**Big Data**

**Big Data Eco-System:**

There are six main components of Big Data tools, namely:

* Data technologies (Hadoop, HDFS, and Spark)
* Analytics and visualization (Tabular)
* Business Intelligence (Power-BI)
* Cloud providers (AWS, GCP, IBM, and Orical)
* NoSQL databases, and (HBase)
* Programming tools (python, r)

Each tooling category plays a very specific and critical role in the Big Data life cycle.

Several major commercial and open source vendors provide tools and support for Big Data processing.

**Open Source and Big Data:**

-Open source runs the world of Big Data.

-Open source projects are free and completely transparent.

-The biggest component of big data is, by far, the Hadoop project including MapReduce, HDFS, and YARN.

-Open source has big data tools like Apache Hive and Apache Spark.

**Big Data Use Cases:**

Companies are relying heavily on Big Data to differentiate themselves from the competition.

There are several ways in which the retail, insurance, telecom, manufacturing, automotive, and finance industries are leveraging Big Data to reduce cost, increase customer satisfaction, and make competitive business decisions.

**Summary of Big Data:**

* Personal assistants like Siri, Alexa and Google Now, use Big Data and IoT to gather data and devise answers.
* Big Data Analytics helps companies gain insights from the data collected by IoT devices.
* Big Data requires parallel processing on account of massive volumes of data that are too large to fit on any one computer.
* "Embarrassingly parallel” calculations are the kinds of workloads that can easily be divided and run independently of one another. If any single process fails, that process has no impact on the other processes and can simply be re-run.
* Open-source projects, which are free and completely transparent, run the world of Big Data and include the Hadoop project and big data tools like Apache Hive and Apache Spark.
* The Big Data tool ecosystem includes the following six main tooling categories: data technologies, analytics and visualization, business intelligence, cloud providers, NoSQL databases, and programming

**Hadoop:**

* Hadoop is an open-source framework for Big Data that faced challenges when encountering dependencies and low-level latency.
* MapReduce, a parallel computing framework used in parallel computing, is flexible for all data types, addresses parallel processing needs for multiple industries and contains two major tasks, “map” and “reduce.”
* The four main stages of the Hadoop Ecosystem are Ingest, Store, Process and Analyze, and Access.
* Key HDFS benefits include its cost efficiency, scalability, and data storage expansion and data replication capabilities. Rack awareness helps reduce the network traffic and improve cluster performance. HDFS enables “write once, read many”operations.
* Suited for static data analysis and built to handle petabytes of data, Hive is a data warehouse software for reading, writing, and managing datasets. Hive is based on the “write once, read many” methodology, doesn’t enforce the schema to verify loading data and has built-in partitioning support.
* Linearly scalable and highly efficient, HBase is a column-oriented non-relational database management system that runs on HDFS and provides an easy-to-use Java API for client access. HBase architecture consists of HMaster, Region servers, Region, Zookeeper and HDFS. A key difference between HDFS and HBase is that HBase allows dynamic changes compared to the rigid architecture of HDFS.

**Hadoop Environment**

**Hadoop-env.sh** serves as a master file to configure YARN,  HDFS,  MapReduce, and Hadoop-related project settings.

Core-site.xml Defines HDFS and Hadoop core properties

Hdfs-site.xml governs the location for storing node metadata, fsimage file and log file.

Mapred-site-xml lists the parameters for MapReduce configuration.

Yarn-site.xml Defines settings relevant to YARN. It contains configurations for the Node Manager, Resource Manager, Containers, and Application Master.

**Hive:**

**Spark**

**Why use Spark**:

-Spark is an open-source, in-memory application framework for distributed data processing and iterative analysis on massive data volumes.

-Distributed computing is a group of computers or processors working together behind the scenes.

-Distributed Computing Vs Parallel Computing:

Parallel computing shares all the memory, while in distributed computing, each processor accesses its own memory.

Distributed computing scale horizontally. Require fault tolerance and redundancy.

-Spark Vs MapReduce:

MapReduce job creates iterations that require reads and writes to disk or HDFS. These reads and writes are usually time-consuming and expensive. Apache Spark solves the read/write problems encountered with MapReduce by keeping much of the data required in-memory and avoiding expensive disk I/O.

**RDD**

A resilient distributed data set (RDD), is Spark's primary data abstraction. A resilient distributed data set, is a collection of fault tolerant elements partitioned across the cluster's nodes capable of receiving parallel operations. Additionally, resilient distributed databases are immutable, meaning that these databases cannot be changed once created.

RDD's support:

Text,

Sequence files,

Avro,

Parquet and Hadoop input format file types,

Support local,

Cassandra,

H Base,

HDFS,

Amazon S3,

Relational and no SQL databases.

**Create RDDs:**

* First, using an external or local file from a Hadoop supported file system such as HDFS, Cassandra, H Base or Amazon S3.
* A second method is to apply the parallelize function to an existing collection in the driver program. Note(One important parameter for parallel collections is the number of partitions specified to cut the dataset)
* A third method is to apply a transformation on an existing RDD to create a new RDD.

**Parallel Programming**:

Tasks run using multiple processors access a shared pool of memory, which has in place control and coordination mechanisms.

**SparkSQL**:

-Spark modules for structured data processing can run SQL queries on

Spark DataFrames and are usable in Java, Scala, Python, and R.

-Spark SQL supports both temporary views and global temporary views.

-You can use a DataFrame function or a SQL Query plus Table View for data aggregation.

-Spark SQL supports Parquet files, JSON datasets, and Hive tables.

**Summary of Spark:**

* Spark is an open source in-memory application framework for distributed data processing and iterative analysis on massive data volumes. Both distributed systems and Apache Spark are inherently scalable and fault tolerant. ​Apache Spark solves the problems encountered with MapReduce by keeping a substantial portion of the data required in-memory, avoiding expensive and time-consuming disk I/O.​
* Functional programming follows a declarative programming model that emphasizes “what” instead of “how to” and uses expressions.​
* Lambda functions or operators are anonymous functions that enable functional programming. Spark parallelizes computations using the lambda calculus​ and all functional Spark programs are inherently parallel​.
* Resilient distributed datasets, or RDDs, are Spark’s primary data abstraction​consisting of a fault-tolerant collection of elements partitioned across the nodes of the cluster, capable of accepting parallel operations.​You can create an RDD using an external or local Hadoop-supported file, from a collection, or from another RDD. RDDs are immutable and always recoverable, providing resilience in Apache Spark​RDDs can persist or cache datasets in memory across operations, which speeds iterative operations​in Spark.
* Apache Spark architecture consists of components data, compute input, and management. The fault-tolerant Spark Core base engine performs large-scale Big Data worthy parallel and distributed data processing jobs, manages memory, schedules tasks, and houses APIs that define RDDs.
* Spark SQL provides a programming abstraction called DataFrames and can also act as a distributed SQL query engine. Spark DataFrames are conceptually equivalent to a table in a relational database or a data frame in R/Python, but with richer optimizations.

**Pandas UDF**

Apache Spark has become the de-facto standard in processing big data. To enable data scientists to leverage the value of big data, Spark added a Python API in version 0.7, with support for user-defined functions (UDF). These user-defined functions operate one-row-at-a-time, and thus suffer from high serialization and invocation overhead. As a result, many data pipelines define UDFs in Java and Scala and then invoke them from Python.

Pandas UDFs built on top of Apache Arrow bring you the best of both worlds—the ability to define low-overhead, high-performance UDFs entirely in Python. In this simple example, we will build a Scalar Pandas UDF to convert the wT column from imperial units (1000-lbs) to metric units (metric tons).

In addition, UDFs can be registered and invoked in SQL out of the box by registering a regular python function using the @pandas\_udf() decorator. We can then apply this UDF to our wt column.

**Intro to DataFrame and SparkSQL Summary:**

* RDDs are Spark's primary data abstraction partitioned across the nodes of the cluster. Transformations leave existing RDDs intact and create new RDDs based on the transformation function. With a variety of available options, apply functions to transformations perform operations. Next, actions return computed values to the driver program. Transformations undergo lazy evaluation, meaning they are only evaluated when the driver function calls an action.
* A dataset is a distributed collection of data that provides the combined benefits of both RDDs and SparkSQL. Consisting of strongly typed JVM objects, datasets make use of DataFrame typesafe capabilities and extend object-oriented API capabilities. Datasets work with both Scala and Java APIs.  DataFrames are not typesafe. You can use APIs in Java, Scala, and Python. Dataset​s are Spark's latest data abstraction.
* The primary goal of Spark SQL Optimization is to improve the run-time performance of a SQL query, by reducing the query’s time and memory consumption, saving organizations time and money. ​Catalyst is the Spark SQL built-in rule-based *query* optimizer.​ Catalyst performs analysis, logical optimization, physical planning, and code generation.​ Tungsten is the Spark built-in cost-based optimizer for CPU and memory usage that enables cache-friendly computation of algorithms and data structures.
* Basic DataFrame operations are reading, analysis, transformation, loading, and writing. ​You can use a Pandas DataFrame in Python to load a dataset and apply the print schema, select function, or show function for data analysis. ​For transform tasks, keep only relevant data and apply functions such as filters, joins, column operations, grouping and aggregations, and other functions.
* Spark SQL consists of Spark modules for structured data processing that can run SQL queries on Spark DataFrames and are usable in Java, Scala, Python and R. Spark SQL supports both temporary views and global temporary views. Use a DataFrame function or an SQL Query + Table View for data aggregation. Spark SQL supports Parquet files, JSON datasets and Hive tables.

**Spark Architecture**

* Spark Architecture has driver and executor processes, coordinated by the Spark Context in the Driver.
* The Driver creates jobs and the Spark Context splits jobs into tasks which can be run in parallel in the executors on the cluster. Stages are a set of tasks that are separated by a data shuffle. Shuffles are costly, as they require data serialization, disk and network I/O.​ The driver program can be run in either client Mode (connecting the driver outside the cluster) or cluster mode (running the driver in the cluster).
* Cluster managers acquire resources and run as an abstracted service outside the application. Spark can run on Spark Standalone, Apache Hadoop YARN, Apache Mesos or Kubernetes cluster managers, with specific set-up requirements.​ Choosing a cluster manager depends on your data ecosystem and factors such as ease of configuration, portability, deployment, or data partitioning needs. Spark can also run using local mode, which is useful for testing or debugging an application.
* 'Spark-submit’ is a unified interface to submit the Spark application, no matter the cluster manager or application language. Mandatory options include telling Spark which cluster manager to connect to; other options set driver deploy mode or executor resourcing. To manage dependencies, application projects or libraries must be accessible for driver and executor processes, for example by creating a Java or Scala uber-JAR. Spark Shell simplifies working with data by automatically initializing the SparkContext and SparkSession variables and providing Spark API access.

**Kubernetes**

-Kubernetes runs containerized applications on a cluster, managing distributed systems such as Spark in a more flexible, resilient way.

-Kubernetes can run locally as a deployment environment, useful for trying out changes before deploying to clusters in the cloud.

-Kubernetes can be hosted on private or hybrid clouds, set up using existing tools to bootstrap clusters, or turnkey options from certified providers.

-Spark can be launched in client or cluster mode, in Client mode, executors must be able to connect with the driver and pod cleanup settings are required.

Week5 Summary

* Running Spark on IBM Cloud provides enterprise security and easily ties in IBM big data solutions for AIOps, IBM Watson and IBM Analytics Engine​. Spark’s big data processing capabilities work well with AIOps tools, using machine learning to identify events or patterns and help report or fix issues​. IBM Spectrum Conductor manages and deploys Spark resources dynamically on a single cluster and provides enterprise security.​ IBM Watson helps you focus on Spark’s machine learning capabilities by creating automated production-ready environments for AI​. IBM Analytics Engine separates storage and compute to create a scalable analytics solution alongside Spark’s data processing capabilities.
* You can set Spark configuration using properties (to control application behavior), environment variables (to adjust settings on a per-machine basis) or logging properties (to control logging outputs)​. Spark property configuration follows a precedence order, with the highest being configuration set programmatically, then spark-submit configuration and lastly configuration set in the spark-defaults.conf file​. Use Static configuration options for values that don’t change from run to run or properties related to the application, such as the application name​. Use Dynamic configuration options for values that change or need tuning when deployed, such as master location, executor memory or core settings​.
* Use Kubernetes to run containerized applications on a cluster, to manage distributed systems such as Spark with more flexibility and resilience. You can run Kubernetes as a deployment environment, which is useful for trying out changes before deploying to clusters in the cloud​. Kubernetes can be hosted on private or hybrid clouds, and set up using existing tools to bootstrap clusters, or using turnkey options from certified providers​. While you can use Kubernetes with Spark launched either in client or cluster mode, when using Client mode, executors must be able to connect with the driver and pod cleanup settings are required.

Week6 Summary

* To connect to the Apache Spark user interface web server, start your application and connect to the application UI using the following URL:  **http://<driver-node>:4040**
* The Spark application UI centralizes critical information, including status information into the **Jobs**, **Stages**, **Storage**, **Environment** and **Executors** tabbed regions. You can quickly identify failures, then drill down to the lowest levels of the application to discover their root causes. If the application runs SQL queries, select the **SQL** tab and the **Description** hyperlink to display the query’s details.​
* The Spark application workflow includes jobs created by the Spark Context in the driver program, jobs in progress running as tasks in the executors, and completed jobs transferring results back to the driver or writing to disk​.
* Common reasons for application failure on a cluster include user code, system and application configurations, missing dependencies, improper resource allocation, and network communications. Application log files, located in the Spark installation directory, will often show the complete details of a failure​.
* User code specific errors include syntax, serialization, data validation. Related errors can happen outside the code If a task fails due to an error, Spark can attempt to rerun tasks for a set number of retries.​ If all attempts to run a task fail, Spark reports an error to the driver and terminates the application. The cause of an application failure can usually be found in the driver event log. ​
* Spark enables configurable memory for executor and driver processes. ​​Executor memory and Storage memory share a region that can be tuned​.
* Setting data persistence by caching data is one technique used to improve application performance​​.
* The following code example illustrates configuration of executor memory on submit for a Spark Standalone cluster: **$ ./bin/spark-submit \​ --class org.apache.spark.examples.SparkPi \​ --master spark://<spark-master-URL>:7077 \​ --executor-memory 10G \​ /path/to/examples.jar \​1000**
* The following code example illustrates setting Spark Standalone worker memory and core parameters: **# Start standalone worker with MAX 10Gb memory, 8 cores​ $ ./sbin/start-worker.sh \​   spark://<spark-master-URL> \​ –-memory 10G –-cores 8**
* Spark assigns processor cores to driver and executor processes during application processing. Executors process tasks in parallel according to the number of cores available or as assigned by the application. ​
* You can apply the argument **‘--executor-cores 8 \’** to set executor cores on submit per executor. This example specifies eight cores.
* You can specify the executor cores for a Spark standalone cluster for the application using the argument ‘**‘--total-executor-cores 50’** followed by the number of cores for the application. This example specifies 50 cores.
* When starting a worker manually in a Spark standalone cluster, you can specify the number of cores the application uses by using the argument **‘--cores‘** followed by the number of cores. ​Spark’s default behavior is to use all available cores. ​