Azure Data + Analytics
HDInsight
Lecture 12
Deep.Azure@McKesson

Zoran B. Djordjević

Data + Analytics

Azure has a large family of data and analytics services containing offerings like:

- HDInsight
  - Provisions Hadoop, Spark, R Server, HBase, and Storm clusters.
  - HDInsight is a Big Data service that uses 100% Apache Hadoop and other popular Big Data solutions to the cloud. A modern, cloud-based data platform that manages structured or unstructured data, and of any size.
  - HDInsight platform provides comparable simplicity, and manageable administrative interface. HDInsight provides a platform for various Big Data needs including Batch, Interactive, No SQL and Streaming.
  - HDINsight comes with a strong eco-system of tools and developer environment.
  - Supported cluster types include: Hadoop (Hive), HBase, Storm, Spark, Kafka, Interactive Hive (LLAP), and R Server (with R Studio, R 9.1).
- Machine Learning
  - Provides open and elastic AI development tools spanning the cloud and the edge. Azure Machine Learning provide access to tools, on a per-seat basis, that allow you to prepare data, engineer features, and author models in the compute environment of your choice with built-in analytics such as metrics and run history.
  - Azure ML offers the experimentation capabilities along with Model Management capabilities to complete your data science workflow from data preparation and model creation through model deployment and management in production.
- Stream Analytics
  - Real-time data stream processing from millions of IoT devices.
  - Azure Stream Analytics is a fully managed, cost effective real-time event processing engine that helps to unlock deep insights from data. Stream Analytics makes it easy to set up real-time analytic computations on data streaming from devices, sensors, web sites, social media, applications, infrastructure systems, and more.
  - You can author a Stream Analytics job specifying the input source of the streaming data, the output sink for the results of your job, and a data transformation expressed in a SQL-like language. You can monitor and adjust the scale/speed of your job to scale from a few kilobytes to a gigabyte or more of events processed per second

Data + Analytics

- Data Lake Analytics
  - Distributed analytics service. Analyze any kind of data of any size
  - Only pay for the processing power that you use
  - Introduce U-SQL: a language that combines declarative **SQL** with imperative C# to let **you** process data at any scale.
  - Data Lake Analytics service scales dynamically to match your needs
- Data Lake Store
  - Hyperscale repository for big data analytics workloads. Azure Data Lake Store is designed to be an enterprise-wide, hyper scale repository for big data analytic workloads.
  - Data Lake Store is Built atop of HDFS (Hadoop Distributed File System)
  - Unlimited storage: No fixed limits on file size, account size, or the number of files.
  - Performance Tuned for Big Data: Optimized for massive throughput to query and analyze any amount of data
  - Enterprise Grade Security: Azure Active Directory authentication and role-based access control
  - All Data: Store data in its native format without prior transformation

Data + Analytics

- Azure Analysis Services
  - Enterprise grade analytics engine as a service.
  - Azure Analysis Services is built on the proven analytical engine in Microsoft SQL Server Analysis Services, and delivers enterprise-grade BI semantic modeling capabilities with the scale, flexibility and management benefits of the cloud.
  - Azure Analysis Services helps you transform complex data into actionable insights. With this service one could model and shape virtually any data of any size.
  - Provide secured access, anytime from virtually anywhere
  - Visualize your data using your favorite data visualization tool
  - Get started quickly without the need to manage infrastructure
  - Scale resources to match your business needs
  - Govern, deploy, test and deliver your BI solution with confidence
- Event Hubs
  - Receive telemetry from millions of devices
- Data Factory
  - Orchestrate and manage data transformation and movement
- SQL Data Warehouse
  - Elastic data warehouse as a service with enterprise-class features

Data + Analytics

- Azure Bot Service
  - Intelligent serverles bot service that scales on demand. Azure Bot Service provides an integrated environment that's purpose-built for bot development with the Microsoft Bot Framework connectors and BotBuilder SDKs. Developers can get started with out-of-the-box templates for scenarios including basic, form, language understanding, question and answer, and proactive bots.
- Power BI Embedded
  - Power BI Embedded provides Power BI capabilities by helping you quickly add visuals, reports, and dashboards into your apps. Power BI can be used with any Microsoft Azure use services like Machine Learning and IoT.
- Data Catalog
  - Azure Data Catalog is a fully managed service that serves as a system of registration and system of discovery for enterprise data sources.
  - Data Catalog lets users, analysts, data scientists and develop, register, discover, understand, and consume data sources.
  - Data Catalog uses crowdsourced annotations and metadata to capture tribal knowledge within your organization, shine light on hidden data, and get more value from your enterprise data sources.

Data + Analytics

- Log Analytics
  - Collect, search and visualize machine data from on-premise and cloud.
  - Can collect and correlate data from multiple sources to get a unified dashboard view and gain insights to detect and mitigate IT issues.
  - Azure Log Analytics offers:
    - Advanced diagnostic and analytics capabilities
    - A refined search experience
    - Interactive and expressive query language with machine learning constructs
- Text Analytics API
  - A cloud-based service that provides advanced natural language processing over raw text, and includes three main functions: sentiment analysis, key phrase extraction, and language detection.
  - The API is backed by resources in Microsoft Cognitive Services, a collection of machine learning and AI algorithms in the cloud, consumable in your projects.
- Custom Speech Service
  - Customize the language model of the speech recognizer by tailoring it to the vocabulary of your application and the speaking style of your users.
  - Customize the acoustic model of the speech recognizer to better match the expected environment and user population of your application.
  - Deploy your models to create a speech recognition endpoint that's customized to your application.
  - Send requests to your custom endpoint using the Microsoft Cognitive Services speech client library. Overcome speech recognition barriers like speaking style, background noise,

HDInsight

In this lecture we will concentrate on HDInsight

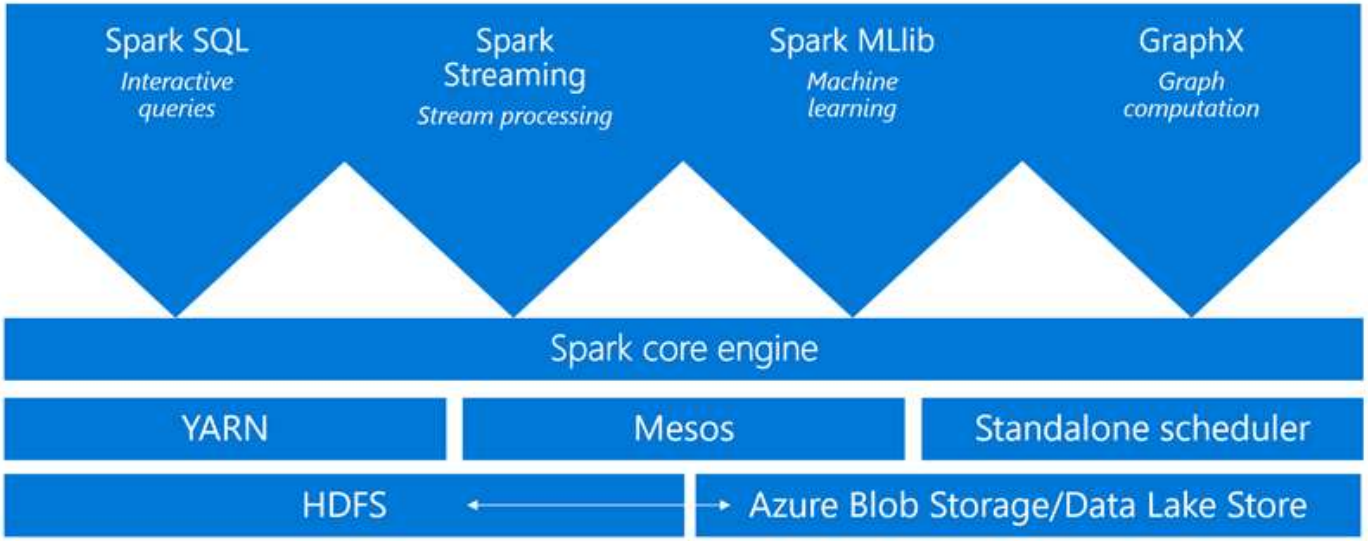
- Azure HDInsight is a fully-managed cloud service that makes it easy, fast, and cost-effective to process massive amounts of data.
- HDInsight provides popular open-source frameworks such as Hadoop, Spark, Hive, LLAP, Kafka, Storm, R & more.
- Azure HDInsight enables a broad range of scenarios such as ETL, Data Warehousing, Machine Learning, IoT and more.
- HDInsight provides optimized clusters for Hadoop, Spark, Hive, LLAP, Kafka, Storm, Hbase and R.
- Available in most Azure regions around the globe. Also available in Azure government clouds in the US, China, and Germany to meet sovereign compliance requirements.
- With HDInsight you can use your preferred productivity tools such as Visual Studio, Eclipse, and IntelliJ, Jupyter and Zeppelin and write code in familiar languages such as Scala, Python, R, Java and .NET.
- HDInsight is a cost-effective service that is powerful and reliable. You pay only for what you use.
- You create clusters on demand, then scale them up or down. Decoupled compute and storage provide better performance and flexibility.

Hadoop

- Apache Hadoop was the original open-source framework for distributed processing and analysis of big data sets on clusters.
- The Hadoop technology stack includes related software and utilities, including Apache Hive, HBase, Spark, Kafka, and many others.

Spark

- Apache Spark is an open-source parallel processing framework that supports in-memory processing to boost the performance of big-data analytic applications. Spark cluster on HDInsight is compatible with Azure Storage as well as Azure Data Lake Store. Hence, your existing data stored in Azure can easily be processed via a Spark cluster.
- When you create a Spark cluster on HDInsight, you create Azure compute resources with Spark installed and configured. The data to be processed is stored in Azure Storage or Azure Data Lake Store.

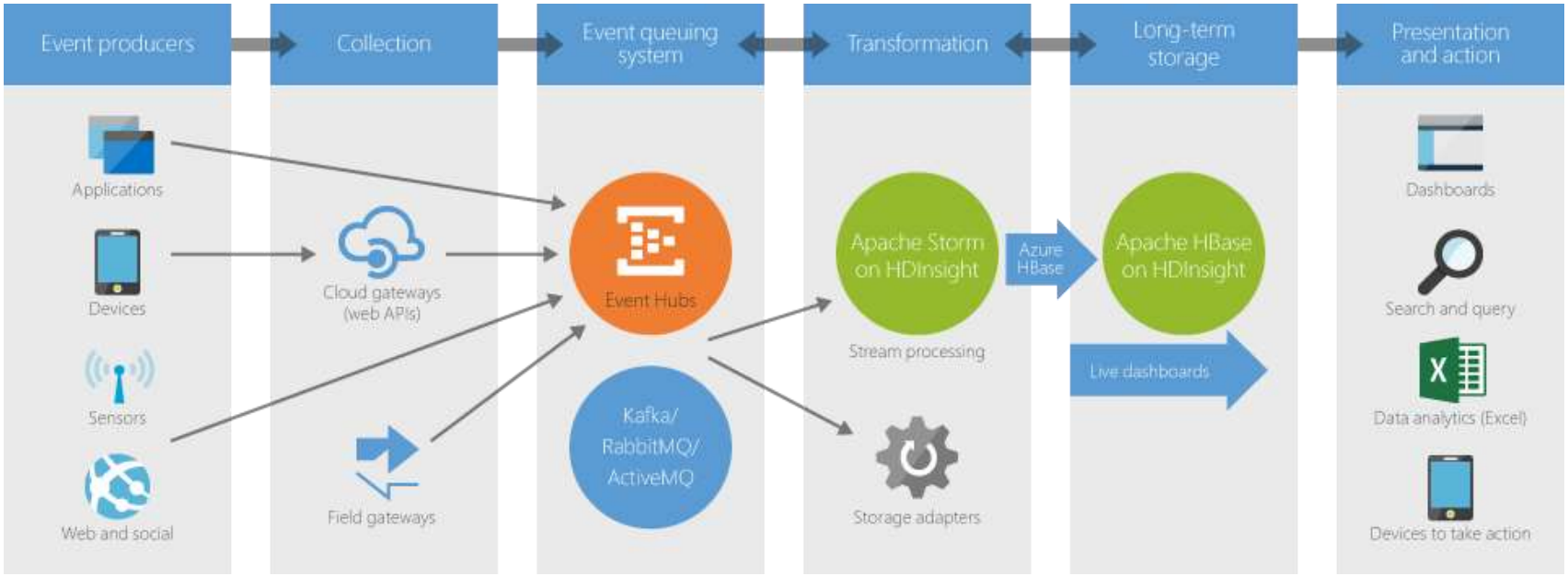


Hive

- Apache Hive is a data warehouse system for Hadoop. Hive enables data summarization, querying, and analysis of data. Hive queries are written in HiveQL, which is a query language similar to SQL.+
- Hive allows you to project structure on largely unstructured data. After you define the structure, you can use HiveQL to query the data without knowledge of Java or MapReduce.
- HDInsight provides several cluster types, which are tuned for specific workloads. The following cluster types are most often used for Hive queries:
  - **Interactive Query**: A Hadoop cluster that provides Low Latency Analytical Processing (LLAP) functionality to improve response times for interactive queries.
  - **Hadoop**: A Hadoop cluster that is tuned for batch processing workloads.
  - **Spark**: Apache Spark has built-in functionality for working with Hive. Spark can read from Hive data warehouse and write into Hive.
  - **HBase**: HiveQL can be used to query data stored in HBase. HBase is a fully transactional (ACID-ic) database built atop of Hadoop. Hbase might lack performance of some other NoSQL solutions, but, does provide full ACID-s features.

Storm

- Apache Storm is a distributed, fault-tolerant, open-source, real-time event processing solution for large, fast streams of data.
- First made famous by Twitter, which used the technology on its massive tweet streams, Storm is a project of The Apache Software Foundation.
- Azure makes Apache Storm easy and cost-effective to deploy, with no hardware to buy, no software to configure, your choice of development tools (Java or C#), and deep integration with Visual Studio.

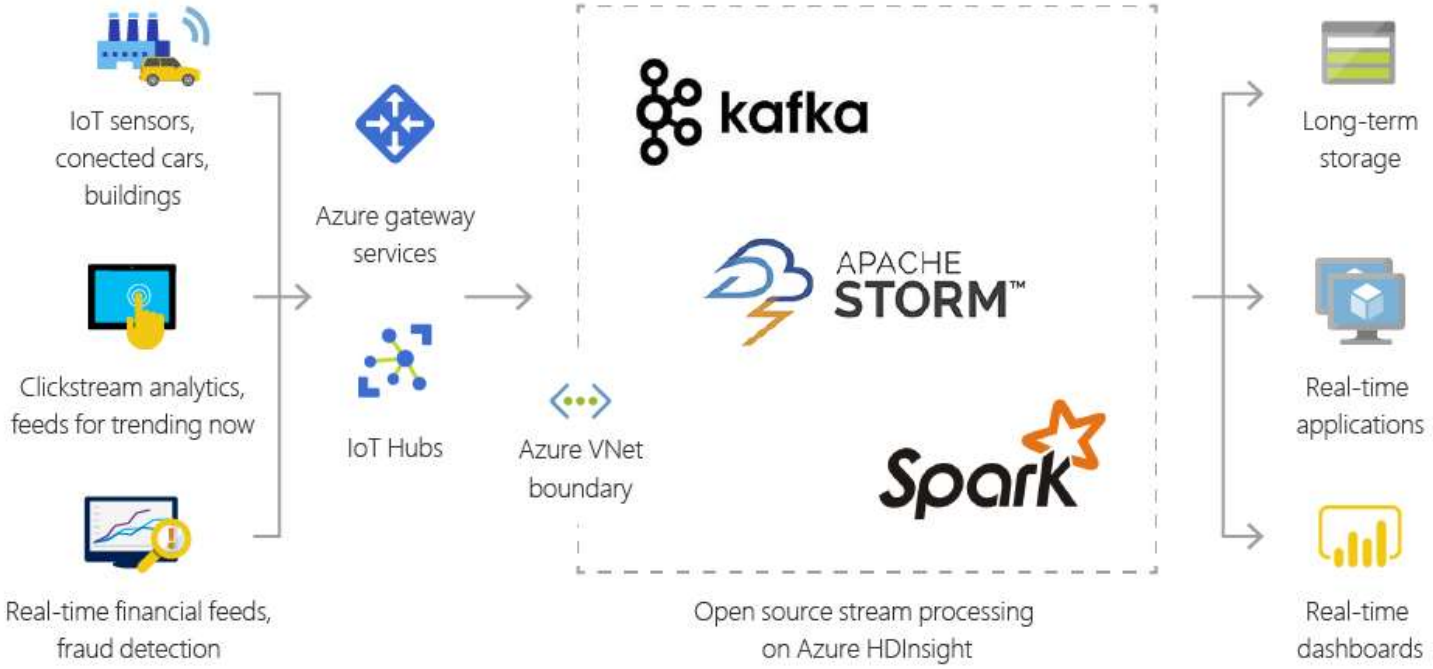


R

- R is a most popular language for statistical, machine learning and general data analysis.
- R on HDInsight combines enterprise-scale R analytics software with the power of Apache Hadoop and Apache Spark.
- Microsoft R Server for HDInsight gives you unlimited scalability and performance
- Multi-threaded math libraries and transparent parallelization in R Server handle up to 1000x more data and up to 50x faster speeds than open-source R, which helps you to train more accurate models for better predictions.
- R Server works with the open-source R language, so all of your R scripts run without changes.
- World is divided into two big tribes: people who do their data analysis in R and people who do their analysis in Python.
- There are some people who do not have access to the Internet and they do their data analysis in Java, C#, C++, Perl, VB and other languages.

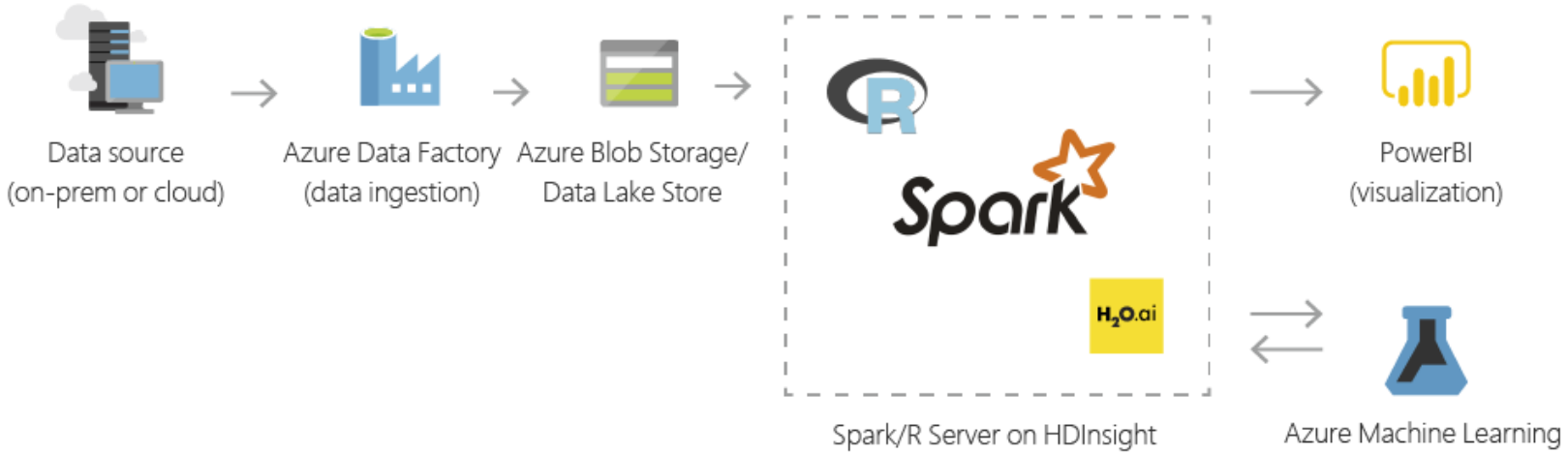
HDInsight, Use case: IoT

- Toyota's Connected Car, Office 365, Bing Ads, are processing millions of events/sec for real time big data processing on HDInsight through Kafka, Storm and Spark Streaming.



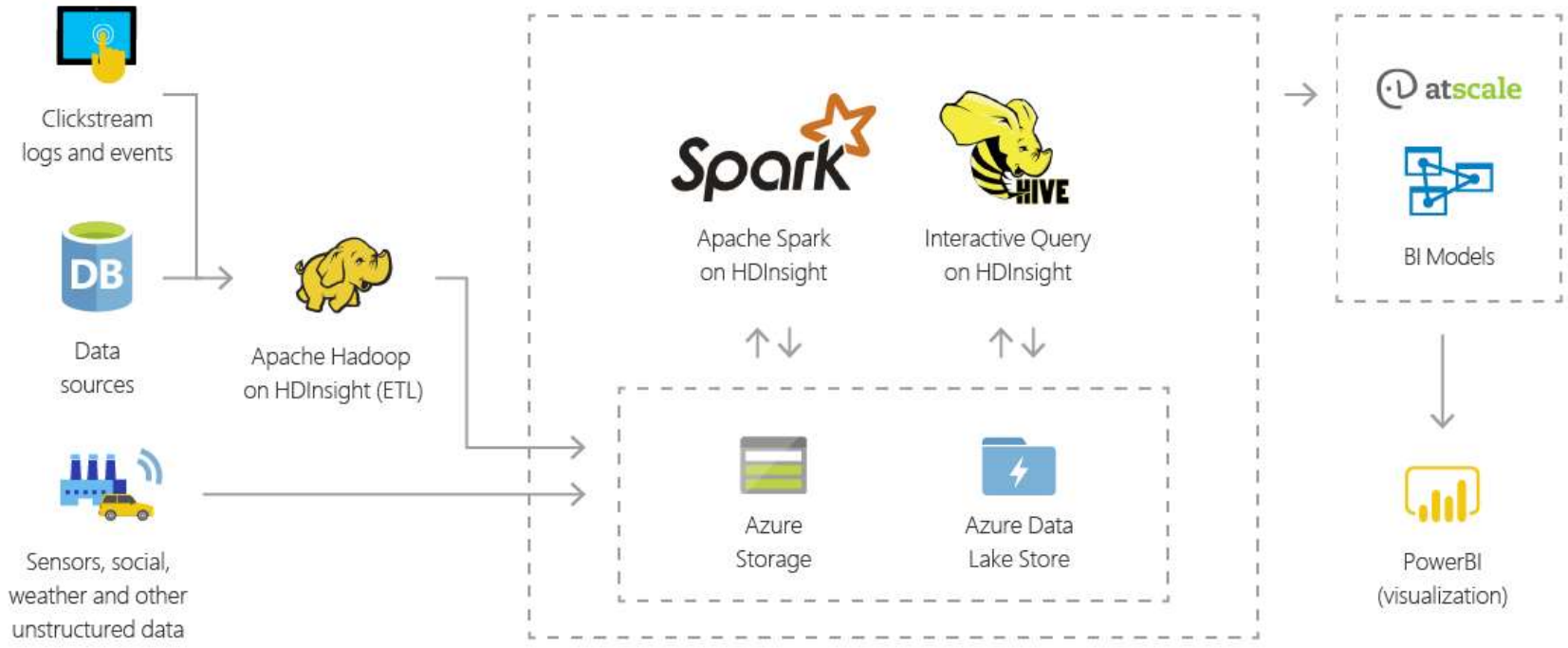
HDInsight Use Case Machine Learning

- Transform your business by adding intelligence to your applications and organization.



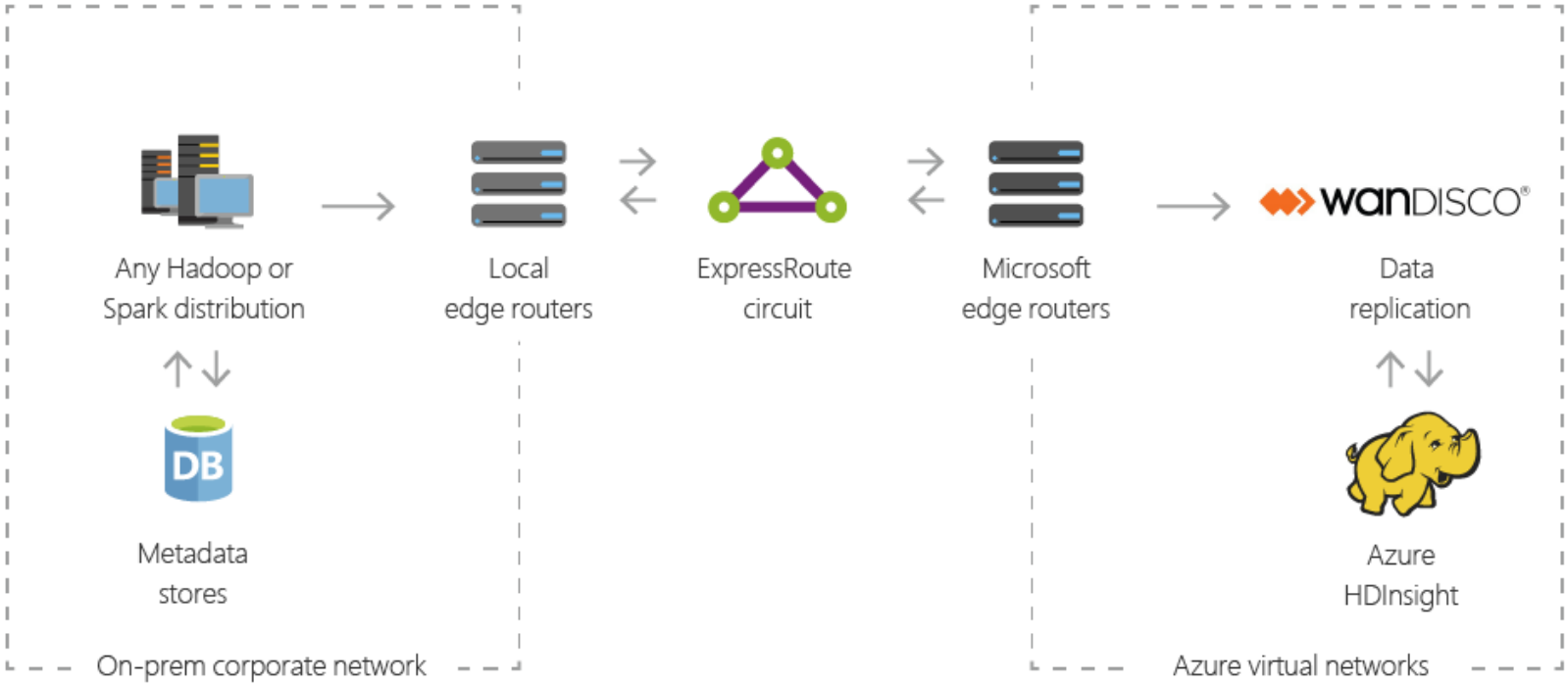
HDInsight Use Case: Data Warehouse

- Perform interactive query at petabyte scale over structured or unstructured data in any format, build models while connecting with your favorite BI tool.



HDInsight Use Case: Hybrid BI

- Extend your on-premises investments to the cloud and transform your business by leveraging the advanced analytics and BI offerings in the cloud.



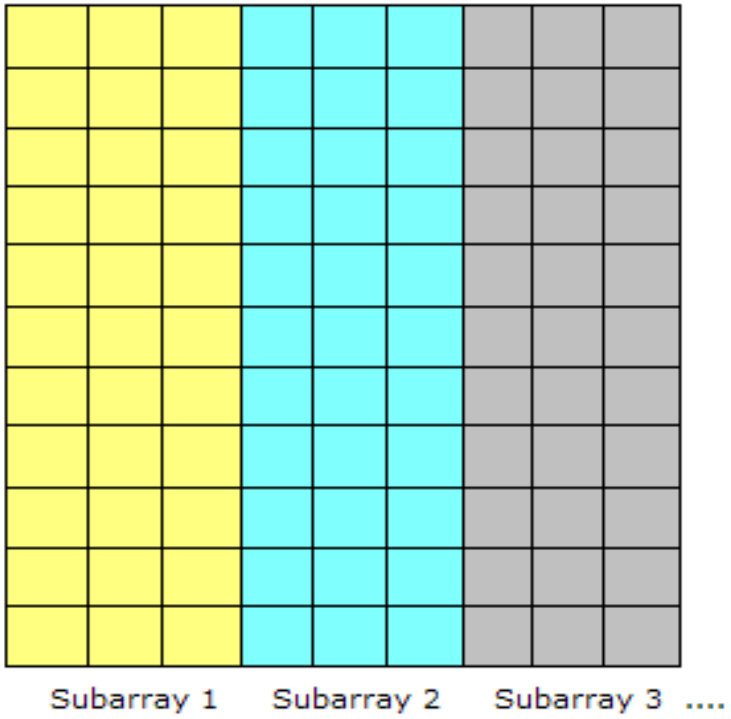
MapReduce and Hadoop

`Serial vs. Parallel` Programming Model

- Many or most of our programs are S*erial*.
  - A `Serial Program` consists of a sequence of instructions, where each instruction executes one after the other.
  - Serial programs run from start to finish on a single processor.
- *Parallel programming* developed as a means of improving performance and efficiency.
  - In a `Parallel Program`, the processing is broken up into parts, each of which could be executed concurrently on a different processor. Parallel programs could be faster.
  - Parallel Programs could also be used to solve problems involving large datasets and non-local resources.
  - Parallel Programs are usually ran on a set of computers connected on a network (a pool of CPUs), with an ability to read and write large files supported by a distributed file system.

Common Situation

- A common situation involves processing of a large amount of consistent data.
- If the data could be decomposed into equal-size partitions, we could devise a parallel solution. Consider a huge array which can be broken up into sub-arrays



If the same processing is required for each array element, with no dependencies in the computations, and no communication required between tasks, we have an ideal parallel computing opportunity, the so called *Embarrassingly Parallel problem.*
A common implementation of this approach is a technique called *Master/Worker*.

Google's MapReduce Programming Model

- `MapReduce` programming model is derived as a technique for solving embarrassingly and not-so-embarrassingly parallel problems.
- The idea stems from the `map` and `reduce` combinators in Lisp programming language.
- In Lisp, a `map` takes as input a function and a sequence of values. It then applies the function to each value in the sequence. A `reduce` combines all the elements of a sequence using a binary operation. For example, it can use "+" to add up all the elements in the sequence.
- `MapReduce` was developed within Google as a mechanism for processing large amounts of raw data, for example, crawled documents or web request logs.

http://static.googleusercontent.com/media/research.google.com/en//archive/mapreduce-osdi04.pdf

- Google data is so large, it must be distributed across tens of thousands of machines in order to be processed in a reasonable time.
- The distribution implies parallel computing since the same computations are performed on each CPU, but with a different portion of data.

MapReduce Programming Model

MapReduce Programming model is made of specific data types, map functions and reduce functions:

- *Data types*: key-value *records (Keys and Values could be of different data types: ints, dates, strings, etc.)*
- *Map function*:

$$(K_{in}, V_{in}) \rightarrow list(K_{inter}, V_{inter})$$

- Intermediate keys do not have to be related to the initial keys in shape or form.
- *Reduce function* is fed sorted collection of intermediate values for each intermediate key.

$$(K_{inter}, list(V_{inter})) \rightarrow list(K_{out}, V_{out})$$

- Reduce function transforms that collection into a final result, a list of key-value pairs.

Vocabulary and Number of Words in all Documents

- Consider the problem of counting the number of occurrences of each word in a large collection of documents

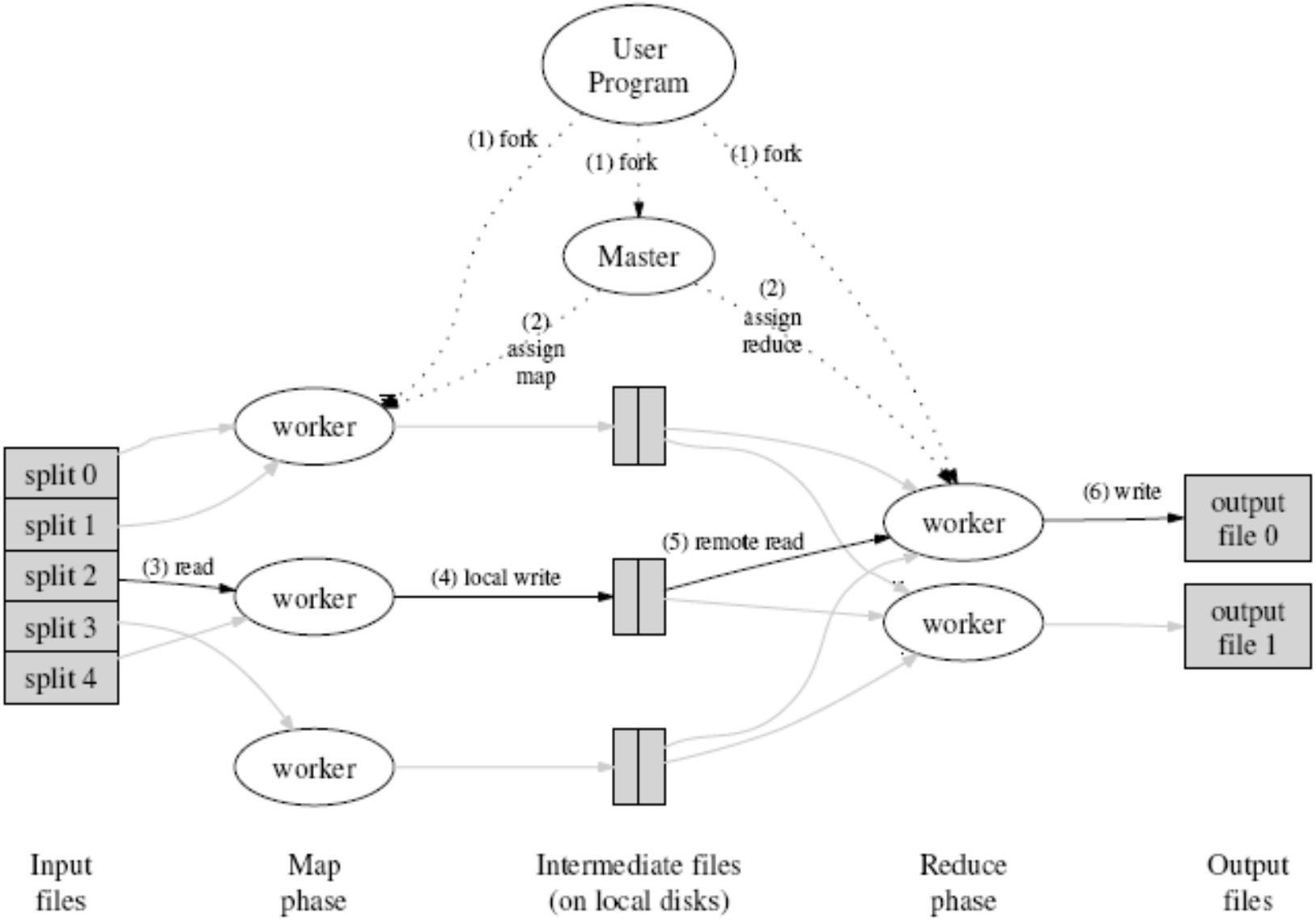
```
map(String documentName, String documentContent):
//key: document name, value: document content
for each word w in documentContent:
//key: word, value: number of occurances
  EmitIntermediate(w, wordCount);

reduce(String w, Iterator values):
// key: a word, // values: a list of counts
int result = 0;
for each v in values:
  result += v;
Emit(w, result));
```

- The map function emits each word plus an associated count of occurrences in a document.
- The reduce function sums all the counts for every word giving us the number of occurrances of each word in the entire set of documents.

MapReduce Execution

- The `map` invocations are distributed across multiple machines by automatically partitioning the input data into a set of M splits or ***shards*.**
- The input shards can be processed in parallel on different machines.
- The `reduce` invocations are distributed by partitioning the intermediate key space into `r` pieces using a partitioning function (e.g., `hash(key) mod r`).
- The number of partitions (`r`) and the partitioning function are specified by the user.

Flow of Execution



Hadoop, Open Source MapReduce

- We do not build MapReduce frameworks.
- We use an open source MapReduce Frameworks, like Hadoop which is offered as Apache project, as well by commercial vendors: Cloudera, Hortonworks, MapR, IBM, and many other vendors.
- Non commercial version of Hadoop source code is available at ***`hadoop.apache.org`***.
- Each vendor improves on original, open source design and sells its improved version commercially.
- **Hadoop made massive parallel computing on large clusters (thousands of cheap, commodity machines) trouble free, efficient and most importantly possible.**
- Hadoop initiated creation of a very large and very dynamic echo system made of may frameworks and APIs

Welcome to Apache™ Hadoop®!

What Is Apache Hadoop?

The Apache™ Hadoop® project develops open-source software for reliable, scalable, distributed computing.

The Apache Hadoop software library is a framework that allows for the distributed processing of large data sets across clusters of computers usi is designed to scale up from single servers to thousands of machines, each offering local computation and storage. Rather than rely on hardwar library itself is designed to detect and handle failures at the application layer, so delivering a highly-available service on top of a cluster of con prone to failures.

The project includes these modules:

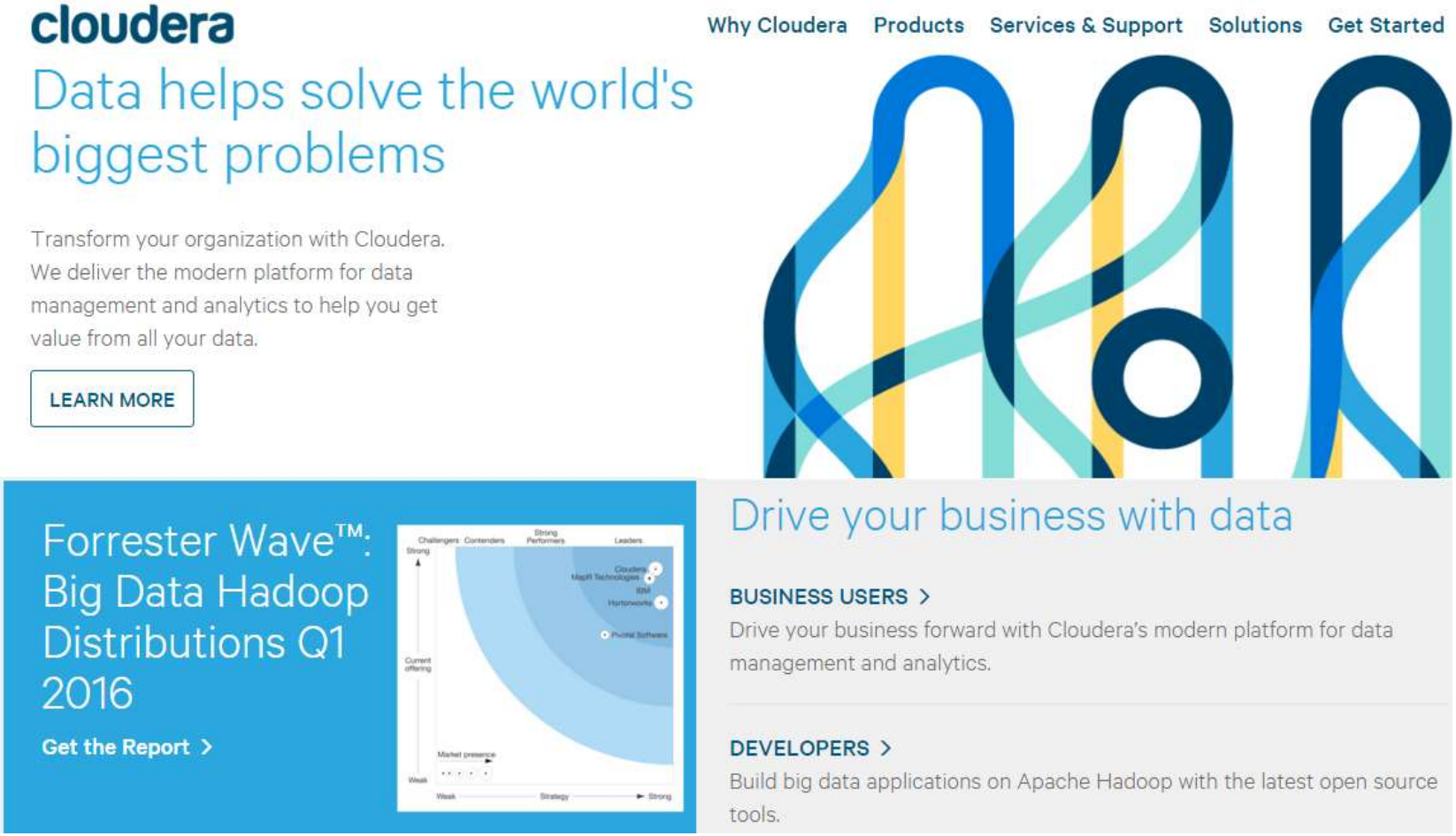
- **Hadoop Common**: The common utilities that support the other Hadoop modules.
- **Hadoop Distributed File System (HDFS™)**: A distributed file system that provides high-throughput access to application data.
- **Hadoop YARN**: A framework for job scheduling and cluster resource management.
- **Hadoop MapReduce**: A YARN-based system for parallel processing of large data sets.

Other Hadoop-related projects at Apache include:

- **Ambari™**: A web-based tool for provisioning, managing, and monitoring Apache Hadoop clusters which includes support for Hadoop HDFS, HCatalog, HBase, ZooKeeper, Oozie, Pig and Sqoop. Ambari also provides a dashboard for viewing cluster health such as heatmaps and ab and Hive applications visually alongwith features to diagnose their performance characteristics in a user-friendly manner.
- **Avro™**: A data serialization system.
- **Cassandra™**: A scalable multi-master database with no single points of failure.
- **Chukwa™**: A data collection system for managing large distributed systems.
- **HBase™**: A scalable, distributed database that supports structured data storage for large tables.
- **Hive™**: A data warehouse infrastructure that provides data summarization and ad hoc querying.
- **Mahout™**: A Scalable machine learning and data mining library.
- **Pig™**: A high-level data-flow language and execution framework for parallel computation.
- **Spark™**: A fast and general compute engine for Hadoop data. Spark provides a simple and expressive programming model that supports a including ETL, machine learning, stream processing, and graph computation.
- **Tez™**: A generalized data-flow programming framework, built on Hadoop YARN, which provides a powerful and flexible engine to execute a process data for both batch and interactive use-cases. Tez is being adopted by Hive™, Pig™ and other frameworks in the Hadoop ecosyste commercial software (e.g. ETL tools), to replace Hadoop™ MapReduce as the underlying execution engine.
- **ZooKeeper™**: A high-performance coordination service for distributed applications.

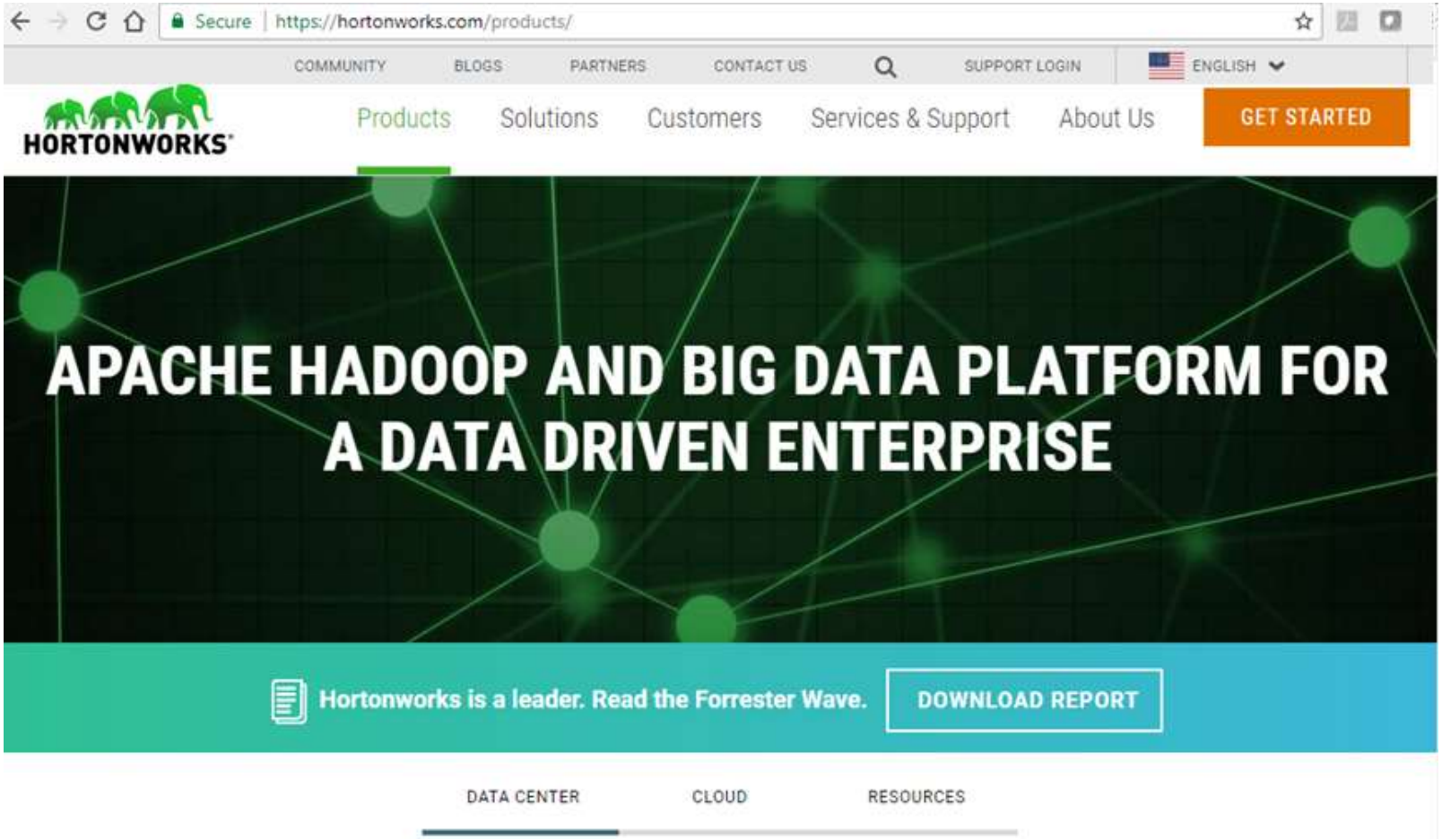
Cloudera

- Doug Cutting who invented Hadoop is at Cloudera. A few years ago Cloudera led the way. One has an impression that they lost steam.
- If you need a smaller VM with Hadoop, older version of Spark (1.6) and other relevant applications, use Cloudera VM.



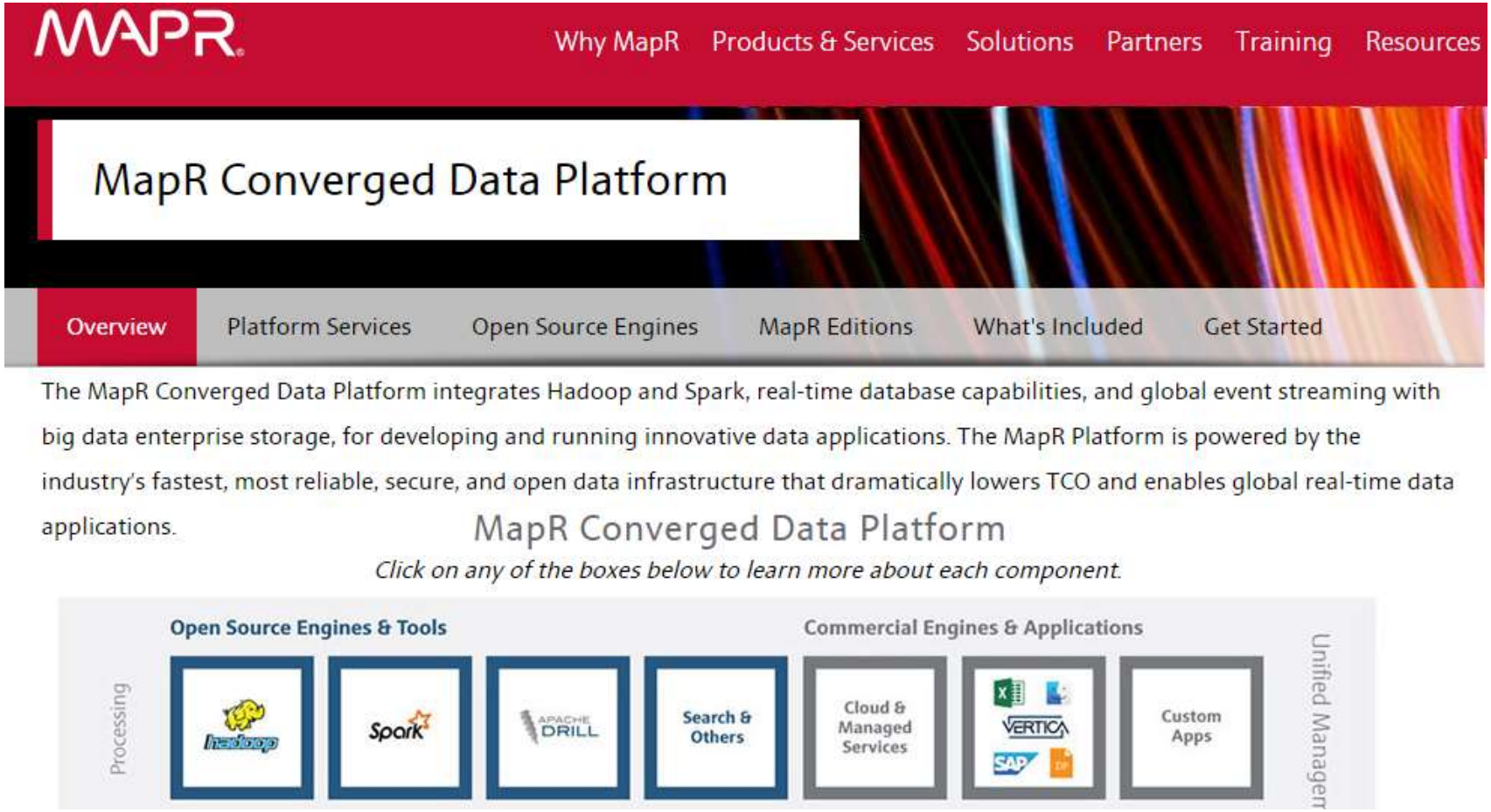
Hortonworks

- If you need a VM with close to latest versions of relevant Big Data APIs and Frameworks go to Hortonworks and download one of their VMs.
- Unfortunately, Hortonworks VMs require machines with more than 8 GB of RAM.

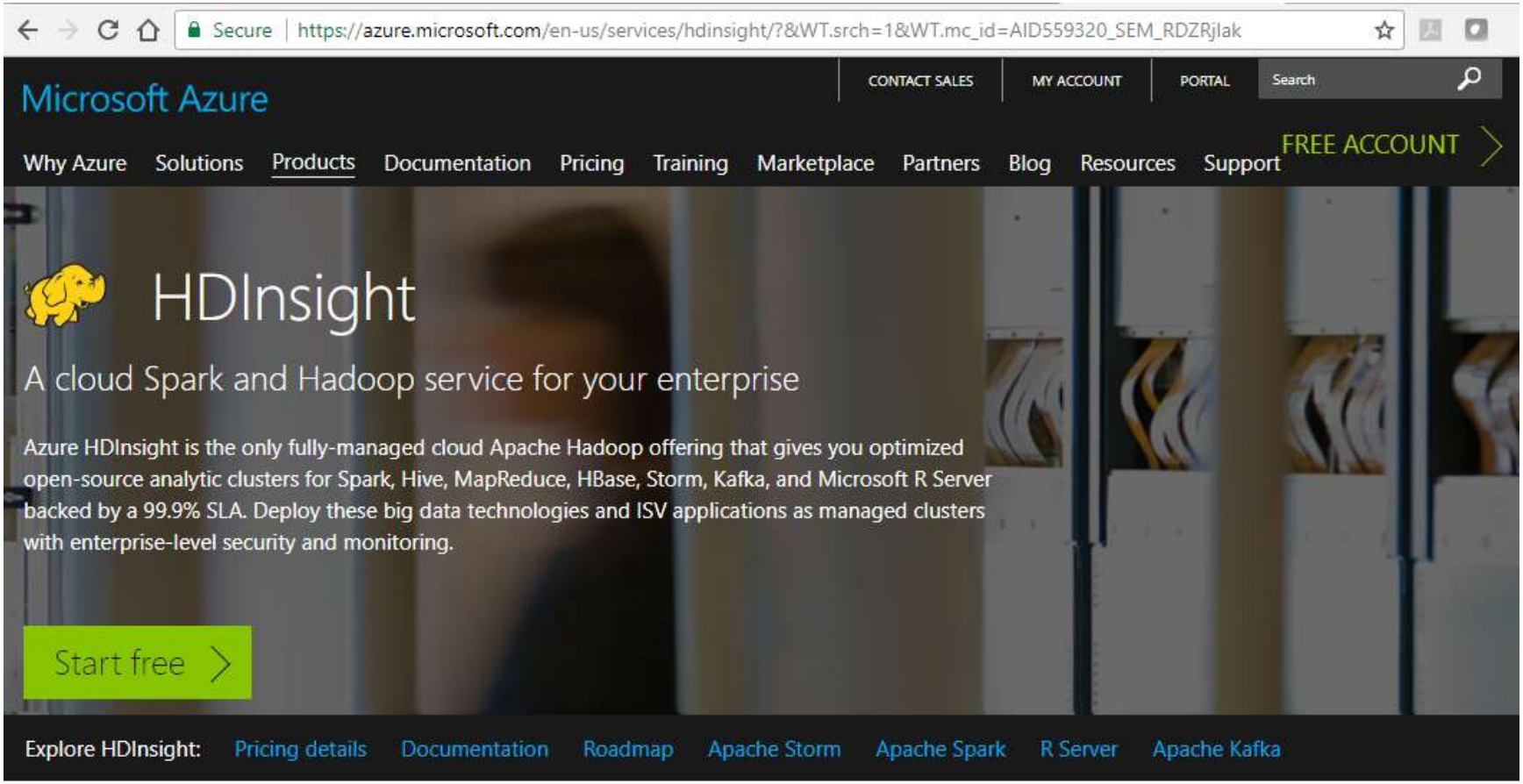


MapR

- If you are the Government of India and want to store and use biometric information of 1.2 billion people, you use MapR Hadoop



Microsoft Azure, HDInsight

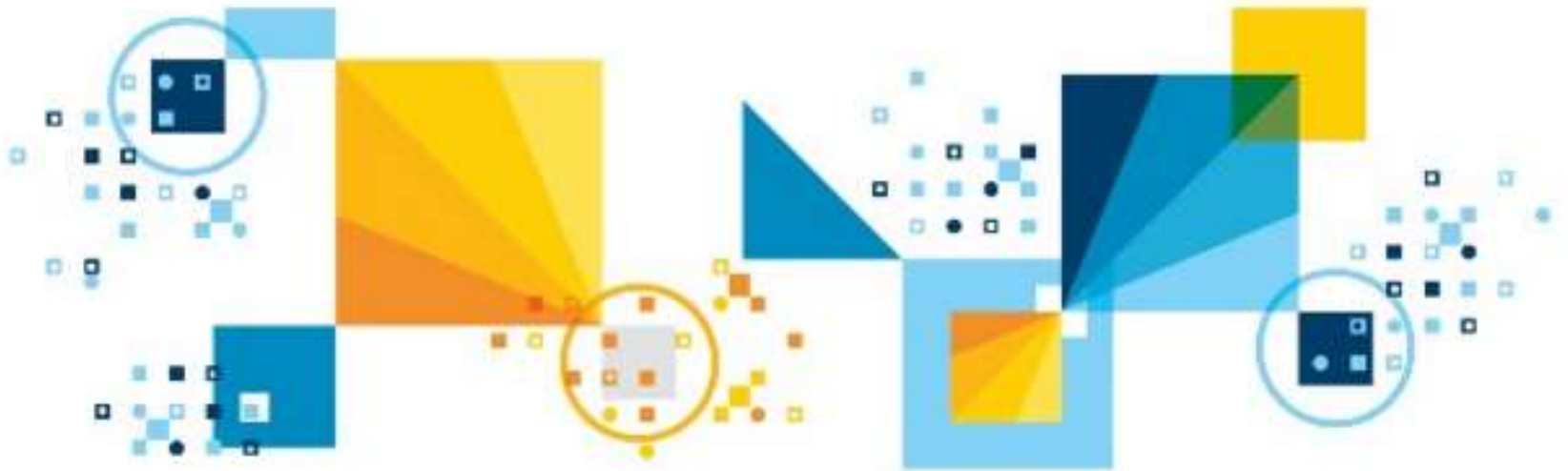


Reliable open-source analytics with an industry-leading SLA

Spin up enterprise-grade, open-source cluster types with a 99.9% SLA and 24/7 support. Our SLA

IBM BigInsight

IBM's BigInsights: Smart Analytics for Big Data



References

- Hadoop's MpReduce API-s are still relevant.
- Many, many companies use Hadoop for batch processing and not only as a storage system.
- Many frameworks like: Hive, Hbase, Impala, and many others use Hadoop's MapReduce (mostly Java) API.
- If you want to do moderate scale batch and streaming analysis you are better of using Spark.

- Hadoop, The Definitive Guide, 3rd Edition, by Tom White, O'Reilly 2012
- Hadoop in Action, by Chuck Lam, Manning 2011
- Hadoop in Practice, by Alex Holmes, Manning 2012
- Cloudera, CDH5 Quick Start Guide

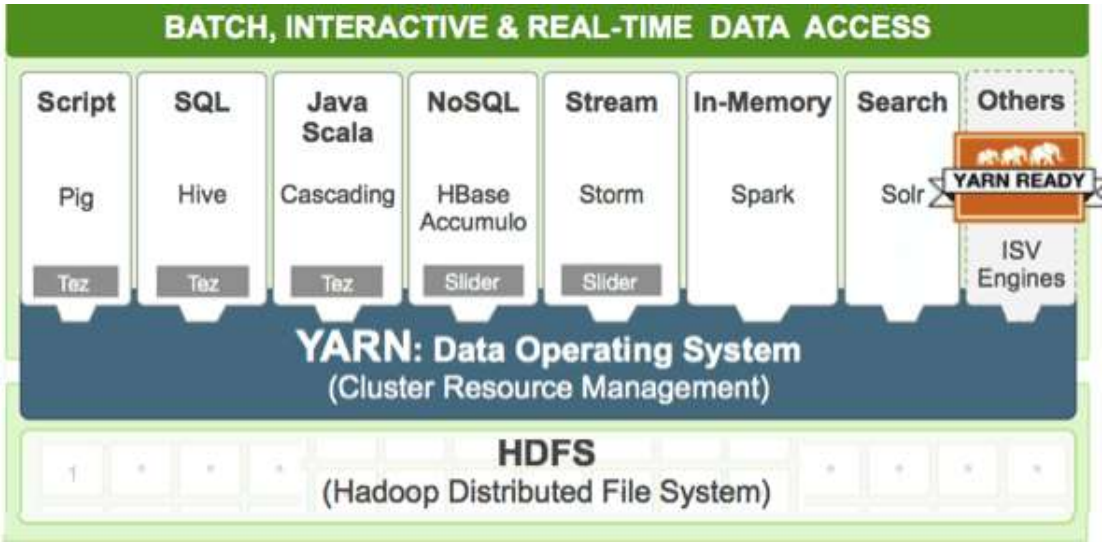
Hadoop Distributed File System

HDFS

Hadoop and HDFS

- We already discussed Hadoop, the premier MapReduce open source framework, and concluded that Spark could do practically everything at a higher speed. Therefor, we decided to "ignore" Hadoop and concentrate on Spark.
- Hadoop is not used only for processing of data. Many organization use Hadoop and HDFS as a "cheap" and reliable storage. As we will demonstrate, Spark can read and store data in HDFS.
- There are many other tools based on Hadoop and HDFS which found widespread use in Big Data processing.
- We will operate Hadoop in pseudo cluster in one machine (VM). Configuration files controlling this pseudo cluster reside in `/etc/hadoop/conf.pseudo/` directory.
- In the following slides we refer to YARN. YARN is a Cluster Resource Manager which allows multiple data processing engines such as interactive SQL, real-time streaming, data science and batch processing to handle data stored in a single platform. YARN is very important and will run as a service on our VM. Otherwise, we will not engage in configuring it or using its facilities beyond the ones already used by the system.
- In the future, we will come back to YARN and its cousin Mesos.

YARN



Feature	Description
Multi-tenancy	YARN allows multiple access engines (either open-source or proprietary) to use Hadoop as the common standard for batch, interactive and real-time engines that can simultaneously access the same data set. Multi-tenant data processing improves an enterprise's return on its Hadoop investments.
Cluster utilization	YARN's dynamic allocation of cluster resources improves utilization over more static MapReduce rules used in early versions of Hadoop
Scalability	Data center processing power continues to rapidly expand. YARN's ResourceManager focuses exclusively on scheduling and keeps pace as clusters expand to thousands of nodes managing petabytes of data.
Compatibility	Existing MapReduce applications developed for Hadoop 1 can run YARN without any disruption to existing processes that already work

`hdfs` script and its commands

Script hdfs performs many important administrative tasks on Hadoop cluster. To see the list, type:

```
[root]# hdfs
```

```
Usage: hdfs [--config confdir] COMMAND  where COMMAND is one of:
  dfs                  run a filesystem command on the file systems supported in Hadoop.
  namenode  -format    format the DFS filesystem
  secondarynamenode    run the DFS secondary namenode
  namenode             run the DFS namenode
  journalnode          run the DFS journalnode
  zkfc                 run the ZK Failover Controller daemon
  datanode             run a DFS datanode
  dfsadmin             run a DFS admin client
  haadmin              run a DFS HA admin client
  fsck                 run a DFS filesystem checking utility
  balancer             run a cluster balancing utility
  jmxget               get JMX exported values from NameNode or DataNode.
  mover                run a utility to move block replicas across storage types
  oiv                  apply the offline fsimage viewer to an fsimage
  oiv_legacy           apply the offline fsimage viewer to an legacy fsimage
  oev                  apply the offline edits viewer to an edits file
  fetchdt              fetch a delegation token from the NameNode
  getconf              get config values from configuration
  groups               get the groups which users belong to
  snapshotDiff         diff two snapshots of a directory or diff the current directory contents with a snapshot
  lsSnapshottableDir   list all snapshottable dirs owned by the current user
  portmap              run a portmap service
  nfs3                 run an NFS version 3 gateway
  cacheadmin           configure the HDFS cache
  crypto               configure HDFS encryption zones
  storagepolicies      list/get/set block storage policies
  version              print the version
```

/user,/tmp, /var & /var/log HDFS directories

- On your VM these directories already exist
- For a variety of users: `hadoop-yarn`, `hbase`, `benchmarks`, `jenkins`, `hive`, `root`, `hue`, and `oozie`, we need user home directories in HDFS of the form `/user/$USER` (e.g. `/user/hbase`) and a series of directories of the type: `/tmp`, `/tmp/hbase`, `/var`, `/var/log`, `var/log/$USER` (e.g. `/var/log/hbase`).
- These fundamental directories are created by the special HDFS administrative user `hdfs` which then changes the mode (access to those directories to `777`, typically) and transfers ownership of those directories to above mentioned users.
- On a Cloudera VM with YARN (MRv2) and MRv1 these tasks are accomplished by script init-hdfs.sh which resides in directory `/usr/lib/hadoop/libexec`, i.e.

```
$ sudo /usr/lib/hadoop/libexec/init-hdfs.sh
```

- Examples of commands used inside `init-hdfs.sh` script are given on the next slide. DO NOT Run `init-hdfs.sh` script on your VM. All needed directories are already created. You might need to

Examine Directories in your HDFS

- User `hdfs` has complete insight into all directories in your HDFS. To see those, type:

```
$ sudo –u hdfs hdfs dfs –ls /user/
drwxr-xr-x   - cloudera cloudera       0 2017-07-19 06:28 /user/cloudera
drwxr-xr-x   - mapred   hadoop         0 2017-07-19 06:29 /user/history
drwxrwxrwx   - hive     supergroup     0 2017-07-19 06:31 /user/hive
drwxrwxrwx   - hue      supergroup     0 2017-07-19 06:30 /user/hue
drwxrwxrwx   - jenkins  supergroup     0 2017-07-19 06:29 /user/jenkins
drwxrwxrwx   - oozie    supergroup     0 2017-07-19 06:30 /user/oozie
drwxrwxrwx   - root     supergroup     0 2017-07-19 06:29 /user/root
drwxr-xr-x   - hdfs     supergroup     0 2017-07-19 06:31 /user/spark
```

- You can see inside all of those directories, just type

```
$ sudo –u hdfs hdfs dfs –ls  -R /user/hive
drwxrwxrwx   - hive supergroup      0 2017-07-19 06:31 /user/hive/warehouse
```

Create Linux user `chuck`

- If you need to create a new Linux user, e.g. `chuck`, log in as `root`, or type:

```
$ su –
```

and enter `root`'s password.

- Now, as user `root`, type:

```
$ useradd –g mapred chuck
```

- The above creates new user `chuck`, as a member of group `mapred`.
- Please note that a user running Hadoop MapReduce programs must be a member of `mapred` group. To create password for new user, type:

```
$ passwd chuck
```

- At the `New password:` prompt , enter a password for user `chuck`, press [Enter].
- At the `Retype new password:` prompt, enter the same password to confirm.
- If user `cloudera` wants to become user `chuck`, `cloudera` types:

```
[cloudera@localhost ~]$ su -- chuck
Password: xxxxxxxxxx
[chuck@localhost cloudera]$ exit # turns you back into cloudera
```

- Should you want to remove Linux user `chuck`, type:

```
$ sudo userdel chuck
```

Create HDFS Directories for user `chuck`

- Next, create HDFS home directory for a new MapReduce user `chuck`.
- By the way, on a truly distributed cluster you will do all of this on the Name Node.
- Type:

```
$ sudo –u hdfs hadoop fs –mkdir /user/chuck
```

- `hadoop fs -mkdir` command automatically creates parent directories if they don't already exist. This is similar to the Unix `mkdir` command with the `-p` option.
- Once we have the directory we want to grant the ownership to user `chuck`.

```
$ sudo –u hdfs hadoop fs –chown chuck:mapred /user/chuck
```

- Finally we want to give full read-write-execute right on that directory.

```
$ sudo –u hdfs hadoop fs –chmod 1777 /user/chuck
```

- Alternatively, if the Linux user already exist, you can login as that user and create the home directory as follows:

```
$ sudo –u hdfs hadoop fs –mkdir /user/$USER
$ sudo –u hdfs hadoop fs –chown $USER /user/$USER
```

- If your system does not have HDFS directories for user `cloudera`, you could use the above procedure to create them

`hadoop` script

- On the previous slides user `hdfs` invoked `hadoop` command. Let us find out what `Hadoop` is. If you type:

```
$ which hadoop
/usr/bin/hadoop
```

- If you open `/usr/bin/hadoop` file, you will see that it invokes another file

```
/usr/lib/hadoop/bin/hadoop
```

- That other `hadoop` file is also a script which runs various Java programs depending on the options you pass to it.
- To get those options invoke `hadoop` by itself:

```
$ hadoop
```

Options of `hadoop` script

```
[root]# hadoop
Usage: hadoop [--config confdir] COMMAND
       where COMMAND is one of:
  fs                   run a generic filesystem user client
  version              print the version
  jar <jar>            run a jar file
  checknative [-a|-h]  check native hadoop and compression
                       libraries availability
  distcp <srcurl> <desturl> copy file or directories recursively
  archive -archiveName NAME -p <parent path> <src>* <dest> create
                          a hadoop archive
  classpath            prints the class path needed to get the
  credential           interact with credential providers
                       Hadoop jar and the required libraries
  daemonlog            get/set the log level for each daemon
  trace                view and modify Hadoop tracing settings or
  CLASSNAME            run the class named CLASSNAME

Most commands print help when invoked w/o parameters.
```

Distributed File System Shell, fs command

- Hadoop has access to both local, Linux, file system, and its own distributed file system (HDFS, Hadoop Distributed File System)
- We access HDFS through Hadoop distributes file system shell, `fs` by invoking command

```
$ hadoop  fs
```

- Will get a long list of options. Some of those resemble Unix (Linux) commands. Some are different.
- We use those commands to create directories in the HDFS, copy files between HDFS and the local file system, Internet and AWS S3 buckets.
- When you use `fs`, you always prefix it with `hadoop`.

Options of file system shell, `hadoop fs`

```
[cloudera]$ hadoop fs
Usage: hadoop fs [generic options]
        [-appendToFile <localsrc> ... <dst>]
        [-cat [-ignoreCrc] <src> ...]
        [-checksum <src> ...]
        [-chgrp [-R] GROUP PATH...]
        [-chmod [-R] <MODE[,MODE]... | OCTALMODE> PATH...]
        [-chown [-R] [OWNER][:[GROUP]] PATH...]
        [-copyFromLocal [-f] [-p] [-l] <localsrc> ... <dst>]
        [-copyToLocal [-p] [-ignoreCrc] [-crc] <src> ... <localdst>]
        [-count [-q] [-h] [-v] [-x] <path> ...]
        [-cp [-f] [-p | -p[topax]] <src> ... <dst>]
        [-createSnapshot <snapshotDir> [<snapshotName>]]
        [-deleteSnapshot <snapshotDir> <snapshotName>]
        [-df [-h] [<path> ...]]
        [-du [-s] [-h] [-x] <path> ...]
        [-expunge]
        [-find <path> ... <expression> ...]
        [-get [-p] [-ignoreCrc] [-crc] <src> ... <localdst>]
        [-getfacl [-R] <path>]
        [-getfattr [-R] {-n name | -d} [-e en] <path>]
        [-getmerge [-nl] <src> <localdst>]
```

File system shell `fs`

```
        [-help [cmd ...]]
         [-ls [-C] [-d] [-h] [-q] [-R] [-t] [-S] [-r] [-u] [<path> ...]]
         [-mkdir [-p] <path> ...]
         [-moveFromLocal <localsrc> ... <dst>]
        [-moveToLocal [-crc] <src> <localdst>]
        [-mkdir <path>]
        [-setrep [-R] [-w] <rep> <path/file>]
        [-touchz <path>]
        [-test -[ezd] <path>]
        [-stat [format] <path>]
        [-tail [-f] <file>]
        [-chmod [-R] <MODE[,MODE]... | OCTALMODE> PATH...]
        [-chown [-R] [OWNER][:[GROUP]] PATH...]
        [-chgrp [-R] GROUP PATH...]
        [-help [cmd]]
Generic options supported are
-conf <configuration file>     specify an application configuration file
-D <property=value>            use value for given property
-fs <local|namenode:port>      specify a namenode
-jt <local|jobtracker:port>    specify a job tracker
-files <comma separated list of files>    specify comma separated files to be
copied to the map reduce cluster
-libjars <comma separated list of jars>    specify comma separated jar files to
include in the classpath.
```

Working with HDFS and Files

- If you run `hadoop fs` command as an ordinary user, e.g. `cloudera`, and do not specify the root of HDFS directory tree, you only see the directories that belong to you and above your home directory.
- Let's check
- `[cloudera]$ hadoop fs -ls`
- There might be nothing there if user `cloudera` has no subdirectories.
- If you want to see all the subdirectories, in a way similar to Unix's `ls` with the `-r` option, you can use `hadoop fs -ls -R` command .

`[cloudera]$ hadoop fs -ls -R` You'll see all the files and directories recursively.

`[cloudera]$ hadoop fs -ls /` `# /` is the root of the directory tree

Found 6 items

```
drwxrwxrwx   - hdfs  supergroup    0 2017-07-19 06:29 /benchmarks
drwxr-xr-x   - hbase supergroup    0 2017-09-27 18:27 /hbase
drwxr-xr-x   - solr  solr          0 2017-07-19 06:31 /solr
drwxrwxrwt   - hdfs  supergroup    0 2017-09-27 06:25 /tmp
drwxr-xr-x   - hdfs  supergroup    0 2017-07-19 06:31 /user
drwxr-xr-x   - hdfs  supergroup    0 2017-07-19 06:31 /var
```

Copying a file to new HDFS directory

- We are ready to add files to HDFS.
- We could fetch the .txt version of James Joyce's Ulysses by issuing the following command on the command prompt:

```
$ wget http://www.gutenberg.org/files/4300/4300-0.txt
```

- We can place file `4300-0.txt` into `ulysses` directory of user `cloudera`

```
$ hadoop fs –mkdir ulysses
$ hadoop fs -put 4300-0.txt ulysses
$ hadoop fs –ls ulysses
[cloudera]$ hadoop fs -ls ulysses
Found 1 items
-rw-r--r-- 1 cloudera cloudera 1580890 2017-09-27 19:54 ulysses/4300-0.txt
```

- The number 1 in the above listing tells us how many times is a particular file replicated. Since we have a single machine, 1 is appropriate.
- The replication factor is 3 by default, but could be set to any number.

Fetching and examining Files from HDFS

- The Hadoop command `get` does the exact reverse of `put`. It copies files from HDFS to the local file system.
- To retrieve file `4300-0.txt` from HDFS and copy it into the current local working directory, we run the command

```
hadoop fs -get 4300-0.txt .
```

- A way to examine the data is to display data. For small files, Hadoop `cat` command is convenient.

```
hadoop fs -cat 4300-0.txt
```

- We can use any Hadoop file command with Unix pipes to forward its output for further processing by another Unix commands. For example, if the file is huge (as typical Hadoop files are) and you're interested in a quick check of its content, you can pipe the output of Hadoop's `cat` into a Unix `head`.

```
hadoop fs -cat 4300-0.txt  | head
```

- Hadoop natively supports `tail` command for looking at the last kilobyte of a file.

```
hadoop fs -tail 4300.txt
```

Deleting Files and Directories

- Hadoop command for removing files is `rm`.

```
hadoop fs –rm ulysses/4300-0.txt
```

- To delete files and directories recursively use

```
hadoop fs -rm -R directory/*
```

- To delete empty directories use

```
hadoop fs -rmdir  directory
```

Introduction to Spark

Reference

- These slides follow to a good measure the book
  "*Learning Spark*" by
  Holden Karau, Andy Konwinski, Patrick Wendell & Mathei Zaharia,
  O'Reilly 2015

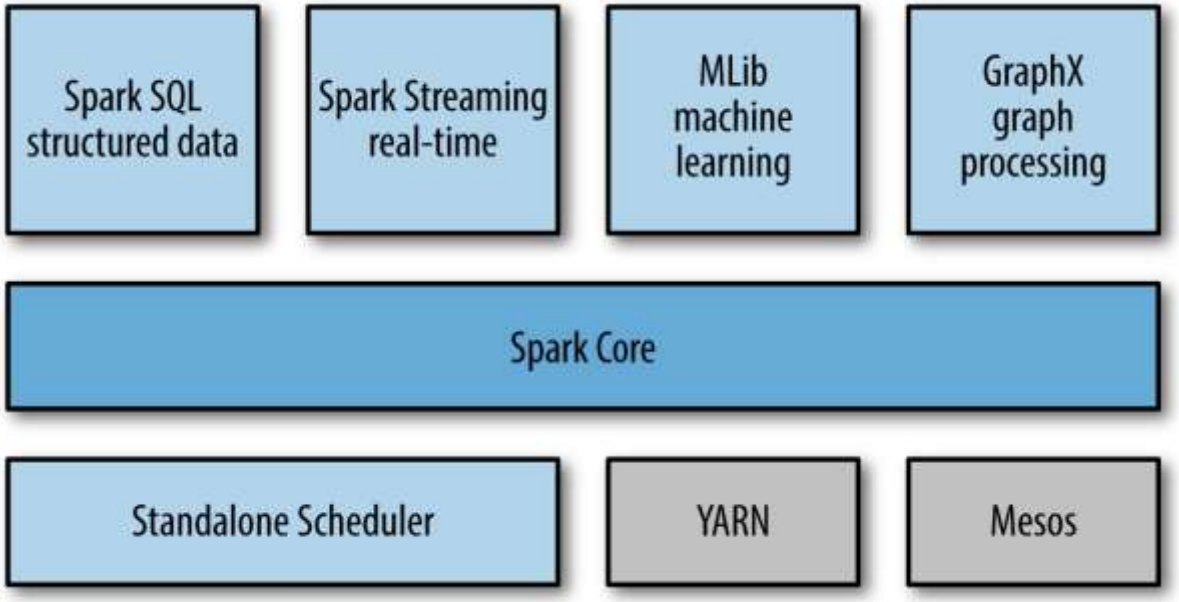
Spark

- Spark is yet another greatest thing ever invented.

- Spark extends MapReduce ideas to efficiently support more types of computations, including interactive queries and stream processing.
- Spark is designed to cover a wide range of workloads that previously required separate distributed systems, including batch applications, iterative algorithms, interactive queries, and streaming.
- By supporting these workloads in the same engine, Spark makes it easy and inexpensive to *combine* different processing types.
- Spark is designed to be highly accessible, offering simple APIs in Python, Java, Scala, R and SQL, and rich built-in libraries.
- Spark integrates closely with other Big Data tools. Spark can run in Hadoop clusters and access any Hadoop data source, including NoSQL DBs like Cassandra.

What is Spark

- Spark is a fast *general* processing engine for large scale data processing
- Spark is designed for iterative computations and interactive data mining.
- Spark supports use of well known languages: Scala, Python, Java and R
- With Spark streaming the same code could be used on data at rest and on data in motion
- Spark has several key modules:



What is RDD, Data Frames

- **Spark is** an *open-source* software solution that performs rapid calculations on *in-memory distributed datasets. Key object used in Spark processing is called RDD*
- RDDs are in-memory distributed datasets.
- *RDDs stands Resilient Distributed Datasets*
  - RDD is the key Spark concept and the basis for what Spark does
  - RDD is a distributed collections of objects that can be cached in memory across cluster and can be manipulated in parallel.
  - RDD could be automatically recomputed on failure
  - RDD is resilient – can be recreated on the fly from known state
  - Immutable – already defined RDDs can be used as a basis to generate derivative RDDs but are never mutated
  - Distributed – the dataset is often partitioned across multiple nodes for increased scalability and parallelism
- Most modern Spark API-s (2.x) emphasize and recommends the use of Spark Data Frames, which are build following the model of R Data Frames and Python Pandas.
- Spark Data Frames have many conveniences like the direct mapping to relational tables and support for standard SQL language.

Key Modules

- **Spark Core** contains the basic functionality of Spark, including components for task scheduling, memory management, fault recovery, interacting with storage systems, and others.
- Spark Core is the home to the API that defines *resilient distributed datasets* (RDDs), which are Spark's main programming abstraction.
- RDDs represent a collection of items distributed across many compute nodes that can be manipulated in parallel. In recent releases less emphasis is placed on RDDs and more and DataFrames and Datasets.
- **Spark SQL** is Spark's package for working with structured data. I
- Spark SQL allows querying data via SQL as well as the Apache Hive variant of SQL—Hive Query Language (HQL)—and it supports many sources of data, including Hive tables, Parquet, and JSON.
- Spark SQL allows developers to intermix SQL queries with the programmatic data manipulations supported by RDDs and DataFrames in Python, Java, and Scala, all within a single application, thus combining SQL with complex analytics.

Key Modules

- **Spark Streaming** is a Spark component that enables processing of live streams of data. Data streams include log files of web servers, or queues of messages.
- Spark Streaming API for manipulating data streams closely matches the Spark Core's RDD API, making it easy to move between apps that manipulate data in memory, on disk, or arriving in real time.
- Spark Streaming provides the same degree of fault tolerance, throughput, and scalability as Spark Core.
- **MLlib** is a library containing common machine learning (ML) functionality.
- MLlib provides multiple types of machine learning algorithms, including classification, regression, clustering, and collaborative filtering, as well as supporting functionality such as model evaluation and data import.
- MLlib provides some lower-level ML primitives, including a generic gradient descent optimization algorithm.
- All of ML methods are designed to scale out across a cluster.

Key Modules

- **GraphX** is a library for manipulating graphs (e.g., social network's graphs) and performing graph-parallel computations.
- GraphX extends the Spark RDD API, allowing us to create a directed graph with arbitrary properties attached to each vertex and edge.
- GraphX also provides various operators for manipulating graphs (e.g., subgraph and mapVertices) and a library of common graph algorithms (e.g., PageRank and triangle counting).

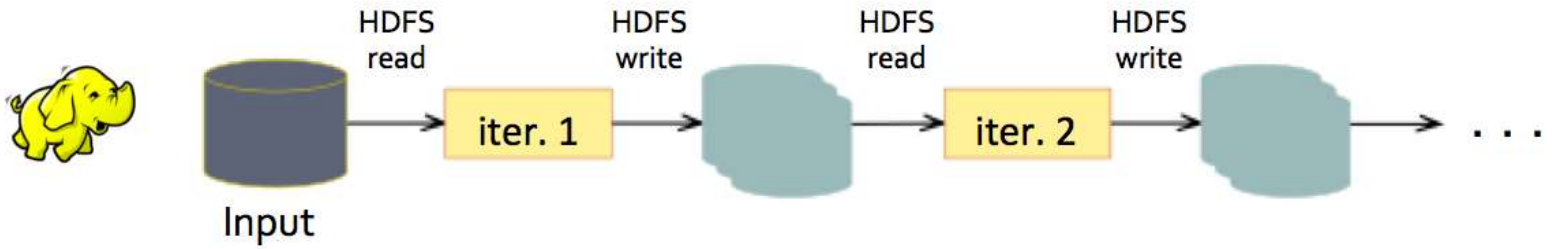
- **Cluster Managers** allow Spark to efficiently scale up from one to many thousands of compute nodes.
- Spark can run over a variety of *cluster managers*, including Hadoop YARN, Apache Mesos, and a simple cluster manager included in Spark itself called the Standalone Scheduler.

How does Spark Work?

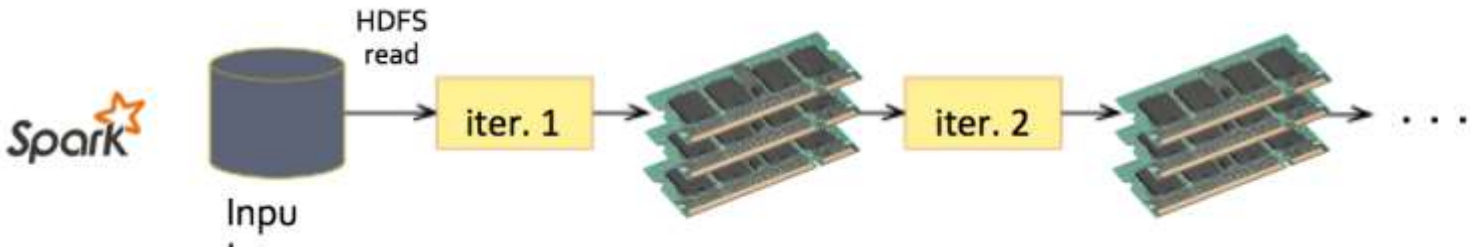
- **RDD**
  - Your data is loaded in parallel into structured collections
- **Actions**
  - Manipulate the state of the working model by forming new RDDs and performing calculations upon them
- **Persistence**
  - Long-term storage of an RDD's state
- **Spark Application is a definition in code of**
  - RDD creation
  - Actions
  - Persistence
- Spark Application results in the creation of a DAG (Directed Acyclic Graph)
- Each DAG is compiled into stages
- Each Stage is executed as a series of Tasks
- Each Task operates in parallel on assigned partitions
- It all starts with the SparkContext 'sc'

Spark vs. Map Reduce

- Map Reduce places every result on the disk



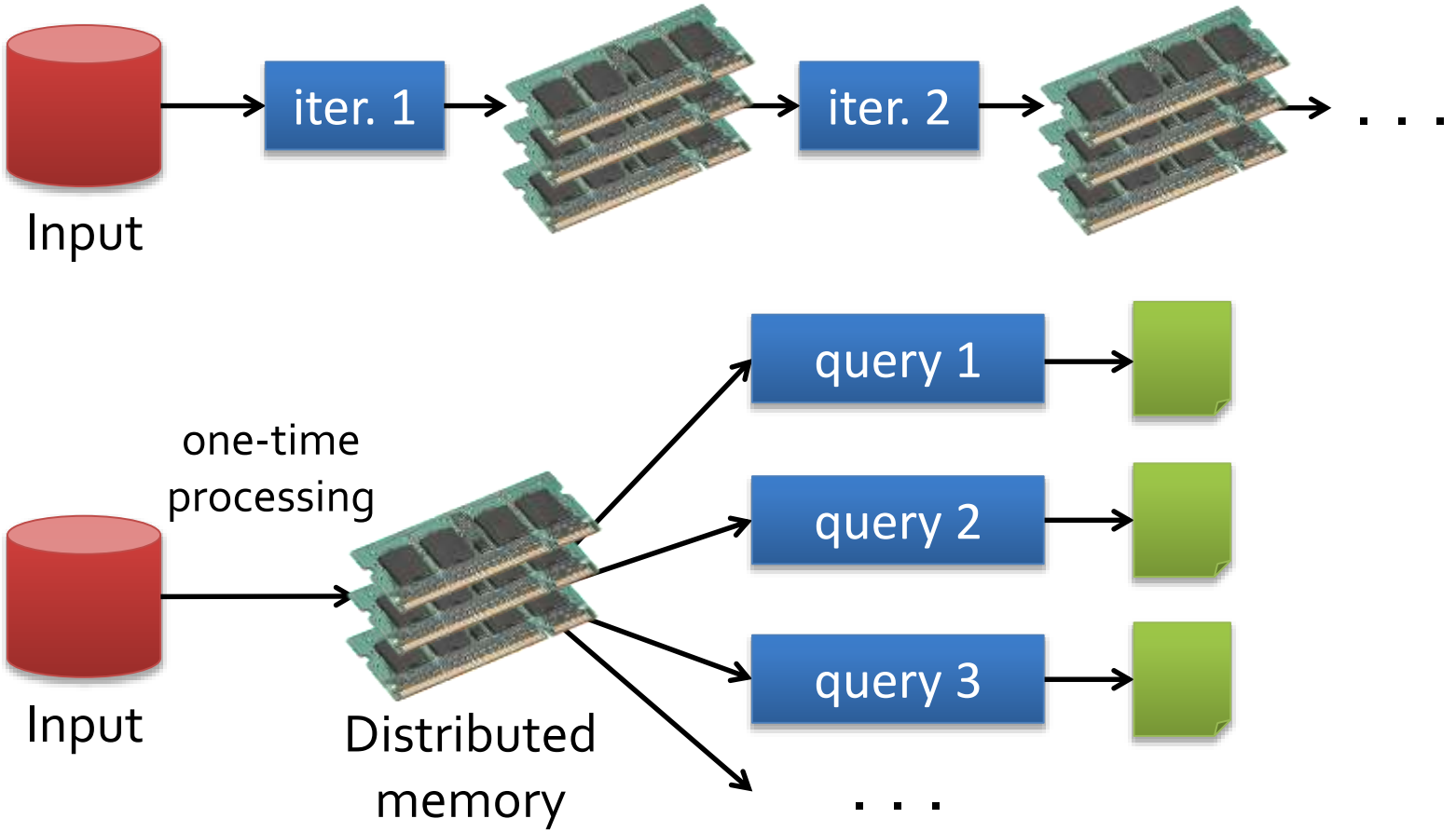
- Sp



- MapReduce greatly simplified big data analysis
- But as soon as it got popular, users wanted more:
  - More **complex**, multi-pass analytics (e.g. ML, graph)
  - More **interactive** ad-hoc queries
  - More **real-time** stream processing
- All need faster **data sharing** across parallel jobs

Data Sharing in Spark

- Distributed memory is 10-100× faster than network and disk

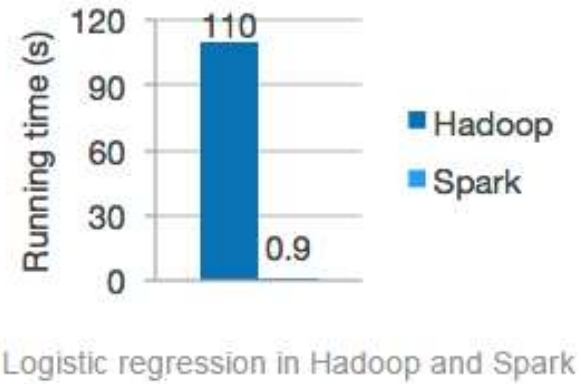


Spark vs. Map Reduce

- Spark extends the popular MapReduce model to efficiently support more types of computations, including interactive queries and stream processing.
- It is said that inventors of Spark noticed that Hadoop (MapReduce) was inefficient for the iterative workflows and they extended Hadoop architecture to make it more efficient.
- Speed is important in processing large datasets, as it means the difference between exploring data interactively and waiting minutes or hours.
- Spark's speed comes from its ability to run computations in memory. Spark appears to be more efficient than MapReduce for complex applications running on disk.
- Spark retains the attractive properties of MapReduce:
  - fault tolerance, data locality, scalability

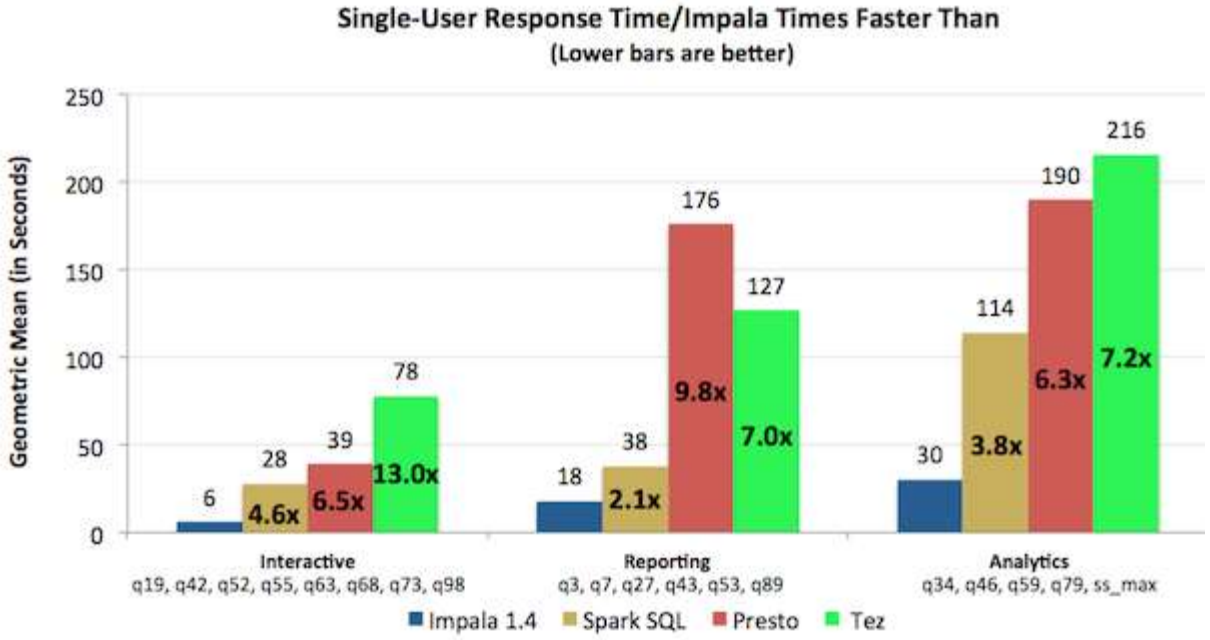
Spark vs. Hadoop

- You will frequently see diagrams like this:



Logistic regression in Hadoop and Spark

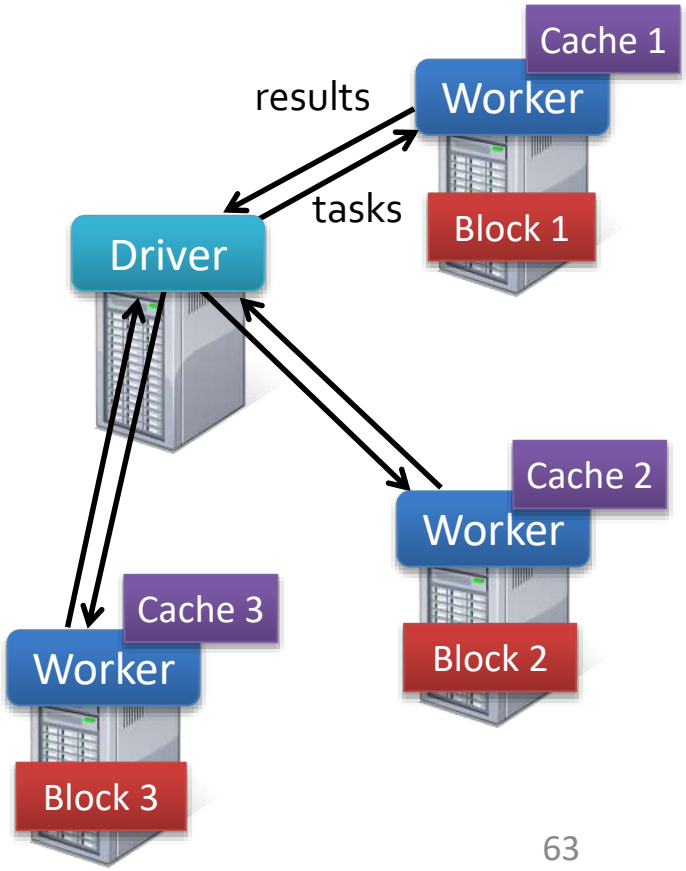
- Do not take this diagrams as a sign of inferiority of people who developed Hadoop. Hadoop people will show you the results below which compare Impala, an SQL engine they developed and Spark. Impala is 4 or 6 X faster,….



- Engineers who developed Hadoop are learning from Spark just like Spark developers learned from Hadoop.
- Cycles keep repeating

Distributed Memory Processing

- Just like MapReduce (Hadoop), Spark has a central controller, called Driver (App Master) which distributes tasks to Workers.
- Data (RDD) is split among memories of many workers.
- Data is originally sourced from the disk (HDFS, S3, regular File System). During processing is shared only between memories (if possible).
- Driver, if it detects that a worker is slacking or not working at all could make the worker redo its work or could move processing to another worker.
- Spark is fault tolerant just like Hadoop.



Spark Shells

- Spark comes with interactive shells that enable ad hoc data analysis.
- Unlike most other shells, which let you manipulate data using the disk and memory on a single machine, Spark's shells allow interaction with data distributed on disks or in memory across many machines.
- Spark can load data into memory on the worker nodes, and distributed computations, even ones that process large volumes of data across many machines, can run in a few seconds. This makes iterative, ad hoc, and exploratory analysis commonly done in shells a good fit for Spark.
- Spark provides both (and only) Python and Scala shells that have been augmented to support access to a cluster of machines.
- In these lecture notes, we will use Python shell, only.
- Python shell opens with `pyspark` command.
- Scala shell is very similar and opens with `spark-shell` command.
- You can program Spark from R using package `SparkR`

Shell verbosity and `log4j.properties` file

- The output is long and annoying. It would have been even longer had we not created `log4j.properties` file in the directory `$SPARK_HOME/conf`.
- You create that file by copying provided file `log4j.properties.template` and by changing line

```
log4j.rootCategory=INFO, console
```

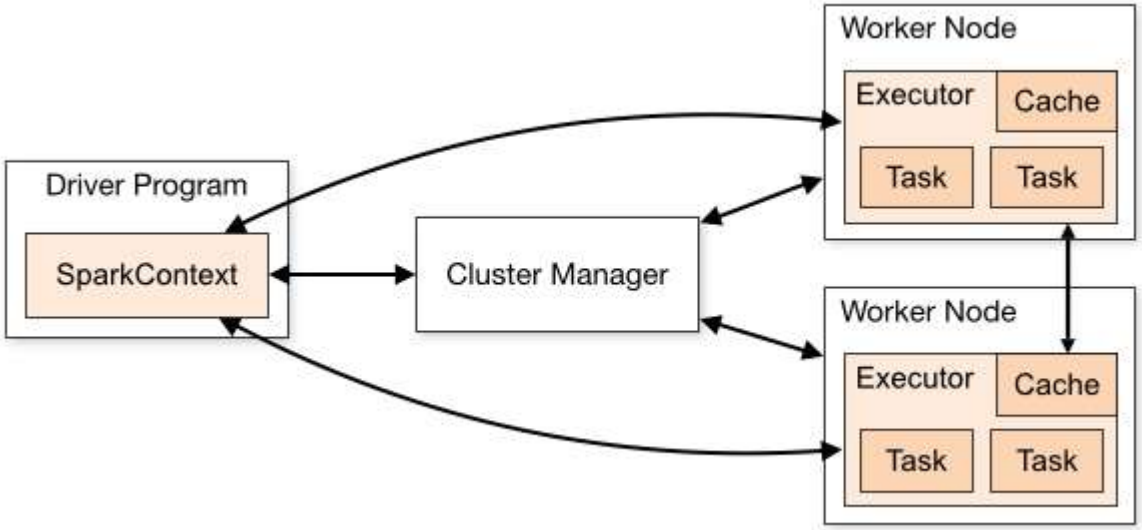
- to read:

```
log4j.rootCategory=ERROR, console
```

- That lowered the logging level so that we see only the ERROR messages, and above.
- You might want to replace all INFO levels with `WARN`, which is more verbose that ERROR but less than INFO. You might also want to replace all WORN levels with ERROR. Adjust those levels as you find convenient.pyspark

Cluster vs. Standalone

- Spark applications run as independent sets of processes on a cluster, coordinated by the `SparkContext` object in your main program (called the driver program).
- Spark clusters could have many thousands of nodes. Initially, we will run a single node standalone "cluster".
- and need to be managed by special resource (cluster) managers such as
- Specifically, to run on a cluster, the SparkContext connects to a cluster manager, such as Spark's own standalone cluster manager, YARN or Mesos. Once connected, Spark acquires *executors* on nodes in the cluster, which are processes that run computations and store data for your application. Next, it sends your application code (defined by JAR or Python files passed to SparkContext) to the executors. Finally, SparkContext sends *tasks* to the executors to run.



Cluster Managers

Spark currently supports three cluster managers:

- Standalone – a simple cluster manager included with Spark that makes it easy to set up a cluster.
- Apache Mesos – a general cluster manager that can also run Hadoop MapReduce and service applications.
- Hadoop YARN – the resource manager in Hadoop 2.
- Kubernetes –Kubernetes is an open-source platform for providing container-centric infrastructure.

- Applications can be submitted to a cluster of any type using the `spark-submit` script.
- Each driver program has a web UI, typically on port 4040, that displays information about running tasks, executors, and storage usage. Simply go to http://<driver-node>:4040 in a web browser to access this UI.

Start your local Cluster of 1

- To start your local cluster with the single machine go to `/opt/spark/sbin` and type:

```
$ sudo ./start-master.sh --master local[2]
```

- Number 2,4 or higher is the number of threads the master will use.
- You could omit `--master local[]` clause.
- In the above command, characters `.` and `/`, `i.e. "./"` are significant.

pyspark

- If you type `pyspark` on the Linux command prompt, you will see the following:

```
[centos@localhost ~]$ pyspark
Python 2.7.5 (default, Aug  4 2017, 00:39:18)
[GCC 4.8.5 20150623 (Red Hat 4.8.5-16)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use
setLogLevel(newLevel).
17/09/16 10:55:40 WARN NativeCodeLoader: Unable to load native-hadoop library for
your platform... using builtin-java classes where applicable
17/09/16 10:55:40 WARN Utils: Your hostname, localhost.localdomain resolves to a
loopback address: 127.0.0.1; using 192.168.135.128 instead (on interface ens33)
17/09/16 10:55:40 WARN Utils: Set SPARK_LOCAL_IP if you need to bind to another
address
17/09/16 10:55:49 WARN ObjectStore: Failed to get database global_temp, returning
NoSuchObjectException
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /__ / .__/\_,_/_/ /_/\_\   version 2.2.0
      /_/

Using Python version 2.7.5 (default, Aug  4 2017 00:39:18)
SparkSession available as 'spark'.
>>>
>>> quit()
$
```

spark-shell

- If your master is started type on the command prompt:

```
$ spark-shell
[centos@localhost ~]$ spark-shell
Spark context Web UI available at http://192.168.135.128:4040
Spark context available as 'sc' (master = local[*], app id = local-
1505573432881).
Spark session available as 'spark'.
Welcome to

      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 2.2.0
      /_/

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> :quit
$
```

Load Data (RDD), Spark 1.6

- In Spark, we express our computation through operations on distributed collections that are automatically parallelized across the cluster.
- These collections are called *resilient distributed datasets*, or RDDs.
- When we load some data, i.e. a file into a shell variable, we are creating an RDD, like

```
>>> lines = sc.textFile("/home/centos/ulysses10.txt")
>>> lines.count()
32742
```

- `ulysses10.txt` is a text file residing in the home directory of user `centos`.
- What we've just done is create and populate an RDD named `lines` using a mysterious object "`sc`" and its method `textFile()`. We populated lines2 = that RDD with data in file `ulysses10.txt`.
- `"sc"` stands for an implicit `SparkContext`.
- `SparkContext` communicates with the execution environment.
- It appears that RDD-s are also (Object Oriented) objects and have methods, such as `count()` which gave use the exact number of lines in file `ulysses10.txt`.
- We could have run the above load process using command:

```
lines = sc.textFile("file:///home/centos/ulysses10.txt")
```

- Emphasizing that the file resides in a regular file system

Load Data (DataFrame), Spark 2.2

- In Spark 2.2 RDDs are still around but you are advised to use a different type of data objects called DataFrame-s. Also, in Spark 2.2 SparkContext (object "sc") is implicit and you rather use spark session object, denoted by "spark". Previous commands could read:

```
>>> lines = spark.read.text("/home/centos/ulysses10.txt")
>>> lines.count()
32742
```

- In `spark-shell` which lets you use Scala, you would use a very similar command:

```
scala> val textFile =
spark.read.textFile("file:///opt/spark/README.md")
textFile: org.apache.spark.sql.Dataset[String] = [value: string]
```

- Spark 2.x DataFrame is strongly-typed like an RDD, but with richer optimizations under the hood. DataFrame objects have better performance than RDDs.

Load Data (DataFrame) from HDFS

- We could load the same data from the Hadoops Distributed File System (HDFS)
- We will learn how to use HDFS in the next lecture.
- If we happen to have the `ulysses10.txt` file in `centos` home directory in HDFS, we could do the following:

```
>>> blines = spark.read.text ("hdfs:///user/centos/ulysses10.txt")
>>> blines.count()
32742
>>> blines.first()
u'The Project Gutenberg EBook of Ulysses, by James Joyce'
```

- What we did above was create an RDD named `blines` and populate that RDD with data from HDFS resident file `ulysses10.txt`.
- We also see in action another method of RDD-s, `first()`, which tells us that the first line in RDD `blines` is some uninterested collection of characters (**`u''`**) .
- Please note that we are not terminating our commands with a semi-colon (";") or anything else aside from the carriage return. Savings on typing all those semi-colons is one of the greatest contributions of Python to the computer science.
- By the way, our commands are in Python

Load Data from the Cloud (AWS S3 Bucket)

- I uploaded the same `ulysses10.txt` file to the Amazon's AWS S3 bucket called `zoran001`.
- On my Linux box (Cloudera VM) I created two new environmental variables in file `.bash_profile`.

```
AWS_ACCESS_KEY_ID=ADTSDYSDUIOSIDOI and
AWS_SECRET_ACCESS_KEY=dsfsfuierfsdfuiorfarifaifaopopa
```

- I source the file:

```
$ source .bash_profile
```

- Then, after reopening Python Spark shell, I issued the command:

```
>>> s3lines = sc.textFile("s3n://zoran001/ulysses10.txt")
>>> s3lines.count()
32742
```

- We did not loose a single line of text while brining the text down from the Cloud.

RDD filter() Method

- RDD method `filter()` takes a function returning `True` or `False` and applies it to a sequence (list) and returns only those members of the sequence for which the function returned `True`.
- So, in the Python code :

```
heavens =lines.filter(lambda line: "Heaven" in line)
```

- Method `filter()` acts on the collection `lines`, and passes every element of that collection as the variable `line` as the argument to the anonymous function created using lambda construct. That anonymous function uses a simple regular expression to test whether string "`Heaven`" exists in variable `line`.
- If the regular expression returns `True` for a particular `line`, an element of collection `lines`, the anonymous function will return `True` and for that particular `line`, `filter()` will return/add variable `line` building up a new collection called `heavens`.
- You can accomplish the same in Java 1.7 and older with several lines of code. In Java 1.8 you have similarly efficient lambda constructs.

RDD `filter()` method

```
>>> lines = sc.textFile("/home/centos/ulysses10.txt")
heaven = lines.filter(lambda line: "heaven" in line)
>>> heaven.count()
50
```

- We see in action a method of RDD-s, `filter()`, which apparently let us inquire how many times is `heaven` mentioned in the Ulysses.
- Heaven is mentioned 52 time, i.e. some 0.15% of the time (> Bible)

DataFrame filtering

- If we load data into a DataFrame, like

```
>>> dset = spark.read.text("/home/centos/ulysses10.txt")
```

- and want to extract all lines with word `heaven` we would use slightly different syntax:

```
>>> dset.count()
32742
>>> heavens = dset.filter(dset.value.contains('heaven'))
>>> heavens.count()
47
```

Python `lambda` Syntax

- Python supports the creation of anonymous inline functions (i.e. functions that are not bound to a name) at runtime, using a construct called "lambda".
- The following code shows the difference between a normal function definition ("f") and a lambda function ("g"):

```
>>> def f(x): return x**2
>>> print f(8)
64
>>> g = lambda x: x**2
>>> print g(8)
64
```

- As you can see, `f()` and `g()` do exactly the same thing. The lambda definition does not include a "return" statement. The last expression is returned.
- You can put a lambda definition anywhere a function is expected, and you don't have to assign it to a variable at all.

Passing Function to Spark in Python

- We want to convince ourselves that rather than using lambda constructs we could define functions and then pass their names to Spark. For example:

```
>>> def hasLife(line):
. . .       return "life" in line    # At the beginning hit a tab
. . .
>>> lines = sc.textFile("file:///home/centos/ulysses10.txt")
>>> lifeLines = lines.filter(hasLife)
>>> lifeLines.count()
203
>>> print lifeLines.first()
u'whom Mulligan was one, and Arius, warring his life long upon
the'
>>>
```

- Function `hasLife()` returns `True` if the line of text in its argument contains string "`life`". We successfully passed that function's name to method `filter()`.
- Above, we have seen a few more crucial features of Python. Python accepted the second line of function definition only when we properly indented `'return "life"…` statement.
- Also, to terminate function definition, we had to hit Carriage Return (Enter) twice.
- Most importantly, there are no semicolons in sight.

Passing Function in Java

- In Java Spark API it is possible to pass functions as well. In that case, functions are defined as Java classes, implementing a Spark interface:
  `org.apache.spark.api.java.function.Function.`
- For example:

```
JavaRDD<String> lifeLines = lines.filter(
    new Function<String, Boolean>() {
     Boolean call(String line){return line.contains("life");}
    }
);
```

- Java 8 introduces shorthand syntax called *lambdas* that looks similar to Python.
- The above Java code in new lambda syntax would look like:

```
JavaRDD<String> lifeLines = lines.filter(line ->
        line.contains("life"));
```

- A lot of Spark's magic is in the fact that function-based operations like `filter()` *also* parallelize across the cluster. That is, Spark automatically takes your function (e.g., `line.contains("Python"))` and ships it to executor nodes.
- You write code in a single driver program and Spark automatically has parts of it running on multiple nodes

Spark Sources and Destinations of Data

- Spark supports a wide range of input and output sources, partly because it builds on the ecosystem made available by Hadoop.
- In particular, Spark can access data through the `InputFormat` and `OutputFormat` interfaces used by Hadoop MapReduce, which are available for many common file formats and storage systems (e.g., S3, HDFS, Cassandra, HBase, etc.).
- For data stored in a local or distributed file system, such as NFS, HDFS, or Amazon S3, Spark can access a variety of file formats including `Text`, `JSON`, `SequenceFiles`, and Google's `Protocol Buffers`.
- The Spark SQL module, provides an efficient API for structured data sources, including JSON and Apache Hive.
- Spark could also use third-party libraries for connecting to Cassandra, HBase, Elasticsearch, and JDBC databases.
- We will analyze some of the above techniques a bit later.

File Formats

- Spark makes it very simple to load and save data in a large number of file formats. Spark transparently handles compressed formats based on the file extension.
- We can use both Hadoop's new and old file APIs for keyed (or paired) data. We can use those only with key/value data.

Format Name	Structured	Comments
Text File	No	Plain old text files. Records assumed to be one per line.
JSON	Semi	Common text-based format, semi-structured; most libraries require one record per line.
CSV	YES	Very common text-based format, often used with spreadsheet applications.
SequenceFile	YES	A common Hadoop file format used for key/value data
Protocol buffer	YES	A fast, space-efficient multi-language format
Object file	YES	Useful for saving data from a Spark job to be consumed by shared code. Breaks if you change your classes, as it relies on Java Serialization.

Standalone Applications

- Spark can be linked into standalone applications in either Java, Scala, or Python. The main difference from using it in a shell is that you need to initialize your own `SparkContext`. After that, the API is the same.
- The process of linking to Spark varies by language. In Java and Scala, you give your application a Maven dependency on the spark-core artifact.
- Maven is a popular package management tool for Java-based languages that lets you link to libraries in public repositories. You can use Maven itself to build your project, or use other tools that can talk to the Maven repositories, including Scala's `sbt` tool or `Gradle`.
- Eclipse also allows you to directly add a Maven dependency to a project.
- In Python, you simply write applications as Python scripts, but you must run them using the `bin/spark-submit` script included in Spark. The `spark-submit` script includes the Spark Python dependencies.
- We simply run your script with the

```
$SPARK_HOME/bin/spark-submit  your_script.py
```

SparkSession in Spark 2, Scala

- In Spark 1.6, you have to create a `SparkConf` and `SparkContext` to interact with Spark, as shown here in Scala snippets:

```
//set up the spark configuration and create contexts
val sparkConf = new
SparkConf().setAppName("SparkSessionZipsExample").setMaster("local")
// your handle to SparkContext to access other context like SQLContext
val sc = new SparkContext(sparkConf).set("spark.some.config.option", "some-
value") val sqlContext = new org.apache.spark.sql.SQLContext(sc)
```

- In Spark 2.0 the same effect will be achieved through `SparkSession`, without explicitly creating `SparkConf, SparkContext or SQLContext`, as they're encapsulated within the `SparkSession`.
- Using a builder design pattern, we instantiate a `SparkSession` object along with its associated underlying contexts.

```
// Create a SparkSession. No need to create SparkContext
import org.apache.spark.sql.SparkSession
val spark = SparkSession
.builder.appName("SparkSessionZipsExample").getOrCreate()
```

Scala Standalone App, Spark 2.x, Example

```
/* SimpleApp.scala */
import org.apache.spark.sql.SparkSession
object SimpleApp {
  def main(args: Array[String]) {
    val logFile = "somefile.txt" // some file on your system
    val spark = SparkSession.builder.appName("Simple Application").getOrCreate()
    val logData = spark.read.textFile(logFile).cache()
    val numAs = logData.filter(line => line.contains("a")).count()
    val numBs = logData.filter(line => line.contains("b")).count()
    println(s"Lines with a: $numAs, Lines with b: $numBs")
    spark.stop()
  }
}
```

- Note that applications defines a `main()` method.
- This program just counts the number of lines containing ‘a’ and the number containing ‘b’ in your file.

Standalone Application in Python 1.6

- To create an application (Python script) we need to import some Python classes and create `SparkContext` object.
- The rest of the application is coded as if you are writing code in PySpark shell

```
from pyspark import SparkConf, SparkContext

conf = SparkConf().setMaster("local").setAppName("MyApp")
sc = SparkContext(conf = conf)
lines = sc.textFile("ulysses10.txt")
lifeLines = lines.filter(lambda line: "life" in line)
print lifeLines.first()
```

- If we invoke the above with:

```
$ spark-submit my_script.py
```

- We get:

```
py4j.protocol.Py4JJavaError: An error occurred while calling
o25.partitions.
: org.apache.hadoop.mapred.InvalidInputException: Input path
does not exist: hdfs://localhost:8020/ulysses/4300.txt
```

Standalone App in Python, Spark 2.2

- Applications can be submitted to a cluster of any type using the `spark-submit` script.

```
from pyspark.sql import SparkSession

logFile = "somefile.txt"  # Should be some file on your system
spark = SparkSession.builder.appName(appName).getOrCreate()
logData = spark.read.text(logFile).cache()

numAs = logData.filter(logData.value.contains('a')).count()
numBs = logData.filter(logData.value.contains('b')).count()

print("Lines with a: %i, lines with b: %i" % (numAs, numBs))

spark.stop()
```

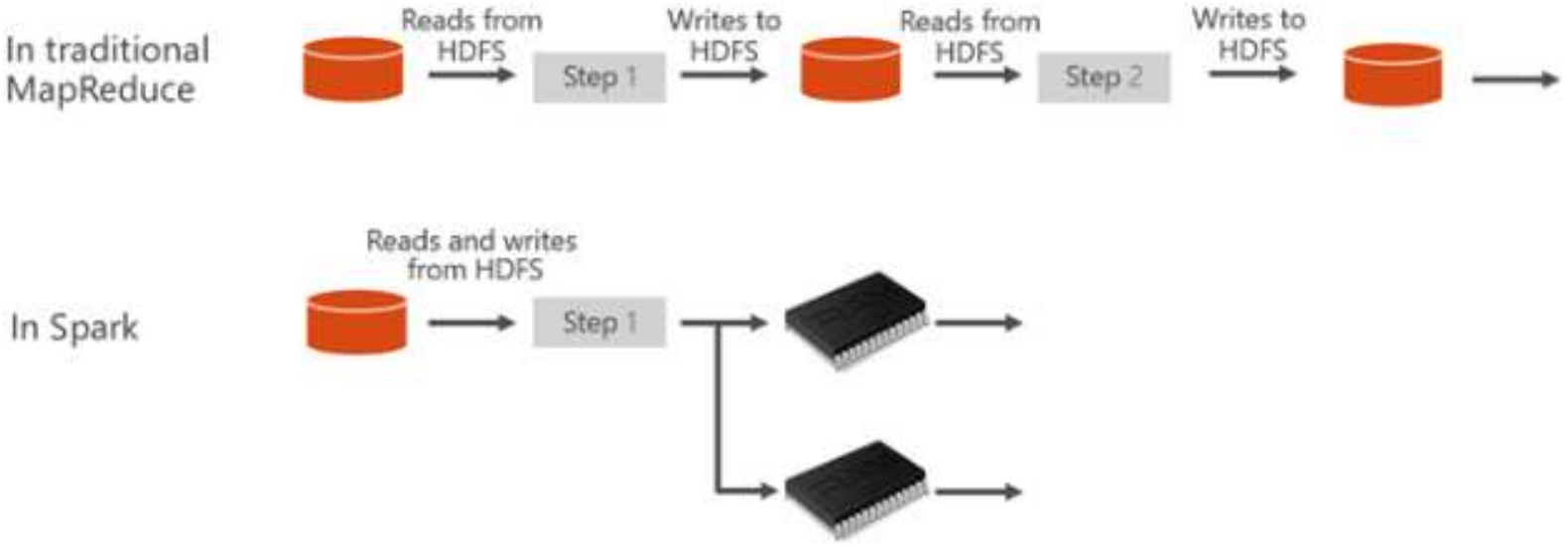
- This program just counts the number of lines containing ‘a’ and the number containing ‘b’ in a text file.

```
$ spark-submit your_script.py
```

Hadoop vs. Spark

Spark vs. MapReduce

- 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 has APIs in Scala, JAVA, Python and partially in R programming languages.
- In Spark you manipulate distributed data sets like local collections. There's no need to structure everything as map and reduce operations.+
- In Spark, data sharing between operations is faster since data is in-memory. In contrast, Hadoop shares data through HDFS, which takes longer to process.



Hadoop vs. Spark

- Hadoop and Apache Spark are both big-data frameworks, but they don't serve the same purposes now a days. Until recently, Hadoop's MapReduce engine was the primary big data analytics tool.
- Hadoop is normally used a distributed data infrastructure or storage framework. Hadoop distributes massive data collections across multiple nodes within a cluster of commodity servers, which means you don't need to buy and maintain expensive custom hardware.
- Hadoop indexes and keeps track of that data, enabling effective big-data processing and analytics.
- Spark, is a data-processing tool that operates on those distributed data collections; it doesn't do distributed storage.
- Hadoop includes not just a storage component, known as the Hadoop Distributed File System (HDFS), but also a processing component called MapReduce, so you don't need Spark to get your processing done.
- Conversely, you can also use Spark without Hadoop. Spark does not come with its own file management system, though, so it needs to be integrated with one -- if not HDFS, then another cloud-based data platform like AWS S3, Azure Storage Blob or Azure Data Lake. Spark was designed for Hadoop, however, so many agree they're better together.

Spark is faster

- Spark is generally a lot faster than Hadoop MapReduce because of the way it processes data. While MapReduce operates in steps, Spark operates on the whole data set in one fell swoop. "The MapReduce workflow looks like this: read data from the cluster, perform an operation, write results to the cluster, read updated data from the cluster, perform next operation, write next results to the cluster, etc.," explained Kirk Borne, principal data scientist at Booz Allen Hamilton. Spark, on the other hand, completes the full data analytics operations in-memory and in near real-time: "Read data from the cluster, perform all of the requisite analytic operations, write results to the cluster, done," Borne said. Spark can be as much as 10 times faster than MapReduce for batch processing and up to 100 times faster for in-memory analytics, he said.
- MapReduce's processing style can be just fine if your data operations and reporting requirements are mostly static and you can wait for batch-mode processing. But if you need to do analytics on streaming data, like from sensors on a factory floor, or have applications that require multiple operations, you probably want to go with Spark. Most machine-learning algorithms, for example, require multiple operations. Common applications for Spark include real-time marketing campaigns, online product recommendations, cybersecurity analytics and machine log monitoring.

Failure Resilience

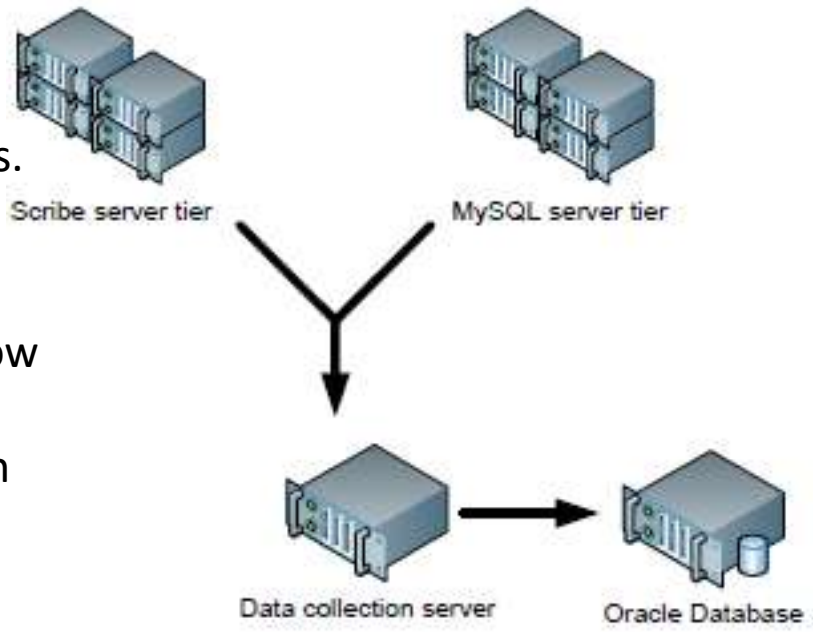
- Hadoop is naturally resilient to system faults or failures since data are written to disk after every operation.
- Spark has similar built-in resiliency by virtue of the fact that its data objects are stored in something called resilient distributed datasets distributed across the data cluster.
- RDD objects can be stored in memory or on disks, and provides full recovery from faults or failures.

Hive

Apache Hive

Facebook 2006

- Started at Facebook.
- At that time, 2006, everything at Facebook was ran on MySQL.
- Facebook has a lot of Web Servers that produce a lot of logs with user related information that they wanted to extract.
- User data were held in farms of MySQL servers.
- All logs went through a locally developed log aggregation framework called Scribe.
- To run large scale reports: how many users, how many emails, who the messages were sent to, etc., a nightly cron job pushed data through an ETL process (Python scripts) into an Oracle database. Oracle produced the reports.



Facebook Scales, 2007 > 2009 >2011> 2016

- In 2006 Facebook produced several tens of GBs of data and schema worked well.
- In mid 2007, Facebook logs accumulated 1TB of data per day.
- In 2009 the volume grew to 10 TB per day.
- January 2011, added 625 TB of compressed data
- In July 2011 Facebook Hadoop cluster had 30 PB of data.
- In 2014 daily volume reached 600 TB
- In 2014, the total size of Hive Warehouse was 300 PB
- In 2016, 500 TB ingested every day.
- Total size of data at Facebook, many Exabytes (1 EB = 1000 PB)
- Since Facebook was aware of Hadoop, they decided to push log data into Hadoop and try processing logs by MapReduce techniques.

Hadoop's MapReduce Jobs

- They started loading data from Scribe and MySQL data into Hadoop HDFS
- Facebook people started writing Hadoop MapReduce jobs to process data.
- To write MapReduce jobs you have to be a relatively sophisticated Java programmer. Hadoop was developed by geeks for geeks.
- It soon became apparent that some things were missing. Most notably:
  - Command-line interface for "end users" was not there.
    - End users are typically "marketing" people and they had no facilities to write queries on the fly. They had to bag real engineers to develop MapReduce programs and even run those programs for them.
    - The above broke the proper social hierarchy of Facebook Corp. Marketing people should never speak with geeks. In military they call that fraternization and can shoot you for it.
  - Information on data structures (schema) was not readily available.
    - Log data files have an implicit schema.
    - That schema is embedded in the code that knows how to read log files.
    - Schema of data is not readily visible.

Hive was Conceived

- In response to those challenges, Facebook developed Hive so that the "users" could perform:
  - Ad-hoc queries without writing full MapReduce jobs
  - Extract or create Schema information

  (Even Hive queries are still written and run by true engineers.)
- Hive could be used for
  - Log processing
  - Text mining
  - Document indexing
  - Customer-facing business intelligence (e.g., Google Analytics)
  - General Statistical Analysis, Predictive modeling, Hypothesis testing, etc.
- Hive has support for various aggregations and joins.
- Hive is considerably closer to standard SQL than PIG.
- Hive is more a data warehouse query language. PIG is more process oriented.

Hive vs. OLAP Warehouses

- Traditional OLAP warehouses create cubes. i.e. materialized views.
- OLAP cubes can not scale to 500 machines. Work on small clusters at best.
- Hive has no automatically generated materialized views.
- If you know what your marketing users are looking for nothing in Hive prevents you from prefabricating data sets that would allow the "users" to quickly find what they are looking for.
- Hive could operate on clusters of 500 or 10,000 machines
- There is no solution that could work on 1TB and return results in minutes.
- If you have only 100 GB of data you are better off loading data into an Oracle database. Let Oracle index everything and then run your queries.
- Once you are in 10TB range Hadoop or Hive are faster than Oracle or any other RDBMS.
- Hive is approximately as fast as Hadoop itself. Hive simplifies your work.
- Hadoop and Hive are not targeting small incremental changes of data sets.
- Hadoop and Hive are meant for global enterprises.

Data Model

- The Major benefit of Hive is that it could make unstructured data look like tables.
- Simply organized data like comma separated values naturally look like tables.
- In `CREATE EXTERNAL TABLE` command you just specify "delimited by", and data will be loaded properly.
- You could also write elaborate serializers/deserializers that could read complex files and populate tables with data contained in those files.
- Hive is a database (warehouse) with strongly typed tables.
- Columns could have atomic types: `int, float, string, date, boolean`
- Composite types: `list, map` (associative array) or `struct` (convenient for JSON-like data).
- Elements of composite types could be any types, including composite types, meaning that types could be arbitrarily complex.

Hive Extends SQL

- Hive has various extensions of SQL. For example:
  - EXPLODE operator would take lists of data and create several columns with atomic data.
  - COLAPSE operator takes lists of data and pushes them into a single column of comma separated data.
- Hives SQL is needs to be learned. For example, in Hive, you could create a table with standard SQL statement.

```
create table people ( name String, id Int, city String);
```

- However, the following, standard SQL, insert statement would not work:

```
insert into people (name, id) values ('Mike', 234)
```

- Hive allowed insert statement is apparently:

```
insert into table people values ('Mike',234,'Boston')
```

- There is a key word TABLE and no list of columns is allowed, meaning that you must provide values for all columns. Nothing difficult, just an annoyance is you are not aware of the syntax differences

Partitions

- You can break large tables by ranges of values in a column, for example by date.
- Date partitioned tables are stored in directories which have subdirectories with names stamped with date.
- You can make queries against individual partitions. Such queries are naturally much faster than queries over entire domain of partition column.
- Within partitions you have sub-partitions called Buckets
- Buckets are Hash partitions within ranges.
- Buckets are useful for sampling. For example: you can perform 5% of query on a valid sample.
- Buckets are also used by optimizer .

Meta Store

- Meta store does not reside in HDFS. Usually it is a Java Derby or MySQL database. Could use almost any other relational databases with a JDBC connector.
- Meta store uses derby by default;
- Meta store is a Database and:
  - Contains a namespace containing a set of tables
  - Holds table definitions (column types, physical layout (where in HDFS tables live as files, etc.))
  - Partitioning data (what are partition boundaries, etc.)
- Database storage location of Meta Store is determined by the hive configuration variable named `javax.jdo.option.ConnectionURL`.
- By default (see `conf/hive-site.xml`), this location is `./metastore_db`
- Right now, in the default configuration, this metadata can only be seen by one user at a time.

Physical Layout

- Warehouse directory is stored in HDFS e.g. `/user/hive/warehouse` or a similarly named directory
- Every table is stored in a subdirectory of `/user/hive/warehouse`
- Partitions, buckets form subdirectories of tables.
- Those files are under Hive control.
- Hive documentation suggests that you could backup those files by just making a copy to another directory or machine.
- Table data stored in
  - Flat Control char-delimited text files (`ctrl A` is the default delimiter)
  - SequenceFiles which are native to Hadoop.
  - With custom serializer-deserilizers, called `SerDe`, files could use arbitrary data organization format

How to access Hive

- On the master node of your EMR master or your Cloudera VM on the command line type `hive` and Hive shell opens:

```
[cloudera@quickstart ~]$ beeline
Beeline version 1.1.0-cdh5.12.0 by Apache Hive
beeline>
```

- To get out of applications we used to type `"quit"` or `"exit".` Quitting is passé, so you do either Ctrl-d or old fashioned Ctrl-c.

Connect beeline to Hive Server

- To connect to beeline, a new client for Hive you do this:

```
$ beeline
beeline>
beeline> !connect jdbc:hive2://127.0.0.1:10000/default hive cloudera
org.apache.hive.jdbc.HiveDriver

"hive" is the username, "cloudera" is its password.
Beeine> show tables;
+------------+--+
|  tab_name  |
+------------+--+
| people  |
+------------+--+
1 rows selected (0.673 seconds)
beeline> describe people;
+-----------+-----------+----------+--+
| col_name  | data_type  | comment  |
+-----------+-----------+----------+--+
| name      | string     |          |
| age       | int        |          |
| salary    | int        |          |
+-----------+-----------+----------+--+
3 rows selected (0.116 seconds)
```

Hive and beeline, Hue

- As user `cloudera` go to `/etc/hive/conf` directory. If there is any file that ends in `.template`, remove the `.template` from the name.
- In `beeline-log4j.properties` and `hive-log4j.properties` files replace INFO and WARN wit ERROR. Do that as a `sudo` user, e.g.:

```
$ sudo vi beeline-log4j.properties
```

- Hue is a Web Application providing graphical user interface to Hive and other database applications in Hadoop eco system.
- Go to Query Editors and select Hive
- You might see several tables in a column on the left. In the query window type:

```
Select * from people
```

Sample Data, Unpack Shakespeare

- On my machine there are two files:

  `shakespeare.tar.gz and bible.tar.gz`

- One contains complete works of Shakespeare and the other King James Bible. Both works use somewhat archaic form of English.
- We can unzip (un tar) both files. Command

  `$ tar zxf shakespeare.tar.gz`

- We will un-tar Shakespeare's works and create loacal directory `~/input` and move `(put)` file `shakespeare` into that directory
- You could also examine the file by perhaps doing:

```
$ cat shakespeare | wc  or
$ cat shakespeare | tail -n 100
```

Copy shakespeare into HDFS

- We will copy local directory "`input`" into the HDFS directory input:

```
$hadoop fs -put input input
```

- We could convince ourselves that the data inside HDFS is still the same Shakespeare by typing something like:
  - `$ hadoop fs -cat input/shakespeare | head -n 20`
  - `1 KING HENRY IV`
  - `DRAMATIS PERSONAE`
  - `KING HENRY the Fourth. (KING HENRY IV:)`

Unpack King James Bible

- We un-tar `bible.tar.gz`, what creates new local file `bible`:

```
$ tar zxf bible.tar.gz
```

- We copy that file to HDFS, as well

```
$hadoop fs –put bible input
```

- If we now list files/directories in HDFS we will see both `input` and files `shakespeare and bible.`

Running a MapReduce `grep` Job

- We can write a Spark script that will behave similarly to Unix grep operation
- The grep script will counts how many times every word appears in the analyzed corpus.
- In our case, Spark `grep` would scan the file (with all Shakespeare's works) placed in the specified (HDFS) directory `"input"` and create a tab delimited report named `shakespeare_freq`.
- Spark `grep` uses regular exp `'\w+'` to select all multi-character words.
- This `grep` is different from Unix (Linux) `grep`. Unix `grep` returns lines where a pattern appears. Hadoop `grep` counts word frequencies.
- The output is placed in HDFS directory `shakespeare_freq`

Examine Result of `grep`

- Examine the output in HDFS directory `shakespeare_freq` by typing:

```
[cloudera]$ hadoop fs -ls
Found 2 items
drwxr-xr-x   - cloudera      input
drwxr-xr-x   - cloudera      shakespeare_freq
$ hadoop fs -ls shakespeare_freq
Found 2 items
drwxr-xr-x   - cloudera        shakespeare_freq/_SUCCESS
-rw-r--r--   1 cloudera        shakespeare_freq/part-r-00000
cloudera@quickstart:~/data/input$
```

Examine Content of the `output` file

- We could also see partial content of the output file:

```
$ hadoop fs -cat shakespeare_freq/part-00000 | head -n 20
25848   the
23031   I
19671   and
18038   to
16700   of
14170   a
12702   you
11297   my
10797   in
6817    his
6773    be
6309    for
cat: Unable to write to output stream.
```

- These are frequency - word pairs, as expected.

Create table to accept `grep` data

- In preparation for import of Shakespeare frequency data we on hive (beeline) prompt we create table `shakespeare`.
- We create the table `shakespeare` by issuing the following command:

```
beeline> create table shakespeare (freq INT, word STRING) ROW
   FORMAT DELIMITED FIELDS TERMINATED BY '\t' stored as textfile;
```

- This created table `shakespeare` with out any data

```
beeline> show tables;
shakespeare
Time taken: 8.268 seconds
beeline> describe shakespeare;
OK
freq    int
word    string
Time taken: 1.253 seconds
beeline>
```

Load grep Data into `shakespeare` Table

- To load data we go back to the `Hue editor` and type:

```
beeline> LOAD DATA INPATH "/user/cloudera/shakespeare_freq" INTO
   TABLE shakespeare;     # From HDFS file system  or
beeline> LOAD DATA LOCAL INPATH "/home/coudera/part-r-00000" INTO
   TABLE shakespeare;     # From the local file system
Loading data to table shakespeare
OK
Time taken: 0.213 seconds
```

- On the load command, Hive moved HDFS content of `shakespeare_freq` into its own HDFS directory. That directory is specified in `hive-site.xml` file

```
cloudera@quickstart:~/etc/hive/conf$ vi hive-site.xml
. . . .
<property>
  <name>hive.metastore.warehouse.dir</name>
  <value>/user/hive/warehouse</value>
  <description>location of default database for the warehouse</description>
</property>
```

- Note again, the directory `/user/hive/warehouse` is in HDFS, not on Linux OS.

Verify that `shakespeare` has `grep` Data

```
beeline> select * from shakespeare limit 10;
OK
25848   the
23031   I
19671   and
18038   to
16700   of
14170   a
12702   you
11297   my
10797   in
8882    is
Time taken: 0.095 seconds
beeline>
```

- This statement read from the table (actually as part of optimization, it read directly from the HDFS file) and presented us with the first 10 lines.
- This is the same data we saw previously.

More Advanced Query

- Slightly more advanced query would perhaps be this one:

```
beeline> SELECT * FROM shakespeare
WHERE freq > 100 SORT BY freq ASC
LIMIT 10;
```

- Notice that for a large data set this is not an entirely trivial job.
- Data has to be sorted before we could see 10 rows of words that have frequency just above 100.
- Notice how hive reports on map-reduce job it is starting.
- If the job takes too long you are given the job id and the command that you could execute to tell Hadoop to kill the job:

```
Starting Job = job_201404021324_0005, Tracking URL =
   http://quickstart:50030/jobdetails.jsp?jobid=job_201404021324_0005
Kill Command = /usr/lib/hadoop/bin/hadoop job  -
   Dmapred.job.tracker=quickstart:8021 -kill job_201404021324_0005
```

Even More Complex Query

- The "users", linguists perhaps, would like to know the number of words which appear with the most common frequencies.

```
beeline> SELECT freq, COUNT(1) AS f2
  FROM shakespeare GROUP BY freq SORT BY f2 DESC LIMIT 10;
```

```
OK
1    13426
2    4274
3    2342
4    1502
5    1111
6    873
7    656
8    598
9    474
10   381
```

- This tells us that there are 13426 words that appears only once.
- 4274 words appear twice. 2342 words appear three times, etc.
- SQL command with minor deviation: ORDER BY is replaced by SORT BY.

Joining Tables

- One of the most powerful feature of Hive is the ability to create queries that joins tables together using regular SQL syntax.
- We have `(freq, word)` data for Shakespeare
- We could generate similar data for King James Bible and then examine which words show up in both volumes of text.
- To generate `grep` data for King James Bible we run Spark grep script:
- This will generate HDFS directory `bible_freq`

```
$ hadoop fs -ls
Found  items
drwxr-xr-x   - cloudera   /user/cloudera/bible_freq
```

- We should remove `_logs and –SUCCESS` directories if present.

```
$ hadoop fs –rm -r bible_freq/_logs
```

Create `bible` Table

```
beeline> CREATE TABLE bible (freq INT, word STRING)
      ROW FORMAT DELIMITED
       FIELDS TERMINATED BY '\t' STORED AS TEXTFILE;
beeline> show tables;
+-------------------------+--+
|        tab_name         |
+-------------------------+--+
| bible                   |
| people                  |
| shakespeare             |
+-------------------------+--+
beeline> desc bible;
OK
freq    int
word    string
Time taken: 0.228 seconds
beeline>
```

- Once you stop believing, you can drop the bible (table) by simply typing

```
beeline> drop table bible;
```

Import data into `bible`

```
beeline> LOAD DATA INPATH "/user/cloudera/bible_freq" INTO TABLE
   Bible;
OK
Time taken: 0.781 seconds
beeline> select * from bible limit 20;
OK
62394    the
38985    and
34654    of
13526    to
12846    And
12603    that
12445    in
6913     be
6884     is
6649     him
6647     LORD
. . .
Time taken: 0.111 seconds
beeline>
```

Examine `bible_freq` directory in HDFS

- Once you imported data into `bible` table examine `bible_freq` directory in HDFS.

```
$ hadoop fs -ls bible_freq
```

- **There is nothing there???**
- Hive took `part-r-00000` out and moved it somewhere else. Where?
- For every table you create, Hive creates a directory in HDFS
- If your table is partitioned, there will be as many directories as partitions
- Those directories live (usually) in HDFS directory `/user/hive/warehouse` we spoke about already
- On some VMs you have to create that directory as the user `hdfs`.
- You do recall commands:

```
$ sudo –u hdfs hadoop fs –mkdir –p /user/hive/warehouse
$ sudo –u hdfs hadoop fs –chown hive /user/hive
$ sudo –u hdfs hadoop fs –chmod 1777 /user/hive
```

- You do not need to do it on Cloudera Quick Start VM

Create an Intermediate Table

- We need a table that will list most common words in both volumes with corresponding frequencies

```
beeline> CREATE TABLE merged
       (word STRING, shake_f INT, kjb_f INT);
```

- For this table we do not need to specify how will date be stored.
- Hive will determine that by itself.

- Next, we will run a query that will select data from tables: `shakespeare` and `bible`, create a join and insert, i.e. overwrite the content of new table.
- In our case the table happens to be empty. If it were not empty and we insist on overwriting, table data would be lost. If we only perform an insert, new data would be appended to the old.

Populate `merged` table

```
beeline> INSERT OVERWRITE TABLE merged
SELECT s.word, s.freq, k.freq FROM
shakespeare s JOIN bible k ON
(s.word = k.word)
WHERE s.freq >= 1 AND k.freq >= 1;
....
Ended Job = job_201404021324_0013
Loading data to table merged
7826 Rows loaded to merged
beeline> . . . .
A       2027    236
AND     102     5
AS      25      2
Aaron   26      350
Abel    2       16
Abhor   2       1
Abide   1       5
About   41      6
Above   25      3
Abraham 4       250
Time taken: 0.107 seconds
```

Most common common words

- What words appeared most frequently in both corpuses?

```
beeline> SELECT word, shake_f, kjb_f,(shake_f + kjb_f) AS ss
FROM merged SORT BY ss DESC LIMIT 20;
the      25848    62394    88242
and      19671    38985    58656
of       16700    34654    51354
I        23031    8854     31885
to       18038    13526    31564
in       10797    12445    23242
a        14170    8057     22227
that     8869     12603    21472
And      7800     12846    20646
is       8882     6884     15766
my       11297    4135     15432
you      12702    2720     15422
he       5720     9672     15392
his      6817     8385     15202
not      8409     6591     15000
be       6773     6913     13686
for      6309     7270     13579
with     7284     6057     13341
it       7178     5917     13095
shall    3293     9764     13057
```

To examine common non-Stop Word, go deeper

```
SELECT word, shake_f, kjb_f, (shake_f + kjb_f) AS ss
FROM merged SORT BY ss DESC LIMIT 200;
. . . . .
heaven  626     578     1204
When    847     349     1196
Of      1006    63      1191
most    1017    135     1152
where   813     335     1148
tell    960     188     1148
blood   699     447     1146
doth    961     63      1146
set     451     694     1145
It      890     241     1131
ever    634     475     1109
Which   977     130     1107
whom    375     732     1107
Time taken: 46.988 seconds
```

Hive's DDL Operations, `Create Table`

- We already know how to create Hive tables and browse through them

```
beeline> CREATE TABLE pokes (foo INT, bar STRING);
```

- Creates a table called `pokes` with two columns, the first being an integer and the other a string

```
beeline> CREATE TABLE invites (foo INT, bar STRING)
  PARTITIONED BY (ds STRING);
```

- Creates a table called `invites` with two columns and a partition column called `ds`.
- The partition column is a virtual column. It is not a part of the data itself but is derived from the partition that a particular dataset is loaded into.
- By default, tables are assumed to be of text input format and the delimiters are assumed to be `^A(ctrl-a)`.

Alter Table Command

- As for altering tables, table names can be changed and additional columns can be dropped:

```
  beeline> ALTER TABLE pokes ADD COLUMNS (new_col INT);
  beeline> ALTER TABLE invites ADD COLUMNS (new_col2 INT COMMENT
   'this is a comment');
  beeline> ALTER TABLE pokes RENAME TO happenings;
OK
Time taken: 0.17 seconds
  beeline> ALTER TABLE happenings RENAME TO pokes;
```

Transfer Data from Hive to Spark

- For Spark to locate your Hive you have to do three things:

1. If Hive services are not running, start Hive metadata server

`$ hiveserver2 &` `# &` at the end means "run in the background"

2. Copy `hive-site.xml` from `$HIVE_HOME/conf`, which is `/etc/hive/conf` to `$SPARK_HOME/conf`, which happens to be `/etc/spark/conf`

3. Create `HiveContext` (in Spark 2, you do not do this, just use `spark (SparkSession)`

```
>>> hivecontext = HiveContext(sc)
```

- In a stand alone Python script you would have to do the import first

```
from pyspark.sql import SQLContext, HiveContext
```

- Now that you are ready, you can query Hive table `shake(speare)` and create DataFrame `dfs` with the content of that Hive table.

```
>>> dfs = hivecontext.sql("select * from shakespeare")
>>> dfs.count()
29183
>>> dfs.first()
Row(freq=25578, word=u'the')
>>> dfs.printSchema()
root
 |-- freq: integer (nullable = true)
 |-- word: string (nullable = true)

>>>
```