually trace the data as it flows through the application. Cask tracker adds **data governance** to the system so that data assets are formally managed throughout the application. You can track its lineage, collect relevant metrics, and audit the data trail throughout the process.

Here is an illustration of how data is flowing in the above pipeline:



Installing the CDAP published application

For step-by-step instructions on installing this and other available ISV applications, please read [Install third-party Hadoop applications](#).

Prerequisites

When creating a new HDInsight cluster, or to install on an existing one, you must have the following configuration to install this app:

- Cluster tier: Standard
- Cluster type: HBase
- Cluster version: 3.4, 3.5

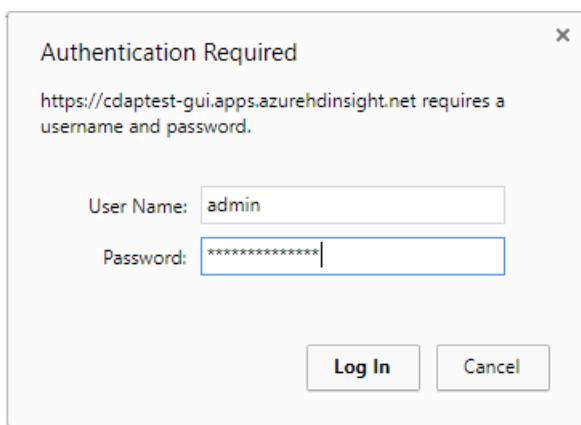
Launching CDA: for the first time

After installation, you can launch CDAP from your cluster in Azure Portal by going to the the **Settings** blade, then clicking **Applications** under the **General** category. The Installed Apps blade lists all the installed applications.

The screenshot shows the Azure portal's 'Properties' view for an application named 'CDAP 4.2 for HDInsight'. The application is listed in the 'NAME' column, published by 'Cask' with version 1.0.2, and is currently 'Installed' with a status of 'Portal'. On the right, under the 'Properties' section, there are three endpoints: 'WEBPAGE' (link to https://cdaptest-gui.apps.azurehdinsight....), 'HTTP ENDPOINT' (link to https://cdaptest-api.apps.azurehdinsig...), and 'SSH ENDPOINT' (link to cdap.cdaptest-ssh.azurehdinsight.net:22). The 'WEBPAGE' link is highlighted with a red box.

When you select CDAP, you'll see a link to the web page, HTTP Endpoint, as well as the SSH endpoint path. Select the WEBPAGE link.

When prompted, enter your cluster admin credentials.



After signing in, you will be presented with the Cask CDAP GUI home page.

The screenshot shows the Cask interface with the following navigation bar:

- CASK
- Overview
- Data
- Pipelines
- Metadata ▾
- Cask Market
- Namespace default
- CDAP Distributed

Below the navigation bar, there are search and filter controls:

- Search
- Filter by
- Sort by: Newest

A green circular button with a plus sign is located in the top right corner.

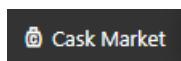
The main content area displays the message: "Entities in Namespace \"default\"". Below this, it says "Displaying All Entities, sorted by Newest".

In the center, it states: "No entities found in namespace \"default\"".

Below this message, there is a section titled "You can try to:" with two options:

- Add new entities; or
- Browse Cask Market

To get an idea of using the CDAP interface, click the **Cask Market** menu link on top of the page.



Select the **Access Log Sample** from the list.

Cask Market

All

- Solutions
- Pipelines
- Applications
- Plugins
- Datapacks
- Drivers
- EDW Offload
- Azure
- AWS



Access Log Sample
v 1.0.0



Browser Mapping
v 1.0.0



Sample Contact Data
v 1.0.0



Organization Data from Crunchbase
v 1.0.0



New York Times World News Feed
v 1.0.0



OmniTure Hits Sample Datapack
v 1.0.0

Click **Load** to confirm.



Access Log Sample
Version 1.0.0
Company Cask Data, Inc.
Author Cask

Sample access logs in Combined Log Format (CLF).

Load **Close**

A sample view of the included data will be displayed. Click **Next**.

Access Log Sample | Upload Data

View Data

View Data
Shows the data that would be uploaded to a destination for your

Select Destination

Download

```
69.181.160.120 - - [08/Feb/2015:04:36:40 +0000]
69.181.160.120 - - [08/Feb/2015:04:36:47 +0000]
69.181.160.120 - - [08/Feb/2015:04:36:56 +0000]
69.181.160.120 - - [08/Feb/2015:04:37:11 +0000]
69.181.160.120 - - [08/Feb/2015:04:37:17 +0000]
69.181.160.120 - - [08/Feb/2015:04:37:27 +0000]
69.181.160.120 - - [08/Feb/2015:04:37:43 +0000]
69.181.160.120 - - [08/Feb/2015:04:37:47 +0000]
69.181.160.120 - - [08/Feb/2015:04:37:58 +0000]
69.181.160.120 - - [08/Feb/2015:04:38:14 +0000]
```

Next >

Crunchbase

Select **Stream** as the Destination Type, enter a Destination Name, then click **Finish**.

Access Log Sample | Upload Data

Select a Destination

2 / 2

Select the destination where the data needs to be uploaded.

Select Destination

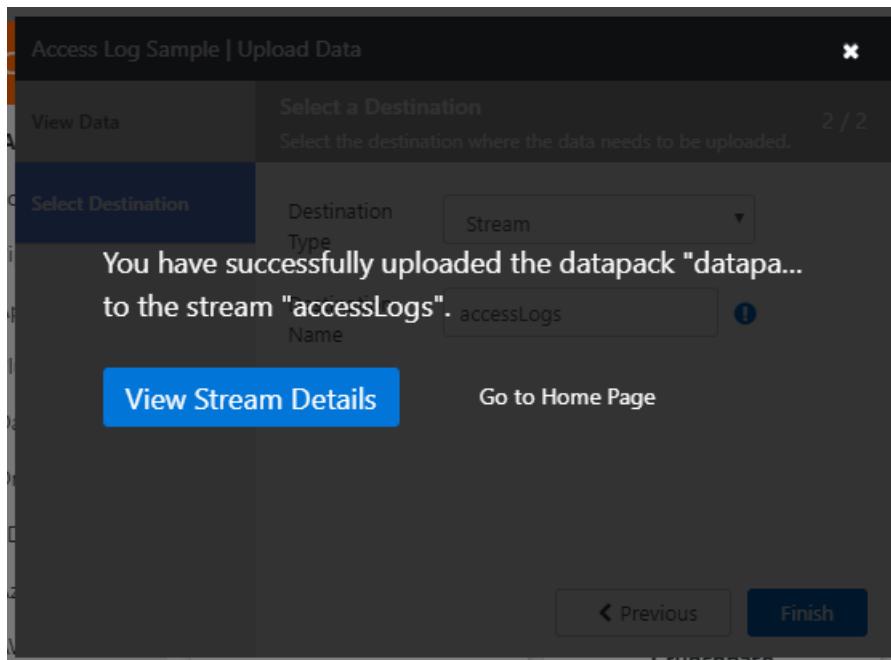
Destination Type Stream

Destination Name accessLogs !

Previous Finish

Crunchbase

Once the datapack has been successfully loaded, click **View Stream Details**.



On the Access Log details page, click **Enable** within the Usage tab to enable metadata for the namespace.

A screenshot of the Cask interface showing the details for the "accessLogs" stream. The top navigation bar includes tabs for Overview, Data, Pipelines, Metadata, Cask Market, Namespace (default), and Distributed. Below the navigation is a breadcrumb trail: < Back | Stream Preparation. On the right is a green circular "Add" button. The main content area shows the stream name "accessLogs", creation date "Created on 08/09/2017, at 06:17 pm", and a row of icons for delete, edit, and other actions. Below this are sections for Description ("No Description available"), Time To Live (TTL) ("Forever"), and Tags ("1: explore"). A "Usage" tab is currently selected, showing sub-links for Schema, Programs (0), Lineage, Audit Log, and Properties. A message states "Metadata is not enabled in this namespace. Please enable it." with an "Enable" button below it. A vertical scrollbar is visible on the right side of the content area.

You will see a graph displaying audit message information, once metadata has been enabled.

CASK Overview Data Pipelines Metadata Cask Market Namespace default Distributed

< Back Stream Preparation 

accessLogs

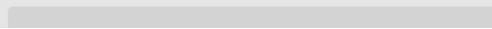
Created on 08/09/2017, at 06:17 pm



Description: No Description available
 Time To Live (TTL): Forever
 Tags(1): [explore](#) 

Usage Schema Programs (0) Lineage Audit Log Properties

Audit Messages Last 7 days 

Most Active Applications All time 

To explore the log data, click the **Explore** icon on top of the page.



You will see a sample SQL query. Feel free to modify, if desired, then click **Execute**.

Explore Dataset 

```
SELECT * FROM default.stream_accesslogs LIMIT 500
```



After the query has finished, click the **View** icon under the Actions column.

Start time	SQL Query	Status	Actions
08-09-2017 18:23:45 PM	SELECT * FROM default.stream_accesslogs LIMIT 500	FINISHED	  

You will now see the query results.

Explore Dataset



```
SELECT * FROM default.stream_accesslogs LIMIT 500
```

Execute

Start time	SQL Query	Status	Actions
08-09-2017 18:23:45 PM	SELECT * FROM default.stream_accesslogs LIMIT 500	FINISHED	
ts	headers	body	
1502317062505	{"content.type":"text/txt"}	69.181.160.120 - - [08/Feb/2015:04:36:40 +0000] "GET /ajax/planStatusHistory/NeighbouringSummaries.action? planKey=COOP-DBT&buildNumber=284&_=1423341312519 HTTP/1.1" 200 508 "http://builds.cask.co/browse/COOP-DBT-284/log" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_10_1) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/38.0.2125.122 Safari/537.36"	
1502317062505	{"content.type":"text/txt"}	69.181.160.120 - - [08/Feb/2015:04:36:47 +0000] "GET /rest/api/latest/server?_=1423341312520 HTTP/1.1" 200 45 "http://builds.cask.co/browse/COOP-DBT-284/log" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_10_1) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/38.0.2125.122 Safari/537.36"	
1502317062505	{"content.type":"text/txt"}	69.181.160.120 - - [08/Feb/2015:04:36:56 +0000] "GET /ajax/planStatusHistory/NeighbouringSummaries.action? planKey=COOP-DBT&buildNumber=284&_=1423341312521 HTTP/1.1" 200 508 "http://builds.cask.co/browse/COOP-DBT-284/log" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_10_1)	

Next steps

- Read the Cask documentation
- [Install custom HDInsight applications](#): learn how to deploy an un-published HDInsight application to HDInsight.
- [Publish HDInsight applications](#): Learn how to publish your custom HDInsight applications to Azure Marketplace.
- [MSDN: Install an HDInsight application](#): Learn how to define HDInsight applications.
- [Customize Linux-based HDInsight clusters using Script Action](#): learn how to use Script Action to install additional applications.
- [Use empty edge nodes in HDInsight](#): learn how to use an empty edge node for accessing HDInsight cluster, testing HDInsight applications, and hosting HDInsight applications.

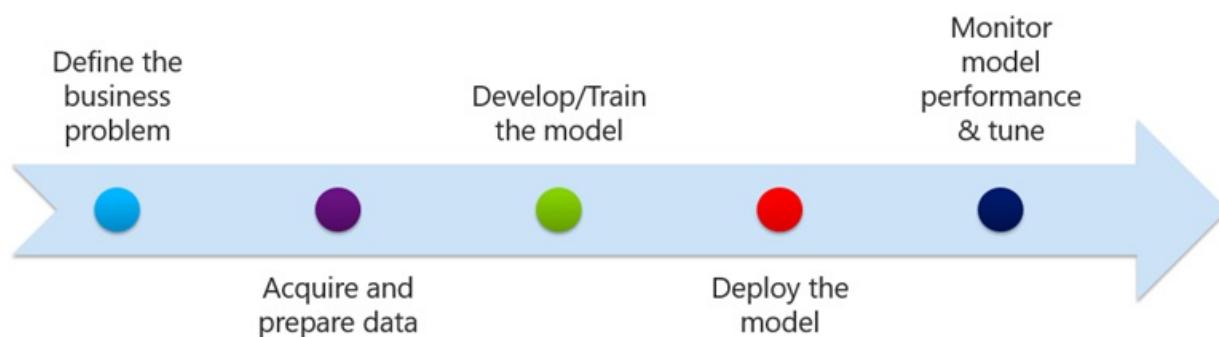
Deep Dive - Advanced Analytics

8/16/2017 • 7 min to read • [Edit Online](#)

What is Advanced Analytics for HDInsight?

HDInsight gives you the power to work with big data, providing the ability to obtain valuable insight from large amounts of structured, unstructured, and fast-moving data. Advanced Analytics is the use of highly scalable architectures, statistical and machine learning models, and intelligent dashboards to provide you with meaningful insights had been out of reach until now. Machine learning, or predictive analytics, are algorithms that identify and learn from relationships in your data to make predictions and guide your decisions.

The advanced analytics process



After you've identified the business problem and have started collecting and processing your data, you need to create a model that represents the question you wish to predict. Your model will use one or more machine learning algorithms to make the type of prediction that best fits your business needs. The majority of your data should be used to train your model, with the smaller portion being used to test or evaluate it.

After you create, load, test and evaluate your model, the next step is to deploy your model so that it can be used for supplying answers to your question. The last step is to monitor your model's performance and tune, if necessary.

Common types of algorithms

An important aspect of any advanced analytics solution is the selection of one or more machine learning algorithms that best suits your requirements. Shown below is a summary of the categories of algorithms and associated common business use cases.



Along with selecting the best-fitting algorithm(s), you need to consider whether or not you'll need to provide data for training. Machine Learning algorithms are categorized in one of several types in this area.

- Supervised - algorithm needs to be trained on a set of labeled data before it can provide results
- Unsupervised - algorithm does not require training data
- Reinforcement - algorithm uses software agents to determine idea behavior within a specific context (often used in robotics)
- Semi-supervised - algorithm can be augmented by extra targets through interactive query by trainer, which were not available during initial stage of training

ALGORITHM CATEGORY	USE	LEARNING TYPE	ALGORITHMS
Classification	Classify people or things into groups	Supervised	Decision trees, Logistic regression, neural networks
Clustering	Dividing a set of examples into homogenous groups	Unsupervised	K-means clustering
Pattern detection	Identify frequent associations in the data	Unsupervised	Association rules
Regression	Predict numerical outcomes	Supervised	Linear regression, neural networks
Reinforcement	Determine optimal behavior for robots	Reinforcement	Monte Carlo Simulations, DeepMind

Machine learning on HDInsight

There are several machine learning options that can be used as part of an advanced analytics workflow in HDInsight:

1. Machine Learning and Spark

[HDInsight Spark](#) is an Azure-hosted offering of [Spark](#), a unified, open source, parallel data processing framework supporting in-memory processing to boost Big Data analytics. The Spark processing engine is built for speed, ease of use, and sophisticated analytics. Spark's in-memory distributed computation capabilities make it a good choice for the iterative algorithms used in machine learning and graph computations.

There are three scalable machine learning libraries that bring the algorithmic modeling capabilities to this

distributed environment:

- **MLlib** - MLlib contains the original API built on top of Spark RDDs
- **SparkML** - SparkML is a newer package that provides a higher-level API built on top of Spark DataFrames for constructing ML pipelines
- **MMLSpark** - The Microsoft Machine Learning library for Apache Spark or MMLSpark is designed to make data scientists more productive on Spark, increase the rate of experimentation, and leverage cutting-edge machine learning techniques, including deep learning, on very large datasets. The MMLSpark library simplifies common modeling tasks for building models in PySpark.

2. R and R Server

As part of HDInsight, you can create an HDInsight cluster with [R Server](#) ready to be used with massive datasets and models. This new capability provides data scientists and statisticians with a familiar R interface that can scale on-demand through HDInsight, without the overhead of cluster setup and maintenance.

3. Azure Machine Learning and Hive

[Azure Machine Learning Studio](#) provides tools to model predictive analytics, as well as a fully managed service you can use to deploy your predictive models as ready-to-consume web services. Azure Machine Learning provides tools for creating complete predictive analytics solutions in the cloud to quickly create, test, operationalize, and manage predictive models. Select from a large algorithm library, use a web-based studio for building models, and easily deploy your model as a web service.

4. Spark and Deep Learning

[Deep learning](#) is a branch of machine learning that uses deep neural networks (or DNNs), inspired by the biological processes of the human brain. Many researchers see deep learning as a very promising approach for making artificial intelligence better. Some examples of deep learning are spoken language translators, image recognition systems, and machine reasoning. To help advance its own work in deep learning, Microsoft has developed the free, easy-to-use, open-source [Microsoft Cognitive Toolkit](#). The toolkit is being used extensively by a wide variety of Microsoft products, by companies worldwide with a need to deploy deep learning at scale, and by students interested in the very latest algorithms and techniques.

In the next section of this article, we'll review an example of an advanced analytics machine learning pipeline using HDInsight.

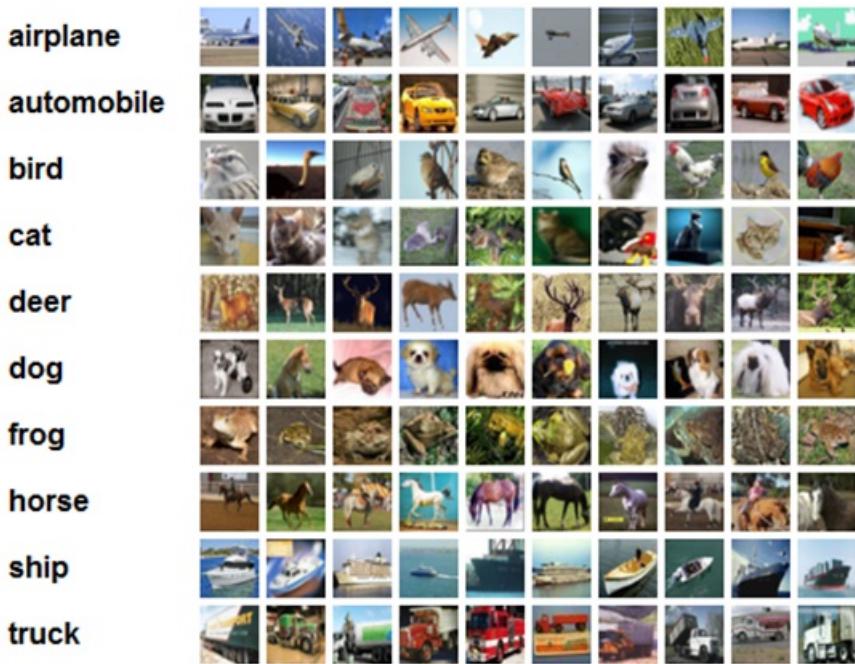
Scenario - Score Images to Identify Patterns in Urban Development

In this scenario you will see how DNNs produced in a deep learning framework, Microsoft's Cognitive Toolkit (CNTK) can be operationalized for scoring large image collections stored in an Azure BLOB Storage Account using PySpark on an HDInsight Spark cluster. This approach is applied to a common DNN use case, aerial image classification, and can be used to identify recent patterns in urban development. You will use a pre-trained image classification model. The model is pre-trained on the [CIFAR-10 dataset](#) and has been applied to 10,000 withheld images.

There are three key tasks in this advanced analytics scenario:

1. Provision an Azure HDInsight Hadoop cluster with an Apache Spark 2.1.0 distribution.
2. Run a custom script to install Microsoft Cognitive Toolkit on all nodes of an Azure HDInsight Spark cluster.
3. Upload a pre-built Jupyter notebook to your HDInsight Spark cluster to apply a trained Microsoft Cognitive Toolkit deep learning model to files in an Azure Blob Storage Account using the Spark Python API (PySpark).

This example uses the recipe on the CIFAR-10 image set compiled and distributed by Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. The CIFAR-10 dataset contains 60,000 32×32 color images belonging to 10 mutually exclusive classes:

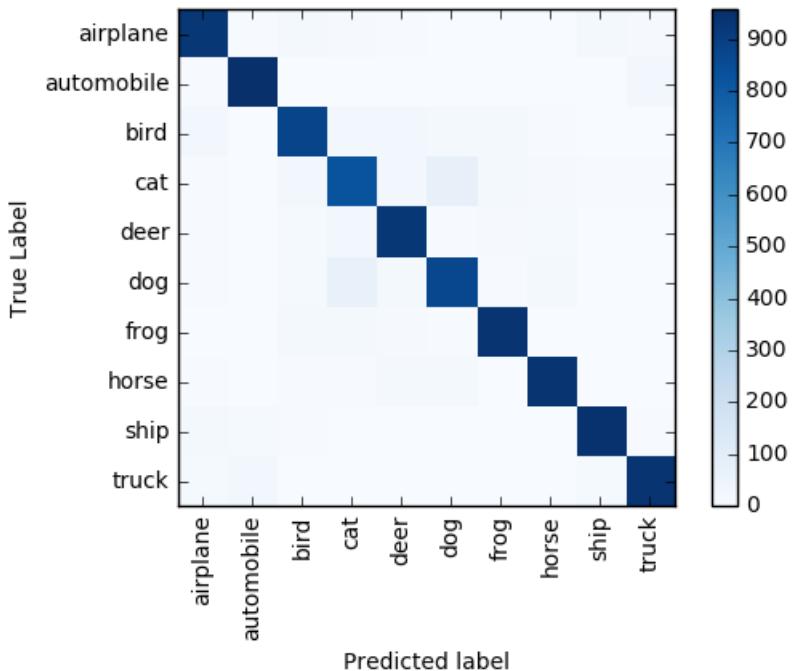


For more details on the dataset, see Alex Krizhevsky's [Learning Multiple Layers of Features from Tiny Images](#).

The dataset was partitioned into a training set of 50,000 images and a test set of 10,000 images. The first set was used to train a twenty-layer deep convolutional residual network (ResNet) model using Cognitive Toolkit by following [this tutorial](#) from the Cognitive Toolkit GitHub repository. The remaining 10,000 images were used for testing the model's accuracy. This is where distributed computing comes into play: the task of pre-processing and scoring the images is highly parallelizable. With the saved trained model in hand, we used:

- PySpark to distribute the images and trained model to the cluster's worker nodes.
- Python to pre-process the images on each node of the HDInsight Spark cluster.
- Cognitive Toolkit to load the model and score the pre-processed images on each node.
- Jupyter Notebooks to run the PySpark script, aggregate the results and use [Matplotlib](#) to visualize the model performance.

The entire preprocessing/scoring of the 10,000 images takes less than one minute on a cluster with 4 worker nodes. The model accurately predicts the labels of ~9,100 (91%) images. A confusion matrix illustrates the most common classification errors. For example, the matrix shows that mislabeling dogs as cats and vice versa occurs more frequently than for other label pairs.



Try it Out!

Follow [this tutorial](#) to implement this solution end-to-end: setup an HDInsight Spark cluster, install Cognitive Toolkit, and run the Jupyter Notebook that scores 10,000 CIFAR images.

See also

Scenarios

Hive and Azure Machine Learning

- [Hive and Azure Machine Learning end-to-end](#)
- [Using an Azure HDInsight Hadoop Cluster on a 1 TB dataset](#)

Spark and MLLib

- [Machine learning with Spark on HDInsight](#)
- [Spark with Machine Learning: Use Spark in HDInsight for analyzing building temperature using HVAC data](#)
- [Spark with Machine Learning: Use Spark in HDInsight to predict food inspection results](#)

Deep Learning, CNTK and others

- [Embarrassingly Parallel Image Classification, Using Cognitive Toolkit and TensorFlow on Azure HDInsight Spark](#)
- [Data Science Azure Virtual Machine](#)
- [Introducing H2O.ai on Azure HDInsight](#)

ETL at Scale

8/16/2017 • 7 min to read • [Edit Online](#)

In this scenario, HDInsight is used to perform an Extract, Transform, and Load (ETL) process that filters and shapes the source data, and then uses it to populate a database table. Specifically, this scenario describes:

- Introduction to racecar telemetry
- ETL process goals and data sources
- The ETL workflow
- Encapsulating the ETL tasks in an Oozie workflow
- Automating the ETL workflow
- Analyzing the loaded data

The scenario demonstrates how you can:

- Use the .NET Library for Avro to serialize data for processing in HDInsight.
- Use the classes in the .NET API for Hadoop WebClient package to upload files to Azure storage.
- Use an Oozie workflow to define an ETL process that includes Pig, Hive, and Sqoop tasks.
- Use the classes in the .NET API for Hadoop WebClient package to automate execution of an Oozie workflow.

Introduction to racecar telemetry

This scenario is based on a fictitious motor racing team that captures and monitors real-time telemetry from sensors on a racecar as it is driven around a racetrack. To perform further analysis of the telemetry data, the team plans to use HDInsight to filter and shape the data before loading it into Azure SQL Database, from where it will be consumed and visualized in Excel. Loading the data into a database for analysis enables the team to decommission the HDInsight server after the ETL process is complete, and makes the data easily consumable from client applications that have the ability to connect to a SQL Server data source.

In this simplified example, the racecar has three sensors that are used to capture telemetry readings at one second intervals: a global positioning system (GPS) sensor, an engine sensor, and a brake sensor. The telemetry data captured from the sensors on the racecar includes:

- From the GPS sensor:
 - The date and time the sensor reading was taken.
 - The geographical position of the car (its latitude and longitude coordinates).
 - The current speed of the car.
- From the engine sensor:
 - The date and time the sensor reading was taken.
 - The revolutions per minute (RPM) of the crankshaft.
 - The oil temperature.
- From the brake sensor:
 - The date and time the sensor reading was taken.
 - The temperature of the brakes.

The sensors used in this scenario are deliberately simplistic. Real racecars include hundreds of sensors emitting thousands of telemetry readings at sub-second intervals.

ETL process goals and data sources

Motor racing is a highly technical sport, and analysis of how the critical components of a car are performing is a major aspect of how teams refine the design of the car, and how drivers optimize their driving style. The data captured over a single lap consists of many telemetry readings, which must be analyzed to find correlations and patterns in the car's performance.

In this scenario, an application has produced a text file. We're going to upload that text file to Azure Blob Storage and simulate an actual ETL pipeline.

The file contains captured sensor readings as objects. Note that the Position property of the GpsReading class is based on the Location struct. C# (Program.cs in RaceTracker project)

Download the [sensor.csv](#) file.

As part of the ETL processing workflow in HDInsight, the captured readings must be filtered to remove any null values caused by sensor transmission problems. At the end of the processing the data must be restructured to a tabular format that matches the following Azure SQL Database table definition. Transact-SQL (Create LapData Table.sql)

```
CREATE TABLE [LapData]
(
    [LapTime] [varchar](25) NOT NULL PRIMARY KEY CLUSTERED,
    [Lat] [float] NULL,
    [Lon] [float] NULL,
    [Speed] [float] NULL,
    [Revs] [float] NULL,
    [OilTemp] [float] NULL,
    [BrakeTemp] [float] NULL,
);
```

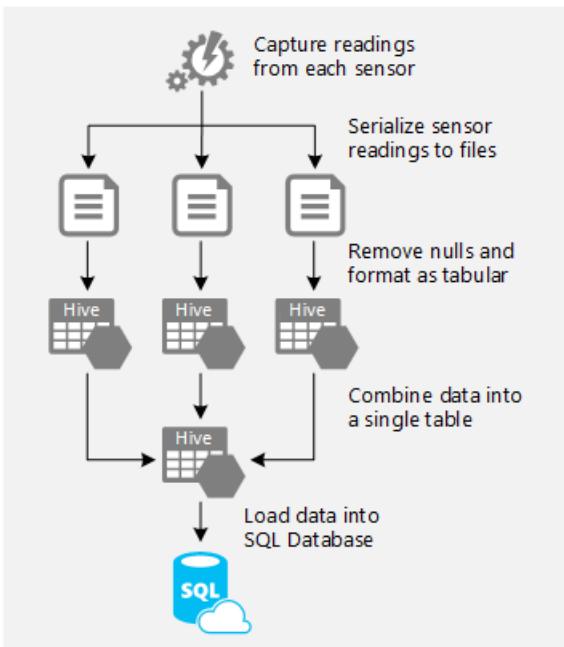
The workflow and its individual components are described in following section.

The ETL workflow

The key tasks that the ETL workflow for the racecar telemetry data must perform are:

- Uploading the telemetry to Azure storage.
- Filtering the data to remove readings that contain null values, and restructuring it into a tabular format.
- Combine the sensor data into a a single table.
- Load the sensor readings data into the table in an Azure SQL Database.

Figure 1 shows this workflow. We will only use one file for simplicity.



The team wants to integrate these tasks into the existing console application so that, after a test lap, the telemetry data is loaded into the database for later analysis.

Serializing and uploading the sensor readings

The first challenge in implementing the ETL workflow is to serialize each list of captured sensor reading objects into a file, and upload the files to Azure blob storage. There are numerous serialization formats that can be used to achieve this objective, but the team decided to use a CSV.

After the data for each sensor has been serialized to a file, the program must upload the files to the Azure blob storage container used by the HDInsight cluster.

To accomplish this the developer imported the Microsoft .NET API for Hadoop WebClient package and added using statements that reference the Microsoft.Hadoop.WebHDFS and Microsoft.Hadoop.WebHDFS.Adapters namespaces. The developer can then use the WebHDFSClient class to connect to Azure storage and upload the files. The following code shows how this technique is used to upload the file containing the GPS sensor readings.

Download the [LoadSensorData.zip](#) Solution.

Notice that the settings used by the WebHDFSClient object are retrieved from the App.Config file. These settings include the credentials required to connect to the Azure storage account used by HDInsight and the path for the folder to which the files should be uploaded. In this scenario the InputDir configuration settings has the value input, so the sensor data file will be saved as /users/admin/sensor/sensor.csv.

Filtering and restructuring the data

After the data has been uploaded to Azure storage, HDInsight can be used to process the data and upload it to Azure SQL Database.

To read the data in these files the developers decided to use Hive because of the simplicity it provides when querying tabular data structures. The first stage in this process was to create a script that builds Hive tables over the input records from the race car sensors. For example, the following HiveQL statements define a table over the filtered sensor data.

```
DROP TABLE IF EXISTS sensor;

CREATE EXTERNAL TABLE sensor
(laptime STRING, lat DOUBLE, lon DOUBLE, speed FLOAT)
COMMENT 'from csv file'
ROW FORMAT DELIMITED FIELDS TERMINATED BY ',' STORED AS TEXTFILE LOCATION '/user/admin/input/';
```

Sometimes the sensor data malfunctions and we get nulls in the value. We want to use Hive to clean the records before loading it into our data warehouse. We'll use a query like this:

```
INSERT OVERWRITE DIRECTORY '/user/admin/output/'
ROW FORMAT DELIMITED FIELDS TERMINATED BY ','
SELECT * FROM sensor
WHERE LapTime IS NOT NULL;
```

Loading the combined data to SQL Database

After the ETL process has filtered the data, it loads it into the LapData table in Azure SQL Database. To accomplish this the developers used Sqoop to copy the data from the folder on which the lap Hive table is based and transfer it to the database. The following command shows an example of how Sqoop can be used to perform this task. For more information on creating an Azure SQL Database, [see here](#).

```
```Command Line sqoop export --connect
"jdbc:sqlserver://{{yourserver}.database.windows.net:1433;databaseName=deepdiveetl;username=
{YourUserAccount};password={{yourpassword}};logintimeout=30" --table LapData --export-dir
/user/admin/output/ --input-fields-terminated-by , --input-null-non-string \N
```

Now that each of the tasks for the workflow have been defined, they can be combined into a workflow definition.

## Using Power BI

Now that your data is in the final resting place, we can use Power BI to create visualizations.

You can download [Power BI here.](<https://powerbi.microsoft.com/en-us/downloads/>)

You can see detailed instructions on how to use it [here:](<https://powerbi.microsoft.com/en-us/blog/using-power-bi-to-visualize-and-explore-azure-sql-databases/>)

To see the speed of the car over time, follow these steps:

1. Open up Power BI Desktop.
2. On the top bar, click Get Data.
3. In the Get Data window, click Azure on the left. Then click Azure SQL Database.
4. Type the full name of your server, for instance: deepdiveetl.database.windows.net.
5. Fill in your database name, for instance deepdiveetl. Click OK.
6. Fill in your Azure security credentials on the database credentials pane. Click OK.
7. In the Visualizations pane, click Line Chart.
8. Drag lap time over to the Axis and speed over to the Values section of the chart.

![Power BI Visualization Pane](./media/hdinsight-deep-dive-etl/powerbivisualizationpane.png)

9. You should have a visualization that looks like this:

![Power BI Lap Race Line Chart](./media/hdinsight-deep-dive-etl/lapracedata.png)

# Operationalize the Data Pipeline

8/16/2017 • 19 min to read • [Edit Online](#)

A key component of any data analytics solution is the data pipeline that acquires data, cleans and shapes the data and performs and desired calculations or aggregations prior to landing the data in the serving location, from which it will ultimately be consumed by clients, reports or API's. Fundamental to the success of the data pipeline is repeatability, enabling the data movement and processing performed by the pipeline to be performed on a schedule or triggered by the availability of new data.

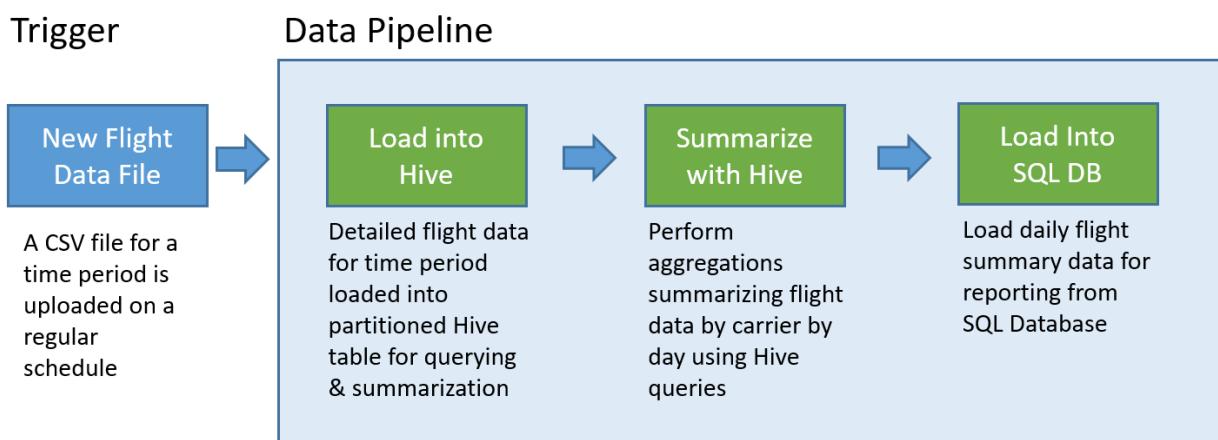
This article introduces how to operationalize your data pipelines that achieve this repeatability using Oozie running on HDInsight Hadoop clusters to build a complete data pipeline to prepare and process airline flight time-series data.

In this scenario, suppose that you receive a flat file containing a batch of flight data for a time period (e.g., monthly). This flight data includes information such as the origin and destination airport, the miles flown, the departure and arrival times, and a whole host of other data. Your goal with this pipeline is to be able to summarize airline performance by day, such that each airline has one row for each day describing the total miles they flew that day, as well as the average departure and arrival delays in minutes.

YEAR	MONTH	DAY_OF_MONTH	CARRIER	Avg Dep Delay	Avg Arr Delay	Total Distance
2017	1	3	AA	10.142229	7.862926	2644539
2017	1	3	AS	9.435449	5.482143	572289
2017	1	3	DL	6.935409	-2.1893024	1909696

You want to build a pipeline that whenever a new time period worth of flight data arrives, the detailed flight information is stored with your Hive data warehouse to support long term analytics and enables you to ask new questions of the data. At the same time you are creating a much smaller, summarized version of the flight data that has just the daily flight summaries you need at the moment. You want to be able to store the daily flight summary data in a SQL Database that provides the reports for your website.

The following diagram illustrates the desired pipeline that is implemented in this article:



This recurring data pipeline can be implemented using Oozie, and this article shows how to implement it in a step by step fashion.

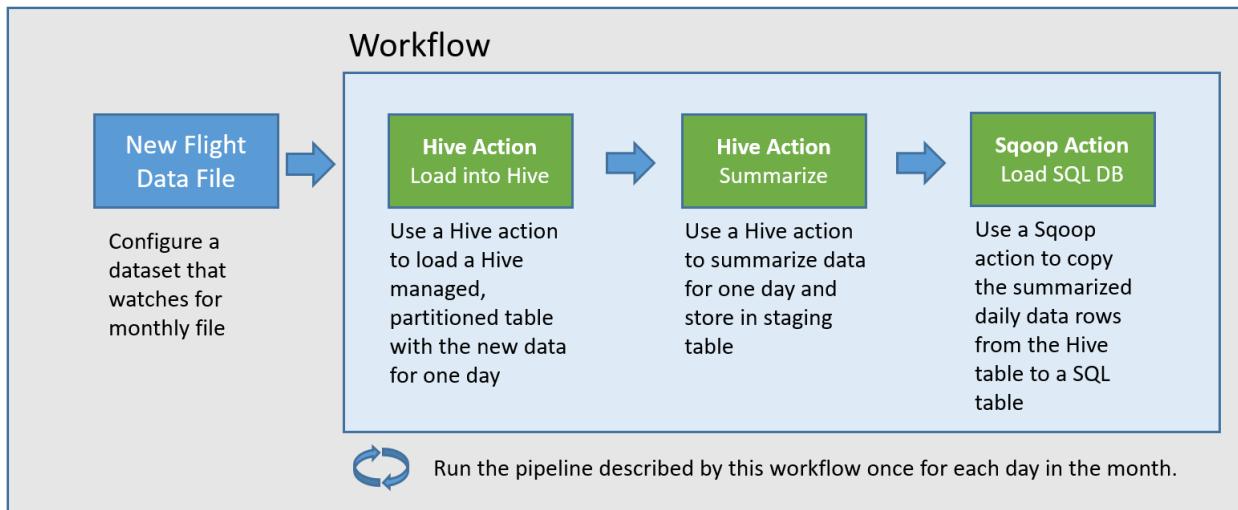
# Oozie Solution Overview

The desired pipeline can be achieved using Apache Oozie running within an HDInsight Hadoop cluster.

Oozie describes its pipelines in terms of actions, workflows and coordinators. Actions are what describe the actual work to perform, such as running a Hive query. Workflows define the sequence of actions. A coordinator defines the schedule overwhich the workflow is run, and can check for the availability of new data prior to launching an instance of the workflow, delaying the run as needed.

The following diagram captures the high level design of the pipeline using Oozie:

## Coordinator



The sections that follow walk thru the implementation of this pipeline.

## Provision Azure Resources

To follow along with this article, you will need to provision an Azure SQL Database and an HDInsight Hadoop cluster in the same location. The Azure SQL Database will be used both to store the summary data produced by the pipeline, as well as the metadata store for Oozie.

### Provision SQL Database

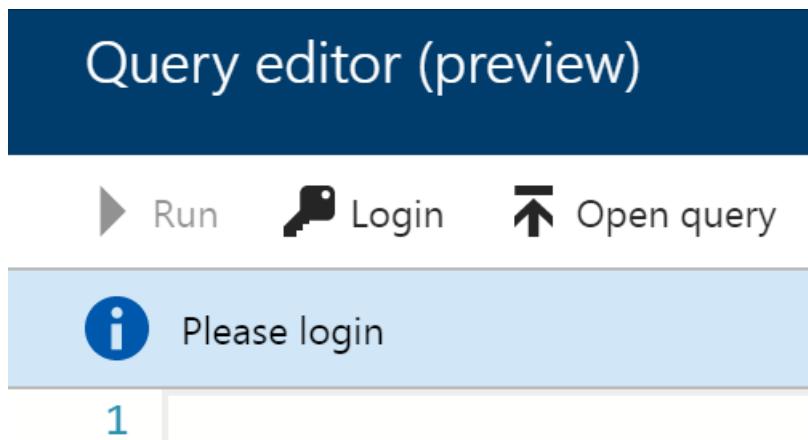
1. Using the Azure Portal, create a new Resource Group called oozie that will contain all of the resources used by this article.
2. Within the oozie resource group, provision an Azure SQL Server and Database. You do not need a database larger than the S1 Standard pricing tier.
3. Using the Azure Portal, navigate to the blade of your newly deployed SQL Database, and select Tools.



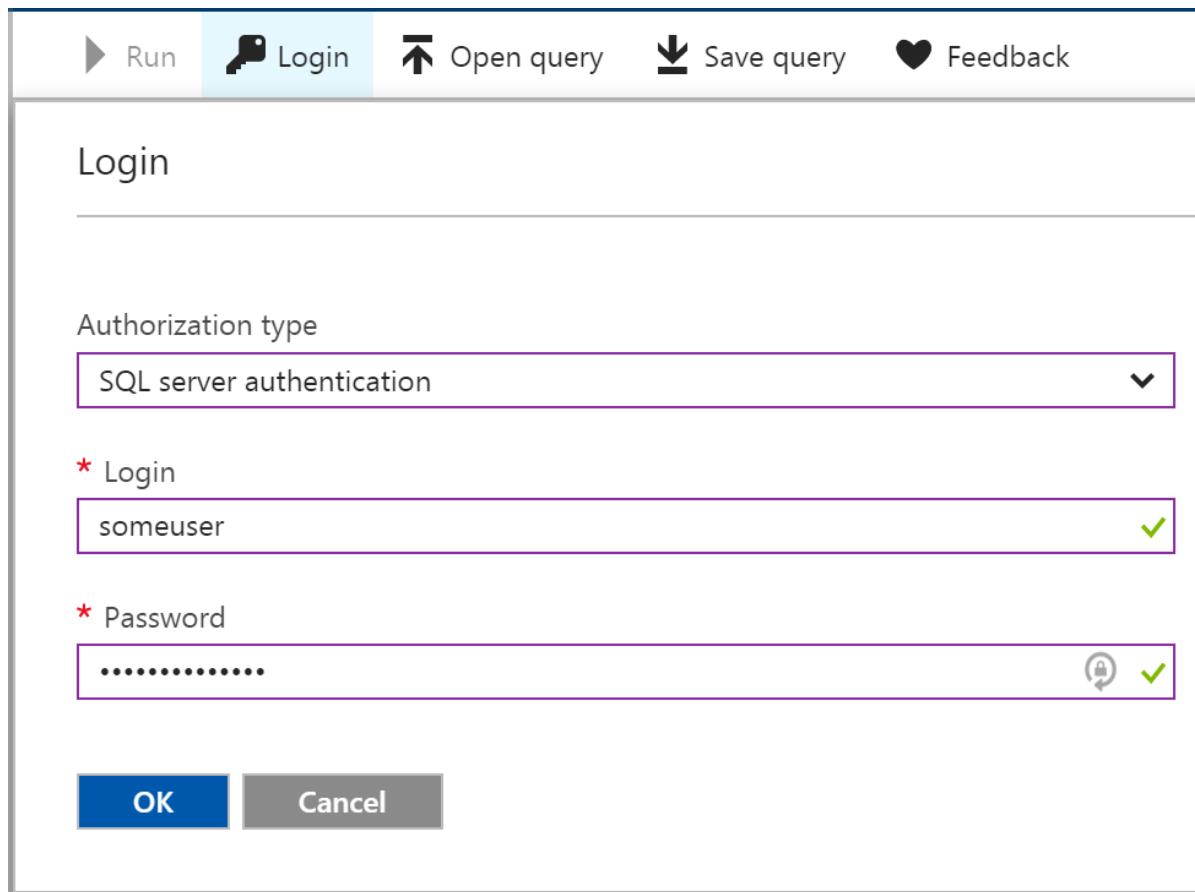
4. Select Query Editor.



5. In the Query Editor blade, select Login.



6. Enter your SQL Database credentials and select OK.



7. In the query editor text area, enter the following SQL statements to create the dailyflights table which will store the summarized data that results from each run of the pipeline.

```

CREATE TABLE dailyflights
(
 YEAR INT,
 MONTH INT,
 DAY_OF_MONTH INT,
 CARRIER CHAR(2),
 AVG_DEP_DELAY FLOAT,
 AVG_ARR_DELAY FLOAT,
 TOTAL_DISTANCE FLOAT
)
GO

CREATE CLUSTERED INDEX dailyflights_clustered_index on dailyflights(YEAR,MONTH,DAY_OF_MONTH,CARRIER)
GO

```

8. Select Run to execute the SQL statements.



9. Your SQL Database is now ready.

#### Provision an HDInsight Hadoop Cluster

1. Using the Azure Portal, select +New and search for HDInsight.
2. Select Create.
3. On the basics blade provide a unique name for your cluster and choose your Azure Subscription.

\* Cluster name

 ✓  
.azurehdinsight.net

\* Subscription

 ▼

4. Select Cluster type.

5. On the Cluster type blade, select the Hadoop cluster type, Linux operating system and the latest version of the HDInsight cluster. Leave the Cluster tier at Standard.



#### Cluster configuration

\* Cluster type i

 ▼

\* Operating system

 Linux  Windows

\* Version

 ▼

\* Cluster tier i

 STANDARD  PREMIUM

6. Choose select to apply your cluster type selection.

7. Complete the Basics blade by providing a login password and selecting your oozie resource group from the list and select Next.

# Basics

\* Cluster name  
myoozie ✓

.azurehdinsight.net

\* Subscription  
Microsoft Azure Enterprise ▼

---

\* Cluster type >  
Hadoop 2.7 on Linux (HDI 3.6)

---

\* Cluster login username i  
admin ✓

\* Cluster login password i  
..... (padlock icon) ✓

Secure Shell (SSH) username i  
sshuser

Use same password as cluster login i

\* Resource group i  
 Create new  Use existing

oozie ▼

\* Location  
West US ▼

8. On the Storage blade, leave the primary storage type set to Azure Storage, select Create new and provide a

name for the new account.

## Storage Account Settings

\* Primary storage type

Azure Storage  Data Lake Store

\* Selection method i

My subscriptions  Access key

---

\* Create a new Storage account

ooziestore



Select existing

---

\* Default container i

myoozie



---

Additional storage accounts >

Optional

---

Data Lake Store access i >

Optional

9. For the metastore settings, under the Select a SQL database for Hive, choose the database you previously created.

## Metastore Settings (optional)

Filtered to location and subscription of cluster.



To preserve your metadata outside this cluster, link a SQL database to this account.

Select a SQL database for Hive

zoozie/oozietest



10. Select Authenticate SQL Database

\* Authenticate SQL Database



*Not configured*

11. Enter your SQL database username and password, and choose select.

Authenticate

SQL Database username

someuser

SQL Database password



12. Back on the Metastore Settings, select your database for the Oozie metadata store and authenticate as you did previously.

## Metastore Settings (optional)

Filtered to location and subscription of cluster.



To preserve your metadata outside this cluster, link a SQL database to this account.

Select a SQL database for Hive

zoozie/oozietest



\* Authenticate SQL Database

Configured



Select a SQL database for Oozie

zoozie/oozietest



\* Authenticate SQL Database

Configured



13. Select Next.

14. On the Summary blade, select Create to deploy your cluster.

### Verify SSH Tunneling Setup

In order to utilize the Oozie Web Console to view the status of your coordinator and workflow instances, you will need to setup an SSH tunnel to your HDInsight cluster.

Detailed step by step instructions are available in the article [SSH Tunnel](#).

Note that you can also use Chrome with the [Foxy Proxy](#) extension to browse your cluster's web resources across the SSH tunnel. You need to configure it to proxy all request thru the host "localhost" on port 9876, which is the port upon which your tunnel is open.

This approach is compatible when using the Windows Subsystem for Linux, also known as Bash on Windows 10.

In summary, you will need to run the following command to open the tunnel to your cluster:

```
ssh -C2qTnNf -D 9876 sshuser@[CLUSTERNAME]-ssh.azurehdinsight.net
```

Once you have setup your proxy, you can verify the tunnel is operational by navigating to Ambari on your head

node by browsing to:

<http://headnodehost:8080>

To access the Oozie Web Console, from Ambari, select Oozie, Quick Links and then select Oozie Web Console.

If you can access the Oozie Web Console, you are good to go.

### Configure Hive

You will need to upload the CSV file `2017-01-FlightData.csv` containing sample data for one month of flight data.

Copy this file up to the Azure Storage account attached to your HDInsight cluster and place it in the `/example/data/flights` folder.

One way to accomplish this is using SCP in your bash shell session.

First, use SCP to copy the files from your local machine to the local storage of your HDInsight cluster head node:

```
scp ./2017-01-FlightData.csv sshuser@[CLUSTERNAME]-ssh.azurehdinsight.net:2017-01-FlightData.csv
```

Second, use the HDFS command to copy the file from your head node local storage to Azure Storage:

```
hdfs dfs -put ./2017-01-FlightData.csv /example/data/flights/2017-01-FlightData.csv
```

The sample data you will process is now available. However, the pipeline requires two Hive tables to be created prior to its execution. You can create these using Ambari by following these steps:

1. Login to Ambari by navigating to <http://headnodehost:8080>
2. From the list of services, select Hive.

- HDFS
- YARN
- MapReduce2
- Tez
- Hive
- Pig
- Sqoop
- Oozie
- ZooKeeper
- Ambari Metrics
- Slider

3. Select Go To View next to the Hive View 2.0 label.

## Summary

[Hive Metastore](#) Started **No alerts**

[Hive Metastore](#) Started **No alerts**

[HiveServer2](#) Started **No alerts**

[HiveServer2](#) Started **No alerts**

[WebHCat Server](#) Started **No alerts**

[WebHCat Server](#) Started **No alerts**

[HCat Client](#) 1 HCat Client Installed

[Hive Clients](#) 3 Hive Clients Installed

HiveServer2 JDBC URL [jdbc:hive2://zk1-zoozie:10000](#)

[Hive View 2.0](#) [Go To View](#)

[Debug Hive Query](#) [Go To View](#)

4. In the query text area, paste the following statements to create the RawFlights table. The RawFlights table provides a schema on read over any of the CSV files that appear within the /example/data/flights folder in Azure Storage.

```
CREATE EXTERNAL TABLE IF NOT EXISTS rawflights (
 YEAR INT,
 MONTH INT,
 DAY_OF_MONTH INT,
 FL_DATE STRING,
 CARRIER STRING,
 FL_NUM STRING,
 ORIGIN STRING,
 DEST STRING,
 DEP_DELAY FLOAT,
 ARR_DELAY FLOAT,
 ACTUAL_ELAPSED_TIME FLOAT,
 DISTANCE FLOAT)
ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.OpenCSVSerde'
WITH SERDEPROPERTIES
(
 "separatorChar" = ",",
 "quoteChar" = "\""
)
LOCATION '/example/data/flights'
```

5. Select Execute to create the table.

 Ambari    zoozie 0 ops 0 alerts

## HIVE

[QUERY](#)[JOBS](#)[TABLES](#)[SAVED QUERIES](#)[UDFs](#)[SETTINGS](#)

Worksheet1 \*

**DATABASE**

Select or search database/schema

default

```
1 CREATE EXTERNAL TABLE IF NOT EXISTS rawflights (
2 YEAR INT,
3 MONTH INT,
4 DAY_OF_MONTH INT,
5 FL_DATE STRING,
6 CARRIER STRING,
7 FL_NUM STRING,
8 ORIGIN STRING,
9 DEST STRING,
10 DEP_DELAY FLOAT,
11 ARR_DELAY FLOAT,
12 ACTUAL_ELAPSED_TIME FLOAT,
13 DISTANCE FLOAT)
14 ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.OpenCSVSerde'
15 WITH SERDEPROPERTIES
```

[Execute](#)[Save As](#)[Insert UDF](#) ▾[Visual Explain](#)

6. Replace the text in the query text area with the following statements to create the Flights table. The Flights table is a Hive managed table that will partition data loaded into it by year, month, and day of month. It will contain all historical flight data, with the lowest granularity present in the source data of one row per flight.

```
SET hive.exec.dynamic.partition.mode=nonstrict;

CREATE TABLE flights
(
 FL_DATE STRING,
 CARRIER STRING,
 FL_NUM STRING,
 ORIGIN STRING,
 DEST STRING,
 DEP_DELAY FLOAT,
 ARR_DELAY FLOAT,
 ACTUAL_ELAPSED_TIME FLOAT,
 DISTANCE FLOAT
)
PARTITIONED BY (YEAR INT, MONTH INT, DAY_OF_MONTH INT)
ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.OpenCSVSerde'
WITH SERDEPROPERTIES
(
 "separatorChar" = ",",
 "quoteChar" = "\""
);
```

7. Select Execute to create the table.

You are now ready to begin constructing your pipeline.

### Create the Oozie workflow

When defining a pipeline, it is typical to batch the data flowing thru the pipeline according to a time interval. In the case of the scenario, the pipeline processes the flight data on a daily interval. This approach provides flexibility in that it allows the input CSV files to arrive daily, weekly, monthly or annually and the pipeline continues to work with little modification.

The workflow you will be build processes the flight data in this day by day fashion, following three major steps:

1. Extract the data in the day date range from the source CSV file represented by the RawFlights table and insert it into Flights table. This is accomplished by running a Hive query.
2. Dynamically create a staging table in Hive for the day which contains a copy of the flight data summarized by day and carrier. This is also accomplished by running a Hive query.
3. Copy all the data from the daily staging table in Hive to the destination dailyflights table in Azure SQL Database. This is accomplished by using Apache Sqoop, which reads the source rows from the data behind the Hive table residing in Azure Storage and loads them into SQL Database using a JDBC connection.

These three steps are coordinated by an Oozie workflow.

The first step is a query in the file `hive-load-flights-partition.hql` as follows:

```
SET hive.exec.dynamic.partition.mode=nonstrict;

INSERT OVERWRITE TABLE flights
PARTITION (YEAR, MONTH, DAY_OF_MONTH)
SELECT
 FL_DATE,
 CARRIER,
 FL_NUM,
 ORIGIN,
 DEST,
 DEP_DELAY,
 ARR_DELAY,
 ACTUAL_ELAPSED_TIME,
 DISTANCE,
 YEAR,
 MONTH,
 DAY_OF_MONTH
FROM rawflights
WHERE year = ${year} AND month = ${month} AND day_of_month = ${day};
```

Observe the values present with the syntax  `${}`, these are Oozie variables that are configured later and Oozie will substitute with the actual values at runtime.

The second step is a query in the file `hive-create-daily-summary-table.hql` as follows:

```

DROP TABLE ${hiveTableName};
CREATE EXTERNAL TABLE ${hiveTableName}
(
 YEAR INT,
 MONTH INT,
 DAY_OF_MONTH INT,
 CARRIER STRING,
 AVG_DEP_DELAY FLOAT,
 AVG_ARR_DELAY FLOAT,
 TOTAL_DISTANCE FLOAT
)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '\t' STORED AS TEXTFILE LOCATION '${hiveDataFolder}';
INSERT OVERWRITE TABLE ${hiveTableName}
SELECT year, month, day_of_month, carrier, avg(dep_delay) avg_dep_delay,
 avg(arr_delay) avg_arr_delay, sum(distance) total_distance
FROM flights
GROUP BY year, month, day_of_month, carrier
HAVING year = ${year} AND month = ${month} AND day_of_month = ${day};

```

This query creates a staging table that will store only the summarized data for one day, take note of the SELECT statement that computes the average delays and total of distance flown by carrier by day. The data inserted into this table stored at a known location (the path indicated by the hiveDataFolder variable) so that it can be used as the source for Sqoop in the next step.

The third step is effectively to run the following sqoop command:

```

sqoop export --connect ${sqlDatabaseConnectionString} --table ${sqlDatabaseTableName} --export-dir
${hiveDataFolder} -m 1 --input-fields-terminated-by "\t"

```

These three steps are expressed as three separate actions in the Oozie workflow (a file named workflow.xml):

```

<workflow-app name="loadflightstable" xmlns="uri:oozie:workflow:0.5">
 <start to = "RunHiveLoadFlightsScript"/>
 <action name="RunHiveLoadFlightsScript">
 <hive xmlns="uri:oozie:hive-action:0.2">
 <job-tracker>${jobTracker}</job-tracker>
 <name-node>${nameNode}</name-node>
 <configuration>
 <property>
 <name>mapred.job.queue.name</name>
 <value>${queueName}</value>
 </property>
 </configuration>
 <script>${hiveScriptLoadPartition}</script>
 <param>year=${year}</param>
 <param>month=${month}</param>
 <param>day=${day}</param>
 </hive>
 <ok to="RunHiveCreateDailyFlightTableScript"/>
 <error to="fail"/>
 </action>
 <action name="RunHiveCreateDailyFlightTableScript">
 <hive xmlns="uri:oozie:hive-action:0.2">
 <job-tracker>${jobTracker}</job-tracker>
 <name-node>${nameNode}</name-node>
 <configuration>
 <property>
 <name>mapred.job.queue.name</name>
 <value>${queueName}</value>
 </property>
 </configuration>
 <script>${hiveScriptCreateDailyTable}</script>

```

```

<param>hiveTableName=${hiveDailyTableName}</param>
<param>year=${year}</param>
<param>month=${month}</param>
<param>day=${day}</param>
<param>hiveDataFolder=${hiveDataFolder}/${year}/${month}/${day}</param>
</hive>
<ok to="RunSqoopExport"/>
<error to="fail"/>
</action>
<action name="RunSqoopExport">
 <sqoop xmlns="uri:oozie:sqoop-action:0.2">
 <job-tracker>${jobTracker}</job-tracker>
 <name-node>${nameNode}</name-node>
 <configuration>
 <property>
 <name>mapred.compress.map.output</name>
 <value>true</value>
 </property>
 </configuration>
 <arg>export</arg>
 <arg>--connect</arg>
 <arg>${sqlDatabaseConnectionString}</arg>
 <arg>--table</arg>
 <arg>${sqlDatabaseTableName}</arg>
 <arg>--export-dir</arg>
 <arg>${hiveDataFolder}/${year}/${month}/${day}</arg>
 <arg>-m</arg>
 <arg>1</arg>
 <arg>--input-fields-terminated-by</arg>
 <arg>"\t"</arg>
 <archive>sqljdbc41.jar</archive>
 </sqoop>
</action>
<kill name="fail">
 <message>Job failed, error message[${wf:errorMessage(wf:lastErrorNode())}] </message>
</kill>
<end name="end"/>
</workflow-app>

```

The two Hive queries are accessed by their path in Azure Storage, and the remaining variable values are provided by the job.properties file, for example:

```

nameNode=wasbs://[CONTAINERNAME]@[ACCOUNTNAME].blob.core.windows.net
jobTracker=hn0-[CLUSTERNAME].[UNIQUESTRING].dx.internal.cloudapp.net:8050
queueName=default
oozie.use.system.libpath=true
appBase=wasbs://[CONTAINERNAME]@[ACCOUNTNAME].blob.core.windows.net/oozie
oozie.wf.application.path=${appBase}/load_flights_by_day
hiveScriptLoadPartition=wasbs://[CONTAINERNAME]@[ACCOUNTNAME].blob.core.windows.net/oozie/load_flights_by_day/
hive-load-flights-partition.hql
hiveScriptCreateDailyTable=wasbs://[CONTAINERNAME]@[ACCOUNTNAME].blob.core.windows.net/oozie/load_flights_by_d
ay/hive-create-daily-summary-table.hql
hiveDailyTableName=dailyflights${year}${month}${day}
hiveDataFolder=wasbs://[CONTAINERNAME]@[ACCOUNTNAME].blob.core.windows.net/example/data/flights/day/${year}/${
month}/${day}
sqlDatabaseConnectionString="jdbc:sqlserver://[SERVERNAME].database.windows.net;user=[USERNAME];password=
[PASSWORD];database=[DATABASENAME]"
sqlDatabaseTableName=dailyflights
year=2017
month=01
day=03

```

Notice that the above job.properties file configures the workflow to run for the date January 3rd, 2017. The

following table summarizes each of the properties in more detail, and indicates from where you need to acquire the value for your own environment.

PROPERTY	VALUE SOURCE
nameNode	The full path to the Azure Storage Container attached to your HDInsight cluster.
jobTracker	The internal hostname to your active cluster YARN head node. Get this from Ambari- on the Ambari home page, select YARN from the list of services, then choose Active Resource Manager. The URI displayed at the top of the page is what you are after. Append the port 8050.
queueName	The name of the YARN queue that is used when scheduling the Hive actions. Leave as default.
oozie.use.system.libpath	Leave as true.
appBase	The path to the subfolder in Azure Storage to which you deploy the oozie workflow and supporting files
oozie.wf.application.path	Indicates the location of the workflow.xml containing the Oozie workflow you want to run
hiveScriptLoadPartition	The path in Azure Storage to the hive-load-flights-partition.hql Hive query file
hiveScriptCreateDailyTable	The path in Azure Storage to the hive-create-daily-summary-table.hql Hive query file
hiveDailyTableName	The dynamically generated name to use for the staging table
hiveDataFolder	The path in Azure Storage to the data contained by the staging table
sqlDatabaseConnectionString	The JDBC syntax connection string to your Azure SQL Database
sqlDatabaseTableName	The name of the table in Azure SQL Database into which summary rows are inserted. Leave as dailyflights.
year	The year component of the day for which flight summaries are computed. Leave as is.
month	The month component of the day for which flight summaries are computed. Leave as is.
day	The day of month component of the day for which flight summaries are computed. Leave as is.

Before you can deploy and run your Oozie workflow, be sure to update your copy of the job.properties file with the values specific to your environment.

## Deploy and run the Oozie Workflow

To deploy your Oozie workflow (workflow.xml), the Hive queries (hive-load-flights-partition.hql and hive-create-

`daily-summary-table.hql`) and the `job.properties` configuration file, you can use SCP from your bash session.

With Oozie, the only the `job.properties` file must exist on the local storage of the headnode. All other files must be stored in HDFS (Azure Storage). Additionally, the Sqoop action used by the workflow depends on a JDBC driver for communicating with SQL Database, which must be copied from the head node to HDFS. The following instructions walk thru each of these steps:

1. Create the `load_flights_by_day` subfolder underneath the user's path in the local storage of the head node:

```
ssh sshuser@[CLUSTERNAME]-ssh.azurehdinsight.net 'mkdir load_flights_by_day'
```

2. Copy the workflow and `job.properties` files up to the `load_flights_by_day` subfolder:

```
scp ./* sshuser@[CLUSTERNAME]-ssh.azurehdinsight.net:load_flights_by_day
```

3. SSH into your head node and naviagate into the `load_flights_by_day` folder:

```
ssh sshuser@[CLUSTERNAME]-ssh.azurehdinsight.net
cd load_flights_by_day
```

4. Copy workflow files to HDFS:

```
hdfs dfs -put ./* /oozie/load_flights_by_day
```

5. Copy the `sqljdbc41.jar` from the local head node to the workflow folder in HDFS:

```
hdfs dfs -put /usr/share/java/sqljdbc_4.1/enu/sqljdbc*.jar /oozie/load_flights_by_day
```

6. Run the workflow:

```
oozie job -config job.properties -run
```

7. Observe the status using the Oozie Web Console. Navigate to the console as previously described and look at the listing in the Workflow Jobs tab and selecting the All Jobs toggle.

**Oozie Documentation**

**Oozie Web Console**

**Workflow Jobs Coordinator Jobs Bundle Jobs System Info Instrumentation Settings**

**All Jobs Active Jobs Done Jobs Custom Filter ▾ Server version [4.2.0.2.6.0.10-29]**

Job Id	Name	Status	R...	User	Group	Created	Started
1	0000068-170707154033584-oozie-oozi-W	loadflightstable	SUCCEEDED	0	sshuser	Mon, 17 Jul 2017 18:25:48 GMT	Mon, 17...
2	0000067-170707154033584-oozie-oozi-W	loadflightstable	SUCCEEDED	0	sshuser	Mon, 17 Jul 2017 18:22:46 GMT	Mon, 17...
3	0000066-170707154033584-oozie-oozi-W	loadflightstable	SUCCEEDED	0	sshuser	Mon, 17 Jul 2017 18:19:35 GMT	Mon, 17...
4	0000065-170707154033584-oozie-oozi-W	loadflightstable	SUCCEEDED	0	sshuser	Mon, 17 Jul 2017 18:16:31 GMT	Mon, 17...
5	0000063-170707154033584-oozie-oozi-W	loadflightstable	SUCCEEDED	0	sshuser	Mon, 17 Jul 2017 18:07:37 GMT	Mon, 17...
6	0000062-170707154033584-oozie-oozi-W	loadflightstable	SUCCEEDED	0	sshuser	Mon, 17 Jul 2017 18:04:31 GMT	Mon, 17...
7	0000059-170707154033584-oozie-oozi-W	loadflightstable	SUCCEEDED	0	sshuser	Mon, 17 Jul 2017 17:08:44 GMT	Mon, 17...
8	0000058-170707154033584-oozie-oozi-W	loadflightstable	SUCCEEDED	0	sshuser	Mon, 17 Jul 2017 17:03:11 GMT	Mon, 17...
9	0000057-170707154033584-oozie-oozi-W	loadflightstable	SUCCEEDED	0	sshuser	Mon, 17 Jul 2017 16:51:13 GMT	Mon, 17...
10	0000056-170707154033584-oozie-oozi-W	loadflightstable	KILLED	0	sshuser	Mon, 17 Jul 2017 16:47:16 GMT	Mon, 17...
11	0000055-170707154033584-oozie-oozi-W	loadflightstable	SUCCEEDED	0	sshuser	Mon, 17 Jul 2017 16:42:05 GMT	Mon, 17...
12	0000054-170707154033584-oozie-oozi-W	loadflightstable	SUCCEEDED	0	sshuser	Mon, 17 Jul 2017 16:23:06 GMT	Mon, 17...
13	0000053-170707154033584-oozie-oozi-W	loadflightstable	KILLED	0	sshuser	Mon, 17 Jul 2017 16:19:51 GMT	Mon, 17...
14	0000052-170707154033584-oozie-oozi-W	exportwf	SUCCEEDED	0	sshuser	Mon, 17 Jul 2017 14:48:51 GMT	Mon, 17...
15	0000050-170707154033584-oozie-oozi-W	exportwf	SUCCEEDED	0	sshuser	Mon, 17 Jul 2017 14:39:39 GMT	Mon, 17...
16	0000002-170707154033584-oozie-oozi-W	useooziewf	PREP	0	sshuser	Sun, 09 Jul 2017 14:15:40 GMT	

◀ | Page 1 of 1 | ▶ | 🔍 | 1 - 16 of 16

8. When the status reads SUCCEEDED, query the SQL database table (you can use the Query Editor in the Azure Portal as described previously) to view the inserted rows:

```
SELECT * FROM dailyflights
```

Now that you have the workflow running for a single day, you can wrap this workflow with a coordinator that will effectively schedule the workflow so that it runs daily.

### Run the workflow with a coordinator

When you want to schedule this workflow so that it runs daily (or to run for all days in a data range), for example once a day as new data arrives, you need to leverage a coordinator.

In the case of the scenario, we want the workflow to run daily and so you create a coordinator that has a daily frequency.

The following is the contents of coordinator.xml:

```
<coordinator-app name="daily_export" start="2017-01-01T00:00Z" end="2017-01-05T00:00Z"
frequency="${coord:days(1)}" timezone="UTC" xmlns="uri:oozie:coordinator:0.4">
<datasets>
 <dataset name="ds_input1" frequency="${coord:days(1)}" initial-instance="2016-12-31T00:00Z"
timezone="UTC">
 <uri-template>${sourceDataFolder}${YEAR}-${MONTH}-FlightData.csv</uri-template>
 <done-flag></done-flag>
 </dataset>
</datasets>
<input-events>
 <data-in name="event_input1" dataset="ds_input1">
 <instance>${coord:current(0)}</instance>
 </data-in>
</input-events>
<action>
 <workflow>
```

```

<app-path>${appBase}/load_flights_by_day</app-path>
<configuration>
 <property>
 <name>year</name>
 <value>${coord:formatTime(coord:nominalTime(), 'yyyy')}</value>
 </property>
 <property>
 <name>month</name>
 <value>${coord:formatTime(coord:nominalTime(), 'MM')}</value>
 </property>
 <property>
 <name>day</name>
 <value>${coord:formatTime(coord:nominalTime(), 'dd')}</value>
 </property>
 <property>
 <name>hiveScriptLoadPartition</name>
 <value>${hiveScriptLoadPartition}</value>
 </property>
 <property>
 <name>hiveScriptCreateDailyTable</name>
 <value>${hiveScriptCreateDailyTable}</value>
 </property>
 <property>
 <name>hiveDailyTableNamePrefix</name>
 <value>${hiveDailyTableNamePrefix}</value>
 </property>
 <property>
 <name>hiveDailyTableName</name>
 <value>${hiveDailyTableNamePrefix}${coord:formatTime(coord:nominalTime(),
'yyyy')}${coord:formatTime(coord:nominalTime(), 'MM')}${coord:formatTime(coord:nominalTime(), 'dd')}</value>
 </property>
 <property>
 <name>hiveDataFolderPrefix</name>
 <value>${hiveDataFolderPrefix}</value>
 </property>
 <property>
 <name>hiveDataFolder</name>
 <value>${hiveDataFolderPrefix}${coord:formatTime(coord:nominalTime(),
'yyyy')}/${coord:formatTime(coord:nominalTime(), 'MM')}/${coord:formatTime(coord:nominalTime(), 'dd')}</value>
 </property>
 <property>
 <name>sqlDatabaseConnectionString</name>
 <value>${sqlDatabaseConnectionString}</value>
 </property>
 <property>
 <name>sqlDatabaseTableName</name>
 <value>${sqlDatabaseTableName}</value>
 </property>
</configuration>
</workflow>
</action>
</coordinator-app>

```

As you can see, the majority of the coordinator is just passing thru configuration to the workflow instance.

However, there are a few important items to call out.

First, the start and end attributes on the coordinator element itself control the time interval over which the coordinator runs.

```
<coordinator-app ... start="2017-01-01T00:00Z" end="2017-01-05T00:00Z" frequency="${coord:days(1)}" ...>
```

A coordinator is responsible for scheduling actions within the start and end date range and according to the interval specified by the frequency attribute. Each action scheduled ultimately runs the workflow configured. In the coordinator definition above, the coordinator is configured to run actions from January 1st, 2017 to January 5th,

2017. The frequency is set to 1 day by the [Oozie Expression Language](#) frequency expression \${coord:days(1)}. This results in the coordinator scheduling an action (and ultimately the workflow) once per day. For date ranges that are in the past, as in the example, the action will be scheduled to run without delay. The start of the date from which an action is scheduled to run is referred to as the nominal time. For example, to process the data for January 1st, 2017 the coordinator will schedule action with a nominal time of 2017-01-01T00:00:00 GMT.

Second, within the date range of the workflow, the dataset element defines where to look in HDFS for the data for a particular date range, and configures how Oozie determines if the data is available yet for processing.

```
<dataset name="ds_input1" frequency="${coord:days(1)}" initial-instance="2016-12-31T00:00Z" timezone="UTC">
 <uri-template>${sourceDataFolder}${YEAR}-${MONTH}-FlightData.csv</uri-template>
 <done-flag></done-flag>
</dataset>
```

The path to the data in HDFS is built dynamically according to the expression provided in the uri-template element. In the above coordinator, a frequency of one day is also used with the dataset. While the start and end dates on the coordinator element control when the actions are scheduled (and defines their nominal time), the initial-instance and frequency on the dataset control the calculation of the date that is used in constructing the uri-template. In this case, we set the initial instance to one day before the start of the coordinator to ensure that we pick up the first day's (e.g., 1/1/2017) worth of data. The dataset's date calculation rolls forward from the value of initial-instance (e.g., 12/31/2016) advancing in increments of dataset frequency (e.g., 1 day) until it finds the latest date that does not pass the nominal time set by the coordinator (e.g., 2017-01-01T00:00:00 GMT for the first action).

Notice also the use of the empty done-flag element- this means that when Oozie checks for the presence of input data at the appointed time, it determines data is available by presence of a directory or file- in this case it is the presence of the csv file. If present it assumes the data is ready and can launch a workflow instance to process it, otherwise it assumes the data is not yet ready and that run of the workflow goes into a waiting state.

Third, the data-in specifies the particular timestamp to use as the nominal time when replacing the values in uri-template for the associated dataset.

```
<data-in name="event_input1" dataset="ds_input1">
 <instance>${coord:current(0)}</instance>
</data-in>
```

In this case, we set the instance to the expression \${coord:current(0)} which translates to using the nominal time of the action as originally scheduled by the coordinator. In other words, when the coordinator schedules the action to run with a nominal time of 01/01/2017, then 01/01/2017 is what is used to replace the YEAR (2017) and MONTH (01) variables in the URI template. Once the URI template is computed for this instance, Oozie checks if the expected directory or file is available and schedules the run of the workflow accordingly.

These three points all combine to yield a situation where the coordinator schedules processing of the source data in a day by day fashion. For example:

1. The coordinator starts with a nominal date of 2017-01-01.
2. Oozie looks for data available in the sourceDataFolder/2017-01-FlightData.csv.
3. When it sees that file, it schedules an instance of the workflow that will process the data for 2017-01-01.

Then it continues processing for 2017-01-02. This evaluation continues up to, but excluding 2017-01-05.

As for workflows, the configuration of a coordinator is also provided in a job.properties file, which has a superset of the settings used by the workflow:

```

nameNode=wasbs://[CONTAINERNAME]@[ACCOUNTNAME].blob.core.windows.net
jobTracker=hn0-[CLUSTERNAME].[UNIQUESTRING].dx.internal.cloudapp.net:8050
queueName=default
oozie.use.system.libpath=true
appBase=wasbs://[CONTAINERNAME]@[ACCOUNTNAME].blob.core.windows.net/oozie
oozie.coord.application.path=${appBase}
sourceDataFolder=wasbs://[CONTAINERNAME]@[ACCOUNTNAME].blob.core.windows.net/example/data/flights/
hiveScriptLoadPartition=wasbs://[CONTAINERNAME]@[ACCOUNTNAME].blob.core.windows.net/oozie/load_flights_by_day/
hive-load-flights-partition.hql
hiveScriptCreateDailyTable=wasbs://[CONTAINERNAME]@[ACCOUNTNAME].blob.core.windows.net/oozie/load_flights_by_d
ay/hive-create-daily-summary-table.hql
hiveDailyTableNamePrefix=dailyflights
hiveDataFolderPrefix=wasbs://[CONTAINERNAME]@[ACCOUNTNAME].blob.core.windows.net/example/data/flights/day/
sqlDatabaseConnectionString="jdbc:sqlserver://[SERVERNAME].database.windows.net;user=[USERNAME];password=
[PASSWORD];database=[DATABASENAME]"
sqlDatabaseTableName=dailyflights

```

The only new properties introduced in this job.properties are:

PROPERTY	VALUE SOURCE
oozie.coord.application.path	Indicates the location of the coordinator.xml containing the Oozie coordinator you want to run
hiveDailyTableNamePrefix	The prefix that will be used when dynamically creating the table name of the staging table
hiveDataFolderPrefix	The prefix of the path to where all the staging tables will be stored

### Deploy and run the Oozie Coordinator

To run the pipeline with a coordinator, you proceed in a similar fashion as for the workflow, except you work from a folder one level above the folder that contains your workflow. This convention enables you to separate the coordinators from the workflows on disk, and also enables you to easily associate a coordinator with different "child" workflows.

1. Use SCP from your local machine to copy the coordinator files up to the local storage of the head node of your cluster:

```
scp ./* sshuser@[CLUSTERNAME]-ssh.azurehdinsight.net:~
```

2. SSH into your head node:

```
ssh sshuser@[CLUSTERNAME]-ssh.azurehdinsight.net
```

3. Copy the coordinator files to HDFS:

```
hdfs dfs -put .//* /oozie/
```

4. Run the coordinator

```
oozie job -config job.properties -run
```

5. Verify the status using the Oozie Web Console, this time selecting the Coordinator Jobs tab, and then the All jobs toggle.

Coordinator Jobs										Server version [4.2.0.2.6.0.10-29]	
All Jobs		Active Jobs		Done Jobs		Custom Filter ▾					
Job Id	Name	Status	User	Group	Frequency	Unit	Started	Next Materialization			
1 0000064-170707154033584-oozie-oozi-C	daily_export	SUCCEEDED	sshuser		1	DAY	Sun, 01 Jan 2017 00:00:00 GMT	Thu, 05 Jan 2017 00:00:00 GMT			
2 0000061-170707154033584-oozie-oozi-C	daily_export	KILLED	sshuser		1	DAY	Sun, 01 Jan 2017 00:00:00 GMT	Thu, 05 Jan 2017 00:00:00 GMT			
3 0000060-170707154033584-oozie-oozi-C	daily_export	FAILED	sshuser		1	DAY	Sun, 01 Jan 2017 00:00:00 GMT				
4 0000051-170707154033584-oozie-oozi-C	daily_export	DONEWITHERROR	sshuser		1	DAY	Sun, 01 Jan 2017 00:00:00 GMT	Thu, 05 Jan 2017 00:00:00 GMT			
5 0000049-170707154033584-oozie-oozi-C	daily_export	DONEWITHERROR	sshuser		1	DAY	Sun, 01 Jan 2017 00:00:00 GMT	Thu, 05 Jan 2017 00:00:00 GMT			

6. Select the coordinator instance to display the list of scheduled actions, you should see four actions with nominal times in the range from 1/1/2017 to 1/4/2017, similar to the following:

Job (Name: daily\_export/coordJobId: 0000064-170707154033584-oozie-oozi-C)

Coord Job Info	Coord Job Definition	Coord Job Configuration	Coord Job Log	Coord Error Log	Coord Audit Log	Coord Action Reruns																														
Job Id: 0000064-170707154033584-oozie-oozi-C Name: daily_export Status: SUCCEEDED User: sshuser Group: Frequency: 1 Unit: DAY Parent Bundle:  Start Time: Sun, 01 Jan 2017 00:00:00 GMT Next Matd: Thu, 05 Jan 2017 00:00:00 GMT End Time: Thu, 05 Jan 2017 00:00:00 GMT Pause Time: Concurrency: 1																																				
<b>Actions</b>																																				
<table border="1"> <thead> <tr> <th>Action Id</th> <th>Status</th> <th>Ext Id</th> <th>Error Code</th> <th>Created Time</th> <th>Nominal Time</th> </tr> </thead> <tbody> <tr> <td>1 0000064-170707154033584-oozie-oozi-C@4</td> <td>SUCCEEDED</td> <td>0000068-170707154033584-oozie-oozi-W</td> <td></td> <td>Mon, 17 Jul 2017 18:16:26 GMT</td> <td>Wed, 04 Jan 2017 00:00:00 GMT</td> </tr> <tr> <td>2 0000064-170707154033584-oozie-oozi-C@3</td> <td>SUCCEEDED</td> <td>0000067-170707154033584-oozie-oozi-W</td> <td></td> <td>Mon, 17 Jul 2017 18:16:26 GMT</td> <td>Tue, 03 Jan 2017 00:00:00 GMT</td> </tr> <tr> <td>3 0000064-170707154033584-oozie-oozi-C@2</td> <td>SUCCEEDED</td> <td>0000066-170707154033584-oozie-oozi-W</td> <td></td> <td>Mon, 17 Jul 2017 18:16:26 GMT</td> <td>Mon, 02 Jan 2017 00:00:00 GMT</td> </tr> <tr> <td>4 0000064-170707154033584-oozie-oozi-C@1</td> <td>SUCCEEDED</td> <td>0000065-170707154033584-oozie-oozi-W</td> <td></td> <td>Mon, 17 Jul 2017 18:16:26 GMT</td> <td>Sun, 01 Jan 2017 00:00:00 GMT</td> </tr> </tbody> </table>	Action Id	Status	Ext Id	Error Code	Created Time	Nominal Time	1 0000064-170707154033584-oozie-oozi-C@4	SUCCEEDED	0000068-170707154033584-oozie-oozi-W		Mon, 17 Jul 2017 18:16:26 GMT	Wed, 04 Jan 2017 00:00:00 GMT	2 0000064-170707154033584-oozie-oozi-C@3	SUCCEEDED	0000067-170707154033584-oozie-oozi-W		Mon, 17 Jul 2017 18:16:26 GMT	Tue, 03 Jan 2017 00:00:00 GMT	3 0000064-170707154033584-oozie-oozi-C@2	SUCCEEDED	0000066-170707154033584-oozie-oozi-W		Mon, 17 Jul 2017 18:16:26 GMT	Mon, 02 Jan 2017 00:00:00 GMT	4 0000064-170707154033584-oozie-oozi-C@1	SUCCEEDED	0000065-170707154033584-oozie-oozi-W		Mon, 17 Jul 2017 18:16:26 GMT	Sun, 01 Jan 2017 00:00:00 GMT						
Action Id	Status	Ext Id	Error Code	Created Time	Nominal Time																															
1 0000064-170707154033584-oozie-oozi-C@4	SUCCEEDED	0000068-170707154033584-oozie-oozi-W		Mon, 17 Jul 2017 18:16:26 GMT	Wed, 04 Jan 2017 00:00:00 GMT																															
2 0000064-170707154033584-oozie-oozi-C@3	SUCCEEDED	0000067-170707154033584-oozie-oozi-W		Mon, 17 Jul 2017 18:16:26 GMT	Tue, 03 Jan 2017 00:00:00 GMT																															
3 0000064-170707154033584-oozie-oozi-C@2	SUCCEEDED	0000066-170707154033584-oozie-oozi-W		Mon, 17 Jul 2017 18:16:26 GMT	Mon, 02 Jan 2017 00:00:00 GMT																															
4 0000064-170707154033584-oozie-oozi-C@1	SUCCEEDED	0000065-170707154033584-oozie-oozi-W		Mon, 17 Jul 2017 18:16:26 GMT	Sun, 01 Jan 2017 00:00:00 GMT																															

Each action in this list correlates to an instance of the workflow intended to process one day's worth of data, where the start of that day is indicated by the nominal time.

## Next steps

In this article, we took a deep dive into operationalizing a data pipeline using Oozie.

- Read the [Apache Oozie Documentation](#)

# Streaming and Business Intelligence

8/16/2017 • 27 min to read • [Edit Online](#)

As IoT (Internet of Things) devices, such as internet-connected sensors, appliances, business and consumer devices, continue to gain popularity, so does the need for businesses to act on this data in motion. Whether the device sending real-time data over the internet is a surgical robot, or one of hundreds of sensors on a race car, it is often necessary to rapidly collect and make sense of the data it produces.

Streaming data falls in the realm of real-time processing, which can be defined as processing a typically infinite stream of input data, whose time until results are ready is short - measured in milliseconds or seconds in the longest of cases. The sample in this article uses [Event Hubs](#) to ingest simulated telemetry data from a temperature sensor. Another common entry point one might use for ingesting this type of data is [IoT Hub](#), which offers bi-directional communication with billions of devices, and exposes an Event Hubs-compatible endpoint. Apache [Kafka](#) is yet another stream-ingestion broker one could use. We will be using Apache Spark [Structured Streaming](#) to process the real-time telemetry data, and [HBase](#) for long-term storage.

Collecting all of this raw data alone does not help business leaders make informed decisions, however. The data needs to be analyzed to detect anomalies, or other thresholds that trigger business rules. It also needs to be condensed down to manageable components so it can be visualized and made sense of. This end of the spectrum is known as Business Intelligence (BI).

The scenario we are using for this exercise, we are monitoring temperature readings from a fictitious Blue Yonders airport terminal. We have created a Node.js app to generate sample temperature telemetry and insert them into an Event Hubs instance. You will learn how to process this real-time data using Spark Structured Streaming. Data will be written to an HBase table as it is being processed, and visualized using [OpenTSDB](#).

To accomplish these tasks, you will perform the following high-level steps:

1. Provision two HDInsight clusters, one with Spark 2.1, and the other with HBase.
2. Provision an Event Hubs instance.
3. Compile and run on your local workstation a sample Event Producer application that generates events to send to Event Hubs.
4. Use the [Spark Shell](#) to define and run a simple Spark Structured Streaming application.
5. Modify and compile a deployable Spark Structured Streaming application that writes processed data to HBase by sending event data to OpenTSDB through its HTTP API.
6. Configure and use OpenTSDB for BI reporting.

## Prerequisites

- An Azure subscription. See [Get Azure free trial](#).

### IMPORTANT

You do not need an existing HDInsight cluster. The steps in this document create the following resources:

- A Spark on HDInsight cluster (four worker nodes)
- An HBase on HDInsight cluster (four worker nodes)

- [Node.js](#): Used generate sample temperature event data from your local machine.
- [Maven](#): Used to build and compile the project.

- **Git:** Used to download the project from GitHub.
- An **SSH** client: Used to connect to the HDInsight clusters. For more information, see [Use SSH with HDInsight](#).

## Provision an Event Hubs namespace

In this task you will provision the Event Hubs namespace that will ultimately contain your Event Hubs instance.

1. Continue in the [Azure Portal](#).
2. Select + New, Internet of Things, Event Hubs.

The screenshot shows the Azure Marketplace interface. At the top, there's a navigation bar with 'New' and 'Internet of Things' tabs. Below the navigation bar is a search bar labeled 'Search the marketplace'. On the left, there's a sidebar titled 'MARKETPLACE' with categories: Compute, Networking, Storage, Web + Mobile, Databases, Data + Analytics, AI + Cognitive Services, Internet of Things, and Enterprise Integration. The 'Internet of Things' category is highlighted with a blue dashed border. On the right, under 'FEATURED APPS', there are three items: 'IoT Hub' (with a blue icon), 'Event Hubs' (with a blue icon), and 'Time Series Insights (preview)' (with a green icon). The 'Event Hubs' item is also highlighted with a blue border. Below each app listing is a brief description.

Category	App	Description
MARKETPLACE	IoT Hub	Connect, monitor and manage IoT devices
MARKETPLACE	Event Hubs	Cloud-scale telemetry ingestion from websites, apps, and devices.
MARKETPLACE	Time Series Insights (preview)	Azure Time Series Insights is a fully managed analytics, storage, and visualization service that makes it
MARKETPLACE	Stream Analytics job	Unlock real-time insights from

3. On the Create namespace blade, provide a unique name for your Event Hubs namespace.
4. Leave the Pricing tier at Standard.
5. Choose a Subscription and Resource Group as appropriate.
6. For the Location, choose the same Location as you used for your HDInsight cluster.

**Create namespace**

Event Hubs - PREVIEW

\* Name  
sparkstreaminghub 

.servicebus.windows.net

\* Pricing tier  
Standard >

\* Subscription 

\* Resource group   Create new  Use existing

Spark 

\* Location  
West US 2 

Throughput Units   
 1

Enable auto-inflate?

7. Select Create.

## Provision an Event Hub

In this task you will provision the Event Hub instance that will receive events from a sample application that generates random events, and that you will use as the source for events to process using Spark Structured Streaming.

- Once your Event Hubs namespace has provisioned, navigate to it in the [Azure Portal](#).
- At the top of the blade, select + Event Hub.

 Event Hub

- In the Create Event Hub blade, enter the name "sensordata" for your Event Hub.
- Leave the remaining settings at their defaults. Note that your Event Hub will have 2 partitions (as set in Partition Count).

5. Select Create.
6. On your Event Hubs blade for your namespace, select **Event Hubs**. Select the **sensordata** entry.
7. Select **Shared access policies** from the side menu.

SETTINGS

💡 Shared access policies

8. In the list of Shared Access Policies, click the **+ Add** link to add the following policies:

POLICY	CLAIMS
spark	Listen
devices	Send

9. Select both policies and copy the value under Primary Key for both, then paste them into a temporary text file. These values are your Policy Key for each policy. Also, take note that the Policy Names are "devices" and "spark".

PRIMARY KEY

10. Close the Policy blade and select Properties from the side menu.

The screenshot shows the Azure portal's 'Properties' blade for an Event Hub. On the left is a navigation menu with options like Overview, Access control (IAM), Tags, Diagnose and solve problems, Shared access policies, Scale, Properties (which is selected and highlighted in blue), Locks, and Automation script. The main area displays the following details:

NAME	sparkstreaminghub
LOCATION	West US 2
SUBSCRIPTION ID	[REDACTED]
CREATED AT	8/2/2017 10:27:36 AM
UPDATED AT	8/2/2017 10:28:01 AM
STATUS	Active

11. The value under Name is your namespace, take note of this value along with your Policy Keys and Names.  
Also, take note that the name of your Event Hub itself is "sensordata".

## Create an HBase cluster

The steps in this section use an [Azure Resource Manager template](#) to create an Azure Virtual Network and a Spark and HBase cluster on the virtual network.

The Resource Manager template used in this document is located in the article's code repo located at (TODO: UPDATE URI HERE AND FOR BUTTON BELOW) <https://raw.githubusercontent.com/ZoinerTejada/hdinsight-docs/master/code/hdinsight-streaming-and-business-intelligence/create-hbase-cluster-with-opentsdb.json>.

1. Click the following button to sign in to Azure and open the Resource Manager template in the Azure portal.

[Deploy to Azure >](#)

2. From the **Custom deployment** blade, enter the following values:

## Custom deployment

Deploy from a custom template

### TEMPLATE

 Customized template  
3 resources

 Edit template

 Edit parameters

 Learn more

### BASICS

\* Subscription

\* Resource group  Create new  Use existing

\* Location

### SETTINGS

\* Cluster Name

Cluster Login User Name

\* Cluster Login Password

Ssh User Name

\* Ssh Password

Cluster Version

Cluster Worker Node Count

Open Tsdb Version

Open Tsdb Topology

Edge Node Size

Tsd Port Number

### TERMS AND CONDITIONS

[Azure Marketplace Terms](#) | [Azure Marketplace](#)

By clicking "Purchase," I (a) agree to the applicable legal terms associated with the offering; (b) authorize Microsoft to charge or bill my current payment method for the fees associated the offering(s), including applicable taxes, with the same billing frequency as my Azure subscription, until I discontinue use of the offering(s); and (c) agree that, if the deployment involves 3rd party offerings, Microsoft may share my contact information and other details of such deployment with the publisher of that offering.

I agree to the terms and conditions stated above

Pin to dashboard

**Purchase**

- **Cluster Name:** This value (letters and numbers, no spaces) is used as the name of your provisioned HBase cluster. Be sure not to add any dashes or other special characters in the name.
- **Cluster Login User Name:** The admin user name for the Spark and HBase clusters.

- **Cluster Login Password:** The admin user password for the Spark and HBase clusters.
- **SSH User Name:** The SSH user to create for the Spark and HBase clusters.
- **SSH Password:** The password for the SSH user for the Spark and HBase clusters.
- **Location:** The region that the clusters are created in.
- **Open Tsdb Version:** The version of OpenTSDB you wish to install.
- **Open Tsdb Topology:** The deployment topology for OpenTSDB on the cluster. 'Edge Node' installs 1 TSD instance on the edge node. 'Load Balanced' installs a TSD instance of every region server in the cluster and a load balancer on the edge node distributes traffic.
- **Edge Node Size:** Size of the edge node that hosts the OpenTSDB application.
- **Tsd Port Number:** The port number on which the TSD services should listen to requests.

Click **OK** to save the parameters.

3. Use the **Basics** section to select the Resource Group you used for your Event Hub.
4. Read the terms and conditions, and then select **I agree to the terms and conditions stated above**.
5. Finally, check **Pin to dashboard** and then select **Purchase**. It takes about 20 minutes to create the cluster.

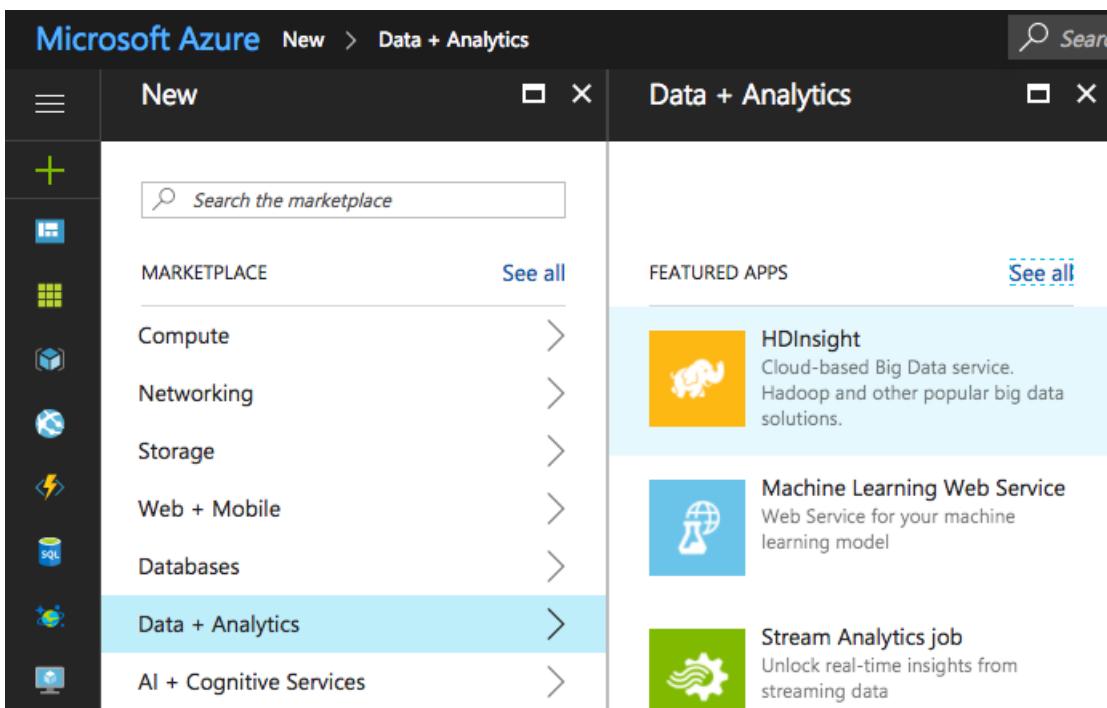
Once the resources have been created, you are redirected to a blade for the resource group that contains the cluster.

Make note of the cluster name for later.

## Provision an HDInsight cluster with Spark 2.1

To provision your HDInsight Spark cluster, follow these steps:

1. Sign in to the [Azure Portal](#).
2. Select + New, Data + Analytics, HDInsight.



3. On the basics blade, provide a unique name for your cluster.
4. Select the Subscription in which to create the cluster.
5. Select Cluster type.
6. On the Cluster configuration blade, select Cluster type of Spark and Version Spark 2.1.0 (HDI 3.6). Click select to apply the cluster type.

## Cluster configuration

* Cluster type ⓘ	* Operating system	* Version
Spark	Linux	Spark 2.1.0 (HDI 3.6)
* Cluster tier ⓘ		
<input checked="" type="radio"/> STANDARD <input type="radio"/> PREMIUM		

**Spark** : Fast data analytics and cluster computing using in-memory processing.

7. Leave the Cluster login username as "admin".
8. Provide a password for the Cluster login password. This will be the password used for both admin and the sshuser accounts.
9. Leave the Secure Shell (SSH) username as "sshuser".
10. Select the same resource group you used when you created the HBase cluster.
11. Choose the same Location you selected for the HBase cluster, in which to deploy your Spark cluster.

**Basics** □ X

---

\* Cluster name  .azurehdinsight.net

\* Subscription

\* Cluster type ⓘ > Spark 2.1 on Linux (HDI 3.6)

\* Cluster login username ⓘ  admin

\* Cluster login password ⓘ

Secure Shell (SSH) username ⓘ  sshuser

Use same password as cluster login ⓘ

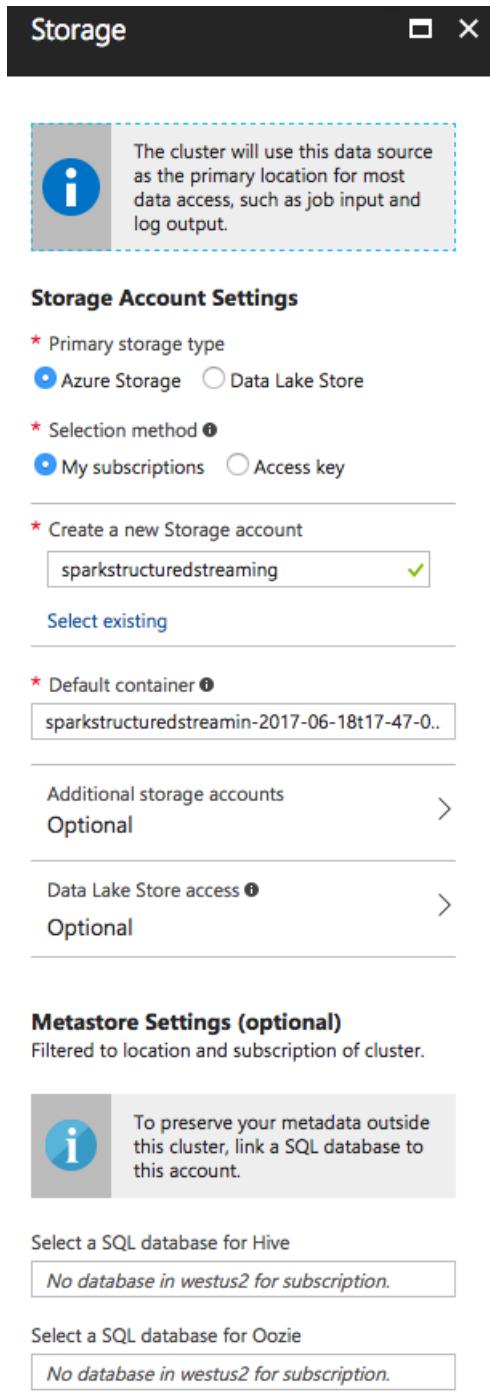
\* Resource group ⓘ  Create new  Use existing  streaming

\* Location  West US 2

**Click here** to view cores usage.

12. Select Next.
13. On the Storage blade, leave Primary Storage type set to Azure Storage and Selection method set to My subscriptions.
14. Under Select a storage account, select Create new and provide a name for the new Storage Account. This account will act as the default storage for your cluster. Optionally, you can select the same storage account provisioned for the HBase cluster. Just be sure to enter a different container name.
15. Enter a unique name for the new Storage Account.

16. Leave the remaining settings at their defaults and select Next.



17. On the Cluster summary blade, select Create.

It will take about 20 minutes to provision your cluster. Continue on to the next section while the cluster provisions.

## Download and configure the project

Use the following command to download the project from GitHub.

```
git clone <project-repo-URL>
```

After the command completes, you have the following directory structure:

```
hdinsight-streaming-and-business-intelligence/
 - SendAirportTempEvents/ <-- sends mock airport temperature sensor data to Event Hub.
 - BlueYonder/ <-- Apache Spark Structured Streaming app that processes the event data.
 -src/
 -main/
 -resources/
 -hive-config.xml
 -scala/com/microsoft/examples
 -BlueYonderMain
 -HBaseSink
 -OpenTSDBSink
```

#### NOTE

This document does not go into full details of the `SendAirportTempEvents` code included in this sample, but the code is fully commented. However, we will walk through the important aspects of the `BlueYonder` Structured Streaming app later on.

### Start generating data

1. Open a new command prompt, shell, or terminal, and change directories to **hdinsight-streaming-and-business-intelligence/SendAirportTempEvents/**. To install the dependencies needed by the application, use the following command:

```
npm install
```

2. Open the **app.js** file in a text editor and add the Event Hub information you obtained earlier:

```
// Event Hub Namespace
var namespace = 'YOUR_NAMESPACE';
// Event Hub Name
var hubname ='sensordata';
// Shared access Policy name and key (from Event Hub configuration)
var my_key_name = 'devices';
var my_key = 'YOUR_KEY';
```

#### NOTE

This example assumes that you have used **sensordata** as the name of your Event Hub, and **devices** as the name of the policy that has a **Send** claim.

3. Use the following command to insert new entries in Event Hub:

```
node app.js
```

You will see several lines of output that contain the data sent to Event Hub:

```

...
...
{
 "TimeStamp": "2017-08-03T04:29:50Z", "DeviceId": "3", "Temperature": 62}
{
 "TimeStamp": "2017-08-03T04:30:00Z", "DeviceId": "1", "Temperature": 65}
{
 "TimeStamp": "2017-08-03T04:30:00Z", "DeviceId": "2", "Temperature": 65}
{
 "TimeStamp": "2017-08-03T04:30:00Z", "DeviceId": "3", "Temperature": 62}
{
 "TimeStamp": "2017-08-03T04:30:10Z", "DeviceId": "1", "Temperature": 65}
{
 "TimeStamp": "2017-08-03T04:30:10Z", "DeviceId": "2", "Temperature": 65.05555555555556}
{
 "TimeStamp": "2017-08-03T04:30:10Z", "DeviceId": "3", "Temperature": 62}
{
 "TimeStamp": "2017-08-03T04:30:20Z", "DeviceId": "1", "Temperature": 65}
{
 "TimeStamp": "2017-08-03T04:30:20Z", "DeviceId": "2", "Temperature": 65.11111111111111}
{
 "TimeStamp": "2017-08-03T04:30:20Z", "DeviceId": "3", "Temperature": 62}
{
 "TimeStamp": "2017-08-03T04:30:30Z", "DeviceId": "1", "Temperature": 65}
{
 "TimeStamp": "2017-08-03T04:30:30Z", "DeviceId": "2", "Temperature": 65.16666666666667}
{
 "TimeStamp": "2017-08-03T04:30:30Z", "DeviceId": "3", "Temperature": 62}
{
 "TimeStamp": "2017-08-03T04:30:40Z", "DeviceId": "1", "Temperature": 65}
{
 "TimeStamp": "2017-08-03T04:30:40Z", "DeviceId": "2", "Temperature": 65.2222222222223}
{
 "TimeStamp": "2017-08-03T04:30:40Z", "DeviceId": "3", "Temperature": 62}
{
 "TimeStamp": "2017-08-03T04:30:50Z", "DeviceId": "1", "Temperature": 65}
{
 "TimeStamp": "2017-08-03T04:30:50Z", "DeviceId": "2", "Temperature": 65.2777777777779}
{
 "TimeStamp": "2017-08-03T04:30:50Z", "DeviceId": "3", "Temperature": 62}
{
 "TimeStamp": "2017-08-03T04:31:00Z", "DeviceId": "1", "Temperature": 65}
{
 "TimeStamp": "2017-08-03T04:31:00Z", "DeviceId": "2", "Temperature": 65.333333333334}
{
 "TimeStamp": "2017-08-03T04:31:00Z", "DeviceId": "3", "Temperature": 62}
{
 "TimeStamp": "2017-08-03T04:31:10Z", "DeviceId": "1", "Temperature": 65}
{
 "TimeStamp": "2017-08-03T04:31:10Z", "DeviceId": "2", "Temperature": 65.388888888889}
{
 "TimeStamp": "2017-08-03T04:31:10Z", "DeviceId": "3", "Temperature": 62}
{
 "TimeStamp": "2017-08-03T04:31:20Z", "DeviceId": "1", "Temperature": 65}
{
 "TimeStamp": "2017-08-03T04:31:20Z", "DeviceId": "2", "Temperature": 65.4444444444446}
{
 "TimeStamp": "2017-08-03T04:31:20Z", "DeviceId": "3", "Temperature": 62}
{
 "TimeStamp": "2017-08-03T04:31:30Z", "DeviceId": "1", "Temperature": 65}
{
 "TimeStamp": "2017-08-03T04:31:30Z", "DeviceId": "2", "Temperature": 65.5000000000001}
{
 "TimeStamp": "2017-08-03T04:31:30Z", "DeviceId": "3", "Temperature": 62}
{
 "TimeStamp": "2017-08-03T04:31:40Z", "DeviceId": "1", "Temperature": 65}
{
 "TimeStamp": "2017-08-03T04:31:40Z", "DeviceId": "2", "Temperature": 65.5555555555557}
{
 "TimeStamp": "2017-08-03T04:31:40Z", "DeviceId": "3", "Temperature": 62}
{
 "TimeStamp": "2017-08-03T04:31:50Z", "DeviceId": "1", "Temperature": 65}
{
 "TimeStamp": "2017-08-03T04:31:50Z", "DeviceId": "2", "Temperature": 65.611111111113}
{
 "TimeStamp": "2017-08-03T04:31:50Z", "DeviceId": "3", "Temperature": 62}
{
 "TimeStamp": "2017-08-03T04:32:00Z", "DeviceId": "1", "Temperature": 65}
{
 "TimeStamp": "2017-08-03T04:32:00Z", "DeviceId": "2", "Temperature": 65.66666666666669}
{
 "TimeStamp": "2017-08-03T04:32:00Z", "DeviceId": "3", "Temperature": 62}
{
 "TimeStamp": "2017-08-03T04:32:10Z", "DeviceId": "1", "Temperature": 65}
{
 "TimeStamp": "2017-08-03T04:32:10Z", "DeviceId": "2", "Temperature": 65.7222222222224}
{
 "TimeStamp": "2017-08-03T04:32:10Z", "DeviceId": "3", "Temperature": 62}
{
 "TimeStamp": "2017-08-03T04:32:20Z", "DeviceId": "1", "Temperature": 65}
{
 "TimeStamp": "2017-08-03T04:32:20Z", "DeviceId": "2", "Temperature": 65.7777777777778}
{
 "TimeStamp": "2017-08-03T04:32:20Z", "DeviceId": "3", "Temperature": 62}
{
 "TimeStamp": "2017-08-03T04:32:30Z", "DeviceId": "1", "Temperature": 65}
{
 "TimeStamp": "2017-08-03T04:32:30Z", "DeviceId": "2", "Temperature": 65.8333333333336}
...
...

```

#### 4. Leave the event generator running while you continue the following steps.

Sample data is loaded one at a time (8,640 \* # of devices total) to simulate a data stream, for a period representing 24 hours of temperature data from an airport terminal. Rooms start at an ambient temperature (such as 65 F), and depending on the room, flights occur every 90 minutes between 5:00am and 12:00am. As people arrive 30 minutes prior to a flight, the temperature starts to rise (5 degrees or more, depending on number of people), due to warmth from bodies. 30 minutes after the flight arrives, people are boarded, and the temperature starts to drop towards ambient temperature.

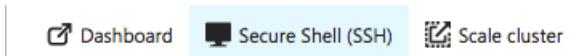
If you wish to batch load the data to quickly insert the same amount of data 500 events at a time, execute the app using the `-batch` argument (`node app.js -batch`).

# Run Spark Shell on your HDInsight cluster

In this task, you will SSH into the head node of your HDInsight cluster, launch the Spark Shell and run a Spark Streaming application that will retrieve and process the events from Event Hubs. This step is to demonstrate viewing the generated data stream using Structured Streaming. You will deploy the full streaming app later on.

By this point your HDInsight cluster should be ready. If not, you will need to wait until it finishes provisioning. Once it is ready, proceed with the following steps.

1. Navigate to your deployed HDInsight Spark cluster in the [Azure Portal](#).
  2. Select Secure Shell.



- Follow the instructions displayed for connecting to your cluster via SSH from your local environment. In general, this will mean running SSH as follows:

ssh sshuser@<yourclustername>-ssh.azurehdinsight.net

4. Complete the login by providing the password you supplied when provisioning the cluster.
  5. The application you will build requires the Spark Streaming Event Hubs package. To run the Spark Shell so that it automatically retrieves this dependency from [Maven Central](#), be sure to supply the packages switch with the Maven coordinates as follows:

```
spark-shell --packages "com.microsoft.azure:spark-streaming-eventhubs_2.11:2.1.0"
```

6. Once the Spark Shell is finished loading, you should see:

7. Copy the following code snippet into a text editor and modify it so it has the Policy Key and Namespace set as appropriate for your Event Hub.

```
val eventhubParameters = Map[String, String] (
 "eventhubs.policyname" -> "spark",
 "eventhubs.policykey" -> "<policyKey>",
 "eventhubs.namespace" -> "<namespace>",
 "eventhubs.name" -> "sensordata",
 "eventhubs.partition.count" -> "2",
 "eventhubs.consumergroup" -> "$Default",
 "eventhubs.progressTrackingDir" -> "/eventhubs/progress",
 "eventhubs.sql.containsProperties" -> "true"
)
```

8. Paste the modified snippet into the waiting scala> prompt and press return. You should see output similar

to:

```
scala> val eventhubParameters = Map[String, String] (
| "eventhubs.policynname" -> "spark",
| "eventhubs.policykey" -> "HgpaccWculG1B78BaYSKkhk3YZMogAKd5BJnLcVr/h8=",
| "eventhubs.namespace" -> "sparkstreaminghub",
| "eventhubs.name" -> "sensordata",
| "eventhubs.partition.count" -> "2",
| "eventhubs.consumergroup" -> "$Default",
| "eventhubs.progressTrackingDir" -> "/eventhubs/progress",
| "eventhubs.sql.containsProperties" -> "true"
|)
eventhubParameters: scala.collection.immutable.Map[String, String] = Map(eventhubs.sql.containsProperties -> true, eventhubs.name -> sensordata, eventhubs.consumergroup -> $Default, eventhubs.partition.count -> 2, eventhubs.progressTrackingDir -> /eventhubs/progress, eventhubs.policykey -> HgpaccWculG1B78BaYSKkhk3YZMogAKd5BJnLcVr/h8=, eventhubs.namespace -> sparkstreaminghub, eventhubs.policynname -> spark)
```

9. Next, you will begin to author a Spark Structured Streaming query by specifying the source. Paste the following into Spark Shell and press return.

```
val inputStream = spark.readStream.
format("eventhubs").
options(eventhubParameters).
load()
```

10. You should see output similar to:

```
inputStream: org.apache.spark.sql.DataFrame = [body: binary, offset: bigint ... 5 more fields]
```

11. As you can see in the previous output, the `body` is in binary format. To view our device data, we need to parse its contents. To do this, paste the following into Spark Shell and press return.

```
var inputSelect = inputStream.select(get_json_object($"body").cast("string"),
("$.Temperature").alias("Temperature"), get_json_object($"body").cast("string"),
("$.TimeStamp").alias("TimeStamp"), get_json_object($"body").cast("string"),
"$.DeviceId").alias("DeviceId"))
```

12. You should see output similar to:

```
inputSelect: org.apache.spark.sql.DataFrame = [Temperature: string, TimeStamp: string ... 1 more field]
```

13. Next, author the query so that it writes its output to the Console. Do this by pasting the following into Spark Shell and pressing return.

```
val streamingQuery1 = inputSelect.writeStream.
outputMode("append").
format("console").start().awaitTermination()
```

14. You should see some batches start with output similar to the following

```

Batch: 0

[Stage 0:> (0 + 2) / 2]
```

15. This will be followed by the output results of the processing of each microbatch of events.

```

Batch: 0

17/08/03 14:14:51 WARN TaskSetManager: Stage 1 contains a task of very large size (274 KB). The maximum recommended task size is 100 KB.
+-----+-----+
| Temperature | TimeStamp | DeviceId |
+-----+-----+
| 65.0277777777776 | 2017-08-04T21:18:10Z | 3 |
| 66.944444444444 | 2017-08-04T21:18:20Z | 1 |
| 71.1111111111128 | 2017-08-04T21:18:20Z | 2 |
| 65.0555555555537 | 2017-08-04T21:18:20Z | 3 |
| 66.91666666666623 | 2017-08-04T21:18:30Z | 1 |
| 71.16666666666684 | 2017-08-04T21:18:30Z | 2 |
| 65.0833333333314 | 2017-08-04T21:18:30Z | 3 |
| 66.8888888888846 | 2017-08-04T21:18:40Z | 1 |
| 71.222222222224 | 2017-08-04T21:18:40Z | 2 |
| 65.1111111111092 | 2017-08-04T21:18:40Z | 3 |
| 66.8611111111069 | 2017-08-04T21:18:50Z | 1 |
| 71.27777777777796 | 2017-08-04T21:18:50Z | 2 |
| 65.1388888888869 | 2017-08-04T21:18:50Z | 3 |
| 66.8333333333292 | 2017-08-04T21:19:00Z | 1 |
| 71.3333333333351 | 2017-08-04T21:19:00Z | 2 |
| 65.16666666666646 | 2017-08-04T21:19:00Z | 3 |
| 66.8055555555515 | 2017-08-04T21:19:10Z | 1 |
| 71.3888888888907 | 2017-08-04T21:19:10Z | 2 |
| 65.19444444444423 | 2017-08-04T21:19:10Z | 3 |
| 66.7777777777737 | 2017-08-04T21:19:20Z | 1 |
+-----+
only showing top 20 rows
```

16. As new events arrive from the Event Producer, they will be processed by this Structured Streaming query.

17. End this process by entering `ctrl+c` (Windows) / `cmd+c` (Mac), since we will be deploying our processor app.

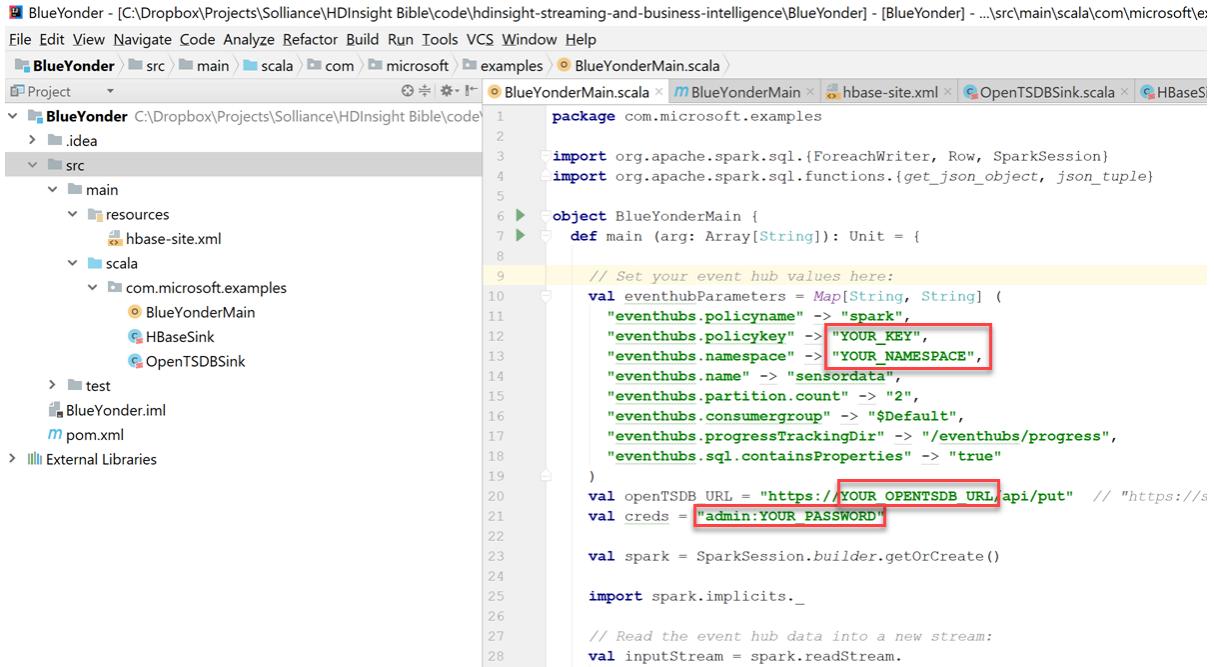
### Configure and build the Spark app

1. Open the BlueYonder project in your favorite scala IDE. In our case, we'll open it in [IntelliJ](#) Community edition.
2. Open the **BlueYonderMain.scala** file and add the Event Hub information you obtained earlier:

```
// Set your event hub values here:
val eventhubParameters = Map[String, String] (
 "eventhubs.policyname" -> "spark",
 "eventhubs.policykey" -> "YOUR_KEY",
 "eventhubs.namespace" -> "YOUR_NAMESPACE",
 "eventhubs.name" -> "sensordata",
 "eventhubs.partition.count" -> "2",
 "eventhubs.consumergroup" -> "$Default",
 "eventhubs.progressTrackingDir" -> "/eventhubs/progress",
 "eventhubs.sql.containsProperties" -> "true"
)
val openTSDB_URL = "https://YOUR_OPENTSDB_URL/api/put" // "https://streamingbi-
tdb.apps.azurehdinsight.net/api/put"
val creds = "admin:YOUR_PASSWORD"
```

## NOTE

This example assumes that you have used **sensordata** as the name of your Event Hub, and **spark** as the name of the policy that has a **Listen** claim.



```
1 package com.microsoft.examples
2
3 import org.apache.spark.sql.{ForeachWriter, Row, SparkSession}
4 import org.apache.spark.sql.functions.{get_json_object, json_tuple}
5
6 object BlueYonderMain {
7 def main (arg: Array[String]): Unit = {
8
9 // Set your event hub values here:
10 val eventhubParameters = Map[String, String] (
11 "eventhubs.policyname" -> "spark",
12 "eventhubs.policykey" -> "YOUR_KEY",
13 "eventhubs.namespace" -> "YOUR_NAMESPACE",
14 "eventhubs.name" -> "sensordata",
15 "eventhubs.partition.count" -> "2",
16 "eventhubs.consumergroup" -> "$Default",
17 "eventhubs.progressTrackingDir" -> "/eventhubs/progress",
18 "eventhubs.sql.containsProperties" -> "true"
19)
20 val openTSDB_URL = "https://YOUR_OPENTMDB_URL/api/put" // "https://s
21 val creds = "admin:YOUR_PASSWORD"
22
23 val spark = SparkSession.builder.getOrCreate()
24
25 import spark.implicits._
26
27 // Read the event hub data into a new stream:
28 val inputStream = spark.readStream.
```

Change the following:

- **eventhubs.policykey**: Change "YOUR\_KEY" to your listener policy's (such as *spark*) key.
- **eventhubs.namespace**: Change "YOUR\_NAMESPACE" to the Event Hubs namespace.
- **val creds**: Change "admin:YOUR\_PASSWORD" to the admin password you set when you provisioned your HBase cluster. Used for authenticating OpenTSDB POST requests.
- **val openTSDB\_URL**: Change "YOUR\_OPENTMDB\_URL" to the URL for your installed OpenTSDB app. To find this, go to your HBase cluster in the Azure Portal. Click **Applications** on the left-hand menu, then select the **opentsdb** application to view its properties. Copy its WEBPAGE URL. The `openTSDB_URL` value should be similar to `https://streamingbi-tdb.apps.azurehdinsight.net/api/put`. Make sure the URL ends with `/api/put`.

The screenshot shows the Azure HDInsight portal's 'Applications' page. On the left, a sidebar lists various management options like 'Locks', 'Automation script', 'Quick start', 'Tools for HDInsight', 'Cluster login', 'Subscription cores usage', 'Scale cluster', 'Secure Shell (SSH)', 'HDInsight partner', 'Script actions', and 'Applications'. The 'Applications' link is highlighted with a red circle labeled '1'. The main area displays a table with columns 'NAME', 'PUBLISHER', 'VERSION', and 'STATUS'. An entry for 'opentsdb' is shown with its status as 'Installed' and a 'Portal' link.



APP NAME

opentsdb

STATUS

Installed

WEBPAGE

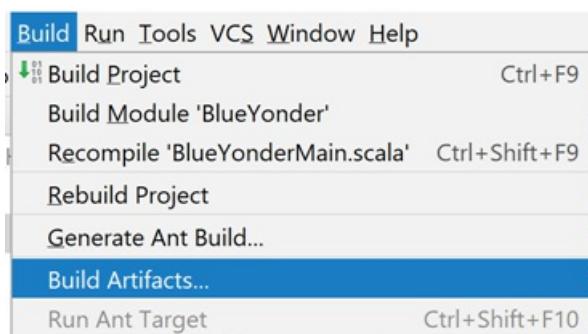
<https://streamingbi-tdb.apps.azurehdinsig...>

SSH ENDPOINT

opentsdb.streamingbi-ssh.azurehdinsig...



3. Build the application by clicking **Build** --> **Build Artifacts...** from the menu.



Then click the **Build** action underneath the BlueYonder\_DefaultArtifact artifact within the Build Artifact menu.



## Upload and execute the app

From a bash shell prompt for SCP, or command prompt for PSCP, upload the compiled JAR file to the local storage of your HDInsight cluster head node. As done earlier, replace **USERNAME** with the SSH user you provided when creating the cluster, and **SPARKCLUSTERNAME** with the name you provided earlier for your Spark cluster. When prompted, enter the password for the SSH user. Replace the

```
/path/to/BlueYonder/out/artifacts/BlueYonder_DefaultArtifact/default_artifact.jar
```

 with the path to this file in the BlueYonder project.

```
scp /path/to/BlueYonder/out/artifacts/BlueYonder_DefaultArtifact/default_artifact.jar
USERNAME@SPARKCLUSTERNAME-ssh.azurehdinsight.net:default_artifact.jar
```

Example using PSCP:

```
pscp "c:\path\to\BlueYonder\out\artifacts\BlueYonder_DefaultArtifact\default_artifact.jar"
USERNAME@SPARKCLUSTERNAME-ssh.azurehdinsight.net:default_artifact.jar
```

Now, SSH into your Spark cluster and use the **HDFS** command to copy the file from your head node local storage to Azure Storage:

```
hdfs dfs -put ./default_artifact.jar /example/jars/default_artifact.jar
```

It is preferable to store the application in Azure Storage so that it is still accessible across your cluster's lifecycle. Meaning, if you were to delete and recreate your cluster, your app will still be available to run.

Next, execute the following command to submit and run the uploaded app, replacing **SPARKCLUSTERNAME** with the Spark cluster name you specified earlier, and **STORAGENAME** with the name of your cluster's default storage account:

```
spark-submit --class com.microsoft.examples.BlueYonderMain --master yarn-cluster --packages
"com.microsoft.azure:spark-streaming-eventhubs_2.11:2.1.0,org.apache.hbase:hbase-
common:1.3.1,org.apache.hbase:hbase-server:1.3.1"
wasb://SPARKCLUSTERNAME@STORAGENAME.blob.core.windows.net/example/jars/default_artifact.jar
```

Note that when submitting spark jobs that use Azure Storage as the JAR's source, you need to specify the **--master** URL as either **yarn** or **yarn-cluster**. Also notice that we're using the **--packages** switch as maven coordinates to our app's dependencies.

You should see output that is constantly scrolling, similar to the following snapshot:

```

...
17/08/06 18:36:19 INFO Client: Application report for application_1502027043716_0004 (state: ACCEPTED)
17/08/06 18:36:19 INFO Client:
 client token: N/A
 diagnostics: AM container is launched, waiting for AM container to Register with RM
 ApplicationMaster host: N/A
 ApplicationMaster RPC port: -1
 queue: default
 start time: 1502044578458
 final status: UNDEFINED
 tracking URL: http://hn1-
spark.4tssacnxawrelhhk15p2yjjggf.xx.internal.cloudapp.net:8088/proxy/application_1502027043716_0004/
 user: sshuser
17/08/06 18:36:20 INFO Client: Application report for application_1502027043716_0004 (state: ACCEPTED)
17/08/06 18:36:21 INFO Client: Application report for application_1502027043716_0004 (state: ACCEPTED)
17/08/06 18:36:22 INFO Client: Application report for application_1502027043716_0004 (state: ACCEPTED)
17/08/06 18:36:23 INFO Client: Application report for application_1502027043716_0004 (state: ACCEPTED)
...
...

```

Let this run in the background. At this time, the data generator should still be running as well, sending simulated streaming data to your event hub. If it has finished running, execute it once more from the command prompt with this command:

```
node app.js
```

## What the Spark app is doing

The `com.microsoft.azure.spark-streaming-eventhubs` library processes streaming data from Event Hubs, and acts as an input source through the `DataStreamReader` interface. The result is a stream of untyped `Dataframe`s, meaning that the schema of the `DataFrame` is not checked at compile time. The Event Hubs source is defined through the `format` property of the streaming `DataFrame` by setting the value to "eventhubs".

```
// Read the event hub data into a new stream:
val inputStream = spark.readStream.
 format("eventhubs").
 options(eventhubParameters).
 load()
```

The `inputStream` `DataFrame` represents an unbounded table containing the Event Hubs data, whose source definition was passed to the `options` property. We capitalize on the fact that most common operations on the `DataFrame` are supported for streaming. One of the columns of our unbounded table is "body", which contains the binary representation of the JSON-formatted data we sent to Event Hubs from our simulated sensors. To make it easier to extract that data, we cast the value to String format, then parse it using `get_json_object` to retrieve our telemetry properties. It is important to note that at this point, we're just setting up the data transformation. Data is not being received yet, because we have not started it.

```
var inputSelect = inputStream.select(
 get_json_object($"body").cast("string"), ".$.Temperature".alias("Temperature"),
 get_json_object($"body").cast("string"), ".$.TimeStamp".alias("TimeStamp"),
 get_json_object($"body").cast("string"), ".$.DeviceId".alias("DeviceId"))
```

At this point, we've set up the data transformations on our streaming data. What is left is to actually start receiving the data and processing it, sending it to our presentation/BI layer.

There are a few built-in output sinks you can choose from, such as:

1. **File sink**: Stores the output to a directory in various file formats.
2. **Console sink**: Used for debugging purposes, as we did within the `spark-shell` session earlier.
3. **Memory sink**: The output is stored in memory as an in-memory table, and should be used for debugging purposes only.
4. **Foreach sink**: Allows you to run arbitrary computation on the records in the output. This is what we are using for our custom sinks.

The project contains two custom Foreach sinks: **HBaseSink** for writing directly to an HBase table, and **OpenTSDBSink** that sends data to OpenTSDB via its HTTP API.

If you wish to use the `Foreach` sink by implementing the `ForeachWriter` interface, the writer must be serializable. It is not enough to simply declare the writer in-line to the `writeStream` object, or as a separate class within your executing class. You must create a separate file for it, as we have done with the `HBaseSink` and `OpenTSDBSink` classes. Furthermore, your writer must do all of the initializataion within the overridden `open` method (such as opening connections, starting a transaction, etc.).

When we start receiving data, we first instantiate a new `OpenTSDBSink` class, passing in the OpenTSDB HTTP API URL and credentials, then pass that in to the `foreach` sink.

```
// Send data to OpenTSDB, using our custom OpenTSDBSink class:
val writer = new OpenTSDBSink(openTSDB_URL, creds)
inputSelect.writeStream.outputMode("append").foreach(writer).start().awaitTermination()
```

The `start()` method starts the streaming computation in the background, and `.awaitTermination()` prevents the process from exiting while the query is active.

When you look at the `OpenTSDBSink` class, you will see that it extends the `ForeachWriter` interface, news up a `DefaultHttpClient` and `HttpPost` object, sets up the telemetry payload (including converting the date/time entry to the required timestamp/epoch format), and POSTs the request to our hosted OpenTSDB service on the HBase cluster.

```

class OpenTSDBSink(url:String, usernamePassword:String) extends ForeachWriter[org.apache.spark.sql.Row] {
 var encoding:String = _
 var httpClient:DefaultHttpClient = _
 var post:HttpPost = _

 override def process(value: Row): Unit = {
 val temp = value(0)
 val time = value(1)
 val device = value(2)

 if (temp != null && time != null) {
 // Convert time to epoch (timestamp)
 val timestamp = Instant.parse(time.toString).getEpochSecond
 val body = f"""{
 | "metric": "sensor.temperature",
 | "value": "$temp",
 | "timestamp": $timestamp,
 | "tags": {"deviceid": "$device"}
 |}""".stripMargin
 post.setEntity(new StringEntity(body))
 httpClient.execute(post)
 }
 }

 override def close(errorOrNull: Throwable): Unit = {
 }

 override def open(partitionId: Long, version: Long): Boolean = {
 encoding = Base64.getEncoder.encodeToString(usernamePassword.getBytes(StandardCharsets.UTF_8))
 httpClient = new DefaultHttpClient()
 post = new HttpPost(url)
 post.setHeader("Content-type", "application/json")
 post.setHeader("Authorization", "Basic " + encoding)

 true
 }

}

```

This `OpenTSDBSink` class can easily be customized to send this streaming data to the Microsoft Power BI REST API, or any other BI, reporting, or visualization platform that exposes an HTTP endpoint.

The `BlueYonderMain.scala` file also contains commented code that demonstrates how to use the `file` sink to save Parquet-formatted files to the cluster's default Azure Storage account (as indicated by the `wasb:` paths). This opens up additional opportunities for working with the streaming data.

```

// Alternate method: output to parquet files
inputSelect.writeStream.outputMode("append").
 option("checkpointLocation", "wasb:///temp").
 format("parquet").
 option("path", "wasb:///StreamOutNew").
 start().awaitTermination()

```

## Business Intelligence using OpenTSDB

[OpenTSDB](#) is a high performance, open source storage engine that allows users to 'Store and serve massive amounts of time series data without losing granularity.'

OpenTSDB uses Apache HBase to store its data. The ARM template you used at the beginning of this article installed OpenTSDB on every HBase Region Server in your cluster.

The OpenTSDB service is installed as a service of Ambari, which effectively makes OpenTSDB a full Platform As A

Service (PaaS) offering. No manual configuration, management or monitoring is required to keep the service running as the integration with Ambari ensures that these functions are performed without manual intervention. Additionally, if desired, the OpenTSDB daemons (TSDs) may be configured and monitored via the Ambari web interface.

The ARM template also installed an HTTP proxy on an Edge Node, which allows time-series collectors outside of the HBase cluster, such as our Spark Structured Streaming app, to be able to send metrics to the system. The OpenTSDB web UX is accessible via this proxy as well as through an Ambari View installed on the Ambari web interface.

If you receive the following error when trying to display data on the OpenTSDB UI:

```
'gnuplot': No such file or directory
```

, you need to execute the following command from each node of your cluster (head nodes, worker nodes, and edge node): `sudo apt-get install gnuplot-x11`

## OpenTSDB in Ambari

When you provisioned the HBase cluster with the ARM template, a few Script Actions were executed that integrated OpenTSDB with Ambari. Log into the Ambari web interface by going to <HTTPS://CLUSTERNAME.azurehdinsight.net>, where CLUSTERNAME is the name of your HBase cluster. You will need to input your admin username and password you specified at cluster creation.

When the dashboard loads, you will see OpenTSDB in the list of services. Click on the menu item to view a summary. Here you can view which nodes have OpenTSDB services running, and their status.

The screenshot shows the Ambari web interface with the following details:

- Header:** Ambari streamingbi 0 ops 0 alerts
- Left sidebar (Services):** A list of services with green checkmarks: HDFS, YARN, MapReduce2, Tez, Hive, HBase, Pig, Sqoop, Oozie, ZooKeeper, Ambari Metrics, and OpenTSDB. The "OpenTSDB" service is highlighted with a red box.
- Summary Tab:** Contains tabs for Summary and Configs. The Summary tab is active, showing the following data:
  - OpenTSDB TSD:** 4 instances, all Started, No alerts.
  - OpenTSDB External Proxy:** 1/1 OpenTSDB External Proxy Live.
- Metrics Tab:** Contains sections for Writes and Deletes. The Writes section shows "No Data Available".

Click on the Configs tab to view basic and advanced configuration options.

The advanced tab provides an easy interface for making changes to the OpenTSDB configs, compared to modifying the individual configuration files on each node.

You can view the OpenTSDB interface by selecting OpenTSDB View from the list of available Ambari views.

The screenshot shows the Ambari interface with the 'Your Views' section. It lists several views: 'YARN Queue Manager (1.0.0)', 'Hive View 2.0 (2.0.0)', 'OpenTSDB View (1.0.0)', and 'Tez View (0.7.0.2.6.1.0-129)'. The 'OpenTSDB View' option is highlighted with a red box.

When you select the view, you will be prompted to enter the same admin credentials you used to connect to Ambari.

## Displaying telemetry data in OpenTSDB

The top of the OpenTSDB UI page contains a form through which you can specify the time series date range, metric, tags, chart positioning, etc.

The screenshot shows the OpenTSDB UI configuration form. It includes fields for 'From' (2017/08/03-00:00:00) and 'To' ((now)). The 'Metric' field is set to 'sensor.temperature'. In the 'Tags' section, there are three entries: 'deviceid' with value '3', 'deviceid' with value '2', and 'deviceid' with value '1'. To the right, there is a 'Global annotations' section with 'WxH: 1542x868' and a checked checkbox. Below the main form, a message says '111952 points retrieved, 1869 points plotted in 222ms.'

Configure the following options to display the telemetry data:

- From:** Enter the start date and time of your first device data entry.
- To:** Enter the end date of the series data. Remember, by default, the generated data begins at the start of the current day (12:01am), and ends at the end of the day. If you choose (now), you may not see all of the available data. Autoreload can be used to refresh at defined intervals. For now, leave this unchecked.
- Metric:** Enter **sensor.temperature**, which is the value we sent from the **OpenTSDBSink** writer in the Spark application.
- Tags:** Enter a tag for each temperature sensor device by Id:
  - key: **deviceid**, value: **1**
  - key: **deviceid**, value: **2**
  - key: **deviceid**, value: **3**
- Aggregator:** Set to sum.
- Downsample:** Check this box to reduce noise from all of the data points (one for every 10 seconds).

On the right-hand side of this form is a section containing 3 tabs: Axes, Key, and Style. Within the **Axes** tab, set the following values:

- Label (Y):** Set this to **Temperature**.
- Range (Y):** Set to [55:85]. This constrains the displayed Y-axis range to temperatures between 55 and 85 degrees fahrenheit.

Now select the **Key** tab and configure the following:

- Key location:** This places the key (or graph legend) in your desired location over top of the chart.
- Box:** Check this to place a box around the key.

WxH: 1542x868  Global annotations

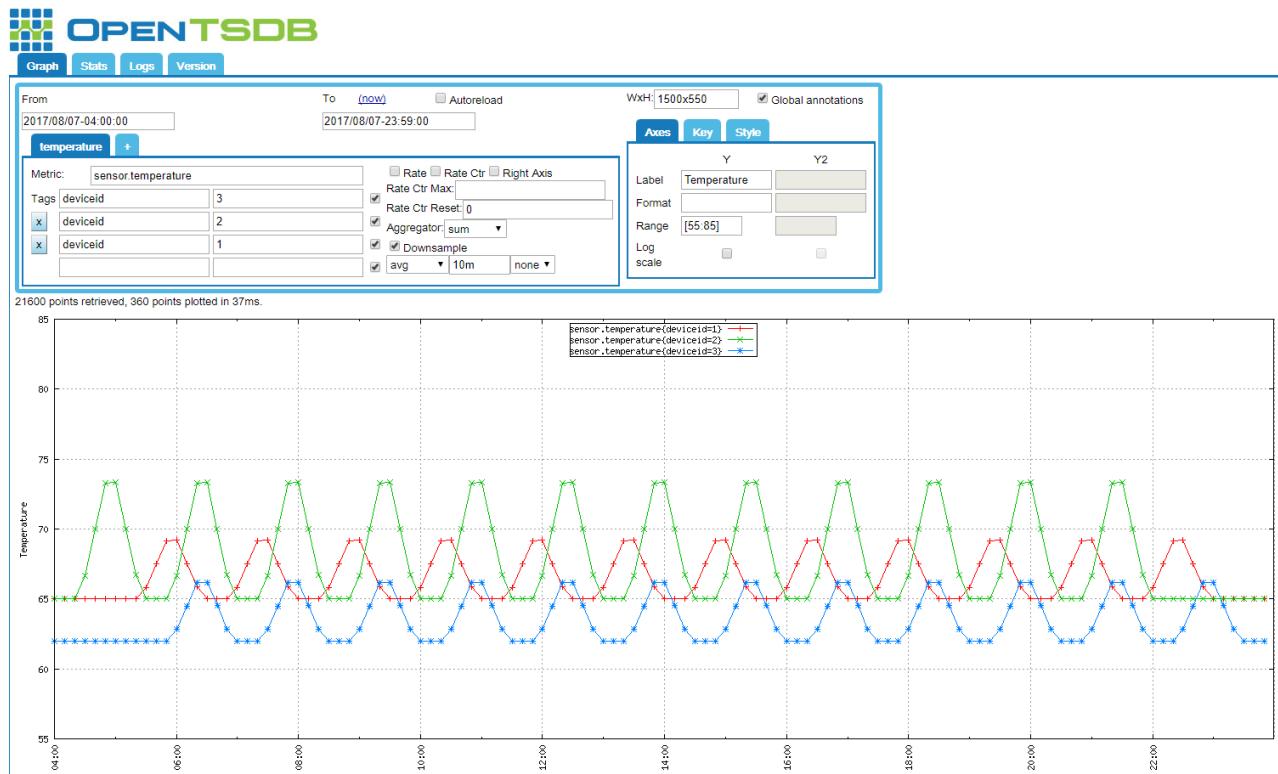
Axes Key Style

Key location:

Horizontal layout  
 Box  
 No key (overrides others)

Configure other options as desired.

Once you have set the filter and display options, you should see a graph similar to the following, once an ample amount of data has been collected and stored:



Notice that OpenTSDB automatically color-coded each of the device data series. With this visualization, you can quickly infer the differing flight schedules, ambient room temperatures, and general sense of crowd sizes in each of the rooms in which each sensor is located.

One thing we would like to know is what the overall average temperature of all devices looks like with respect to the current graph. To show this new data point, click the + symbol text to the **temperature** tab in the form you just modified. This allows us to add a new metric.

From  
2017/08/03-00:00:00

temperature +

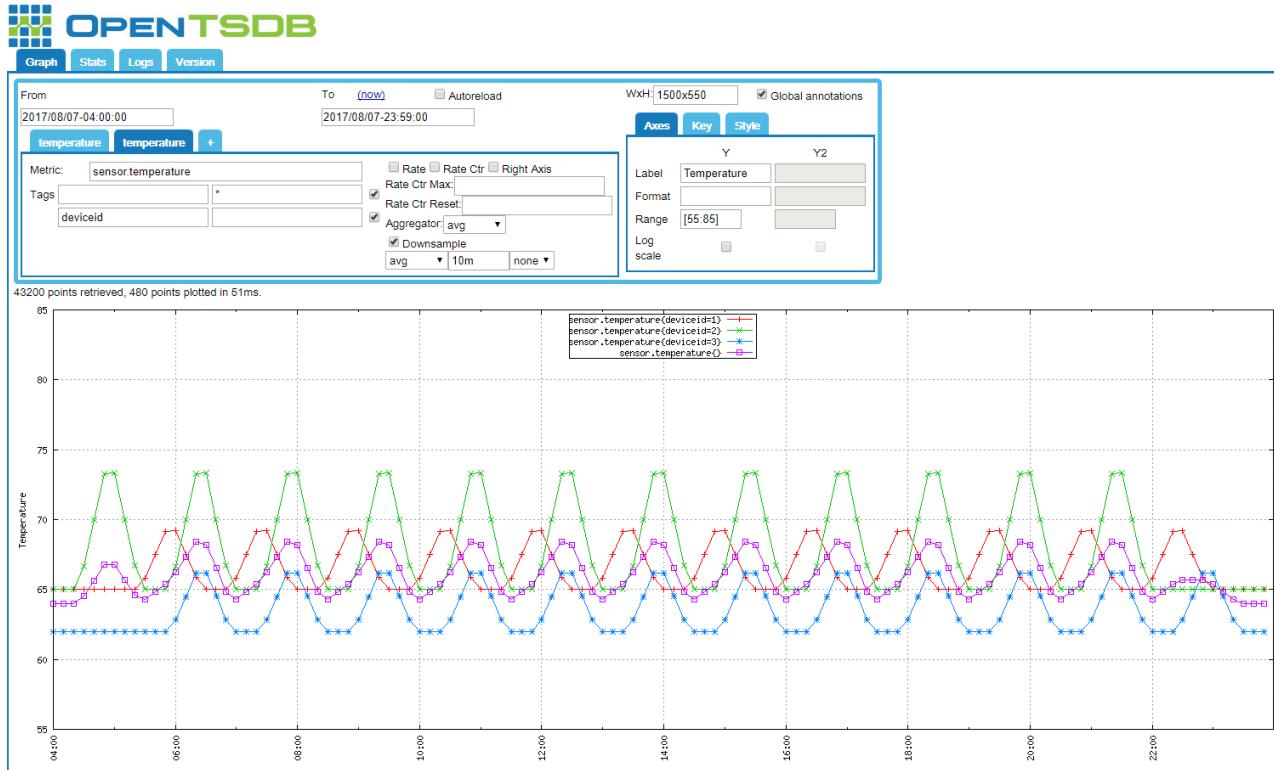
Metric:

In the new metric form, enter the following:

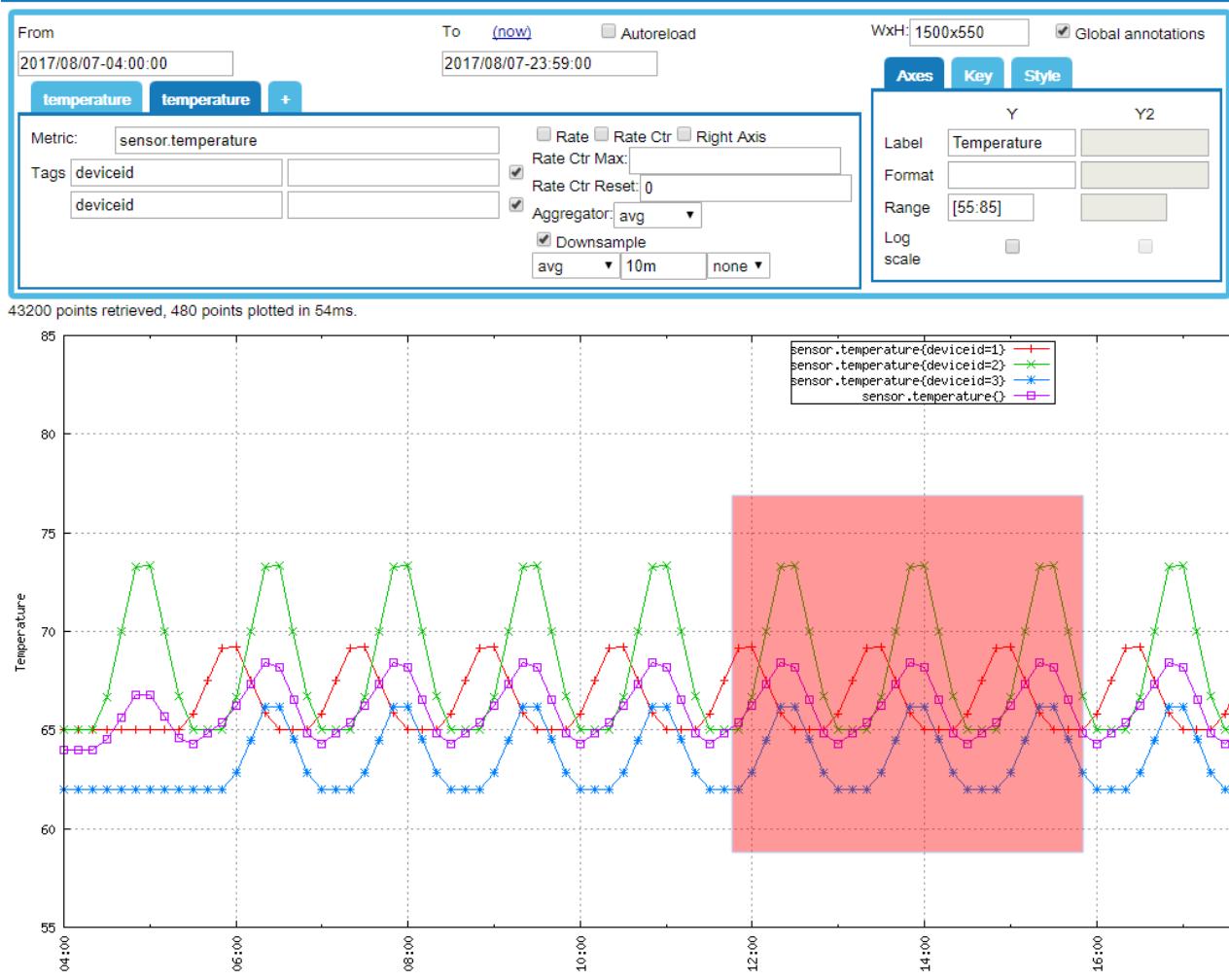
1. **Metric:** Enter **sensor.temperature**.

2. **Tags:** Enter an asterisk (\*) in the value field and check the group checkbox to the right. This ensures all devices are combined:
  - key: (blank), value: \*
  - key: **deviceid**, value: (blank)
3. **Aggregator:** Set to avg (gets the average temperature for all devices).
4. **Downsampling:** Check this box so reduce noise from all of the data points (one for every 10 seconds \* 3).

You will now see a 4th line on the chart (in purple) for the average temperature of all devices.



Now click and drag with your mouse to zoom in on a smaller time slice of data.



The page will be refreshed to display just the zoomed in time range. Notice that the From and To form values have been updated accordingly.



Another thing to note is that each time you make a change to the chart values or displayed range, the page refreshes with a custom URL. This allows you to bookmark or save the displayed chart to share or view later.

The tabs at the top of the page allow you to view statistics, logs, and version information. When you click on the **Stats** tab, you will see interesting information about the number of data input requests received from the Spark app, how many failed or caused exceptions, if any, the number of graph requests, latency, and other interesting bits of information.



The screenshot shows the OpenTSDB stats interface with the 'Stats' tab selected. The interface includes a navigation bar with 'Graph', 'Stats' (selected), 'Logs', and 'Version' buttons. Below the navigation bar is a table displaying various metrics and their corresponding values and types.

Metric	Value	Count	Type
tsd.connectionmgr.connections	1502111526	1	type=open host=wn1-stream
tsd.connectionmgr.connections	1502111526	0	type=rejected host=wn1-stream
tsd.connectionmgr.connections	1502111526	52981	type=total host=wn1-stream
tsd.connectionmgr.exceptions	1502111526	0	type=closed host=wn1-stream
tsd.connectionmgr.exceptions	1502111526	0	type=reset host=wn1-stream
tsd.connectionmgr.exceptions	1502111526	0	type=timeout host=wn1-stream
tsd.connectionmgr.exceptions	1502111526	0	type=unknown host=wn1-stream
tsd.rpc.received	1502111526	0	type=telnet host=wn1-stream
tsd.rpc.received	1502111526	52315	type=http host=wn1-stream
tsd.rpc.received	1502111526	0	type=http_plugin host=wn1-stream
tsd.rpc.exceptions	1502111526	0	host=wn1-stream
tsd.http.latency_50pct	1502111526	2	type=all host=wn1-stream
tsd.http.latency_75pct	1502111526	2	type=all host=wn1-stream
tsd.http.latency_90pct	1502111526	2	type=all host=wn1-stream
tsd.http.latency_95pct	1502111526	2	type=all host=wn1-stream
tsd.http.latency_50pct	1502111526	36	type=graph host=wn1-stream
tsd.http.latency_75pct	1502111526	48	type=graph host=wn1-stream
tsd.http.latency_90pct	1502111526	72	type=graph host=wn1-stream
tsd.http.latency_95pct	1502111526	94	type=graph host=wn1-stream
tsd.http.latency_50pct	1502111526	20	type=gnuplot host=wn1-stream
tsd.http.latency_75pct	1502111526	20	type=gnuplot host=wn1-stream
tsd.http.latency_90pct	1502111526	30	type=gnuplot host=wn1-stream
tsd.http.latency_95pct	1502111526	36	type=gnuplot host=wn1-stream
tsd.http.graph.requests	1502111526	4350	cache=disk host=wn1-stream
tsd.http.graph.requests	1502111526	782	cache=miss host=wn1-stream
tsd.rpc.received	1502111526	46479	type=put host=wn1-stream
tsd.rpc.errors	1502111526	0	type=hbase_errors host=wn1-stream
tsd.rpc.errors	1502111526	0	type=invalid_values host=wn1-stream
tsd.rpc.errors	1502111526	0	type=illegal_arguments host=wn1-stream
tsd.rpc.errors	1502111526	0	type=unknown_metrics host=wn1-stream
tsd.rpc.errors	1502111526	0	type=socket_writes_blocked host=wn1-stream
tsd.http.query.invalid_requests	1502111526	0	host=wn1-stream
tsd.http.query.exceptions	1502111526	0	host=wn1-stream
tsd.http.query.success	1502111526	0	host=wn1-stream
tsd.jvm.thread.states	1502111526	0	state=new host=wn1-stream
tsd.jvm.thread.states	1502111526	20	state=runnable host=wn1-stream
tsd.jvm.thread.states	1502111526	0	state=blocked host=wn1-stream
tsd.jvm.thread.states	1502111526	6	state=waiting host=wn1-stream
tsd.jvm.thread.states	1502111526	0	state=terminated host=wn1-stream

The **Logs** tab displays entries for every action in the chart's configuration, as well as other items of interest, such as incoming `put` requests to the OpenTSDB HTTP API. In the below screenshot, you can see `put` requests invoked by our running Spark app (highlighted).



Graph Stats Logs Version

Mon Aug 07 11:50:11 GMT-400 2017 [id: 0x716aec7e, /10.0.0.9:43842 => /10.0.0.8:4242] CONNECTED: /10.0.0.9:43842	INFO	OpenTSDB I/O Worker #2	tsd.ConnectionManager
Mon Aug 07 11:50:11 GMT-400 2017 [id: 0x716aec7e, /10.0.0.9:43842 => /10.0.0.8:4242] BOUND: /10.0.0.8:4242	INFO	OpenTSDB I/O Worker #2	tsd.ConnectionManager
Mon Aug 07 11:50:11 GMT-400 2017 [id: 0x716aec7e, /10.0.0.9:43842 => /10.0.0.8:4242] OPEN	INFO	OpenTSDB I/O Boss #1	tsd.ConnectionManager
Mon Aug 07 11:50:11 GMT-400 2017 [id: 0x86908198, /52.183.68.131:1537 => /10.0.0.8:4242] HTTP /api/put done in 0ms	INFO	OpenTSDB I/O Worker #1	tsd.HttpQuery
Mon Aug 07 11:50:11 GMT-400 2017 [id: 0x86908198, /10.0.0.9:43840 > /10.0.0.8:4242] CLOSED	INFO	OpenTSDB I/O Worker #1	tsd.ConnectionManager
Mon Aug 07 11:50:11 GMT-400 2017 [id: 0x86908198, /10.0.0.9:43840 > /10.0.0.8:4242] UNBOUND	INFO	OpenTSDB I/O Worker #1	tsd.ConnectionManager
Mon Aug 07 11:50:11 GMT-400 2017 [id: 0x86908198, /10.0.0.9:43840 > /10.0.0.8:4242] DISCONNECTED	INFO	OpenTSDB I/O Worker #1	tsd.ConnectionManager
Mon Aug 07 11:50:11 GMT-400 2017 [id: 0x86908198, /10.0.0.9:43840 => /10.0.0.8:4242] CONNECTED: /10.0.0.9:43840	INFO	OpenTSDB I/O Worker #1	tsd.ConnectionManager
Mon Aug 07 11:50:11 GMT-400 2017 [id: 0x86908198, /10.0.0.9:43840 => /10.0.0.8:4242] BOUND: /10.0.0.8:4242	INFO	OpenTSDB I/O Worker #1	tsd.ConnectionManager
Mon Aug 07 11:50:11 GMT-400 2017 [id: 0x86908198, /10.0.0.9:43840 => /10.0.0.8:4242] OPEN	INFO	OpenTSDB I/O Boss #1	tsd.ConnectionManager
Mon Aug 07 11:50:11 GMT-400 2017 [id: 0x277a1750, /52.183.68.131:1537 => /10.0.0.8:4242] HTTP /api/put done in 0ms	INFO	OpenTSDB I/O Worker #8	tsd.HttpQuery
Mon Aug 07 11:50:11 GMT-400 2017 [id: 0x277a1750, /10.0.0.9:43838 > /10.0.0.8:4242] CLOSED	INFO	OpenTSDB I/O Worker #8	tsd.ConnectionManager
Mon Aug 07 11:50:11 GMT-400 2017 [id: 0x277a1750, /10.0.0.9:43838 > /10.0.0.8:4242] UNBOUND	INFO	OpenTSDB I/O Worker #8	tsd.ConnectionManager
Mon Aug 07 11:50:11 GMT-400 2017 [id: 0x277a1750, /10.0.0.9:43838 > /10.0.0.8:4242] DISCONNECTED	INFO	OpenTSDB I/O Worker #8	tsd.ConnectionManager
Mon Aug 07 11:50:11 GMT-400 2017 [id: 0x277a1750, /10.0.0.9:43838 => /10.0.0.8:4242] CONNECTED: /10.0.0.9:43838	INFO	OpenTSDB I/O Worker #8	tsd.ConnectionManager
Mon Aug 07 11:50:11 GMT-400 2017 [id: 0x277a1750, /10.0.0.9:43838 => /10.0.0.8:4242] BOUND: /10.0.0.8:4242	INFO	OpenTSDB I/O Worker #8	tsd.ConnectionManager
Mon Aug 07 11:50:11 GMT-400 2017 [id: 0x277a1750, /10.0.0.9:43838 => /10.0.0.8:4242] OPEN	INFO	OpenTSDB I/O Boss #1	tsd.ConnectionManager
Mon Aug 07 11:50:11 GMT-400 2017 [id: 0x255c4515, /52.183.68.131:1537 => /10.0.0.8:4242] HTTP /api/put done in 0ms	INFO	OpenTSDB I/O Worker #7	tsd.HttpQuery
Mon Aug 07 11:50:11 GMT-400 2017 [id: 0x255c4515, /10.0.0.9:43836 > /10.0.0.8:4242] CLOSED	INFO	OpenTSDB I/O Worker #7	tsd.ConnectionManager
Mon Aug 07 11:50:11 GMT-400 2017	INFO	OpenTSDB I/O Worker #7	tsd.ConnectionManager

Learn more about using the [OpenTSDB GUI](#).

## Optional steps to directly write to HBase

If you wish to write directly to an HBase table of your own choosing, you will need to follow the steps below **on a new set of clusters**, following [these instructions](#), or using the ARM template [in this article](#).

### NOTE

If you choose to go this route, you will need to first provision a virtual network, then add both the Spark and HBase clusters to it so that the structured streaming app running on the Spark cluster can directly communicate with the HBase cluster using the HBase Java API.

To write to HBase from the Spark cluster, you must provide the Spark app with the configuration details of your HBase cluster. This example uses the **hbase-site.xml** file from the HBase cluster.

### Download the hbase-site.xml

If you wish to write your event data directly to HBase, you will need to copy the **hbase-site.xml** file to the Spark application's resource directory.

From a bash shell prompt, use SCP to download the **hbase-site.xml** file from the cluster. Alternately, if using Windows, you can use **PSCP** from the command prompt. In the following example, replace **USERNAME** with the SSH user you provided when creating the cluster, and **BASENAME** with the base name you provided earlier. When prompted, enter the password for the SSH user. Replace the

`/path/to/BlueYonder/src/main/resources/hbase-site.xml` with the path to this file in the BlueYonder project.

```
scp USERNAME@hbase-BASENAME-ssh.azurehdinsight.net:/etc/hbase/conf/hbase-site.xml
/path/to/BlueYonder/src/main/resources/hbase-site.xml
```

This command downloads the **hbase-site.xml** to the path specified.

Example using PSCP:

```
pscp USERNAME@hbase-BASENAME-ssh.azurehdinsight.net:/etc/hbase/conf/hbase-site.xml
c:\path\to\Blueprint\src\main\resources\hbase-site.xml
```

## Create the HBase table

To store data in HBase, we must first create a table. Pre-create resources that the Spark Structured Streaming app needs to write to, as trying to create resources from inside the app can result in multiple instances trying to create the same resource. Learn more about HBase with [an Apache HBase example in HDInsight](#).

1. Use SSH to connect to the HBase cluster using the SSH user and password you supplied to the template during cluster creation. For example, if connecting using the `ssh` command, you would use the following syntax:

```
ssh USERNAME@hbase-BASENAME-ssh.azurehdinsight.net
```

In this command, replace **USERNAME** with the SSH user name you provided when creating the cluster, and **BASENAME** with the base name you provided. When prompted, enter the password for the SSH user.

2. From the SSH session, start the HBase shell.

```
hbase shell
```

Once the shell has loaded, you see an `hbase(main):001:0>` prompt.

3. From the HBase shell, enter the following command to create a table with a Device column family, to store the sensor data:

```
create 'SensorData', 'Device'
```

4. Verify that the table has been created by using the following command:

```
scan 'SensorData'
```

This returns information similar to the following example, indicating that there are 0 rows in the table.

ROW	COLUMN+CELL	0 row(s) in 0.1900 seconds

5. Enter `exit` to exit the HBase shell:

## Modify the Spark Structured Streaming app to use the HBaseSink

Modify the `BlueYonderMain.scala` file in the Spark Structured Streaming app to use the `HBaseSink` class instead of the currently referenced `OpenTSDBSink` implementation.

```
// Alternate method 2: use the custom HBaseSink to write directly to HBase
val writer = new HBaseSink()

inputSelect.writeStream.outputMode("append").foreach(writer).start().awaitTermination()
```

## Next steps

- Learn more about [Structured Streaming](#)
- Learn more about [connecting to and using Ambari](#) to manage your clusters

# Troubleshooting a Failed or Slow HDInsight Cluster

8/16/2017 • 17 min to read • [Edit Online](#)

This article walks you through the process of troubleshooting an HDInsight cluster that is either in the failed state, or running slowly. A 'Failed Cluster' is defined as one that has terminated with an error code. If your jobs are taking longer to run than expected, or you are seeing slow response times in general, you may be experiencing failures upstream from your cluster, such as the services on which the cluster runs. However, the most common cause of these slowdowns have to do with scale. When you provision a new HDInsight cluster, you have many options for selecting [virtual machine sizes](#) to preserve your metadata when you delete and recreate your cluster.

There are a set of general steps to take when diagnosing a failed or slow cluster. They involve getting information about all aspects of the environment, including, but not limited to, all associated Azure Services, cluster configuration, job execution information, and reproducability of error state. The most common steps taken in this process are listed below.

## General troubleshooting steps to diagnose an HDInsight cluster

- Step 1: Gather data about the issue
- Step 2: Validate the HDInsight cluster environment
- Step 3: View your cluster's health
- Step 4: Review the environment stack and versions
- Step 5: Examine the cluster log files
- Step 6: Check configuration settings
- Step 7: Reproduce the failure on a different Cluster

## Step 1: Gather data about the issue

HDInsight provides many tools that you can use to identify and troubleshoot issues like cluster failures and slow response times. The key is to know what these tools are and how to use them. The following steps guide you through these tools and some options for pinpointing the issue for resolution.

### Identify the problem

Take note of the problem to assist yourself and others during the troubleshooting process. It's easy to miss key details, so be as clear and concise as possible. Here are a few questions that can help you with this process:

- What did I expect to happen? What happened instead?
- How long did the process take to run? How long should it have run?
- Have my tasks always run slowly on this cluster? Did they run faster on a different cluster?
- When did this problem first occur? How often has it happened since?
- Has anything changed in my cluster configuration?

### Cluster details

Gather key information about your cluster, such as:

- Name of the cluster
- The cluster's region (you can check for [region outages](#)).
- The HDInsight cluster type and version.
- Type and number of HDInsight instances specified for head and worker nodes.

You can quickly get much of this top level information via the Azure portal. A sample screen is shown below:

## Essentials ^

Resource group (change)	Cluster type, HDI version
lynn-demo-2	Spark 2.1 on Linux (HDI 3.6)
Status	URL
Running	<a href="https://lynn-for-destruction.azurehdinsight.net">https://lynn-for-destruction.azurehdinsight.net</a>
Location	Learn more
East US	<a href="#">Documentation</a>
Subscription name (change)	Getting started
<a href="#">Microsoft Azure Sponsorship</a>	<a href="#">Quickstart</a>
Subscription ID	Head Nodes, Worker nodes
[REDACTED]	D12 v2 (x2), D4 v2 (x4)

Alternatively, you can use the Azure cli to get information about a HDInsight cluster by running the following commands:

```
azure hdinsight cluster list
azure hdinsight cluster show <Cluster Name>
```

Or, you can use PowerShell to view this type of information. See [Manage Hadoop clusters in HDInsight by using Azure PowerShell](#) for details.

## Step 2: Validate the HDInsight cluster environment

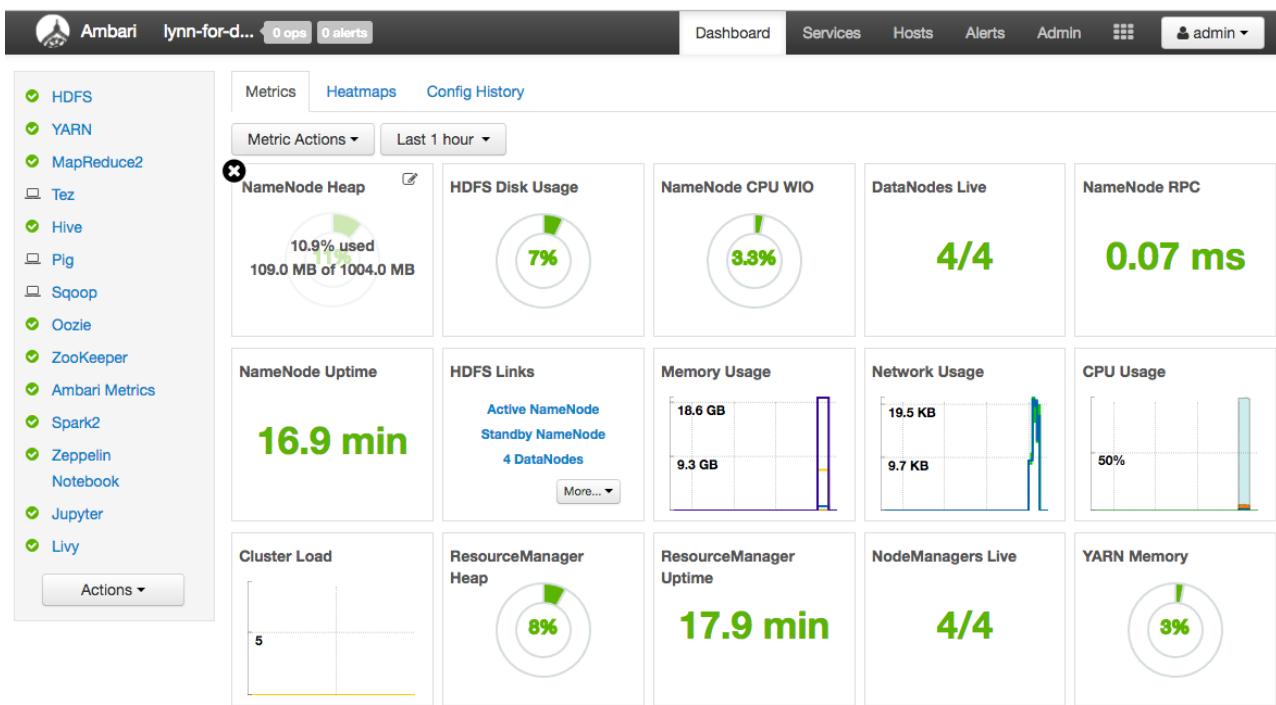
A typical HDInsight cluster uses a number of services and open-source software packages (such as Apache HBase, Apache Spark, etc...). In addition, it's common to find that a HDInsight cluster interoperates with other Azure services, such as Azure Virtual Networks and others. Failures in any of the running services on your cluster or any external services can result in a cluster failure. Additionally, a requested cluster service configuration change could also cause the cluster to fail.

### Service details

- Check the Open-source library Release Versions
- Check for [Azure Service Outages](#)
- Check for Azure Service Usage Limits
- Check the Azure Virtual Network Subnet Configuration

### Viewing cluster configuration settings with the Ambari UI

Apache Ambari simplifies the management and monitoring of a HDInsight cluster by providing an easy to use web UI and REST API. Ambari is included on Linux-based HDInsight clusters, and is used to monitor the cluster and make configuration changes. Click on the 'Cluster Dashboard' blade on the Azure Portal HDInsight page to open the 'Cluster Dashboards' link page. Next, click on the 'HDInsight cluster dashboard' blade to open the Ambari UI. You'll be prompted for your cluster login credentials. An example Ambari HDInsight Dashboard is shown below.



Also, you can click the blade named 'Ambari Views' on the Azure portal page for HDInsight to open a list of service views. This list will vary, depending on which libraries you've installed. For example, you may see YARN Queue Manager, Hive View and Tez View, if you've installed these services. Click any service link of interest to drill down to see configuration and service information.

#### Check for Azure service outages

HDInsight relies on several Azure services. It runs virtual servers on Azure HDInsight, stores data and scripts on Azure Blob storage or Azure Datalake Store, and indexes log files in Azure Table storage. Disruptions to these services, although rare, can cause issues in HDInsight. The first thing you should do when encountering unexpected slowdowns or failures in your cluster, is to check the [Azure Status Dashboard](#). The status of each service is listed by region. Be sure to check your cluster's region, as well as regions for any related services, for outages.

#### Check Azure service usage limits

If you are launching a large cluster, or have launched many clusters simultaneously, the cluster may have failed because you exceeded an Azure service limit. Service limits can vary based on your Azure subscription. Read more about [Azure subscription and service limits, quotas, and constraints](#). You can request that Microsoft increase the number of HDInsight resources available (such as VM cores and VM instances) by completing a [Resource Manager core quota increase request](#).

#### Check the release version

Compare the release label that you used to launch the cluster with the latest HDInsight release. Each release of HDInsight includes improvements such as new applications, features, patches, and bug fixes. The issue that is affecting your cluster may have already been fixed in the latest release version. If possible, re-run your cluster using the latest version of HDInsight and associated libraries (i.e. Apache HBase, Apache Spark, etc..).

#### Restart your cluster services

If you are experiencing slowdowns in your cluster, consider restarting your services through the Ambari interface or CLI. Your cluster may be experiencing transient errors, and oftentimes this is the fastest path to stabilizing your environment and improving performance.

## Step 3: View your cluster's health

HDInsight clusters are composed of different types of nodes running on Virtual Machine instances. Each of these nodes can be monitored for resource starvation, network connectivity issues, or other problems that can slowdown your cluster. Every cluster contains two head nodes, and most cluster types contains some combination of worker and edge nodes. For a description of the various nodes each cluster type uses, see [HDInsight Architecture](#). You will

want to check the health of each node, as well as the overall cluster health, using the following tools:

### Get a snapshot of the cluster health via the Ambari UI dashboard

As shown in the reference image in Step 2 (above), the Ambari UI dashboard (

`https://<clusternode>.azurehdinsight.net`) provides a bird's-eye view of your overall cluster health, such as uptime, memory, network, and CPU usage, HDFS disk usage, etc. You can use the Hosts section of Ambari to view resources at a host level to try and determine which may be causing issues or slowdowns. From here, you also have the ability to stop and restart services.

### Check your WebHCat service

One common scenario for Hive, Pig, or Scoop jobs failing is due to a failure with the **WebHCat** (sometimes referred to as Templeton) service. WebHCat is a REST interface for remote jobs (Hive, Pig, Scoop, MapReduce) execution. WebHCat translates the job submission requests into YARN applications and reports the status based on the YARN application status. WebHCat results come from YARN, and troubleshooting some of them requires going to YARN.

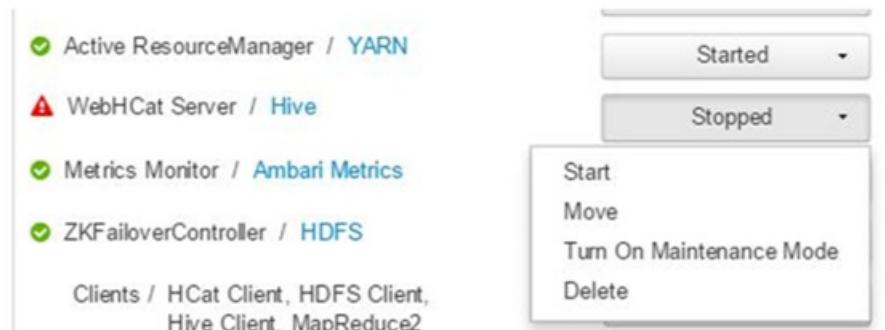
WebHCat may respond with the following HTTP status codes:

#### BadGateway (502 status code)

This is a generic message from Gateway nodes, and is the most common status code you may see. One possible cause for this is due to the WebHCat service being down on the active head node. This can be quickly verified by running the following CURL command:

```
$ curl -u admin:{HTTP PASSWD} https://{CLUSTERNAME}.azurehdinsight.net/templeton/v1/status?user.name=admin
```

Ambari will also display an alert showing the hosts on which the WebHCat service is down. You can attempt to bring the service back up by restarting the service on the host for which the alert was raised:



If WebHCat server still does not come up, then clicking through operations will show the failures. For more detailed information, refer to the `stderr` and `stdout` files referenced on the node.

#### WebHCat times out

The HDInsight Gateway times out responses which take longer than 2 minutes, resulting in `502 BadGateway`. WebHCat queries YARN services for job status, and if they take longer than two minutes, the request might timeout.

When this happens, review the following logs for further investigation:

```
/var/log/webhcatt
```

You will typically find the following in this directory:

- **webhcatt.log** is the log4j log to which server writes logs
- **webhcatt-console.log** is stdou of the server when started
- **webhcatt-console-error.log** is stderr of the server process

NOTE: webhcat.log will roll-over daily, generating files like `webhcat.log.YYYY-MM-DD`. Select the appropriate file, given the time range you are investigating.

Here are some possible causes for timeouts:

#### WebHCat Level Timeout

When WebHCat is under load, meaning there are more than 10 open sockets at any given time, it will take longer to establish new socket connections, which might result in a time out. A quick way to validate is to check the connection status using the below command on the current active headnode:

```
$ netstat | grep 30111
```

30111 is the port WebHCat listens to, and the above command lists network connections to and from WebHCat. This command will show you the current open sockets on port 30111. The number of open sockets should be less than 10.

Sometimes when debugging using the above command, you will receive no result. That doesn't mean that nothing is listening on port 30111. Because the command `netstat` only prints out open sockets. No result simply means for that given time, there are no open sockets. To check if Templeton is up and listening on port 30111, use:

```
$ netstat -l | grep 30111
```

#### YARN level timeout

Because Templeton is calling YARN to run jobs, the communication between Templeton and YARN is another source that can cause a timeout.

At the YARN level, there are two types of timeouts:

1. Submitting a YARN job might take long enough to cause a timeout.

If you open the **webhcat.log** file mentioned earlier, and search for "queued job", you may see multiple entries where the execution time is excessively long (>2000 ms), with each entry showing even longer wait times.

The reason the time for the queued jobs continues to increase, is the rate at which new jobs get submitted is much higher than the rate at which the old jobs are completed. Because of this, once the Yarn Memory is 100% used, there is no way the `joblauncher queue` can borrow capacity from the `default queue`. Thus, no more new jobs can be accepted (meaning being added to the joblauncher queue). This would cause the waiting time to become longer and longer, leading up to a `timeout` error, usually followed by many others.

As an illustration, the following shows the joblauncher queue at 714.4% over used. This is okay as long as there is still free capacity in the default queue, meaning new jobs can still be added by borrowing capacity from the default queue. But, if the YARN memory is at 100% capacity, meaning the cluster is fully used, new jobs must wait, eventually causing timeouts as described above.



There are two ways to resolve this issue: one is to reduce the speed of new jobs being submitted, the second one is to increase the consumption speed of old jobs in joblauncher queue, which is basically increasing the processing power of your cluster by scaling up.

2. YARN processing might take a long time, which makes another source of timeouts.

- List all jobs: This is a very expensive call. This call enumerates the applications from YARN

ResourceManager and for each completed application, gets the status from JobHistoryServer. In cases of higher numbers of jobs, this call might timeout, resulting in a 502.

- List jobs older than 7 days: The HDInsight YARN JobHistoryServer is configured (`mapreduce.jobhistory.max-age-ms`) to retain completed job information for 7 days. Trying to enumerating purged jobs results in a timeout, causing a 502.

Your process will involve the following:

- Figure out the UTC time range to troubleshoot
- Select the **webhcat.log** file, based on the time range
- Look for WARN/ERROR messages during that period of time

#### Other failures

##### 1. HTTP Status code 500

In most cases where WebHCat returns 500, the error message contains details on the failure. Otherwise, looking through the **webhcat.log** for WARN/ERROR messages will reveal the issue.

##### 2. Job failures

There may be cases where interactions with WebHCat are successful, but the jobs are failing.

Templeton collects the job console output as `stderr` in "statusdir" which will most often be useful for troubleshooting. `stderr` contains the YARN application id of the actual query, which can be used for troubleshooting.

## Step 4: Review the environment stack and versions

The Ambari UI 'Stack and Version' page provides information about the cluster services configuration and service version history. Incorrect Hadoop service library versions can be a cause of cluster failure. In the Ambari UI, click on the 'Admin' menu and then on 'Stacks and Versions' to navigate to this section. Then click on the 'Versions' tab on the page to see service version information. An example is shown below.

Stack and Versions			
Stack			
Service	Version	Status	Description
HDFS	2.7.3	Installed	Apache Hadoop Distributed File System
YARN	2.7.3	Installed	Apache Hadoop NextGen MapReduce (YARN)
MapReduce2	2.7.3	Installed	Apache Hadoop NextGen MapReduce (YARN)
Tez	0.7.0	Installed	Tez is the next generation Hadoop Query Processing framework written on top of YARN.
Hive	1.2.1000	Installed	Data warehouse system for ad-hoc queries & analysis of large datasets and table & storage management service
Pig	0.16.0	Installed	Scripting platform for analyzing large datasets
Sqoop	1.4.6	Installed	Tool for transferring bulk data between Apache Hadoop and structured data stores such as relational databases
Oozie	4.2.0	Installed	System for workflow coordination and execution of Apache Hadoop jobs. This also includes the installation of the optional Oozie Web Console which relies on and will install the ExtJS Library.
ZooKeeper	3.4.6	Installed	Centralized service which provides highly reliable distributed coordination

## Step 5: Examine the log files

We have demonstrated examining log files in the WebHCat section of Step 3 above. There are several other useful, and oftentimes verbose, log files you can investigate to narrow down issues with your cluster. There are many types of logs that are generated, given the large number of services and components that comprise a Hadoop

cluster. We will go over where you can locate these various logs in this section.

An example log is shown below.

As explained earlier, HDInsight clusters consist of several nodes, most of which are tasked to run submitted jobs. Jobs will run concurrently, but log files can only display results linearly. HDInsight executes new tasks, terminating others that fail to complete first. This activity is logged to `stderr` and `syslog` log files as they occur.

Start by checking the Script Action logs for errors or unexpected configuration changes during your cluster's provisioning process. The next set of logs to check are the step logs to identify Hadoop jobs launched as part of a step with errors.

The following sections cover each of the log files you can use to troubleshoot cluster errors and slowdowns:

#### **Check the Script Action logs**

HDInsight [Script Actions](#) run scripts on the cluster manually or when specified. For example, they can be used to install additional software on the cluster or to alter configuration settings from the default values. Checking these logs may provide insight into errors that occurred during set up of the cluster as well as configuration settings changes that could affect availability. You can view the status of a script action by clicking on the 'ops' button on your Ambari UI or by accessing them from the default storage account.

The storage logs are available at

\STORAGE\_ACCOUNT\_NAME\DEFAULT\_CONTAINER\_NAME\custom-scriptaction-logs\CLUSTER\_NAME\DATE .

## **View logs in HDInsight via Quick Links in Ambari**

The HDInsight Ambari UI includes a number of 'Quick Links' sections. To access the log links for a particular service in your HDInsight cluster, open the Ambari UI for your cluster, then click on the service link from the list at left, next click on the 'Quick Links' drop down and then on the HDInsight node of interest and then on the link for its associated log.

An example, for HDFS logs, is shown below:

The screenshot shows the Ambari interface for an HDInsight cluster named 'lynn-for-d...'. The 'HDFS' service is selected. In the 'Summary' tab, under 'Active NameNode', it shows 'hn0-lynn-f.tho2q1ndp2ue5mgjbp3fuqeaa.bx.internal.cloudapp.net (Active)' with status 'Started' and 'No alerts'. Below it, 'hn1-lynn-f.tho2q1ndp2ue5mgjbp3fuqeaa.bx.internal.cloudapp.net (Standby)' is also listed with 'Started' and 'No alerts'. Other components like 'ZKFailoverController' and 'Standby NameNode' are also shown as started with no alerts. The 'DataNodes' section shows 4/4 nodes started. On the right, a 'Service Actions' dropdown menu is open, listing options like 'NameNode JMX', 'NameNode Logs', 'Thread Stacks', and 'NameNode UI'.

#### **HDInsight logs written to Azure Tables**

The logs written to Azure Tables provide one level of insight into what is happening with an HDInsight cluster. When you create an HDInsight cluster, 6 tables are automatically created for Linux-based clusters in the default Table storage:

- hdinsightagentlog
- syslog
- daemonlog
- hadoopservicelog
- ambariserverlog
- ambariagentlog

#### **HDInsight logs written to Azure Blob Storage**

HDInsight clusters are configured to write task logs to an Azure Blob Storage account for any job that is submitted using the Azure PowerShell cmdlets or the .NET Job Submission APIs. If you submit jobs through RDP/command-line access to the cluster then the execution logging information will be found in the Azure Tables discussed in the previous paragraph.

#### **HDInsight logs generated by YARN**

YARN aggregates logs across all containers on a worker node and stores them as one aggregated log file per worker node. The log is stored on the default file system after an application finishes. Your application may use hundreds or thousands of containers, but logs for ALL containers run on a single worker node are always aggregated to a single file. So there is only one log per worker node used by your application. Log Aggregation is enabled by default on HDInsight clusters version 3.0 and above. Aggregated logs are located in default storage for the cluster. The following path is the HDFS path to the logs:

```
/app-logs/<user>/logs/<applicationId>
```

The aggregated logs are not directly readable, as they are written in a TFile, binary format indexed by container. Use the YARN ResourceManager logs or CLI tools to view these logs as plain text for applications or containers of interest.

#### **YARN CLI tools**

To use the YARN CLI tools, you must first connect to the HDInsight cluster using SSH. Specify the , , , and information when running these commands. You can view these logs as plain text by running one of the following commands:

```
yarn logs -applicationId <applicationId> -appOwner <user-who-started-the-application>
yarn logs -applicationId <applicationId> -appOwner <user-who-started-the-application> -containerId
<containerId> -nodeAddress <worker-node-address>
```

#### YARN ResourceManager UI

The YARN ResourceManager UI runs on the cluster headnode. It is accessed through the Ambari web UI. Use the following steps to view the YARN logs: In your web browser, navigate to <https://CLUSTERNAMESPACE.azurehdinsight.net>. Replace CLUSTERNAMESPACE with the name of your HDInsight cluster. From the list of services on the left, select YARN. Yarn service selected From the Quick Links dropdown, select one of the cluster head nodes and then select ResourceManager Log. Yarn quick links You are presented with a list of links to YARN logs.

#### Other logs

Heap dumps contain a snapshot of the application's memory, including the values of variables at the time the dump was created. So they are useful for diagnosing problems that occur at run-time. See the link at the bottom of this article for the process to enable heap dumps for your HDInsight cluster.

## Step 6: Check configuration settings

HDInsight clusters come preconfigured with default settings for related services, such as Hadoop, Hive, HBase, etc. Depending on your cluster's hardware configuration, number of nodes, types of jobs you are running, the data you are working with (and how that data is being processed), as well as the type of cluster, you may need to optimize your configuration.

Read [Changing Configs via Ambari](#) for detailed instructions on optimizing performance configurations for most scenarios. When using Spark, refer to [Optimizing and configuring Spark Jobs for Performance](#).

## Step 7: Reproduce the Failure on a different cluster

A useful technique when you are trying to track down the source of an error is to restart a new cluster with the same configuration and then to submit the job steps (one-by-one) that caused the original cluster to fail. In this way, you can check the results of each step before processing the next one. This method gives you the opportunity to correct and re-run a single step that has failed. This also has the advantage that you only load your input data once which can save time in the troubleshooting process.

To test a cluster step-by-step:

1. Launch a new cluster, with the same configuration as the failed cluster.
2. Submit the first job step to the cluster.
3. When the step completes processing, check for errors in the step log files. The fastest way to locate these log files is by connecting to the master node and viewing the log files there. The step log files do not appear until the step runs for some time, finishes, or fails.
4. If the step succeeded without error, run the next step. If there were errors, investigate the error in the log files. If it was an error in your code, make the correction and re-run the step. Continue until all steps run without error.
5. When you are done debugging the test cluster, delete it.

## Conclusion

There are a number of considerations you need to pay attention to make sure your HDInsight cluster is operational. You should focus on using the best HDInsight cluster configuration for your particular workload. Along with that, you'll need to monitor the execution of long-running and/or high resource consuming job executions to make sure that they don't fail and possibly bring down your entire cluster. It's also critically important to manage your cluster configuration over time, so that you can revert to working state should the need arise.

## See also

- [Manage HDInsight clusters by using the Ambari Web UI](#)
- [Analyze HDInsight Logs](#)
- [Access YARN application log on Linux-based HDInsight](#)
- [Enable heap dumps for Hadoop services on Linux-based HDInsight](#)
- [Known Issues for Apache Spark cluster on HDInsight](#)

# Analyze HDInsight logs

8/16/2017 • 12 min to read • [Edit Online](#)

Each Hadoop cluster in Azure HDInsight has an Azure storage account used as the default file system. The storage account is referred as the default Storage account. Cluster uses the Azure Table storage and the Blob storage on the default Storage account to store its logs. To find out the default storage account for your cluster, see [Manage Hadoop clusters in HDInsight](#). The logs retain in the Storage account even after the cluster is deleted.

## Logs written to Azure Tables

The logs written to Azure Tables provide one level of insight into what is happening with an HDInsight cluster.

When you create an HDInsight cluster, 6 tables are automatically created for Linux-based clusters in the default Table storage:

- hdinsightagentlog
- syslog
- daemonlog
- hadoopservicelog
- ambariserverlog
- ambariagentlog

The table file names are **uDDMonYYYYatHHMMSSsss**.

These tables contains the following fields:

- ClusterDnsName
- ComponentName
- EventTimestamp
- Host
- MALoggingHash
- Message
- N
- PreciseTimeStamp
- Role
- RowIndex
- Tenant
- TIMESTAMP
- TraceLevel

### Tools for accessing the logs

There are many tools available for accessing data in these tables:

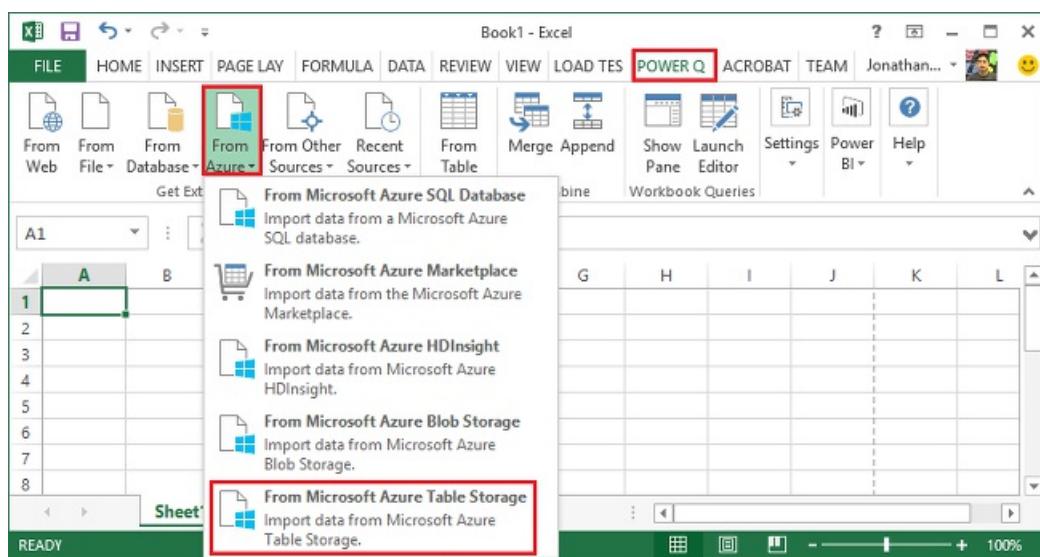
- Visual Studio
- Azure Storage Explorer
- Power Query for Excel

#### Use Power Query for Excel

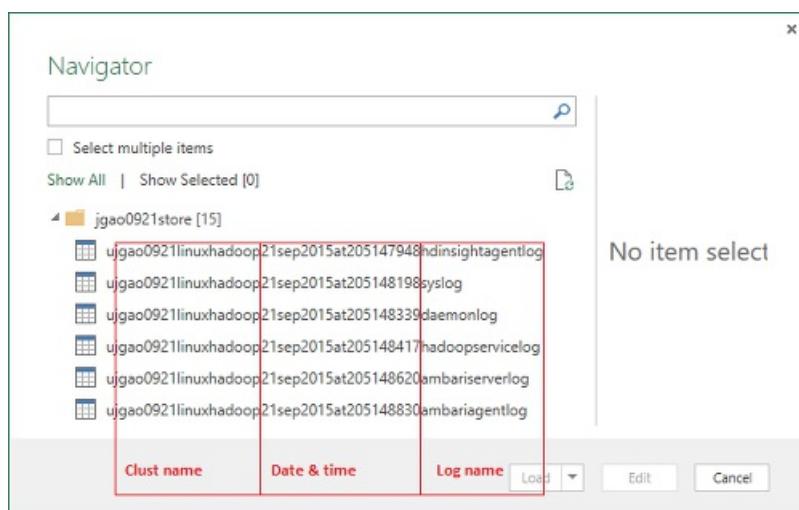
Power Query can be installed from [www.microsoft.com/en-us/download/details.aspx?id=39379](http://www.microsoft.com/en-us/download/details.aspx?id=39379). See the download page for the system requirements

## To use Power Query to open and analyze the service log

1. Open Microsoft Excel.
2. From the **Power Query** menu, click **From Azure**, and then click **From Microsoft Azure Table storage**.



3. Enter the storage account name. This can be either the short name or the FQDN.
4. Enter the storage account key. You shall see a list of tables:



5. Right-click the hadoopservicelog table in the **Navigator** pane and select **Edit**. You shall see 4 columns. Optionally, delete the **Partition Key**, **Row Key**, and **Timestamp** columns by selecting them, then clicking **Remove Columns** from the options in the ribbon.
6. Click the expand icon on the Content column to choose the columns you want to import into the Excel spreadsheet. For this demonstration, I chose TraceLevel, and ComponentName: It can give me some basic information on which components had issues.

1 COLUMN, 999+ ROWS

7. Click **OK** to import the data.
8. Select the **TraceLevel**, **Role**, and **ComponentName** columns, and then click **Group By** control in the ribbon.
9. Click **OK** in the Group By dialog box
10. Click\*\* Apply & Close\*\*.

You can now use Excel to filter and sort as necessary. Obviously, you may want to include other columns (e.g. **Message**) in order to drill down into issues when they occur, but selecting and grouping the columns described above provides a decent picture of what is happening with Hadoop services. The same idea can be applied to the **setuplog** and **hadoopinstalllog** tables.

#### Use Visual Studio

#### To use Visual Studio

1. Open Visual Studio.
2. From the **View** menu, click **Cloud Explorer**. Or simply click **CTRL+\, CTRL+X**.
3. From **Cloud Explorer**, select **Resource Types**. The other available option is **Resource Groups**.
4. Expand **Storage Accounts**, the default storage account for your cluster, and then **Tables**.
5. Double-click **hadoopservicelog**.
6. Add a filter. For example:

```
TraceLevel eq 'ERROR'
```

	PreciseTimeStamp	Role	RowIndex	TIMESTAMP	Tenant	TraceLevel
00000...	9/21/2015 9:05:...	headnode	03964402432233...	9/21/2015 9:05:...	f81f091b6d1644...	WARN
00000...	9/21/2015 9:05:...	headnode	03964402432233...	9/21/2015 9:05:...	f81f091b6d1644...	WARN
00000...	9/21/2015 9:05:...	headnode	03964402432233...	9/21/2015 9:05:...	f81f091b6d1644...	WARN
00000...	9/21/2015 9:05:...	headnode	03964402432233...	9/21/2015 9:05:...	f81f091b6d1644...	ERROR
00000...	9/21/2015 9:05:...	headnode	03964402432233...	9/21/2015 9:05:...	f81f091b6d1644...	ERROR

For more information about constructing filters, see [Construct Filter Strings for the Table Designer](#).

## Logs Written to Azure Blob Storage

The logs written to Azure Tables provide one level of insight into what is happening with an HDInsight cluster. However, these tables do not provide task-level logs, which can be helpful in drilling further into issues when they occur. To provide this next level of detail, HDInsight clusters are configured to write task logs to your Blob Storage account for any job that is submitted through Templeton. Practically, this means jobs submitted using the Microsoft Azure PowerShell cmdlets or the .NET Job Submission APIs, not jobs submitted through RDP/command-line access to the cluster.

To view the logs, see [Access YARN application logs on Linux-based HDInsight](#).

For more information about application logs, see [Simplifying user-logs management and access in YARN](#).

## View cluster health and job logs

### Access the Ambari UI

From the Azure portal, click an HDInsight cluster name to open the cluster blade. From the cluster blade, click **Dashboard**.



### Access the Yarn UI

From the Azure portal, click an HDInsight cluster name to open the cluster blade. From the cluster blade, click **Dashboard**. When prompted, enter the cluster administrator credentials. In Ambari, select **YARN** from the list of services on the left. On the page that appears, select **Quick Links**, then select the Active head node entry and Resource Manager UI.

You can use the YARN UI to do the following:

- **Get cluster status.** From the left pane, expand **Cluster**, and click **About**. This present cluster status details like total allocated memory, cores used, state of the cluster resource manager, cluster version etc.

A screenshot of the Hadoop YARN 'About the Cluster' page. At the top left is the 'hadoop' logo. The main title is 'About the Cluster'. On the left is a sidebar with links: 'Cluster' (selected), 'About', 'Nodes', 'Applications' (highlighted with a red box), 'Schedulers', and 'Tools'. The main content area has two tabs: 'Cluster Metrics' (selected) and 'Logs'. The 'Cluster Metrics' tab shows a table with columns: Apps Submitted, Apps Pending, Apps Running, Apps Completed, Containers Running, Memory Used, Memory Total, Memory Reserved, Vcores Used, Vcores Total, Vcores Reserved, and Active Nodes. The table has one row with values: 0, 0, 0, 0, 0, 0 B, 48 GB, 0 B, 0, 16, 0, 2. Below this is a detailed section with the following information:

Cluster ID:	1442954055645
ResourceManager state:	STARTED
ResourceManager HA state:	active
ResourceManager RMStateStore:	org.apache.hadoop.yarn.server.resourcemanager.recovery.NullRMStateStore
ResourceManager started on:	Tue Sep 22 20:34:15 +0000 2015
ResourceManager version:	2.6.0.2.2.7.1-0004 from 4808dd46ab5198a742112dad272e4ee44b791ab9 by jenkins source check 2015-07-15T23:59Z
Hadoop version:	2.6.0.2.2.7.1-0004 from 4808dd46ab5198a742112dad272e4ee44b791ab9 by jenkins source check 07-15T23:54Z

- **Get node status.** From the left pane, expand **Cluster**, and click **Nodes**. This lists all the nodes in the cluster, HTTP address of each node, resources allocated to each node, etc.
- **Monitor job status.** From the left pane, expand **Cluster**, and then click **Applications** to list all the jobs in the cluster. If you want to look at jobs in a specific state (such as new, submitted, running, etc.), click the appropriate link under **Applications**. You can further click the job name to find out more about the job such including the output, logs, etc.

## Access the HBase UI

From the Azure portal, click an HDInsight HBase cluster name to open the cluster blade. From the cluster blade, click **Dashboard**. When prompted, enter the cluster administrator credentials. In Ambari, select HBase from the list of services. Select **Quick links** on the top of the page, point to the active Zookeeper node link, and then click HBase Master UI.

## HDInsight error codes

The error messages itemized in this section are provided to help the users of Hadoop in Azure HDInsight understand possible error conditions that they can encounter when administering the service using Azure PowerShell and to advise them on the steps which can be taken to recover from the error.

Some of these error messages could also be seen in the Azure portal when it is used to manage HDInsight clusters. But other error messages you might encounter there are less granular due to the constraints on the remedial actions possible in this context. Other error messages are provided in the contexts where the mitigation is obvious.

### AtleastOneSqlMetastoreMustBeProvided

- **Description:** Please provide Azure SQL database details for at least one component in order to use custom settings for Hive and Oozie metastores.
- **Mitigation:** The user needs to supply a valid SQL Azure metastore and retry the request.

### AzureRegionNotSupported

- **Description:** Could not create cluster in region *nameOfYourRegion*. Use a valid HDInsight region and retry request.
- **Mitigation:** Customer should create the cluster region that currently supports them: Southeast Asia, West Europe, North Europe, East US, or West US.

### ClusterContainerRecordNotFound

- **Description:** The server could not find the requested cluster record.
- **Mitigation:** Retry the operation.

### ClusterDnsNameInvalidReservedWord

- **Description:** Cluster DNS name *yourDnsName* is invalid. Please ensure name starts and ends with alphanumeric and can only contain '-' special character
- **Mitigation:** Make sure that you have used a valid DNS name for your cluster that starts and ends with alphanumeric and contains no special characters other than the dash '-' and then retry the operation.

### ClusterNameUnavailable

- **Description:** Cluster name *yourClusterName* is unavailable. Please pick another name.
- **Mitigation:** The user should specify a clustername that is unique and does not exist and retry. If the user is using the Portal, the UI will notify them if a cluster name is already being used during the create steps.

### ClusterPasswordInvalid

- **Description:** Cluster password is invalid. Password must be at least 10 characters long and must contain at least one number, uppercase letter, lowercase letter and special character with no spaces and should not contain the username as part of it.
- **Mitigation:** Provide a valid cluster password and retry the operation.

### ClusterUserNameInvalid

- **Description:** Cluster username is invalid. Please ensure username doesn't contain special characters or spaces.
- **Mitigation:** Provide a valid cluster username and retry the operation.

### ClusterUserNameInvalidReservedWord

- **Description:** Cluster DNS name *yourDnsClusterName* is invalid. Please ensure name starts and ends with

alphanumeric and can only contain '-' special character

- **Mitigation:** Provide a valid DNS cluster username and retry the operation.

#### **ContainerNameMisMatchWithDnsName**

- **Description:** Container name in URI *yourcontainerURI* and DNS name *yourDnsName* in request body must be the same.
- **Mitigation:** Make sure that your container Name and your DNS name are the same and retry the operation.

#### **DataNodeDefinitionNotFound**

- **Description:** Invalid cluster configuration. Unable to find any data node definitions in node size.
- **Mitigation:** Retry the operation.

#### **DeploymentDeletionFailure**

- **Description:** Deletion of deployment failed for the Cluster
- **Mitigation:** Retry the delete operation.

#### **DnsMappingNotFound**

- **Description:** Service configuration error. Required DNS mapping information not found.
- **Mitigation:** Delete cluster and create a new cluster.

#### **DuplicateClusterContainerRequest**

- **Description:** Duplicate cluster container creation attempt. Record exists for *nameOfYourContainer* but Etags do not match.
- **Mitigation:** Provide a unique name for the container and retry the create operation.

#### **DuplicateClusterInHostedService**

- **Description:** Hosted service *nameOfYourHostedService* already contains a cluster. A hosted service cannot contain multiple clusters
- **Mitigation:** Host the cluster in another hosted service.

#### **FailureToUpdateDeploymentStatus**

- **Description:** The server could not update the state of the cluster deployment.
- **Mitigation:** Retry the operation. If this happens multiple times, contact CSS.

#### **HdiRestoreClusterAltered**

- **Description:** Cluster *yourClusterName* was deleted as part of maintenance. Please recreate the cluster.
- **Mitigation:** Recreate the cluster.

#### **HeadNodeConfigNotFound**

- **Description:** Invalid cluster configuration. Required head node configuration not found in node sizes.
- **Mitigation:** Retry the operation.

#### **HostedServiceCreationFailure**

- **Description:** Unable to create hosted service *nameOfYourHostedService*. Please retry request.
- **Mitigation:** Retry the request.

#### **HostedServiceHasProductionDeployment**

- **Description:** Hosted Service *nameOfYourHostedService* already has a production deployment. A hosted service cannot contain multiple production deployments. Retry the request with a different cluster name.
- **Mitigation:** Use a different cluster name and retry the request.

#### **HostedServiceNotFound**

- **Description:** Hosted Service *nameOfYourHostedService* for the cluster could not be found.

- **Mitigation:** If the cluster is in error state, delete it and then try again.

### **HostedServiceWithNoDeployment**

- **Description:** Hosted Service *nameOfYourHostedService* has no associated deployment.
- **Mitigation:** If the cluster is in error state, delete it and then try again.

### **InsufficientResourcesCores**

- **Description:** The SubscriptionId *yourSubscriptionId* does not have cores left to create cluster *yourClusterName*. Required: *resourcesRequired*, Available: *resourcesAvailable*.
- **Mitigation:** Free up resources in your subscription or increase the resources available to the subscription and try to create the cluster again.

### **InsufficientResourcesHostedServices**

- **Description:** Subscription ID *yourSubscriptionId* does not have quota for a new HostedService to create cluster *yourClusterName*.
- **Mitigation:** Free up resources in your subscription or increase the resources available to the subscription and try to create the cluster again.

### **InternalErrorRetryRequest**

- **Description:** The server encountered an internal error. Please retry request.
- **Mitigation:** Retry the request.

### **InvalidAzureStorageLocation**

- **Description:** Azure Storage location *dataRegionName* is not a valid location. Make sure the region is correct and retry request.
- **Mitigation:** Select a Storage location that supports HDInsight, check that your cluster is co-located and retry the operation.

### **InvalidNodeSizeForDataNode**

- **Description:** Invalid VM size for data nodes. Only 'Large VM' size is supported for all data nodes.
- **Mitigation:** Specify the supported node size for the data node and retry the operation.

### **InvalidNodeSizeForHeadNode**

- **Description:** Invalid VM size for head node. Only 'ExtraLarge VM' size is supported for head node.
- **Mitigation:** Specify the supported node size for the head node and retry the operation

### **InvalidRightsForDeploymentDeletion**

- **Description:** Subscription ID *yourSubscriptionId* being used does not have sufficient permissions to execute delete operation for cluster *yourClusterName*.
- **Mitigation:** If the cluster is in error state, drop it and then try again.

### **InvalidStorageAccountBlobContainerName**

- **Description:** External storage account blob container name *yourContainerName* is invalid. Make sure name starts with a letter and contains only lowercase letters, numbers and dash.
- **Mitigation:** Specify a valid storage account blob container name and retry the operation.

### **InvalidStorageAccountConfigurationSecretKey**

- **Description:** Configuration for external storage account *yourStorageAccountName* is required to have secret key details to be set.
- **Mitigation:** Specify a valid secret key for the storage account and retry the operation.

### **InvalidVersionHeaderFormat**

- **Description:** Version header *yourVersionHeader* is not in valid format of yyyy-mm-dd.

- **Mitigation:** Specify a valid format for the version-header and retry the request.

#### **MoreThanOneHeadNode**

- **Description:** Invalid cluster configuration. Found more than one head node configuration.
- **Mitigation:** Edit the configuration so that only one head node is specified.

#### **OperationTimedOutRetryRequest**

- **Description:** The operation could not be completed within the permitted time or the maximum retry attempts possible. Please retry request.
- **Mitigation:** Retry the request.

#### **ParameterNullOrEmpty**

- **Description:** Parameter *yourParameterName* cannot be null or empty.
- **Mitigation:** Specify a valid value for the parameter.

#### **PreClusterCreationValidationFailure**

- **Description:** One or more of the cluster creation request inputs is not valid. Make sure the input values are correct and retry request.
- **Mitigation:** Make sure the input values are correct and retry request.

#### **RegionCapabilityNotAvailable**

- **Description:** Region capability not available for region *yourRegionName* and Subscription ID *yourSubscriptionId*.
- **Mitigation:** Specify a region that supports HDInsight clusters. The publicly supported regions are: Southeast Asia, West Europe, North Europe, East US, or West US.

#### **StorageAccountNotColocated**

- **Description:** Storage account *yourStorageAccountName* is in region *currentRegionName*. It should be same as the cluster region *yourClusterRegionName*.
- **Mitigation:** Either specify a storage account in the same region that your cluster is in or if your data is already in the storage account, create a new cluster in the same region as the existing storage account. If you are using the Portal, the UI will notify them of this issue in advance.

#### **SubscriptionIdNotActive**

- **Description:** Given Subscription ID *yourSubscriptionId* is not active.
- **Mitigation:** Re-activate your subscription or get a new valid subscription.

#### **SubscriptionIdNotFound**

- **Description:** Subscription ID *yourSubscriptionId* could not be found.
- **Mitigation:** Check that your subscription ID is valid and retry the operation.

#### **UnableToResolveDNS**

- **Description:** Unable to resolve DNS *yourDnsUrl*. Please ensure the fully qualified URL for the blob endpoint is provided.
- **Mitigation:** Supply a valid blob URL. The URL MUST be fully valid, including starting with *http://* and ending in *.com*.

#### **UnableToVerifyLocationOfResource**

- **Description:** Unable to verify location of resource *yourDnsUrl*. Please ensure the fully qualified URL for the blob endpoint is provided.
- **Mitigation:** Supply a valid blob URL. The URL MUST be fully valid, including starting with *http://* and ending in *.com*.

#### **VersionCapabilityNotAvailable**

- **Description:** Version capability not available for version *specifiedVersion* and Subscription ID *yourSubscriptionId*.
- **Mitigation:** Choose a version that is available and retry the operation.

#### **VersionNotSupported**

- **Description:** Version *specifiedVersion* not supported.
- **Mitigation:** Choose a version that is supported and retry the operation.

#### **VersionNotSupportedInRegion**

- **Description:** Version *specifiedVersion* is not available in Azure region *specifiedRegion*.
- **Mitigation:** Choose a version that is supported in the region specified and retry the operation.

#### **WasbAccountConfigNotFound**

- **Description:** Invalid cluster configuration. Required WASB account configuration not found in external accounts.
- **Mitigation:** Verify that the account exists and is properly specified in configuration and retry the operation.

## Next steps

- [Use Ambari Views to debug Tez Jobs on HDInsight](#)
- [Enable heap dumps for Hadoop services on Linux-based HDInsight](#)
- [Manage HDInsight clusters by using the Ambari Web UI](#)

# Use Ambari Views to debug Tez Jobs on HDInsight

8/16/2017 • 3 min to read • [Edit Online](#)

The Ambari Web UI for HDInsight contains a Tez view that can be used to understand and debug jobs that use Tez. The Tez view allows you to visualize the job as a graph of connected items, drill into each item, and retrieve statistics and logging information.

## IMPORTANT

The steps in this document require an HDInsight cluster that uses Linux. Linux is the only operating system used on HDInsight version 3.4 or greater. For more information, see [HDInsight component versioning](#).

## Prerequisites

- A Linux-based HDInsight cluster. For steps on creating a cluster, see [Get started using Linux-based HDInsight](#).
- A modern web browser that supports HTML5.

## Understanding Tez

Tez is an extensible framework for data processing in Hadoop that provides greater speeds than traditional MapReduce processing. For Linux-based HDInsight clusters, it is the default engine for Hive.

Tez creates a Directed Acyclic Graph (DAG) that describes the order of actions required by jobs. Individual actions are called vertices, and execute a piece of the overall job. The actual execution of the work described by a vertex is called a task, and may be distributed across multiple nodes in the cluster.

### Understanding the Tez view

The Tez view provides both historical information and information on processes that are running. This information shows how a job is distributed across clusters. It also displays counters used by tasks and vertices, and error information related to the job. It may offer useful information in the following scenarios:

- Monitoring long-running processes, viewing the progress of map and reduce tasks.
- Analyzing historical data for successful or failed processes to learn how processing could be improved or why it failed.

## Generate a DAG

The Tez view only contains data if a job that uses the Tez engine is currently running, or has been ran previously. Simple Hive queries can be resolved without using Tez. More complex queries that do filtering, grouping, ordering, joins, etc. use the Tez engine.

Use the following steps to run a Hive query that uses Tez:

1. In a web browser, navigate to <https://CLUSTERNAME.azurehdinsight.net>, where **CLUSTERNAME** is the name of your HDInsight cluster.
2. From the menu at the top of the page, select the **Views** icon. This icon looks like a series of squares. In the dropdown that appears, select **Hive view**.

- When the Hive view loads, paste the following query into the Query Editor, and then click **execute**.

```
select market, state, country from hivesampletable where deviceplatform='Android' group by market,
country, state;
```

Once the job has completed, you should see the output displayed in the **Query Process Results** section. The results should be similar to the following text:

market	state	country
en-GB	Hessen	Germany
en-GB	Kingston	Jamaica

- Select the **Log** tab. You see information similar to the following text:

```
INFO : Session is already open
INFO :

INFO : Status: Running (Executing on YARN cluster with App id application_1454546500517_0063)
```

Save the **App id** value, as this value is used in the next section.

## Use the Tez View

- From the menu at the top of the page, select the **Views** icon. In the dropdown that appears, select **Tez view**.

- When the Tez view loads, you see a list of hive queries that are currently running, or have been ran on the cluster.

Query ID	User	Status	Query	DAG ID	Tables Read	Tables Written	App ID	Queue	Execution Mode
hive_20170712143133_e0abe8b6-3...	admin	<span style="color: blue;">RUNNING</span>	select market, state...	Not Available!	default.hivesampl...	None			
hive_20170712143026_5baaa2e2-5...	admin	<span style="color: green;">SUCCEEDED</span>	select market, state...	dag_1499865447023_0004_1	default.hivesampl...	None			

- If you have only one entry, it is for the query that you ran in the previous section. If you have multiple entries, you can search by using the fields at the top of the page.
- Select the **Query ID** for a Hive query. Information about the query is displayed.

The screenshot shows the Tez view interface for a specific query. At the top, there are tabs for 'Query Details', 'Timeline', and 'Configurations'. The 'Query Details' tab is selected, showing information such as Query ID (hive\_20170712143133\_e0abe8b6-310b-45df-b6e8-6345766f2bc5), User (admin), Status (SUCCEEDED), Start Time (12 Jul 2017 10:31:33), End Time (12 Jul 2017 10:31:55), Duration (22s 508ms), Application ID (application\_1499865447023\_0005), DAG ID (dag\_1499865447023\_0005\_1), Session ID (Not Available!), Thread Name (HiveServer2-Background-Pool: Thread-494), and Queue (default). To the right of this table is a 'Hive Details' section with tables for Read and Written operations, Client Address (10.0.0.19), Execution Mode (TEZ), Hive Address (10.0.0.19), and Client Type (HS2). Below these sections is a 'Query Text' area containing the SQL query: `1 select market, state, country from hivesamplable where deviceplatform='Android' group by market, country, state`.

5. The tabs on this page allow you to view the following information:

- **Query Details:** Details about the Hive query.
- **Timeline:** Information about how long each stage of processing took.
- **Configurations:** The configuration used for this query.

From **Query Details** you can use the links to find information about the **Application** or the **DAG** for this query.

- The **Application** link displays information about the YARN application for this query. From here you can access the YARN application logs.
- The **DAG** link displays information about the directed acyclic graph for this query. From here you can view a graphical representation of the DAG. You can also find information on the vertices within the DAG.

## Next Steps

Now that you have learned how to use the Tez view, learn more about [Using Hive on HDInsight](#).

For more detailed technical information on Tez, see the [Tez page at Hortonworks](#).

For more information on using Ambari with HDInsight, see [Manage HDInsight clusters using the Ambari Web UI](#)

# Common Problems FAQ

8/15/2017 • 4 min to read • [Edit Online](#)

This guide serves to address common frequently asked questions (FAQs) pertaining to common issues when using and provisioning HDInsight clusters.

## Capacity planning

There are several aspects about your chosen cluster type you must consider when [planning your cluster's capacity](#). Ultimately, you need to consider cost optimization, as well as the performance and usability of your cluster. Below are some common pitfalls when it comes to capacity planning.

### **How do I ensure my cluster can handle the rate of growth of my data set? We don't know what the upper limit will be.**

HDInsight separates compute from data storage by using either Azure Blob Storage or Azure Data Lake Store. This provides many benefits, including scaling out less expensive storage separate from compute. When you provision a new HDInsight cluster, you have the choice between Azure Storage and Azure Data Lake Store as your default data container. Azure Storage has certain [capacity limits](#) that you must consider, whereas Data Lake Store is virtually unlimited. However, since your cluster's default storage must be located in the same location as your cluster, you are limited by locations in which you can provision your cluster when using Data Lake Store. Data Lake Store is currently only available in three locations globally (Central US, East US 2 and North Europe).

Alternately, it is possible to use a combination of different storage accounts with an HDInsight cluster. You might want to use more than one storage account in the following circumstances:

- When the amount of data is likely to exceed the storage capacity of a single blob storage container.
- When the rate of access to the blob container might exceed the threshold where throttling will occur.
- When you want to make data you have already uploaded to a blob container available to the cluster.
- When you want to isolate different parts of the storage for reasons of security, or to simplify administration.

As a general rule, for a 48 node cluster, it is recommended that you have 4-8 storage accounts. This is not due to storage space requirements, per se, but due to the fact that each storage account provides additional networking bandwidth that opens up the pipe as wide as possible for the compute nodes to finish their jobs faster. When you use multiple storage accounts, make the naming convention of the storage account as random as possible, with no prefix. This is to reduce the chance of hitting storage bottlenecks or common mode failures in storage across all accounts at the same time. This type of storage partitioning in Azure Storage meant to avoid storage throttling. Lastly, make sure to only have one container per storage account. This yields better performance.

### **We've selected large VMs and enough nodes to support our batch processing needs, but cost of running the cluster is much higher than we'd like. How do we minimize our cost while meeting our compute needs?**

HDInsight gives you the flexibility to [scale your cluster](#) at will by adjusting the number of worker nodes at any time. A common practice is to scale out your cluster to meet peak load demands, then scale it back down when those extra nodes are no longer needed.

Another option, particularly if there are specific times during the day/week/month that you need your cluster up and running, is to [create on-demand clusters using Azure Data Factory](#). Since you are charged for your cluster for its lifetime, this is one effective way in which you can manage its lifecycle, netting significant cost savings. Since your data is stored on low-cost storage, independent of your cluster, then you don't need to worry about losing valuable data when you delete your cluster. An alternative to using Data Factory is to create PowerShell scripts to provision and delete your cluster, then schedule running those scripts with [Azure Automation](#).

One thing to look out for when deleting and re-creating your clusters, is that the Hive metastore that is created by default with a cluster is transient. When the cluster is deleted, the metastore gets deleted as well. [Use an external database like Azure Database or Oozie to persist the metastore](#) if your cluster lifecycle management process is to run it on-demand, deleting it when not needed.

**My code works fine when running locally, but tends to fail when deploying to a multi-node cluster. How do I isolate the problem to determine if the issue is with my multi-node cluster or something else?**

Sometimes errors can occur due to the parallel execution of multiple map and reduce components on a multi-node cluster. Consider emulating distributed testing by running multiple jobs on a single node cluster at the same time to detect errors, then expand this approach to run multiple jobs concurrently on clusters containing more than one node in order to help isolate the issue.

You can create a single-node HDInsight cluster in Azure by specifying the advanced option when creating the cluster. Alternatively, you can install a single-node development environment on your local computer and execute the solution there. A single-node local development environment for Hadoop-based solutions that is useful for initial development, proof of concept, and testing is available from Hortonworks. For more details, see [Hortonworks Sandbox](#).

By using a single-node local cluster you can rerun failed jobs and adjust the input data, or use smaller datasets, to help you isolate the problem. How you go about rerunning jobs depends on the type of application and on which platform it is running.

## Next steps

Check this FAQ periodically, as we add additional questions and answers to common problems.