HADOOP

**0.CLUSTER SETUP:**

<https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-common/ClusterSetup.html>

400TB DATA  ( number of node+ no of core + memory + network config )

<https://data-flair.training/forums/topic/hadoop-cluster-hardware-planning-and-provisioning/>

**1. Performance Tuning:--**

<http://hadooptutorial.info/hadoop-performance-tuning/>

1) Memory Tuning

2) Improving IO Performance

3) Tuning the Number of Mapper or Reducer Tasks

4) Adjust Replication Factor

5) Writing a Combiner

6) Using Skewed Joins

7) Speculative Execution

**2. Number of Mappers in a Cluster:\_\_**

https://data-flair.training/forums/topic/how-many-mappers-run-on-the-cluster/

**2. Optimize Map-Reduce Jobs:--**

**1)** We can try to write Combiner along with a map and reduce functions if possible. This will reduce shuffling and the map-reduce job can be optimized.

**2)** For huge files in TB, we can try keeping a block size of 256MB or even 512MB.

**3)** Also, we can try to compress map output. Compress the intermediate output, which reduces the amount of data needed to shuffle between Mapper node and reducer node.

**4)** We should keep the number of mappers & reducers to the reasonable value.

Number of mappers:

How long are they (job) running for ? If they are only running for a few seconds on average, then we should see if there’s a way to have fewer mappers and make them all run longer, a minute or so, as a rule of thumb. The extent to which this is possible depends on the input format you are using.

**5)** Combiner: Using a combiner will reduce the amount of data transferred to each of the reducers, since combiner merges the output on the mapper side.

**6)** Number of reducers: Choose optimal number of reducers. If data size is huge, then one reducer is not a good idea. Also, setting the number of reducers to a high number, is not a good idea, since the number of reducers also determines the number of partitions on the mapper side.

**7)** Compress Mapper Output: Its recommended to compress the mapper outputs (determined by configuration: mapreduce.map.output.compress)

so that lesser data gets written to disk and gets transferred to reducers.

**3. Move data inside hadoop cluster/hdfs:**

We can **copy files** or directories between different **clusters** by using the **hadoop** distcp command. We must include a credentials file in our **copy** request so the source **cluster** can validate that we are authenticated to the source **cluster** and the target **cluster**.

The hadoop **distcp(distributed copy) command** is a tool used for large inter- and intra-cluster copying. It uses MapReduce to affect its distribution, error handling and recovery, and reporting. It expands a list of files and directories into input to map tasks, each of which will copy a partition of the files specified in the source list.

**4. Data STAGING environment?(**Hadoop distcp <src> <dist>)

A **staging** area, or landing zone, is an intermediate storage area used for data processing during the extract, transform and load (ETL) process. The data **staging** area sits between the data source(s) and the data target(s), which are often data warehouses, data marts, or other data repositories.

**5. MSCK repair Table?**

It recovers partitions and data associated with partitions. We can use this statement when we add partitions to the catalog. It is possible it will take some time to add all the partitions. If this operation timed out, it will be in an incomplete state where only a few partitions are added to the catalog. We should run the statement on the same table until all partitions are added.

MSCK REPAIR TABLE orders;

**6. FSCK**

Hadoop Admin Commands – FSCK, DFSAdmin

1. Hadoop fsck / fsck command is used to check the HDFS file system. ...
2. hadoop fsck / -files. It displays all the files in HDFS while checking. ...
3. hadoop fsck / -files -blocks. It displays all the blocks of the files while checking.
4. hadoop fsck -move

This command is used to move the corrupted files to a particular directory, by default it will move to the /lost+found directory.

**7. Speculative execution in Hadoop?**

In **Hadoop**, **Speculative Execution** is a process that takes place during the slower **execution** of a task at a node. In this process, the master node starts **executing** another instance of that same task on the other node. And the task which is finished first is accepted and the **execution** of others is stopped by killing that.

**8. RACK & Rack Awareness**

A **rack** is a collection of 30 or 40 nodes that are physically stored close together and are all connected to the same network switch. Network bandwidth between any two nodes in a rack is greater than bandwidth between two nodes on different racks. A Hadoop Cluster is a collection of racks.

**Rack Awareness in Hadoop** is the concept that chooses closer Data Nodes based on the **rack** information. By default, **Hadoop** installation assumes that all the nodes belong to the same **rack**. To improve network traffic while reading/writing **HDFS** files in large clusters of **Hadoop**.

**9. Fault tolerance in new hadoop 3.0**

**Hadoop 3.0** supports 2 or more Standby nodes to provide additional **fault tolerance** unlike **Hadoop 2.0** that supports only two NameNodes. **Fault tolerance** was limited in **Hadoop** 2.0 [for fault tolerance 2.0 was depending on replicas ] as **HDFS** could run only a single standby and a single active NameNode.

**10. Check-Pointing**

**Checkpointing** is basically a process which involves merging the fsimage along with the latest Edit log and creating a new fsimage for the namenode to possess the latest configured metadata of HDFS namespace . Now we can say this task can be performed by a Secondary Namenode or a Standby Namenode as well.

**11. FsImage**

The metadata files ([FsImage](https://datacadamia.com/db/hadoop/hdfs/fsimage) and [EditLog](https://datacadamia.com/db/hadoop/hdfs/log)) are central data structures of HDFS.

**FsImage** is a file stored on the OS file system that contains the complete directory structure (namespace) of the HDFS with details about the location of the data on the Data Blocks and which blocks are stored on which node.

The FsImage is stored as a file in the [NameNode](https://datacadamia.com/db/hadoop/hdfs/namenode)’s local file system.

The location is defined in [HDFS - Configuration (hdfs-site.xml)](https://datacadamia.com/db/hadoop/hdfs/conf).

**12. Edit-Logs**

The [NameNode](https://datacadamia.com/db/hadoop/hdfs/namenode) uses a [transaction log](https://datacadamia.com/data/property/transaction_log) called the EditLog to persistently record every change that occurs to [file system metadata](https://datacadamia.com/db/hadoop/hdfs/metadata).

EditLogs is a transaction log that records the changes in the HDFS file system or any action performed on the HDFS cluster such as addition of a new block, replication, deletion etc. It records the changes since the last FsImage was created, it then merges the changes into the FsImage file to create a new FsImage file.

The default directory of Hadoop log file is $HADOOP\_HOME/logs (i.e. log directory in Hadoop home directory).

**13. Safe-Mode in Hadoop**

**Safe Mode in hadoop** is a maintenance state of NameNode during which NameNode doesn't allow any changes to the file system. **Safe mode** is a mechanism of preventing modifications when the HDFS cluster is unstable, In this **mode HDFS** remains in the read-only **mode** preventing deletion, replication of the Data Blocks.

It loads the file system namespace from the last saved fsimage into its main memory and the edited log file. Applies/merges edits log file on fsimage and results in new file system namespace.

Why is NameNode in safe mode?

During start up, **NameNode** loads the filesystem state from fsimage and edits log file. It then waits for data nodes to report their blocks so that it does not prematurely start replicating the blocks though, enough replicas already exist in the cluster. During this time, **NameNode** stays in **safe mode**.

**14. Journal Node**

**JournalNode** is a daemon that enables[**high availability**](https://datacadamia.com/db/hadoop/hdfs/ha) **of** [**namenode**](https://datacadamia.com/db/hadoop/hdfs/namenode)**. Journal Nodes** are the ones which will perform the synchronisation activities between Active & Passive NameNode. Now imagine a situation where the **JournalNode** fails. The whole purpose of the High availability fails. Again, the **Journal Node** will become a single point of failure.

In order for the Standby node to keep its state synchronized with the Active node, both nodes communicate with a group of separate daemons called JournalNodes (JNs)

**15. Edge Node**

**15. High Availability**

The **high availability** feature in **Hadoop** ensures the **availability** of the **Hadoop** cluster without any downtime, even in unfavorable conditions like NameNode failure, DataNode failure, machine crash, etc. It means if the machine crashes, data will be accessible from another path.

To set up High Availability in a Hadoop cluster we have to use Zookeeper in all the nodes.

16.

HDFS

**0. HDFS TO AMAZON S3:**

Enterprise customers use Hadoop Distributed File System (HDFS) as their data lake storage repository for on-premises Hadoop applications. Customers are migrating their data lakes to AWS for a more secure, scalable, agile, and cost-effective solution.

For HDFS migrations where high-speed transfer rates are high, AWS offers the AWS Snowball Edge service. AWS Snowball Edge is to use an intermediary staging machine for the file transfer. <https://aws.amazon.com/snowball/>

How it works

In the [AWS Snowball console](https://console.aws.amazon.com/importexport/home), select your preferred device, either Snowball Edge Compute Optimized or Snowball Edge Storage Optimized. Create a job with an Amazon S3 bucket, select [Amazon Simple Notification Service (Amazon SNS)](https://aws.amazon.com/sns/) for tracking, and configure options like Amazon EC2 AMIs and a GPU. AWS prepares and ships the device to you, and you receive it in approximately 4-6 days. Once the device arrives, power it up and use AWS OpsHub to unlock it. Connect to your LAN. Use AWS OpsHub to manage the device, transfer data, or launch EC2 instances. When done, shut down and return the device to AWS. The shipping label automatically appears on the E Ink screen. When the device arrives at the AWS Region, any data stored in your on-board bucket(s) is moved to your S3 bucket and verified in about the same time it took you to load the device. All data is then securely erased from the device, and it is sanitized of any customer information.

**1. Fix corrupt HDFS file:**

<https://stackoverflow.com/questions/19205057/how-to-fix-corrupt-hdfs-files>

We can use:--

hdfs fsck /

to determine which files are having problems. We need to look through the output for missing or corrupt blocks. I normally get down to the meaningful output with:--

hdfs fsck / | egrep -v '^\.+$' | grep -v replica

which ignores lines with nothing but dots and lines talking about replication.

When I ran:--

bin/hadoop fsck / -delete

it listed the files that were corrupt or missing blocks.

Once we find a file that is corrupt:--

hdfs fsck /path/to/corrupt/file -locations -blocks -files

We can Use that output to determine where blocks might live. If the file is larger than my block size then it might have multiple blocks...

We can use the reported block numbers to go around to the datanodes and the namenode logs searching for the machine or machines on which the blocks lived. We can try looking for filesystem errors on those machines. Missing mount points, datanode not running, file system reformatted/reprovisioned. If we can find a problem in that way and bring the block back online that file will be healthy again.

Once we determine what happened and we cannot recover any more blocks, just use the:--

hdfs fs -rm /path/to/file/with/permanently/missing/blocks

command to get our HDFS filesystem back to healthy so we can start tracking new errors as they occur.

**2.Copy data HDFS to S3 ?**

If we want to move data from a Hadoop environment into an S3 bucket there is a very simple way to do it. It requires two steps:

STEP1: Create an S3 Bucket &&

STEP2: Use distcp utility to copy data from your hadoop platform to the S3 bucket created in step one.

STEP 1: Create an S3 Bucket

1. Sign in to the preview version of the [AWS Management Console](https://console.aws.amazon.com/).

2. Under Storage & Content Delivery, we need to choose S3 to open the Amazon S3 console.

3. From the Amazon S3 console dashboard, we have to choose Create Bucket.

4. In Create a Bucket, type a bucket name in Bucket Name.

5. Select the region we want to use

6. Click create

STEP 2: Move your data from Hadoop to the new S3 Bucket.

1. Open up a terminal session of the source hadoop system:
2. Use distcp to move data from Hadoop HDFS to the new S3 bucket

**3. Accessing HDFS files from Spark (with AVRO lib)**

I can access my avro files like this:

First, we need to include proper avro lib, in my case I can try:

spark-submit --packages com.databricks:spark-avro\_2.10:2.0.1 --class MyMain MyMain.jar

val df = sqlContext.read.format("com.databricks.spark.avro"). option("header", "true").load("/user/user1/writer\_test.avro") df.select("time").show()

**4. Read a file from HDFS in Spark:**

If we want to make it simple, then just can just use: sc.textFile("hdfs:/input/war-and-peace.txt")

Here is another solution [with absolute path]

sc.textFile("hdfs://nn1home:8020/input/war-and-peace.txt")

How did I find out nn1home:8020?

Just search for the file core-site.xml and look for xml element fs.defaultFS

Without mn1home:8020, it should be sc.textFile("hdfs:////input/war-and-peace.txt")

**5.Write a file to HDFS from Spark Scala?**

// Defining an Hello-world class

case class HelloWorld(message: String)

// Creating a dataframe with 1 partition:-

val df = Seq(HelloWorld("helloworld")).toDF().coalesce(1)

// Writing Dataframe as parquet file:-

df.write.mode(SaveMode.Overwrite).parquet(hdfs\_master + "user/hdfs/wiki/testwiki")

// Writing Dataframe as csv file:-

df.write.mode(SaveMode.Overwrite).csv(hdfs\_master + "user/hdfs/wiki/testwiki.csv")

**6.Read file from HDFS with Spark Scala ?**

// Reading parquet files into a Spark Dataframe

val df\_parquet = session.read.parquet(hdfs\_master + "user/hdfs/wiki/testwiki")

// Reading csv files into a Spark Dataframe

val df\_csv = sparkSession.read.option("inferSchema", "true").csv(hdfs\_master + "user/hdfs/wiki/testwiki.csv")

**7. INDEXING  in HDFS**

In Distributed file system like **HDFS**, **indexing** is different from the local file system. Here **indexing** and searching of data is done using the memory of the **HDFS** node where data is residing. The generated **index** files are stored in a folder in the directory where the actual data is residing.

**Indexes** are used to quickly locate data without having to search every row in a database table every time a database table is accessed. An **indexed file** is a computer **file** with an **index** that allows easy random access to any record given its **file** key. The key must be such that it uniquely identifies a record.

How is it done ?

This can be achieved very well using the Memory-Based Indexing on HDFS:

We cannot directly create an index on the distributed data. In order to do so various steps have to be followed as:

1. Copying of data from HDFS to a local file system,

2. Creating an index of the data present on the local file system, and

3. Finally storing the index files back to HDFS.

The same steps would be required for searches. But this approach is time-consuming and suboptimal, so instead, better option is to search our data using the memory of the HDFS node where data is residing

If we have a data file on HDFS residing inside a working directory, we need to create a folder inside the same working directory of HDFS where all the generated indexes will be stored.

To search the data now, search the indexes stored in HDFS. First, we must make the HDFS index files available in memory for searching. When we have the required index files available in the memory, we can directly perform a search on the index files.

**8. CDC - Change Data Capture**

**Change data capture** (CDC) is the process of **capturing changes** made at the **data** source and applying them throughout the enterprise. CDC minimizes the resources required for **ETL** ( extract, transform, load ) processes because it only deals with **data changes**. The goal of CDC is to ensure **data** synchronicity.

In databases, change data capture (**CDC**) is a set of software design patterns used to determine and track the data that has changed so that action can be taken using the changed data.

The source of **change data** for **change data capture** is the SQL Server transaction log. As inserts, updates, and deletes are applied to tracked source tables, entries that describe those **changes** are added to the log. The log serves as input to the **capture** process.

**9. SCD - Slow Changing Dimension**

<https://www.1keydata.com/datawarehousing/slowly-changing-dimensions.html>

A **Slowly Changing Dimension** (SCD) is a **dimension** that stores and manages both current and historical data over time in a data warehouse. It is considered and implemented as one of the most critical ETL tasks in tracking the history of **dimension** records.

**Slowly Changing Dimensions** (SCD) are the most commonly **used** advanced **dimensional** technique **used** in **dimensional** data warehouses. **Slowly changing dimensions** are **used** when we wish to capture the **changing** data within the **dimension** over time.

**10. Checkpointing in hadoop**

**Checkpointing** is basically a process which involves merging the fsimage along with the latest edit log and creating a new fsimage for the namenode to possess the latest configured metadata of HDFS namespace . Now one can say this task can be performed by a Secondary Namenode or a Standby Namenode as well

SPARK

**1.** **Spark job workflow? (details abt spark job process)**

>When the command is triggered from the client, the Driver receive it and convert the command for creating and transforming RDDs

> Then it Creates RDD from the data.

> DAG is initiated behind the screen (in Driver Program of Master Node)

> Transformation happens (maybe series of transformations)

> Nothing executed in the cluster only DAG gets more constructed

> Perform Action (so spark can materialize the output)

> Job is created

> Job is assigned to DAG scheduler

> DAG divide the plan into stages (and make an output asset of stages which is actually a **physical execution plan**) and filially set of task

> Which is then forwarded to the Task scheduler..

> Task Scheduler launches each task one by one in executer via Cluster Manager (who allocate resource)

> Before executing the task, all executors register themselves with the Driver program

> Task is executed (Data + Computation) in executer (read, process and write)

> While the task is being executed, it is monitored by the Driver program

> Then Report the result to the Driver program and put the result on Memory or Disc or Cache.

> When the task is over, all the resources allocated to the executor are de-allocated.

> Then we can Create a Data Frame on top of RDD.

> We can Perform queries on Data Frame.

> We can also Run SQL queries on Dataframe.

**2. Scheduling a spark job?**

Spark applications run as independent sets of processes on a cluster, coordinated by the SparkContext object in our main program (called the *driver program*).

Specifically, to run on a cluster, the SparkContext can connect to several types of *cluster managers* (Spark’s own standalone cluster manager, Mesos or YARN), which allocate resources across applications. 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.

<https://spark.apache.org/docs/2.2.0/job-scheduling.html>  (see for details)

**2. Spark job fail: (troubleshooting)**

[https://docs.informatica.com/big-data-management/big-data-management/h2l/big-data-management-10-2-1-performance-tuning-and-sizing-guideli/big-data-management-1021-performance-tuning-and-sizing-guideline/tune-the-spark-engine/troubleshooting-spark-job-failures.html#:~:text=A%20job%20fails%20due%20to%](https://docs.informatica.com/big-data-management/big-data-management/h2l/big-data-management-10-2-1-performance-tuning-and-sizing-guideli/big-data-management-1021-performance-tuning-and-sizing-guideline/tune-the-spark-engine/troubleshooting-spark-job-failures.html#:~:text=A%20job%20fails%20due%20to%20Spark%20speculative%20execution%20of%20tasks.&text=Set%20the%20value%20to%20false,information%20to%20isolate%20the%20cause.)

[%20speculative%20execution%20of%20tasks.&text=Set%20the%20value%20to%20false,information%20to%20isolate%20the%20cause.](https://docs.informatica.com/big-data-management/big-data-management/h2l/big-data-management-10-2-1-performance-tuning-and-sizing-guideli/big-data-management-1021-performance-tuning-and-sizing-guideline/tune-the-spark-engine/troubleshooting-spark-job-failures.html#:~:text=A%20job%20fails%20due%20to%20Spark%20speculative%20execution%20of%20tasks.&text=Set%20the%20value%20to%20false,information%20to%20isolate%20the%20cause.)

**Why a job fails**:--  it becomes very difficult when **Spark applications** start to **slow** down or **fail**. Sometimes a well-tuned **application** might **fail** due to a data change, or a data layout change. Sometimes an **application** which was running well so far, starts behaving badly due to resource starvation. In Spark, stage failures happen when there's a problem with processing a Spark task. These failures can be caused by hardware issues, incorrect Spark configurations, or code problems. It could be an OOM problem as well.

 Some of the most common causes of OOM are:

* Incorrect usage of Spark
* High concurrency
* Inefficient queries
* Incorrect configuration

**How to identify:--** If we want to check in general if there are any failures from the side of **Spark Launcher**, we can exit the application started by Jar with exit code different than 0 using kind of **System.exit(1)**, if detected a job failure. The **Process** returned by **SparkLauncher::launch** contains the **exitValue** method, so you can detect if it failed or not.

**Where we can find it:--** We can see any job information using **JavaSparkStatusTracker**. For active jobs nothing additional should be done, since it has the ".getActiveJobIds" method.

The JobExecutionStatus can be one of RUNNING, SUCCEEDED, FAILED, UNKNOWN;

We can always go to spark history server and click on our job id to get the job details. If we are using yarn then we can go to the resource manager web UI to track your job status.

**3. Optimize a Spark Job:**

[Key factors to consider when optimizing Spark Jobs](https://medium.com/datakaresolutions/key-factors-to-consider-when-optimizing-spark-jobs-72b1a0dc22bf)

1. Data Serialization. ...
2. Broadcasting. ...
3. Avoid UDF and UDAF. ...
4. Data locality. ...
5. Dynamic allocation. ...
6. Garbage collection. ...
7. Executor Tuning. ...
8. Parallelism.

**3. Spark Job submit components / parameters ?**

[All about spark submit + Executor & Driver memory](https://sparkbyexamples.com/spark/spark-submit-command/#driver-executor-resources)

spark.master            spark://5.6.7.8:7077

spark.executor.memory   512m

spark.eventLog.enabled  true

spark.serializer        org.apache.spark.serializer.KryoSerializer

This topic describes how to configure spark-submit parameters in E-MapReduce.

Cluster Configuration

Software configuration  
E-MapReduce V1.1.0

Hadoop V2.6.0

* + Spark V1.6.0

Hardware configuration

* + *Master node*
    - 8-core, 16 GB memory, and 500 GB storage space (ultra disk)
    - 1 set
  + *Worker node*
    - 8-core, 16 GB memory, and 500 GB storage space (ultra disk)
    - 10 sets
  + Total: 8-core 16 GB (Worker) × 10 + 8-core 16 GB (Master)
  + **Notice** Only CPU and memory resources are calculated when a job is submitted. Therefore, the disk size is not included in total resource calculation.
  + Total resources available for YARN: 12-core 12.8 GB (worker) × 10
  + **Notice** By default, cores available for YARN = number of cores × 1.5, and memory available for YARN = node memory × 0.8.

**3. Launching a Spark Program**

spark-submit is the single script used to submit a spark program and launch the application on the cluster. There are multiple options through which spark-submit script can connect with different cluster managers and control on the number of resources the application gets. For a few cluster managers, spark-submit can run the driver within the cluster like in YARN on worker nodes while for others it runs only on local machines.

**4. DAG vs Lineage? When to use what?**

Lineage graph:

As we know, whenever a series of transformations are performed on an [RDD](https://data-flair.training/blogs/apache-spark-rdd-tutorial/), they are not evaluated immediately, but lazily([Lazy Evaluation](https://data-flair.training/blogs/apache-spark-lazy-evaluation/)). When a new RDD has been created from an existing RDD, that new RDD contains a pointer to the parent RDD. Similarly, all the dependencies between the RDDs will be logged in a graph, rather than the actual data. This graph is called the lineage graph. In the case of data loss, this lineage graph is used to rebuild the data. I need say spme action… for example. Reduce, count,collect etc

Directed Acyclic Graph(DAG):---

DAG shows the different stages of a spark job. DAG is a combination of Vertices and Edges. In DAG, vertices represent the RDDs and the edges represent the Operation to be applied on RDD. Every edge in DAG is directed from earlier to later in a sequence. When we call an action, the created DAG is submitted to DAG Scheduler which further splits the graph into the stages of the task.

[No they cannot be used interchangeably, because workings are different. Lineage graph deals with RDDs so it is applicable up-till *transformations****,***Whereas, DAG shows the complete task, ex: *transformation + Action****.*]**

**5. Spark window function:**

Window (or windowing or windowed) functions perform a calculation over a set of rows. It is an important tool to do statistics. Most Databases support Windows functions. The spark from version 1.4 starts supporting Windows functions.

Window functions are often used to avoid needing to create an auxiliary dataframe and then joining on that. Window Functions helps us to **compare current rows** with other rows in the same dataframe, calculating **running** **totals**, **sequencing** of events and **sessionization** of transactions, etc.

**6. CATALYST Optimizer in Spark SQL:**

At the core of [Spark SQL](https://databricks.com/glossary/what-is-spark-sql) is the Catalyst optimizer, which leverages advanced programming language features (e.g. Scala’s pattern matching and quasi quotes) in a novel way to build an extensible query optimizer.

Catalyst is based on functional programming constructs in Scala and designed with these key two purposes:

* Easily add new optimization techniques and features to Spark SQL
* Enable external developers to extend the optimizer (e.g. adding data source-specific rules, support for new data types, etc.)

Catalyst contains a general library for representing trees and applying rules to manipulate them. On top of this framework, it has libraries specific to relational query processing (e.g., expressions, logical query plans), and several sets of rules that handle different phases of query execution: analysis. Catalyst also offers several public extension points, including external data sources and user-defined types. As well, Catalyst supports both rule-based and cost-based optimization.

**7. CHECKPOINTING:-**

The need with [the Spark Streaming](http://data-flair.training/blogs/apache-spark-streaming-comprehensive-guide/) application is that it should be operational 24/7. Thus, the system should also be fault-tolerant. If any data is lost, the recovery should be speedy. Spark streaming accomplishes this using checkpointing.

So, *Checkpointing* is a process to truncate [RDD lineage graphs](http://data-flair.training/blogs/directed-acyclic-graph-dag-in-apache-spark/). It saves the application state timely to reliable storage ([HDFS](http://data-flair.training/blogs/comprehensive-hdfs-guide-introduction-architecture-data-read-write-tutorial/)). As the driver restarts the recovery takes place.

There are two types of data that we checkpoint in Spark:

1. Metadata checkpointing    2. Data Checkpointing

**7. Sliding window**

**Spark** streaming leverages the advantage of windowed computations in **spark**. It offers to apply transformations over a **sliding window** of data. As the window slides over a source DStream, the source RDDs that fall within the **window** are combined.

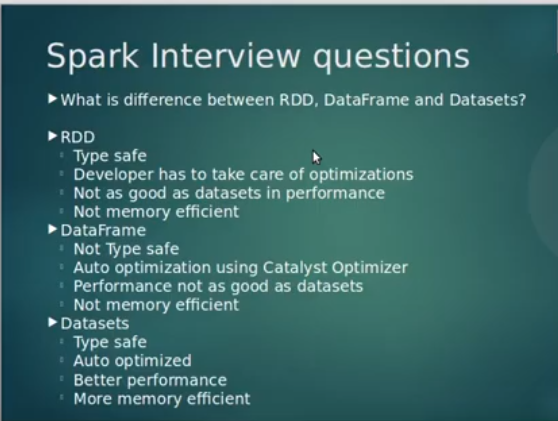
For example if I set batch **interval** 5 seconds - **Spark Streaming** will collect data for 5 seconds and then kick out calculation on RDD with that data. ...

**sliding interval** - is amount of time in seconds for how much the window will shift.

**8. Shuffle read and Shuffle writes in spark:**

Shuffling means the reallocation of data between multiple Spark stages. "Shuffle Write" is the sum of all written serialized data on all executors before transmitting (normally at the end of a stage) and "Shuffle Read" means the sum of reading serialized data on all executors at the beginning of a stage.

**9. RDD vs DF vs DS ? When to use what ?**



<https://databricks.com/glossary/spark-api>

**RDD** – An RDD stands for Resilient Distributed Datasets. It is a Read-only partition collection of records. RDD is the fundamental data structure of Spark. It allows us to perform in-memory computations on large clusters in a [fault-tolerant](http://data-flair.training/blogs/apache-spark-streaming-fault-tolerance/) manner. **RDD**- When we want low-level transformation and actions, we **use RDDs**. Also, when we need high-level abstractions then we **use RDDs**.

**Dataframe** – Data organized under named columns. For example a table in a relational database. It is an immutable distributed collection of data.

We **use dataframe** when we need a high level of abstraction and for unstructured data, such as media streams or streams of text.

The DataFrame API introduces the concept of a schema to describe the data, allowing Spark to manage the schema and only pass data between nodes, in a much more efficient way than using Java serialization. DataFrame in Spark allows us to impose a structure onto a distributed collection of data, allowing higher-level abstraction.

**Dataset**– A Dataset is a distributed collection of data. Dataset is a new interface added in **Spark** 1.6 that provides the benefits of RDDs. **Spark Dataset** provides both type safety and object-oriented programming interface. Dataset can be constructed from JVM objects and then manipulated using complex functional transformations.

Datasets in Apache Spark are an extension of DataFrame API which provides type-safe, object-oriented programming interface. Dataset takes advantage of Spark’s Catalyst optimizer by exposing expressions and data fields to a query planner.

Dataset has built-in encoders. Basically, **encoders** are what convert our data between JVM objects and Spark SQL's specialized internal (tabular) representation. They're required by all **Datasets**! **Encoders** are highly specialized and optimized code generators that generate custom bytecodes for serialization and deserialization of our data.

<https://techvidvan.com/tutorials/apache-spark-dataframe-vs-datasets/>

**10.PARTITIONING in Spark helps achieve more parallelism?**

[Apache Spark](https://www.dezyre.com/apache-spark-tutorial/pyspark-tutorial) allows developers to run multiple tasks in parallel across hundreds of machines in a cluster or across multiple cores on a desktop. Under the hood, these RDDs are stored in partitions and operated in parallel.

[Resilient Distributed Datasets](https://www.dezyre.com/article/working-with-spark-rdd-for-fast-data-processing/273) are collections of various data items that are so huge in size, that they cannot fit into a single node and have to be partitioned across various nodes. Spark automatically partitions RDDs and distributes the partitions across different nodes. A partition in spark is an atomic chunk of data (logical division of data) stored on a node in the cluster. Partitions are basic units of parallelism in Apache Spark. RDDs in Apache Spark are collections of partitions.

Creating a Partition in Spark:--

simple example that creates a list of 10 integers with 3 partitions –

integer\_RDD = sc.parallelize (range (10), 3)

Having too large a number of partitions or too few - is not an ideal solution. The number of partitions in spark should be decided thoughtfully based on the cluster configuration and requirements of the application.

val rdd= sc.textFile (“file.txt”, 5)

Types of Partitioning in Apache Spark:--

1. **Hash Partitioning in Spark:-** Hash Partitioning attempts to spread the data evenly across various partitions based on the key. Object.hashCode method is used to determine the partition in Spark as partition = key.hashCode () % numPartitions.
2. **Range Partitioning in Spark:-** In range partitioning method, tuples having keys within the same range will appear on the same machine. Keys in a range partitioner are partitioned based on the set of sorted range of keys and ordering of keys.

RDDs can be created with specific partitioning in two ways –

1. Providing explicit partitioner by calling partitionBy method on an RDD,
2. Applying transformations that return RDDs with specific partitioners. Some operation on RDDs that hold to and propagate a partitioner are-- Join, LeftOuterJoin, RightOuterJoin, groupByKey, reduceByKey, foldByKey, Sort, partitionBy, foldByKey

11.**SparkContext?**

SparkContext is the entry point of Spark functionality. A SparkContext represents the connection to a Spark cluster and can be used to create RDDs, accumulators and broadcast variables on that cluster. The most important step of any Spark driver application is to generate SparkContext. It allows our Spark Application to access Spark Cluster with the help of the Resource Manager. The resource manager can be [Spark Standalone](http://data-flair.training/blogs/apache-spark-cluster-managers-tutorial/), [YARN](http://data-flair.training/blogs/hadoop-yarn-tutorial/), [Apache or Mesos.](http://data-flair.training/blogs/apache-mesos-tutorial-learn-mesos/)

*Note that we can create only one SparkContext per JVM. We should stop existing Sparkcontext (using stop()) before creating a new one.*

<https://data-flair.training/blogs/learn-apache-spark-sparkcontext/>

**12.Create SparkContext Class?**

If we want to create SparkContext, first SparkConf should be made. The SparkConf has a configuration parameter that our Spark driver application will pass to SparkContext. Some of these parameters define properties of Spark driver application. While some are used by Spark to allocate resources on the cluster, like the number, memory size, and cores used by executors running on the worker nodes.

Once the SparkContext is created, it can be used to [create RDDs](http://data-flair.training/blogs/how-to-create-rdds-in-apache-spark/), broadcast variables, and accumulators, ingress Spark service and run jobs. All these things can be carried out until SparkContext is stopped.

**13. Session vs Context ?**

Prior to [Spark](http://data-flair.training/blogs/apache-spark-introduction-spark-comprehensive-tutorial/) 2.0.0 **SparkContext** was used as a channel to access all spark functionality. The spark driver program uses spark context to connect to the cluster through a resource manager ([YARN](http://data-flair.training/blogs/category/yarn/) or Mesos).sparkConf is required to create the spark context object, which stores configuration parameters like appName. Its object ***sc***is default available in spark-shell and it can be programmatically created using SparkContext class.

When programming wither with Scala, PySpark or Java, first we need to create a SparkConf instance by assigning app name and setting master by using the SparkConf static methods setAppName() and setMaster() respectively and then pass SparkConf object as an argument to SparkContext constructor to create Spark Context.

Val sparkConf = new SparkConf().setAppName("sparkbyexamples.com").setMaster("local[1]")

val sparkContext = new SparkContext(sparkConf )

SparkContext.getOrCreate(sparkConf)

val rdd = sc.textFile("/src/main/resources/text/alice.txt")

**SparkSession** provides a single point of entry to interact with underlying Spark functionality and allows programming Spark with [DataFrame](http://data-flair.training/blogs/apache-spark-dataframe-tutorial/) and Dataset APIs. All the functionality available with sparkContext are also available in sparkSession.

Creating a spark session in SCALA

val spark = SparkSession.builder()

      .master("local[1]")

      .appName("SparkByExamples.com")

      .getOrCreate();

**14. TRANSFORMATION**

* Narrow transformations
* Wide transformations

**Narrow transformations:**

Narrow transformations are the result of map, filter and in which data to be transformed

id from a single partition only, i.e. it is self-sustained.

An output RDD has partitions with records that originate from a

single partition in the parent [RDD](http://data-flair.training/blogs/rdd-in-apache-spark/).

**Wide Transformations:**

Wide transformations are the result of groupByKey and reduceByKey.

The data required to compute the records in a single partition may

reside in many partitions of the parent RDD.

Wide transformations are also called shuffle transformations as they may or may not depend on a shuffle.

**14. BROADCAST && Accumulator**

**What is the need of a Shared variable in Apache Spark?**

Sometimes, a variable needs to be shared across **tasks**, or between **tasks** and the driver program. Spark supports two **types** of shared variables: broadcast variables, which can be used to cache a **value** in memory on all nodes, and accumulators, which are variables that are only “added” to, such as counters and sums.

Shared variables are the variables that are required to be used by many functions & methods in parallel. Shared variables can be used in parallel operations.

Broadcast variables: (also called read-only variable)

Variables allow the programmers to keep a read-only variable cached on each machine. *Broadcast Variables, d*espite shipping a copy of it with tasks. We can use them, for example, to give a copy of a large input dataset in an efficient manner to every node. It helps to reduce communication cost. Hence, creating broadcast variables explicitly is useful in some cases, like while tasks across multiple stages need the same data. While caching the data in the deserialized form is important.

Example:

If we have a big size array that is accessed from Spark Closures, for example some reference data, this array will be shipped to each spark node with closure.

For example, if we have 10 nodes in a cluster with 100 partitions (10 partitions per node), this Array will be distributed at least 100 times (10 times to each node).

If I use a broadcast variable it will be distributed once per node using an efficient manner.

val array: Array[Int] = ??? // some big-size array

val broadcasted = sc.broadcast(array)

val rdd: RDD[Int] = ??? // my array

rdd.map(i => array.contains(i))

rdd.map(i => broadcasted.value.contains(i))

Broadcast variables are:

Immutable, Distributed i.e. broadcasted to the cluster, Fit in memory.

Syntax to create Broadcast variable:

SparkContext.broadcast(Value)

**Accumulators: (for data write purpose)**

As its name suggests, the accumulator's main role is to accumulate values. The accumulator is variables that are used to implement counters and sums. Spark provides accumulators of numeric type only. The user can create named or unnamed accumulators. These accumulator variables can only be used when a user wants to perform associative or commutative operations on the data. The accumulators can be created with or without a name. If the accumulators are created with a name, they can be viewed in Spark’s UI which will be useful to understand the progress of running stages.

The accumulators are created using an initial value v. by calling SparkContext.accumulator(v)

Unlike Broadcast Variables, accumulators are writable. However, written values can only be read in the driver program. It’s why accumulators work pretty well as data aggregators.

Syntax to create accumulator:

SparkContext.accumulator(orgnlValue)

Example:

To create a numeric accumulator, we can call SparkContext.longAccumulator() or SparkContext.doubleAccumulator() to accumulate the values of Long or Double type.

scala> val a=sc.longAccumulator("Accumulator")

scala> sc.parallelize(Array(2,5)).foreach(x=>a.add(x))

scala> a.value

**15. Core engine of sparks runs on ?**

Spark SQL, spark Streaming, MLlib, GraphX

**Spark core API** : R, SQL, Python, Scala, Java

**16.** **Spark SQL work?**

**Spark SQL** integrates relational data processing with the functional programming API of **Spark**. It gives a programming abstraction called Dataframe and allows to query on different nodes of a cluster (acts as a distributed querying engine). It supports querying using either the **SQL** or Hive Query Language (HQL).

**17.STATIC Vs Dynamic resource allocation?**

In static memory allocation once the memory is allocated, the memory size is fixed and it cannot be changed. The memory is not reusable. While in dynamic memory allocation, once the memory is allocated, the memory size can be changed as per we need.

**18. Resource allocation:--**

There are two types of resource allocation: Static and Dynamic.

By default, resources in Spark are allocated statically. It can lead to some problematic cases. Above all, it's difficult to estimate the exact workload. It can produce 2 situations: underuse & starvation of resources. The first one means that too many resources were reserved but only a small part of them is used. The second case means that one process takes all available resources and prevents other applications to start.

Dynamic resource allocation is one of the solutions for the above problems. It adapts resources used in processing according to the workload. This feature is controlled by spark.dynamicAllocation.enabled configuration entry. And thanks to other parameters we can specify the initial/minimal/maximal number-of-executors (spark.dynamicAllocation.(initialExecutors|minExecutors|maxExecutors).

<http://site.clairvoyantsoft.com/understanding-resource-allocation-configurations-spark-application/>

**20. Spark works with YARN?**

There are two deploy modes that can be used to launch Spark applications on YARN. In **cluster mode**, the Spark driver runs inside an application master process which is managed by YARN on the cluster, and the client can go away after initiating the application. In **client mode**, the driver runs in the client process, and the application master is only used for requesting resources from YARN.

<https://spark.apache.org/docs/latest/running-on-yarn.html>

**21.** **Persistence VS Caching in Spark**?

Spark [RDD](http://data-flair.training/blogs/rdd-in-apache-spark/) persistence is an optimization technique in which saves the result of RDD evaluation. Using this we save the intermediate result so that we can use it further if required. It reduces the computation overhead.

We can make persisted RDD through cache() and persist() methods. When we use the cache() method we can store all the RDD in-memory. We can persist the RDD in memory and use it efficiently across parallel operations.

The difference between cache() and persist() is that using cache() the default storage level is MEMORY\_ONLY while using persist() we can use various storage levels (memory only, memory and disk, disk only, memory with serialization, .....)

cache() is the same as calling persist() with the default storage level. The default **persist()** will store the data in the JVM heap as unserialized objects. When you write data to a disk, that data is also always serialized.

<https://data-flair.training/blogs/apache-spark-rdd-persistence-caching/>

22. **Storage levels of Persisted RDDs**

a. MEMORY\_ONLY

b. MEMORY\_AND\_DISK

c. MEMORY\_ONLY\_serialized

d. MEMORY\_AND\_DISK\_serialized

e. DISK\_ONLY

23. **Need of Persistence in Apache Spark**

In Spark, we can use some RDD’s multiple times. Honestly, we repeat the same process of RDD evaluation each time it is required or brought into action. This task can be time and memory consuming, especially for iterative algorithms that look at data multiple times. To solve the problem of repeated computation the technique of persistence came into the picture.

24. **Map vs Flatmap (narrow VS wide)**

## The **map()** transformation takes in a function and applies it to each element in the RDD and the result of the function is a new value of each element in the resulting RDD. The **flatMap()** is used to produce multiple output elements for each input element.

**OR**

## Spark map function (narrow transformation) expresses a one-to-one transformation. It transforms each element of a collection into one element of the resulting collection. While Spark flatMap (wide transformation) function expresses a one-to-many transformation. It transforms each element to 0 or more elements.

**25. ADD a new COLUMN to DataFrame**

[adding a new column in an existing data frame in scala](https://sparkbyexamples.com/spark/spark-add-new-column-to-dataframe/#:~:text=Using%20withColumn()%20to%20add%20a%20new%20column&text=withColumn()%20function%20takes%20two,the%20column%20in%20Column%20type.&text=Here%2C%20we%20have%20added%20a,an%20existing%20column%20%E2%80%9C%20Salary%20%E2%80%9C.)

We can add a new column or multiple columns to a Spark DataFrame using withColumn(), select() and lit() functions and with Scala.

**26. PERFORMANCE TUNING**

<https://www.xenonstack.com/blog/apache-spark-optimisation/>

There may be good results of Spark performance tuning if done properly. likewise:

* By terminating jobs those run long.
* To ensure that jobs are on an accurate execution engine.
* By using all resources in an effective manner (dynamic resources allocation)
* By enhancing the performance time of the system.

\*\*To optimize a Spark application, we should always start with data serialization. It plays a vital role in the performance of any distributed application.  Also, it is a most important key aspect of Apache Spark performance tuning.

**\*\* Memory tuning.**

**\*\* Garbage collection tuning.**

**\*\* Using Accumulators/Broadcast variables**

\*\* Hive Bucketing Performance.

In the meantime of data processing a lot of shuffle happens. More is the shuffles, slower are the things. shuffle is normally required in Wide transformations. People who are not aware, spark has two kinds of transformation: Narrow and wide. Narrow Transformation output is generated using only one partition data. In wide transformation, data from multiple partitions may be required. Eg map is narrow transformation and groupByKey is wide transformation.

**1. Define Right number of Executors and Executor cores** -

 We can change the number of executors while submitting a job or while starting a shell using property  --num-executors. After this job is not done. we need to also define the number of executor cores. Executor core defines the number of concurrent tasks each executor can run. If we give executors cores very high, then spark will create a very high number of tasks for each executor. These tasks will compete with each other for resources and It will reduce data I/O throughput.

Eg. spark-shell --num-executers 15 --executor-cores 5 --master yarn test.jar com.test.Example

2. **Changing No of Partitions** - No of tasks created in a stage is equal to no of partitions of data. So having a very huge number of partitions can reduce our cluster's performance. This is a very common problem when we run spark code locally. If our data has a huge number of partitions , we will see our code running very slowly. change the number of partitions using **coalesce** method. If we are using spark data frames, then we need to use a repartition method.

3. **Use Broadcast Variables** - This is like Distributed cache in Hadoop. We can share data across worker nodes in read only way. If used intelligently , we can optimize our code with this.

4. **Caching your Data** - If you are doing a lot of exploration on a single data frame, caching that data will speed up any execution that you try on that data frame.

5. **Prefer ReduceByKey than groupByKey** - If I explain it in Hadoop terms, then reduceByKey is reduced operation with combiner and groupByKey is reduced without combiner. So when data shuffle happens during these operations as these are wide transformations, more data transfer happens in case of groupByKey. so it is slower compared to reduceByKey.

6. **flatmap-join-groupBy vs cogroup -** Use cogroup wherever possible, because it has better performance than flatmap-join-groupBy pattern. It avoids extra overheads of packing and unpacking of data.

**27. R**[**e-Partition() vs Coalesce()**](https://stackoverflow.com/questions/31610971/spark-repartition-vs-coalesce)

[We use Repartition on an RDD to increase & decrease it’s number of partitions && we use coalesce to decrease it’s partitions…]

repartition is used to increase or decrease the RDD, DataFrame, Dataset partitions whereas Spark coalesce is used to only decrease the number of partitions in an efficient way.

Both are wide transformations. The repartition() algorithm does a full shuffle of the data and creates equal sized partitions of data. Coalesce() combines existing partitions to avoid a full shuffle.

We need to keep in mind that repartitioning our data is a fairly expensive operation. Spark also has an optimized version of repartition() called coalesce() that allows us to avoid data movement, but only if we are decreasing the number of RDD partitions.

One difference I get is that with repartition() the number of partitions can be increased/decreased, but with coalesce() the number of partitions can only be decreased.

What if the partitions are spread across multiple machines and coalesce() is running, how can it avoid data movement?

It avoids a full shuffle. If it's known that the number is decreasing then the executor can safely keep data on the minimum number of partitions, only moving the data off the extra nodes, onto the nodes that we kept.

For example if we have 640 MB file and running it on Hadoop version 2, creates 5 partitions with each consists on 128 MB blocks (5 blocks \* 128 MB = 640 MB). If we repartition to 10 then it creates 2 partitions for each block.

**28. Connect Spark to MongoDB?(plugins)**

MongoDB provides us a plugin called the mongo-spark-connector, which will help us connect MongoDB and Spark without any drama at all. We just need to provide the MongoDB connection URI in the SparkConf object, and create a ReadConfig object specifying the collection name. It might sound complicated right now, but once we look at the code, we will understand how extremely easy it is.

<https://medium.com/@contactsunny/connect-apache-spark-to-your-mongodb-database-using-the-mongo-spark-connector-f826ffa0f991>

**29. BroaDCast join/Map-side-join**

Spark SQL uses **broadcast join** instead of hash join to optimize join queries when the size of one side data is below [spark.sql.autoBroadcastJoinThreshold](https://jaceklaskowski.gitbooks.io/mastering-spark-sql/spark-sql-properties.html#spark.sql.autoBroadcastJoinThreshold).

Broadcast join can be very efficient for joins between a large table with relatively small tables (dimensions) that could then be used to perform a **star-schema join**. It can avoid sending all data of the large table over the network.

We can use [broadcast](https://jaceklaskowski.gitbooks.io/mastering-spark-sql/spark-sql-functions.html#broadcast) function or SQL’s [broadcast hints](https://jaceklaskowski.gitbooks.io/mastering-spark-sql/spark-sql-hint-framework.html#broadcast-hints) to mark a dataset to be broadcast when used in a join query.

**30.Structured\_semi-structured\_unstructured Data**

Spark added different data sources like CSV, JSON, Avro, Parquet and many more. Based on the data source you choose, you may need a third party dependency and Spark can read and write all these files from/to windows(using Uinutils), Linux, HDFS, S3, Azure, GCP.

Unstructured data:- Text file formats are considered unstructured data. In order to process text files use [spark.read.text()](https://sparkbyexamples.com/spark/spark-read-text-file-rdd-dataframe/) and [spark.read.textFile()](https://sparkbyexamples.com/spark/spark-read-text-file-rdd-dataframe/)

Semi-Structured data:-

CSV and TSV is considered as Semi-structured data and to process CSV file, we should use [spark.read.csv()](https://sparkbyexamples.com/spark/spark-read-csv-file-into-dataframe/)

XML and JSON file format is considered semi-structured data as the data in the file can represent as a string, integer, arrays e.t.c but without explicitly mentioning the data types.

Processing JSON files in spark can be done using  [spark.read.json("path")](https://sparkbyexamples.com/spark/spark-read-and-write-json-file/) or [spark.read.format("json").load("path"](https://sparkbyexamples.com/spark/spark-read-and-write-json-file/)). Parsing unstructured and semi-structured data to DataFrame and Dataset is very slow.

*Structured data:-*

Avro and Parquet file formats are considered structured data as these can maintain the structure/schema of the data along with its data types.

avro() function is not provided in Spark DataFrameReader  hence, we should use DataSource format as “avro” or [org.apache.spark.sql.avro](https://sparkbyexamples.com/spark/read-write-avro-file-spark-dataframe/) and load() is used to read the Avro file. We need to pass the HDFS path as an argument to the load function.

DataFrameReader provides a parquet() function ([spark.read.parquet](https://sparkbyexamples.com/spark/spark-read-write-dataframe-parquet-example/)) to read the parquet files and creates a Spark DataFrame.

**31. Nice spark Site with spark core, sql, streaming, data read&write with examples**...

<https://sparkbyexamples.com/>

**32.GroupBykey, ReduceBykey, AggregateByKey, Partitionby() :**

<https://learndbigdata.com/spark-groupbykey-reducebykey-aggregatebykey/>

**Groupbykey:**

In **Spark**, the **groupByKey** function is a frequently used transformation operation that performs shuffling of data. It receives key-value pairs (K, V) as an input, groups the values based on key and generates a dataset of (K, Iterable ) pairs as an output. GroupByKey can cause out of disk problems as data is sent over the network and collected on the reduced workers.

When we perform **groupby**() on **Spark** Dataframe, it returns RelationalGroupedDataset object which contains below aggregate functions. count() - Returns the count of rows for each group. mean() - Returns the mean of values for each group. max() - Returns the maximum of values for each group.

**ReducebyKey:**

reduceByKey() is a transformation which operates on pairRDD (which contains Key/Value). Data are combined at each partition, only one output for one key at each partition to send over the network. reduceByKey required combining all your values into another value with the exact same type.

> PairRDD contains tuples, hence we need to pass the function that operator on tuple instead of each element.

> It merges the values with the same key using associative reduce function.

> It is a wide operation because data shuffles may happen across multiple partitions.

> It merges data locally before sending data across partitions to optimize data shuffling.

> It functions as an input which has two parameters of the same type (values associated with the same key) and one element output of the input type(value).

**Aggregatebykey:**

**Spark aggregateByKey** function aggregates the values of each key, using given combine functions and a neutral “zero value” The **aggregateByKey** function aggregates values for each key and and returns a different type of value for that key

**PartitionBy:**

**partitionBy**() is a DataFrameWriter method that specifies if the data should be written to disk in folders. By default, **Spark** does not write data to disk in nested folders. Memory partitioning is often important independent of disk partitioning.

**Combinebykey:**

Internally **spark combineByKey** function efficiently combines the values of a PairRDD partition by applying aggregation function. The main objective of **combineByKey** transformation is transforming any PairRDD[(K,V)] to the RDD[(K,C)] where C is the result of any aggregation of all values under key K

**Groupbykey VS Reducebykey:**

groupByKey() is just to group our dataset based on a key value. It will result in data shuffling when RDD is not already partitioned. reduceByKey() is something like grouping + aggregation. ... aggregateByKey() is logically the same as reduceByKey() but it lets us return results in different types.

**33. Speculative execution in spark?**

**Speculative execution** of tasks is a health-check procedure that checks for tasks to be speculated, i.e. running slower in a stage than the median of all successfully completed tasks in a taskset (FIXME the setting). Such slow tasks will be re-submitted to another worker. ... It executes periodically every **spark**.

**34. Executor & Driver Memory**

Every **spark** application has the same fixed heap size and fixed number of cores for a **spark executor**. The heap size is what is referred to as the **Spark executor memory** which is controlled with the **spark**.**executor**.**memory** property of the –**executor**-**memory** flag. Every **spark** application will have one **executor** on each worker node.

How is spark executor memory calculated?

Number of available **executors** = (total cores/num-cores-per-**executor**) = 150/5 = 30. Leaving 1 **executor** for ApplicationManager => --num-**executors** = 29. Number of **executors** per node = 30/10 = 3. **Memory** per **executor** = 64GB/3 = 21GB.

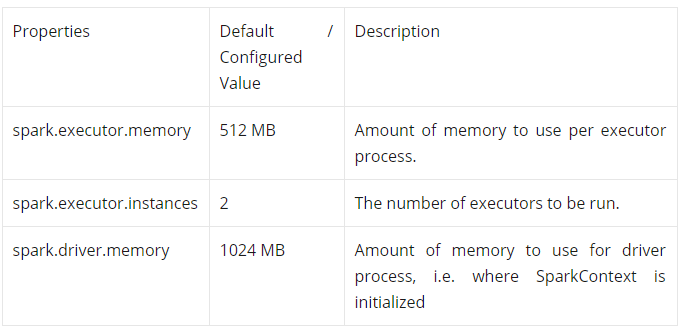
**How do I set executor memory in spark?**

You can do that by either:

1. setting it in the properties file (default is $SPARK\_HOME/conf/spark-defaults.conf ), spark.driver.memory 5g.
2. or by supplying configuration setting at runtime $ ./bin/spark-shell --driver-memory 5g.

What happens if a spark executor fails?

Any of the worker nodes running **executor** can **fail**, thus resulting in loss of in-memory **If** any receivers were running on failed nodes, then their buffer data will be lost.



**Spark driver?**

The **spark driver** is the program that declares the transformations and actions on RDDs of data and submits such requests to the master. In practical terms, the **driver** is the program that creates the SparkContext, connecting to a given **Spark** Master. **Driver memory,** the amount of memory that a driver requires depends upon the job to be executed.

the **driver**-**memory** flag controls the amount of **memory** to allocate for a **driver**, which is 1GB by default and should be increased in case we call a collect() or take(N) action on a large RDD inside your application.

**35. Map-Side Shuffle**

The process of moving the data from partition to partition in order to aggregate, join, match up, or spread out in some other way, is known as **shuffling**. The aggregation/reduction that takes place before data is moved across partitions is known as a **map-side shuffle**.

**36. Tuning Shuffles**

Shuffles are tuned by a few parameters:

1) The shuffle manager (which determines which keys go to which partitions) and

2) The number/size of partitions (these are sort of interchangeable) and

3) The number/size of executors processing our data. When we increase the number of partitions, we inherently reduce (or in certain cases, keep the size the same) the size of each partition. When it comes to shuffle performance is the number of partitions. In practice this is done using the "repartition" or "coalesce" functions.

 37. ORC file format works very well with Apache Hive, Parquet works extremely well with Apache Spark . explain why?

38. When we have spark sql these days then why do we still require Hive can spark SQL replace Hive completely?

39. Spark datasets vs dataframes we know that Dataset provide compile type safety but dataframe do not…. Please demonstrate this with an example.

40. Spark dataframe reader API. Is read a transformation or action ? please explain?

FILE FORMATS

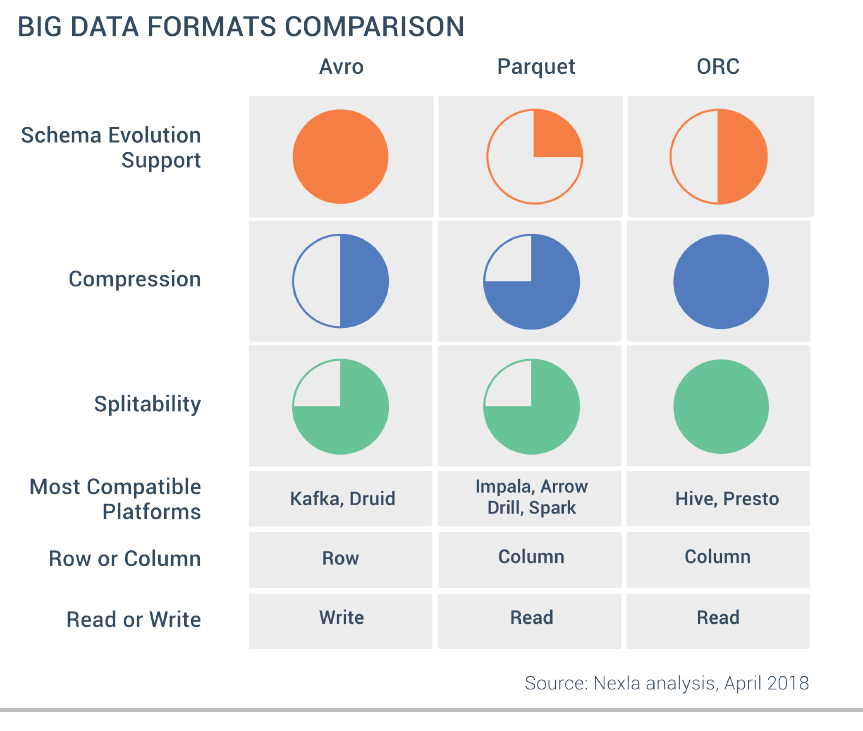
1. <https://blog.clairvoyantsoft.com/big-data-file-formats-3fb659903271>
2. <https://medium.com/@luminousmen/big-data-file-formats-explained-73552c7ef4cd#:~:text=Common%20formats%20used%20primarily%20for,and%20Avro%20with%20Apache%20Spark.>

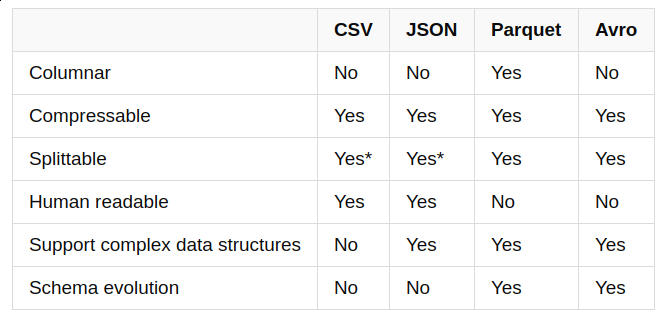
**0**.**Hadoop File Formats, when and what to use?**

* Basic file formats are: **Text** format, Key-Value format, Sequence format.
* Other formats which are used and are well known are: Avro, Parquet, RC or Row-Columnar format, ORC or Optimized Row Columnar format.

Sequence files, Avro data files, and Parquet file formats. Data serialization is a way of representing data in memory as a series of bytes. Avro is an efficient data serialization framework and is widely supported throughout Hadoop and its ecosystem.

**1.AVRO--PARQUET--ORC**





**2.Columnar data storage format?**

Columnar File Formats (**Parquet**, **RCFile**)

The latest hotness in file formats for Hadoop is columnar file storage. Basically this means that instead of just storing rows of data adjacent to one another you also store column values adjacent to each other. So datasets are partitioned both horizontally and vertically.

**3.Predicate Pushdown**

<http://bigdatums.net/2017/08/29/what-is-predicate-pushdown/>

A “**predicate**” (in mathematics and functional programming) is a function that returns a boolean (true or false). In SQL queries predicates are usually encountered in the **WHERE** clause and are used to filter data.

**Predicate pushdown** deals with what values will be scanned and not what columns. So, if we apply a filter on column A to only return records with value V, the predicate push down will make parquet read only blocks that may contain values V. Parquet holds min/max statistics in several levels, and it will compare the value V to the those min/max headers, and only scan blocks where min/max contains the value V. This is for predicate push down.

Another thing with parquet is **"projection pushdown"** - it stores data in columns, so when our projection limits the query to certain columns, only those columns will be returned.

**4.Parsing a JSON file?**

The **JSON**. **parse**() method **parses a JSON** string, constructing the JavaScript value or object described by the string. An optional reviver function can be provided to perform a transformation on the resulting object before it is returned.

**Use** the JavaScript function **JSON**. **parse**() to convert text into a JavaScript object: var obj = **JSON**. **parse**('{ "name":"John", "age":30, "city":"New York"}');

**5.File Compression technique**

**File compression** is a process of "packaging" a **file** (or **files**) to use less disk space. **Compression** software allows you to take many **files** and **compress** them into one **file**, which is smaller than the combined size of the originals. When a **file** or a group of **files** is **compressed**, the resulting "archive" often takes up 50% to 90% less disk space than the original **file**(s).

**Compression** is **useful** because it helps reduce resources usage, such as data storage space or transmission capacity. **File compression** can zip up several small **files** into a single **file** for more convenient email transmission. **Compressed** files transfer faster and use less bandwidth than their uncompressed counterparts.

There are **two** main types of compression: **lossy and lossless.**

Common types of file compression include **Zip, Gzip, RAR, StuffIt, and 7z compression.**

WinZip **compression** is lossless.

If an original file is 1.5 MB, for example, **lossless compression can** reduce it to about half that size, depending on the type of file being **compressed**.

**Compression algorithms** reduce the number of bytes required to represent data and the amount of memory required to store images. **Compression** allows a larger number of images to be stored on a given medium and increases the amount of data that can be sent over the internet.

Spark SQL

[Spark SQL and DataFrames - Spark 2.3.2 Documentation](https://spark.apache.org/docs/2.3.2/sql-programming-guide.html)

Spark SQL is a Spark module for structured data processing. Unlike the basic Spark RDD API, the interfaces provided by Spark SQL provide more information about the structure of both the data and the computation being performed. Internally, Spark SQL uses this extra information to perform extra optimizations. *There are several ways to interact with Spark SQL including SQL and the Dataset API.*

One use of Spark SQL is to execute SQL queries. Spark SQL can also be used to read data from an existing Hive installation. When running SQL from within another programming language the results will be returned as a [Dataset/DataFrame](https://spark.apache.org/docs/latest/sql-programming-guide.html#datasets-and-dataframes). We can also interact / CONNECT with the SQL interface using the [command-line](https://spark.apache.org/docs/latest/sql-distributed-sql-engine.html#running-the-spark-sql-cli) or over [JDBC/ODBC](https://spark.apache.org/docs/latest/sql-distributed-sql-engine.html#running-the-thrift-jdbcodbc-server).

**Datasets and DataFrames**

<https://techvidvan.com/tutorials/apache-spark-dataframe-vs-datasets/>

A Dataset is a distributed collection of data. Dataset is a new interface added in Spark 1.6 that provides the benefits of RDDs (strong typing, ability to use powerful lambda functions) with the benefits of Spark SQL’s optimized execution engine. A Dataset can be [constructed](https://spark.apache.org/docs/latest/sql-getting-started.html#creating-datasets) from JVM objects and then manipulated using functional transformations (map, flatMap, filter, etc.). The Dataset API is available in [Scala](https://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.Dataset) and [Java](https://spark.apache.org/docs/latest/api/java/index.html?org/apache/spark/sql/Dataset.html). Python does not have the support for the Dataset API

A DataFrame is a *Dataset* organized into named columns. It is conceptually equivalent to a table in a relational database or a data frame in R/Python, but with richer optimizations under the hood. DataFrames can be constructed from a wide array of [sources](https://spark.apache.org/docs/latest/sql-data-sources.html) such as: structured data files, tables in Hive, external databases, or existing RDDs. The DataFrame API is available in Scala, Java, [Python](https://spark.apache.org/docs/latest/api/python/pyspark.sql.html#pyspark.sql.DataFrame), and [R](https://spark.apache.org/docs/latest/api/R/index.html).

DataFrames and SQL provide a common way to access a variety of data sources, including Hive, Avro, Parquet, ORC, JSON, and JDBC.

Spark SQL supports the HiveQL syntax as well as Hive SerDes and UDFs, allowing us to access existing Hive warehouses.

**Catalyst Optimizer**

Spark SQL uses an optimizer called catalyst to optimize all the queries written both in spark sql and dataframe dsl. This optimizer makes queries run much faster than their RDD counterparts.

Catalyst is a modular library which is built as a rule based system. Each rule in the framework focuses on the specific optimization.

**Window Functions**

Window aggregate functions (aka window functions or windowed aggregates) are functions that perform a calculation over a group of records called window that are in *some* relation to the current record.

Spark SQL supports three kinds of window functions:

* **ranking** functions
* **analytic** functions
* **aggregate** functions[collect-list/set, count\_distinct, max, min, mean, avg…]

**What is Salting:**

In a SQL join operation, the join key is changed to redistribute data in an even manner so that processing for a partition does not take more time. This technique is called **salting**. ... After the shuffle stage induced by the join operation, all the rows with the same key need to be in the same partition.

**Pushdown Predicate**

 A predicate is a condition on a query that returns true or false, typically located in the WHERE clause. **Predicate Pushdown** gets its name from the fact that portions of SQL statements, ones that filter data, are referred to as **predicates**. ... It can improve query performance by reducing the amount of data read (I/O) from Storage files. **Spark predicate push down** to database allows for better optimized **Spark** SQL queries.

**LAMBDA FUNCTION/expression**

In **Python**, a **lambda** function is a single-line function declared with no name, which can have any number of arguments, but it can only have one expression.

**Lambda Expression** refers to an expression that uses an [anonymous function](https://www.geeksforgeeks.org/anonymous-functions-in-scala/) instead of variable or value. These expressions are faster and more expressive than defining a whole function. We can make our lambda expressions *reusable* for any kind of transformations.

// lambda expression to find double of x

val ex1 = (x:Int) => x + x

 // with multiple parameters

val ex2 = (x:Int, y:Int) => x \* y

**HIGHER ORDER FUNCTION** (example of FUNCTIONAL PROGRAMMING)

A **higher order function** is a **function** that takes a **function** as an argument, or returns a **function as an output**. All other functions are first-order functions. In mathematics higher-order functions are also termed operators or functionals.

const double = n => n \* 2

         [1, 2, 3, 4].map(double) // [ 2, 4, 6, 8 ]

 .map() and .filter(). Both of them take a function as an argument. They're both higher order functions.

**Spark SQL UDF**

In Spark, we create UDF by creating a function in a language we prefer to use for Spark. For example, if we are using Spark with scala, then we create a UDF in scala language and wrap it with udf() function or register it as udf to use it on DataFrame and SQL respectively.

<https://sparkbyexamples.com/spark/spark-sql-udf/>

Why do we need a Spark UDF?

UDF’s are used to extend the functions of the framework and re-use this function on several DataFrame. For example if we wanted to convert the every first letter of a word in a sentence to capital case, spark build-in features does’t have this function hence we can create it as UDF and reuse this as needed on many Data Frames. UDF’s are once created they can be re-use on several DataFrame and SQL expressions.

DATAFRAME

Different ways to create Dataframe in Spark

<https://sparkbyexamples.com/spark/different-ways-to-create-a-spark-dataframe/>

import spark.implicits.\_

val columns = Seq("language","users\_count")

val data = Seq(("Java", "20000"), ("Python", "100000"), ("Scala", "3000"))

val rdd = spark.sparkContext.parallelize(data) //create rdd

val dfFromRDD1 = rdd.toDF() // .toDF function

dfFromRDD1.printSchema()

**From different file formats:**

val df2 = spark.read.csv("/src/resources/file.csv")//from CSV

val df2 = spark.read.json("/src/resources/file.json") // JSON

**Join 2 DF:**

df1.join(df2, $"df1Key" === $"df2Key")

df1.join(df2).where($"df1Key" === $"df2Key")

df1.join(df2).filter($"df1Key" === $"df2Key")

**Union two DF:**

**val** unionDF **=** df1.union(df2)

display(unionDF)

**Write the unionDF to a parquet file:**

// Remove the file if it exists

dbutils.fs.rm("/tmp/databricks-df-example.parquet", **true**)

unionDF.write.parquet("/tmp/databricks-df-example.parquet")

<https://sparkbyexamples.com/spark/spark-streaming-read-json-files-from-directory/>

SPARK STREAMING

<https://spark.apache.org/docs/latest/streaming-programming-guide.html>

Spark Streaming brings Apache Spark's [language-integrated API](https://spark.apache.org/docs/latest/streaming-programming-guide.html) to stream processing, letting us  write streaming jobs the same way we write batch jobs.

Spark Streaming recovers both lost work and operator state (e.g. sliding windows) out of the box.

By running on Spark, Spark Streaming lets us reuse the same code for batch processing, join streams against historical data, or run ad-hoc queries on stream state.

Internally, it works as follows. Spark Streaming receives live input data streams and divides the data into batches, which are then processed by the Spark engine to generate the final stream of results in batches.

**D-STREAM**

Spark Streaming provides a high-level abstraction called *discretized stream* or *DStream*, which represents a continuous stream of data. DStreams can be created either from input data streams from sources such as Kafka, Flume, and Kinesis, or by applying high-level operations on other DStreams. Internally, a DStream is represented as a sequence of [RDDs](https://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.rdd.RDD). Each RDD in a DStream contains data from a certain interval.

**Parallelism**

Clusters will not be fully utilized unless the level of parallelism for each operation is high enough. Spark automatically sets the number of partitions of an input file according to its size and for distributed shuffles.

In order to reduce the processing time, one needs to increase the parallelism.

Spark Streaming provides three ways to increase the parallelism :

(1) Increase the number of receivers: If there are too many records for a single receiver (single machine) to read in and distribute so that is a bottleneck. So we can increase the no. of the receiver depending on the scenario.

(2) Re-partition the receive data: If one is not in a position to increase the no. of receivers, in that case, redistribute the data by re-partitioning.

(3) Increase parallelism in aggregation.

**Sliding Window:**

[Spark streaming](https://techvidvan.com/tutorials/spark-streaming/) leverages the advantage of windowed computations in [spark](https://techvidvan.com/tutorials/why-apache-spark/). It offers to apply transformations over a sliding window of data. This **algorithm** is exactly as it sounds; a **window** is formed over some part of data, and this **window** can slide over the data to capture different portions of it.

Basically, any **Spark window** operation requires specifying two parameters. **Window** length – It defines the duration of the **window**. **Sliding interval** – It defines the **interval** at which the **window** operation is performed.

**Spark** splits the **stream** into micro **batches**. The **batch interval** defines the size of the **batch** in seconds. For example, a **batch interval** of 5 seconds will cause **Spark** to collect 5 seconds worth of data to process.

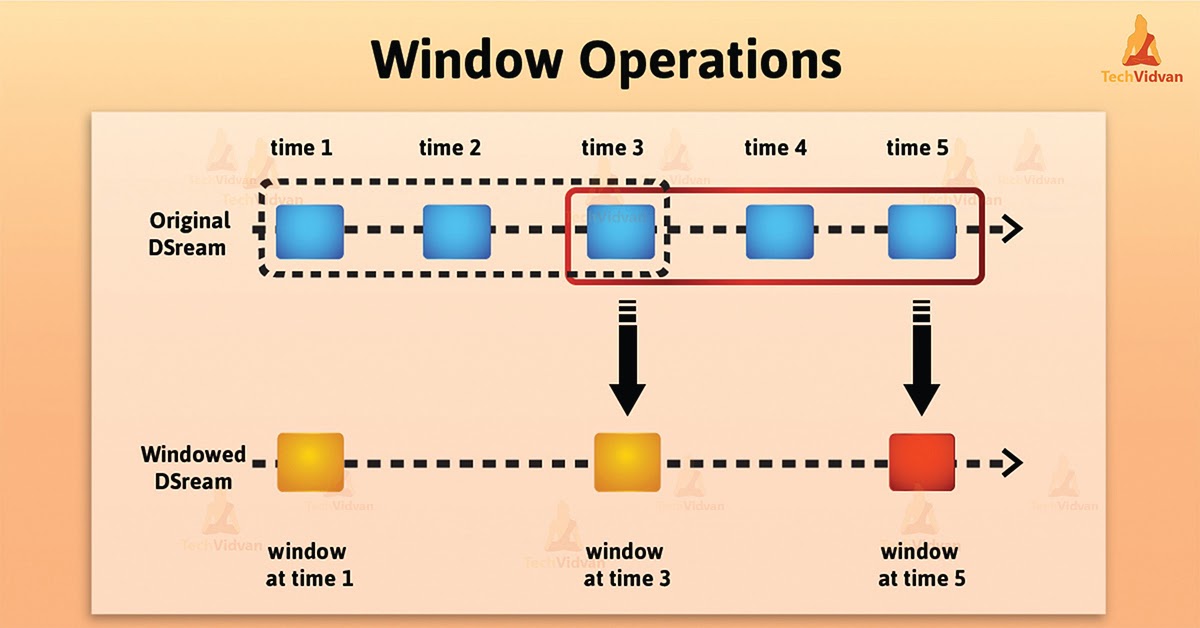
Steps to spark streaming:

1.Initialize a **Spark** StreamingContext object.

2.Apply transformations and output operations to DStreams.

3.Start receiving **data** and processing it using streamingContext. start().

4.Wait for the processing to be stopped using streamingContext. awaitTermination().



*Introduction – Spark Streaming Window operations*

As the window slides over a source [DStream](https://techvidvan.com/tutorials/spark-dstream/), the source RDDs that fall within the window are combined. It also operates upon which produces [spark RDDs](https://techvidvan.com/tutorials/ways-to-create-rdd-in-spark/) of the windowed DStream. Hence, In this specific case, the operation is applied over the last 3 time units of data, also slides by 2-time units.

Basically, any Spark window operation requires specifying two parameters.

* Window length – It defines the duration of the window (3 in the figure).
* Sliding interval – It defines the interval at which the window operation is performed (2 in the figure).

PYSpark

As a future data practitioner, we should be familiar with python's famous libraries: Pandas and scikit-learn. These two libraries are fantastic to explore a dataset up to mid-size. The solution has been evident for a long time, split the problem up onto multiple computers. Parallel computing comes with multiple problems as well.

Pyspark gives the data scientist an API that can be used to solve the parallel data processing problems. Pyspark handles the complexities of multiprocessing, such as distributing the data, distributing code and collecting output from the workers on a cluster of machines.

PySpark, released by Apache Spark community, is basically a Python API for supporting Python with Spark. By utilizing PySpark, we can work and integrate with RDD easily in Python. The library Py4j helps to achieve this feature.

There are several features of PySpark framework:

1. Faster processing than other frameworks.
2. Real-time computations and low latency due to in-memory processing.
3. Polyglot, which means compatible with several languages like Java, Python, Scala and R.
4. Powerful caching and efficient disk persistence.
5. Deployment can be performed by Hadoop through Yarn.

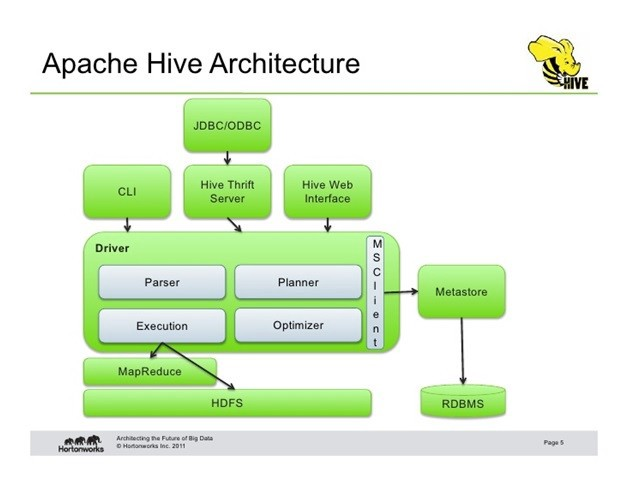
**1.Multiple spark jobs in parallel?**

Within each **Spark** application, **multiple** "**jobs**" (**Spark** actions) may be **running** concurrently if they were submitted by different threads. In other words, a single SparkContext instance can be used by **multiple** threads that gives the ability to submit **multiple Spark jobs** that may or may not be **running** in **parallel**.

**2.How do I create a multiple spark session?**

If we have an existing **spark session** and want to **create a new** one, use the newSession method on the existing SparkSession. The newSession method **creates** a new **spark session** with isolated SQL configurations, temporary tables. The new **session** will share the underlying SparkContext and cached data

HIVE

**0. Architecture**

There are **4 main components** as part of Hive Architecture.

1. Hadoop core components(Hdfs, MapReduce)
2. Metastore
3. Driver
4. Hive Clients

Let’s start off with each of the components individually.

1. Hadoop core components:

i) **HDFS**: When we load the data into a Hive Table it internally stores the data in HDFS path i.e by default in hive warehouse directory.

ii) **MapReduce:** When we Run the below query, it will run a Map Reduce job by converting or compiling the query into a java class file, building a jar and executing this jar file.

**2. Metastore:** is a namespace for tables. This is a crucial part for the hive as all the metadata information related to the hive such as details related to the table, columns, partitions, location is present as part of it. Usually, the Metastore is available as part of the Relational databases eg: MySql

You can check the DataBase configuration via hive-site.xml

**3. Driver:** The component that parses the query, does semantic analysis on the different query blocks and query expressions and eventually generates an execution plan with the help of the table and partition metadata looked up from the metastore. The execution plan created by the compiler is a DAG of stages.

[ A bunch of jar files that are part of hive package help in converting these HiveQL queries into equivalent MapReduce jobs(java) and execute them on MapReduce.

To check if the hive can talk to the appropriate cluster. i.e for hive to interact, query or execute with the existing cluster. You can check details under core-site.xml.

**4. Hive Clients:** It is the interface through which we can submit the hive queries. eg: hive CLI, beeline are some of the terminal interfaces, we can also use the Web-interface like Hue, Ambari to perform the same.

On connecting to Hive via CLI or Web-Interface it establishes a connection to Metastore as well.

**1.When to use External and Managed tables ?**

Managed table:

* Data is temporary
* Hive to Manage the data completely not allowing any external source to use the data
* Don’t want the data after deletion

External table:

# The data is also used outside of Hive. For example, the data files are read and processed by an existing program that doesn’t lock the files

# Hive should not own data and control settings, dirs, etc., we have another program or process that will do those things

# we are not creating table based on existing table (AS SELECT)

# Can create table back and with the same schema and point the location of the data

**2. Performance Tuning**

<http://hadooptutorial.info/hive-performance-tuning/>

1. We can use Apache Tez to Fasten the execution

2. Enable compression in Hive

3. Use suitable file format (ORC file format)

4. Optimize our joins (auto-map-join, skew-join, bucket-map-join)

5. Use partition

6. Bucketing can also be used

7. Parallel execution

8. Cost based optimization

9. Enable vectorization.

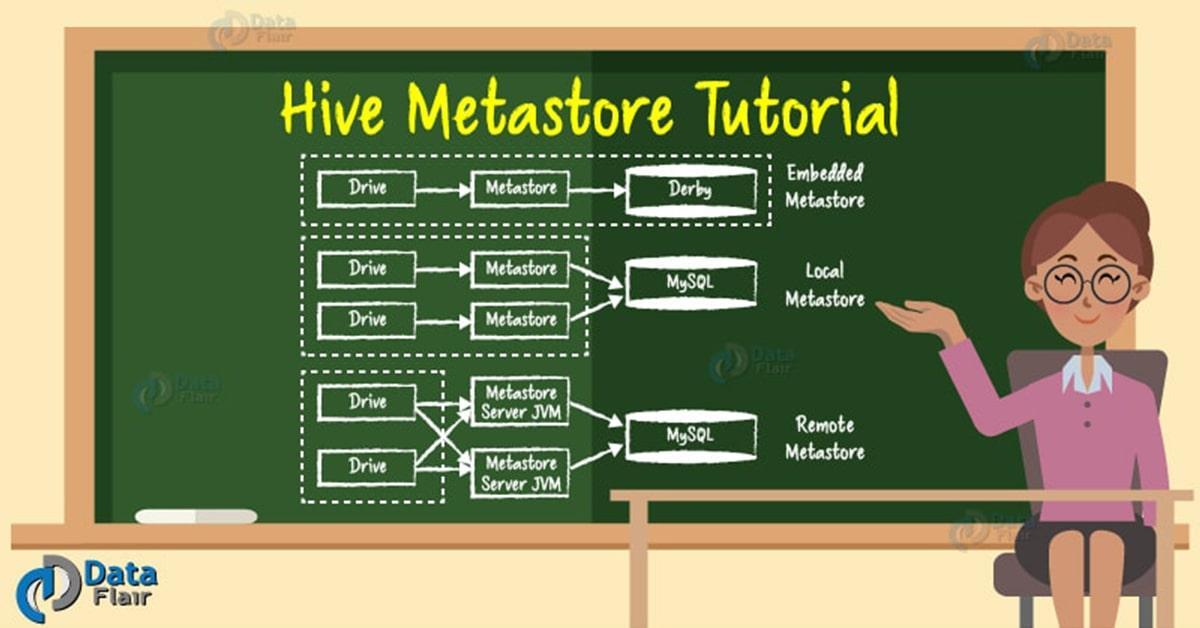
10. Use of indexing

**2. Optimize Hive Query**

* Use Column Names instead of \* in SELECT Clause. This seems to be odd but it will definitely improve the performance of Hive query on TEXT file format. ...
* Use Hive Cost Based Optimizer (CBO) and Update Stats. ...
* Use WHERE instead of HAVING to Define Filters on non-aggregate Columns.

**3. METADATA**

Metastore is the central repository of Hive Metadata. It stores the metadata for Hive tables and relations. For example, Schema and Locations etc.  It is implemented using tables in a relational database. By default, Hive uses a built-in Derby SQL server. It provides single process storage, so when we use Derby, we cannot run instances of Hive CLI. Whenever we want to run Hive on a personal machine or for some developer task, then it is good, but when we want to use it in a cluster, then MySQL or any other similar relational database is required.



There are three modes for Hive Metastore deployment:

* Embedded Metastore
* Local Metastore
* Remote Metastore

**i. Embedded Metastore**

In Hive by default, metastore service runs in the same JVM as the Hive service. It uses an embedded derby database stored on the local file system in this mode. Thus both metastore service and hive service run in the same JVM by using embedded Derby Database. But, this mode also has limitations, as only one embedded Derby database can access the database files on disk at any one time, so only one Hive session could be open at a time.

If we try to start the second session it produces an error when it attempts to open a connection to the metastore. So, to allow many services to connect the Metastore, it configures Derby as a network server. This mode is good for unit testing. But it is not good for the practical solutions.

**ii. Local Metastore**

Hive is the data-warehousing framework, so hive does not prefer a single session. To overcome this limitation of Embedded Metastore, for Local Metastore was introduced. This mode allows us to have many Hive sessions i.e. many users can use the metastore at the same time. We can achieve this by using any JDBC compliant like MySQL which runs in a separate JVM or different machines than that of the Hive service and metastore service which are running in the same JVM.

**iii. Remote Metastore**

Another metastore configuration called Remote Metastore. In this mode, metastore runs on its own separate JVM, not in the Hive service JVM. If other processes want to communicate with the metastore server they can communicate using Thrift Network APIs. We can also have one more metastore server in this case to provide more availability. This also brings better manageability/security because the database tier can be completely firewalled off. And the clients no longer need to share database credentials with each Hiver user to access the metastore database.

**3.Complete delete/purge the external table?**

One solution could be, we have to change the external to internal table before drop it:

**Example:**

beeline> ALTER TABLE $tablename SET TBLPROPERTIES('EXTERNAL'='False');

// make the table as internal and then:

beeline> drop table $tablename;

//if we drop the table data will be dropped as well.

**4.Hive tables from Spark SQL?**

There are various methods that we can follow to connect to Hive metastore or access Hive tables from Apache Spark processing framework.

some of commonly used methods to access hive tables from apache spark:

* Access Hive Tables using Apache Spark Beeline
* Accessing Hive Tables using Apache Spark JDBC Driver
* [Execute Pyspark Script from Python and Examples](https://dwgeek.com/execute-pyspark-script-from-python-example.html/)

……………...OR…………………………………………………………

Step 1: Move hive-site.xml from $HIVE\_HOME/conf/hive-site.xml to $SPARK\_HOME/conf/hive-site.xml. Make an entry regarding hive metastore uris in this file.

Step 2: Extract all the dependencies for required Spark components (in this case Spark SQL and Hive) in the build.sbt file.

Step 3: Start all Hadoop processes in the cluster By Start-All.sh to start all the demons.

Step 4: Start MySQL because Hive needs it to connect to the metastore and because Spark SQL will also need it when it connects to Hive

Step 5: Run the Hive metastore process so that when Spark SQL runs, it can connect to metastore uris and take from it the hive-site.xml.

**5.Access Hive tables on Spark SQL and convert them into DataFrames?**

For large scale projects working with petabytes of data, it is possible to reduce the time taken to process the aggregate function, if we can execute complex queries in an efficient way. Hence, we use Spark SQL, which has an in-built catalyst optimizer that processes all types of queries at a faster pace. It even allows the usage of external DataFrames with Hive tables for purposes such as join, cogroup, etc.

**6. PARTITION / Bucket when to use?**

**Partitioning** helps in elimination of data, if **used** in WHERE clause, whereas **bucketing** helps in organizing data in each **partition** into multiple files, so the same set of data is always written in the same **bucket**. Helps a lot in joining columns.

**Partitioning –**  In the Hive **Partition**, each **partition will** be created as a directory. Hive organizes tables into partitions for grouping the same type of data together based on a column or partition key. Each table in the hive can have one or more partition keys to identify a particular partition. Using partition we can make it faster to do queries on slices of the data.

**Bucketing –**  Here, **CLUSTERED BY** clause is **used** to divide the table into buckets. **Buckets** in hive **is** used in segregating of hive table-data into multiple files or directories. **Bucketing can** also be done even **without partitioning** on **Hive** tables.

In Hive Tables or partitions are subdivided into buckets based on the hash function of a column in the table to give extra structure to the data that may be used for more efficient queries.

**7.Dynamic partitioning and when is it used?**

Dynamic partitioning values for partition columns are known during the runtime. In other words, it is known during loading of the data into a Hive table.

* Usage:

1. While we Load data from an existing non-partitioned table, in order to improve the sampling. Thus it decreases the query latency.

2. Also, while we do not know all the values of the partitions beforehand. Thus, finding these partition values manually from a huge dataset is a tedious task.

**8. STATIC vs DYNAMIC partition:**

**Static Partition in Hive**

* Insert input data files individually into a partition table is Static Partition
* Usually when loading files (big files) into Hive Tables static partitions are preferred
* Static Partition saves your time in loading data compared to dynamic partition
* You 'statically' add a partition in the table and move the file into the partition of the table.
* We can alter the partition in static partition
* If you want to use the Static partition in the hive you should set property set hive.mapred.mode = strict [non-strict for dynamic]

this property set by default in hive-site.xml

* Static partition is in Strict Mode.
* You should use where clause to use limit in the static partition.
* You can perform Static partition on Hive Managed table or external table

**Dynamic Partition in Hive:**

* Single insert to partition table is known as dynamic partition
* Usually dynamic partition load the data from non partitioned table
* Dynamic Partition takes more time in loading data compared to static partition
* When you have large data stored in a table then Dynamic partition is suitable
* If you want to partition number of column but you don’t know how many columns then also dynamic partition is suitable
* Dynamic partition There is no requirement where clause to use limit.
* We can’t perform alter on the Dynamic partition.

**9. HIVE to NoSQL/HBASE/MONGODB**

Hive - HBase Integration (https://cwiki.apache.org/confluence/display/Hive/HBaseIntegration)

* First, we need to create HBase table
* Then, create external Hive Table (as template for the HFile creation)
* set Hive properties for HFile creation
  + **set hfile.family.path=/tmp/my\_test\_table/cf**
  + **set hive.hbase.generatehfiles=true**
* move data from Hive to HBase by INSERT OVERWRITE TABLE ... SELECT FROM ... statement
* insert generated HFiles into HBase by hbase org.apache.hadoop.hbase.mapreduce.LoadIncrementalHFiles tool

**11. Map-Side Join:**

Map side join is a process where joins between two tables are performed in the Map phase without the involvement of Reduce phase.

Map Join:

1. By specifying the keyword, /\*+ MAPJOIN(b) \*/ in the join statement.

2. By setting the property to true.

hive.auto.convert.join=true

For performing Map-side joins, there should be two files, one is of larger size and the other is of smaller size. We can set the small file size by using the following property:

hive.mapjoin.smalltable.filesize=(default it will be 25MB)

**12. Update Millions of records in a table:**

If we use a for loop approach for the update it will take 53.7 years to complete!

We can use the Insert into a dummy table appended with nologging, and were able to complete the "update" in under 30 minutes.

With nologging, if the system aborts, we simply re-run the 'update' again, as we have the original data in the main table. When done, we swap the partition of original data with the 'dummy' table (the one containing new values), rebuild indexes in parallel, and Our update will be complete.

update field2 in a table:

1) First create your dummy hold table: create table xyz\_HOLD as select \* from xyz where rownum<1. Alter tablexyz nologging.

2) insert /\*+ append parallel (xyzhold,12) \*/ into xyz\_hold xyzhold (field1, field2, field3) select /\*+ parallel (x,12) \*/ xyz.field1, my\_new\_value\_for\_field2, xyz.field3 from xyz x where blah blah blah.

3) When done, either rename the table, or swap the partition if your original table is partitioned, and we only updated one partition as we do. Obviously you need to rebuild indexes, etc as required.

HIVE & SPARK

**1.Access Hive tables from spark SQL?**

There are various methods that we can follow to connect to Hive metastore or access Hive tables from Apache Spark processing framework.

some of commonly used methods to access hive tables from apache spark:

* Access Hive Tables using Apache Spark Beeline
* Accessing Hive Tables using Apache Spark JDBC Driver
* [Execute Pyspark Script from Python and Examples](https://dwgeek.com/execute-pyspark-script-from-python-example.html/)

**2.Connect HIVE from SPARK**

JDBC is not required.

Spark can connect directly to the Hive metastore, not through HiveServer2.

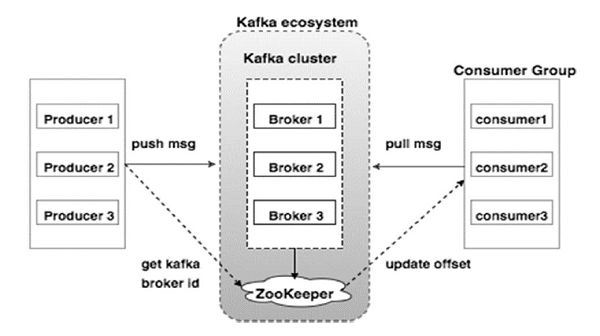
To configure we can try :

1. We have to put hive-site.xml on our classpath, and need to specify hive.metastore.uris to where our hive metastore is hosted. [ [How to connect to a Hive metastore programmatically in SparkSQL?](https://stackoverflow.com/questions/31980584/how-to-connect-to-a-hive-metastore-programmatically-in-sparksql)  ]
2. Need to Import org.apache.spark.sql.hive.HiveContext, as it can perform SQL query over Hive tables.
3. Then we can define val sqlContext = new org.apache.spark.sql.hive.HiveContext(sc)
4. Then we can verify sqlContext.sql("show tables") to see how it works.

KAFKA

**0. Cluster Architecture** -- Role of Zookeeper

<https://dzone.com/articles/kafka-architecture>



**1. Components**

-Kafka Producer

-Kafka Consumer

-Kafka Broker

-kafka Cluster

-Kafka Topic

-Kafka Partitions

-Kafka Consumer Group

-Offset

**Kafka Producer**

The producer acts as a sender. It is responsible for sending a message or data. It does not send messages directly to consumers. It pushes messages to Kafka Server or Broker. The messages or data are stored in the Kafka Server or Broker. There can be multiple producers which can send a message to the same Kafka topic or different Kafka topics.

**Kafka Consumer**

The consumer acts as Receiver. It is responsible for receiving or consuming a message. But It does not consume or receive a message directly from Kafka Producer. Kafka Producer pushes messages to Kafka Server or broker. The consumer can request a message from the Kafka broker. If the Kafka Consumer will have enough permissions, then it gets a message from the Kafka Broker.

**Kafka Broker**

The Kafka Broker is nothing but just a server. In simple words, A broker is just an intermediate entity who exchanges messages between a producer and a consumer. For Kafka Producer, it acts as a receiver and for Kafka Consumer, it acts as a sender. In the Kafka cluster, there can be one or more Kafka brokers.

**Kafka Cluster**

Now first understand, what is a cluster? A cluster is a common terminology in the distributed computing system. It is nothing but just a group of computers which are working for a common purpose. Kafka is also a distributed system, so it also has a cluster having a group of servers called brokers.

There can be one or more brokers in the Kafka cluster.

* Single Broker Cluster: The Kafka cluster having only one broker is called Single Broker Cluster.
* Multi-Broker Cluster: The Kafka cluster having two or more brokers are called Multi-Broker Cluster.

**Kafka Topic**

It is one of the most important components of Kafka. Kafka Topic is a unique name given to a data stream or message stream.

Now let us understand the need for this. As you know Kafka Producer sends a message stream to the Broker and Kafka Consumer receives a message stream from that Broker.

Suppose a consumer wants to consume a message from Broker, but the question is, from which message stream? There can be multiple different message streams on the same Broker, coming from different Kafka producers. Here comes the concept of Topic which is a unique identity of the message stream. The Producer sends a message to a unique name which is called the topic for that message stream. Multiple producers can also send to the same topic. If any consumer wants to consume the message, then it subscribes to the topic present in Kafka Broker. Now all the messages coming to that topic will be delivered to the consumer.

**Kafka Partitions**

Now we know that the producer sends data to the broker with a unique identity called topic and the broker stores the message with that topic. Now consider you have a huge volume of data and it is very challenging for the broker to store data on a single machine. As we already know, Kafka is a distributed system. In such a scenario we can break the Kafka topic in partitions and distribute the partitions on a different machine to store. Based on the use case and data volume, we can decide the number of partitions for a topic during Kafka topic creation.

**Offsets**

In Kafka, a sequence number is assigned to each message in each partition of a Kafka topic. This sequence number is called Offset. As soon as any message arrives in a partition a number is assigned to that message. For a given topic, different partitions have different offsets. The offset number is always local to the topic partition. There is not any offset that is global to the topic or each partition of the topic. Initially, the offset pointer points to the first message. As soon as the consumer reads that message, the offset pointer moves to the next message and so on in the sequence.

So, for any message, the combination of topic name, partition number and offset number is a unique identity. In other words, we can find any message based on these three components.

**Kafka Consumer Group**

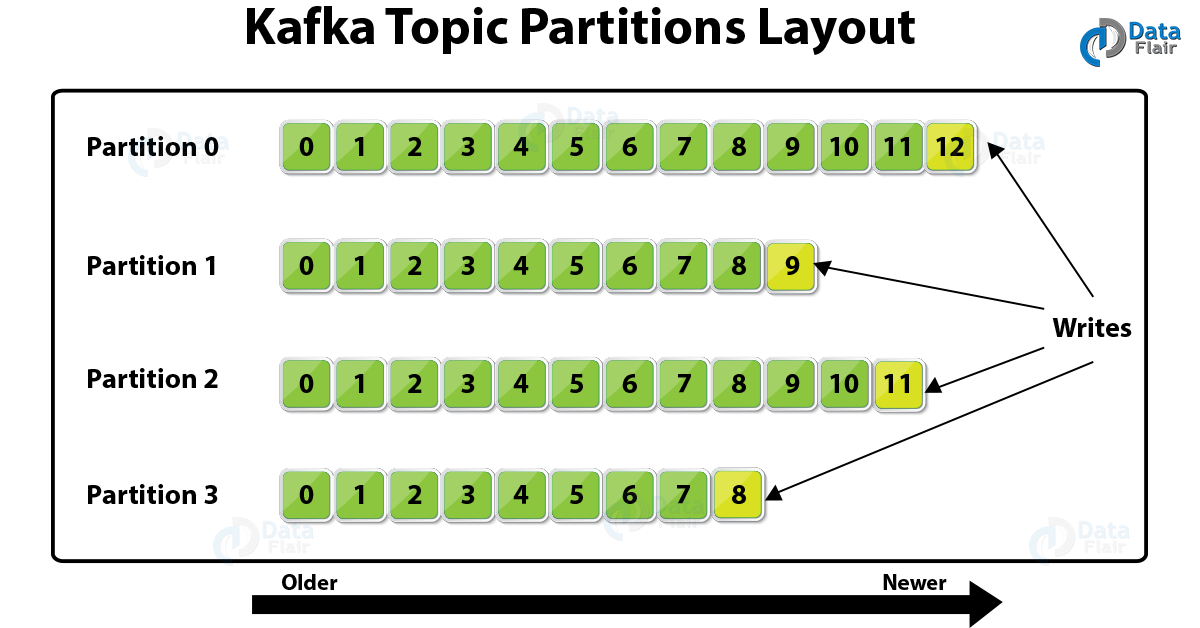
As the name suggests, the [Kafka Consumer group](https://www.educba.com/kafka-consumer-group/) is a group of consumers. Multiple consumers combined to share the workload. It is just like dividing a piece of a large task among multiple individuals. There can be multiple consumer groups subscribing to the same or different topics. Two or more consumers belonging to the same consumer group do not receive the common message. They always receive a different message because the offset pointer moves to the next number once the message is consumed by any of the consumers in that consumer group.

**2.Kafka 4 core APIs**

* The [Producer API](https://kafka.apache.org/documentation.html#producerapi) allows an application to publish a stream of records to one or more Kafka topics. The central part of the Producer API is Producer class. Producer class provides an option to connect Kafka “broker” in its constructor by some methods. KafkaProducer class provides a send method to send messages asynchronously to a topic.
* The [Consumer API](https://kafka.apache.org/documentation.html#consumerapi) allows an application to subscribe to one or more topics and process the stream of records produced to them.
* The [Streams API](https://kafka.apache.org/documentation/streams) allows an application to act as a *stream processor*, consuming an input stream from one or more topics and producing an output stream to one or more output topics, effectively transforming the input streams to output streams.
* The [Connector API](https://kafka.apache.org/documentation.html#connect) allows building and running reusable producers or consumers that connect Kafka topics to existing applications or data systems. For example, a connector to a relational database might capture every change to a table.

**3.Create a topic (command)**

<http://cloudurable.com/blog/kafka-tutorial-kafka-from-command-line/index.html>



**4. Kafka workflow:**

<https://www.tutorialspoint.com/apache_kafka/apache_kafka_workflow.htm>

**4. Message LOST**

<https://developer20.com/when-you-can-nose-messages-in-kafka/>

Kafka is an excellent tool with high capacity, consistency, and latency. On the other hand, a lot of these things depend on the configuration in the producer, consumer and even on the broker(s). What’s even more problematic, the behavior can change depending on the implementation of the producer or the consumer.

**5. Kafka Streams**

**Kafka Streams** uses the concepts of **stream** partitions and **stream** tasks as logical units of its parallelism model. ... Each **stream** partition is a totally ordered sequence of data records and maps **to** a **Kafka** topic partition. A data record in the **stream** maps **to** a **Kafka** message from that topic.

**5.Confluent Kafka?**

**Confluent** is a data streaming platform based on Apache **Kafka**: a full-scale streaming platform, capable of not only publish-and-subscribe, but also the storage and processing of data within the stream. **Confluent** is a more complete distribution of Apache **Kafka**.

**6.Biulding a real time DataPipeLine using Spark streaming & Kafka**

<https://www.opcito.com/blogs/building-a-real-time-data-pipeline-using-spark-streaming-and-kafka/>

**7.Real-Time Twitter Application:**

<https://www.tutorialspoint.com/apache_kafka/apache_kafka_real_time_application.htm>

Spark Streaming + Kafka Integration

**1. Linking:** In our SBT/Maven project definition, we have to link our streaming application against the following artifact (see [Linking section](https://spark.apache.org/docs/1.2.0/streaming-programming-guide.html#linking) ).  
 groupId = org.apache.spark

 artifactId = spark-streaming-kafka\_2.10

 version = 1.2.0

**2.Programming:** In the streaming application code, we need to import *KafkaUtils* and need to create input DStream as follows.

* + [Scala](https://spark.apache.org/docs/1.2.0/streaming-kafka-integration.html#tab_scala_0)

import org.apache.spark.streaming.kafka.\_

val kafkaStream = KafkaUtils.createStream(

streamingContext, [zookeeperQuorum], [group id of the consumer], [per-topic number of Kafka partitions to consume])

Points to remember:

* + Topic partitions in Kafka do not correlate to partitions of RDDs generated in Spark Streaming. So increasing the number of topic-specific partitions in the KafkaUtils.createStream() only increases the number of threads using which topics that are consumed within a single receiver. It does not increase the parallelism of Spark in processing the data.
  + Multiple Kafka input DStreams can be created with different groups and topics for parallel receiving of data using multiple receivers.

**3. Deploying:** Package spark-streaming-kafka\_2.10 and its dependencies (except spark-core\_2.10 and spark-streaming\_2.10 which are provided by spark-submit) into the application JAR. Then we have to use spark-submit to launch our application (see [Deploying section](https://spark.apache.org/docs/1.2.0/streaming-programming-guide.html#deploying-applications) )

**4. Spark integration with Kafka:**

**Approach 1: Receiver-based Approach**

This approach uses a Receiver to receive the data. The Receiver is implemented using the Kafka high-level consumer API. As with all receivers, the data received from Kafka through a Receiver is stored in Spark executors, and then jobs launched by Spark Streaming processes the data.

However, under default configuration, this approach can lose data under failures. To ensure zero-data loss, we have to additionally enable Write Ahead Logs in Spark Streaming (introduced in Spark 1.2). This synchronously saves all the received Kafka data into write ahead logs on a distributed file system (e.g HDFS), so that all the data can be recovered on failure.

**Approach 2: Direct Approach (No Receivers)**

This new receiver-less “direct” approach has been introduced in Spark 1.3 to ensure stronger end-to-end guarantees. Instead of using receivers to receive data, this approach periodically queries Kafka for the latest offsets in each topic+partition, and accordingly defines the offset ranges to process in each batch. When the jobs to process the data are launched, Kafka’s simple consumer API is used to read the defined ranges of offsets from Kafka (similar to read files from a file system).

**5. Commit-log / Write-ahead-log / Transaction-log**

In Spark 1.2, we got introduced a new experimental feature of *write ahead logs* for achieving strong fault-tolerance guarantees. If enabled, all the data received from a receiver gets written into a write ahead log in the configuration checkpoint directory. This prevents data loss on driver recovery, thus ensuring zero data loss ( [Fault-tolerance Semantics](https://spark.apache.org/docs/1.2.0/streaming-programming-guide.html#fault-tolerance-semantics) section). This can be enabled by setting the [configuration parameter](https://spark.apache.org/docs/1.2.0/configuration.html#spark-streaming) spark.streaming.receiver.writeAheadLogs.enable to true

Conceptually there's no difference between the "commit log" that Kafka provides and the commit log/transaction log/write ahead log that a DBMS uses: *They're both about recording the changes made to something so that it can be replayed later.*

Kafka can serve as a kind of external **commit-log** for a distributed system. The log helps replicate data between nodes and acts as a re-syncing mechanism for failed nodes to restore their data.

**6.Leader & Followers ?**

Each partition has one server which acts as the "leader" and zero or more servers which act as "followers". The leader handles all read and write requests for the partition while the followers passively replicate the leader. If the leader fails, one of the followers will automatically become the new leader. Each server acts as a leader for some of its partitions and a follower for others so load is well balanced within the cluster.

**7.Partition VS Replica ?**

* partition: each topic can be splitted up into partitions for load balancing (we could write into different partitions at the same time) & scalability (the topic can scale up without the instance limitations); within the same partition the records are ordered; Each record is assigned and identified by an unique offset.
* replica: Replication is implemented at partition level. The redundant unit of topic partition is called replica. For fault-tolerant durability mainly;

**8.Consumer group:**

In a queue messaging system instead of a single consumer, a group of consumers having the same Group ID will subscribe to a topic. In simple terms, consumers subscribing to a topic with the same Group ID are considered as a single group and the messages are shared among them.

**9. REST Proxy:**

The **Kafka REST Proxy** provides a **RESTful** interface to a **Kafka** cluster. It makes it easy to produce and consume messages, view the state of the cluster, and perform administrative actions without using the native **Kafka** protocol or clients. Examples of use cases include reporting data to Kafka from any frontend app built in any language, ingesting messages into a stream processing framework that doesn’t yet support Kafka, and scripting administrative actions.

**10.Schema Registry: (SR):**

**Schema Registry** provides a serving layer for our metadata. It provides a RESTful interface for storing and retrieving Avro **schemas**. ... It provides serializers that plug into **Kafka** clients that handle **schema** storage and retrieval for **Kafka** messages that are sent in the Avro format.

The Schema Registry provides data compatibility. It's a separate component and it must be able to reject bad data. Producers and Consumers need to be able to talk to it. With the Schema Registry, the Kafka Clients can process messages with an Avro Schema without including the schema with every message. Instead, the schemas are stored in the Schema Registry.

**11.Start the Kafka schema registry?**

Start each Confluent Platform service in its own terminal using this order of operations:

1. Start ZooKeeper. Run this command in its own terminal. bin/zookeeper-server-start ./etc/kafka/zookeeper.properties.
2. Start Kafka. Run this command in its own terminal. ...
3. Start Schema Registry. Run this command in its own terminal.

**11. ISR (in sync replica)**

**Kafka** replicates writes to the leader partition to followers (node/partition pair). A follower that is in-**sync** is called an ISR (in-**sync replica**). If a partition leader fails, **Kafka** chooses a new ISR as the new leader.

In-**sync replicas** are the subset of all the **replicas** for a partition having the same messages as the leader. Now let's see what happens when a broker goes down. If for some reason let's say Broker 2 goes down. The access to partition 1 is now lost since broker 2 was the leader for partition 1.

**12. Mirror-Maker**

Kafka MirrorMaker is a stand-alone tool for copying data between two Apache Kafka® clusters. It is a Kafka consumer and producer hooked together. We need to use Mirror Maker for disaster recovery. Mirror Maker replicates a Kafka cluster to another datacenter or AWS region. They call what Mirror Maker does mirroring as not to be confused with replication.

**MirrorMaker** is essentially a **Kafka** high-level consumer and producer pair, efficiently moving data from the source cluster to the destination cluster and not offering much else.

**12. Zookeeper role in kafka:**

**Kafka** uses **ZooKeeper to** manage the cluster. Kafka uses **Zookeeper** to track status of **kafka** cluster nodes. **ZooKeeper** is used **to** coordinate the brokers/cluster topology. **ZooKeeper** is a consistent file system for configuration information.

**ZooKeeper** gets used for leadership elections for Broker Topic Partition Leaders. **Zookeeper** also plays a vital **role** for serving many other purposes, such as leader detection, configuration management, synchronization, detecting when a new node joins or leaves the cluster, etc.

ZooKeeper is a high-performance coordination service for distributed applications and Kafka uses ZooKeeper to store the metadata information of the cluster. Kafka comes with the Zookeeper built-in, all we need is to start the service with default configuration.

**13. Zookeeper goes down in Kafka?**

For example, **if** we lost the **Kafka** data in **ZooKeeper**, the mapping of replicas to Brokers and topic configurations would be lost as well, making our **Kafka** cluster no longer functional and potentially resulting in total data loss.

**14. Can Kafka work without a zookeeper?**

**Kafka** 0.9 **can run without Zookeeper** after all **Zookeeper** brokers are down. After killing all three **Zookeeper** nodes the **Kafka** cluster continues functioning.

**15.** [**Kafka → Kerberos Authentication on Windows using the Cofluent library?**](https://stackoverflow.com/questions/56395729/how-to-connect-to-kafka-cluster-with-kerberos-authentication-on-windows-using-th)

In the kafka environment, I had changed some parameters in server.properties file for enabling SASL and then created the jaas file for kafka.

On the zookeeper side, I also did some changes so that zookeeper runs with a jaas file.

In our client.properties file we need do the following configuration:

sasl.mechanism=GSSAPI

## Configure SASL\_SSL if SSL encryption is enabled, otherwise configure

SASL\_PLAINTEXT security.protocol=SASL\_SSL sasl.kerberos.service.name=kafka sasl.jaas.config=com.sun.security.auth.module.Krb5LoginModule required \ useKeyTab=true \ storeKey=true \ keyTab="/etc/security/keytabs/kafka\_client.keytab" \ principal="[kafkaclient1@EXAMPLE.COM](mailto:kafkaclient1@EXAMPLE.COM)";

## optionally - kafka-console-consumer or kafka-console-producer, kinit can be used along with useTicketCache=true sasl.jaas.config=com.sun.security.auth.module.Krb5LoginModule required \ useTicketCache=true;

For Windows: bin/kafka-console-consumer.bat --bootstrap-server localhost:9092 --topic test --consumer.config client.properties --from-beginnin

**16. Performance Tuneing in Kafka:**

Here are ten specific tips to help keep your Kafka deployment optimized and more easily managed:

1. Set log configuration parameters to keep logs manageable
2. Know Kafka’s (low) hardware requirements
3. Leverage Apache ZooKeeper to its fullest
4. Set up replication and redundancy the right way
5. Take care with topic configurations
6. Use parallel processing
7. Configure and isolate Kafka with security in mind
8. Avoid outages by raising the Ulimit
9. Maintain a low network latency
10. Utilize effective monitoring and alerts

**17.Configure Kafka to ensure that events are stored reliably?**

The following recommendations for Kafka configuration settings make it extremely difficult for data loss to occur.

1. Producer
   1. block.on.buffer.full=true
   2. retries=Long.MAX\_VALUE
   3. acks=all
   4. max.in.flight.requests.per.connections=1
   5. Remember to close the producer when it is finished or when there is a long pause.
2. Broker
   1. Topic replication.factor >= 3
   2. Min.insync.replicas = 2
   3. Disable unclean leader election
3. Consumer
   1. Disable enable.auto.commit
   2. Commit offsets after messages are processed by your consumer client(s).

If we have more than 3 hosts, we can increase the broker settings appropriately on topics that need more protection against data loss.

**18.“Right” number of partitions for a topic?**

Choosing the proper number of partitions for a topic is the key to achieving a high degree of parallelism with respect to writes to and reads and to distribute load. Evenly distributed load over partitions is a key factor to have good throughput.

For example, if we want to be able to read 1 GB/sec, but our consumer is only able to process 50 MB/sec, then we need at least 20 partitions and 20 consumers in the consumer group.

Similarly, if we want to achieve the same for producers, and 1 producer can only write at 100 MB/sec, we need 10 partitions. In this case, if we have 20 partitions, we can maintain 1 GB/sec for producing and consuming messages. We should adjust the exact number of partitions to the number of consumers or producers, so that each consumer and producer achieve their target throughput.

So a simple formula could be:

Number of Partitions = Max( NP, NC)

where:

* NP is the number of required producers determined by calculating: TT/TP
* NC is the number of required consumers determined by calculating: TT/TC
* TT is the total expected throughput for our system
* TP is the max throughput of a single producer to a single partition
* TC is the max throughput of a single consumer from a single partition

NOSQL

**1. What is NoSql DB ?**

A **NoSQL**  (aka "not only SQL") non tabular database is a database designed to allow for scalable data storage that can handle. Compared to other databases, NoSQL databases don't use tabular relationships. Querying for NoSQL databases is handled differently depending on the software. **NoSQL** databases come in a variety of types based on their data model. The main types are document, key-value, wide-column, and graph.

**2. Types on NoSql**

Broadly speaking there are 4 different models of NoSQL databases:

* Key-Value pair-based databases
* Column-based databases
* Document-oriented databases
* Graph databases

**3. NoSQL DB list:-**

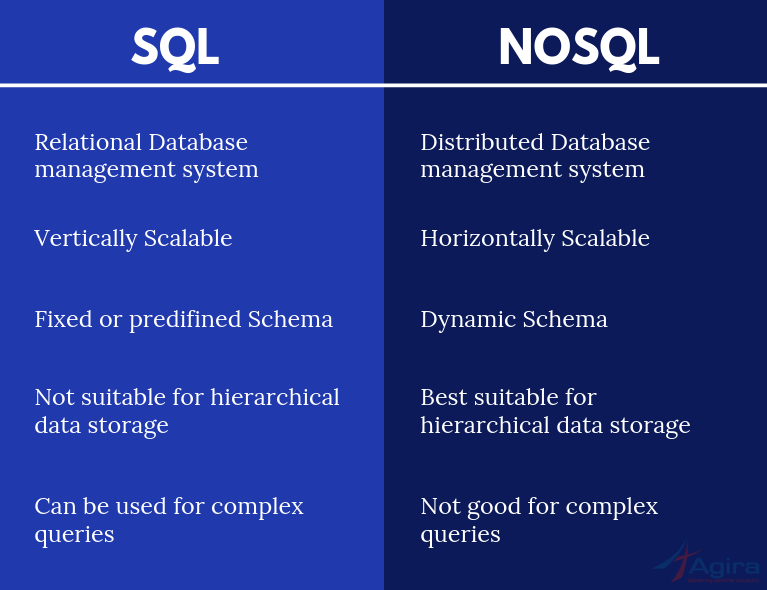
* Hadoop/**Hbase**. Use **Apache HBase** when we need random, real-time read/write access to your Big Data environment.
* Cassandra. The Apache Cassandra database is the right choice when we need scalability and high availability without compromising performance. ...
* Hypertable. ...
* Accumulo. ...
* Amazon SimpleDB. ...
* Cloud Data. ...
* HPCC. ...
* Flink.
* Redis
* Couch DB
* MongoDB
* Raven DB
* Neo4j
* Amazon DynamoDB

**4. Need of NoSql DB**

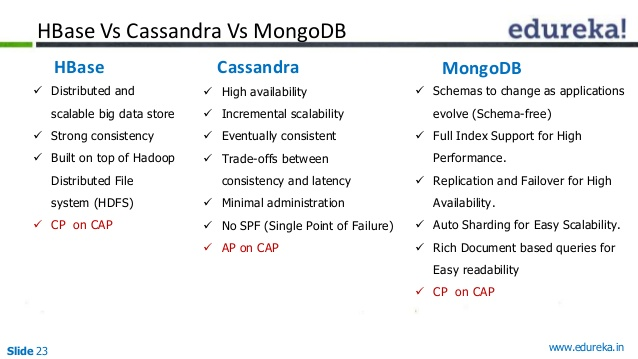
**NoSQL databases** do not limit the types of data that we can store together. **NoSQL databases** also enable us to add new data types as our **needs** change. With document-oriented **databases**, we can store data in one place without having to define the data type in advance. To make the most of cloud computing and storage.

**NoSQL databases** are a great fit for many modern applications such as mobile, web, and gaming that require flexible, scalable, high-performance, and highly functional **databases** to provide great user experiences. A **NoSQL database** is exactly the type of **database** that can handle the sort of unstructured, messy and unpredictable data that our system of engagement requires.

**5. SQL vs NoSQL**



**6**. **Hbase VS MangoDB:-**



**7. CAP-Theorem-in-NoSQL-DB**(consistency,availability, partition tolerance)

**CAP Theorem** is a concept that a distributed database system can only have 2 of the 3: Consistency, Availability and Partition Tolerance. **CAP Theorem** is very important in the Big Data world, especially when we need to make trade offs between the three, based on our unique use case.

* **Consistency** — A guarantee that every node in a distributed cluster returns the same, most recent, successful write. Consistency refers to every client having the same view of the data. There are various types of consistency models. Consistency in CAP (used to prove the theorem) refers to linearizability or sequential consistency, a very strong form of consistency.
* **Availability** — Every non-failing node returns a response for all read and write requests in a reasonable amount of time. The key word here is every. To be available, every node on (either side of a network partition) must be able to respond in a reasonable amount of time.
* **Partition Tolerant** — The system continues to function and upholds its consistency guarantees in spite of network partitions. Network partitions are a fact of life. Distributed systems guaranteeing partition tolerance can gracefully recover from partitions once the partition heals.

**8. Region Server in HBase?**

**HBase** Tables are divided horizontally by row key range into “**Regions**.” A **region** contains all rows in the table between the **region's** start key and end key. **Regions** are assigned to the nodes in the cluster, called “**Region Servers**,” and these serve data for reads and writes. A **region server** can serve about 1,000 **regions**.

**HBase Crash Recovery:--** ZooKeeper **notifies the** HMaster about the failure, whenever a **Region Server fails**. Afterward, too many active **Region Servers**, HMaster distributes and allocates the **regions** of crashed **Region Server**.

**10. HFILE in HBASE ?**

**HFile** is simply a specialised file based data structure that is used to store data in **Hbase**. **Hbase** previously used Hadoop Map file format(sorted sequence file). However it has performance limitations. It was replaced by the HFILE format which is designed specific to **HBASE**.

SQOOP

**0. All about Sqoop:  RDBMS-->HDFS / HDFS-->RDBMS**

[Apache Sqoop. RDBMS to HDFS and back | by Prathamesh Nimkar](https://towardsdatascience.com/apache-sqoop-1113ce453639)

**1. Create a Sqoop job?**

We can create and maintain the Sqoop jobs. Sqoop job creates and saves the import and export commands. It specifies parameters to identify and recall the saved job. This re-calling or re-executing is used in the incremental import, which can import the updated rows from the RDBMS table to HDFS.

We are creating a sqoop job with the name *myjob*, which can import the table data from the RDBMS table to HDFS. The following command is used to create a job that is importing data from the employee table in the db database to the HDFS file.

$ sqoop job

--create myjob \

-- import \

--connect jdbc:mysql://localhost/db \

--username root \

--table employee

--m 1

**2. Manage a Sqoop job?**

‘--list’ argument is used to verify the saved jobs. This command is used to verify the list of saved Sqoop jobs.

$ sqoop job --list

Inspect Job (--show)

‘--show’ argument is used to inspect or verify particular jobs and their details. This command is used to verify a sqoop job called myjob.

$ sqoop job --show myjob

Execute Job (--exec)

‘--exec’ option is used to execute a saved job. The following command is used to execute a saved job called myjob.

$ sqoop job --exec myjob

**2. If a JOB Fails**

Sqoop is meant to “Fail Fast” unless you use Oozie or another workflow engine to configure actions to take upon a failure. A typical Sqoop job that is ingesting data from a source database into HDFS will copy the data to a target directory. The copied file is going to be deleted if sqoop fails without completing.

There are different stages involved in the sqoop job and we can restart the jobs. We can use the sqoop for performing the incremental and full refresh operation and save the data to the HDFS or the Hive tables.

We can rerun Sqoop to recover the job, but before that, we need to check the jobs for the reason for the failure. If there is any issue with the syntax or service you need to fix that before running the job again.

**3. No of Mappers**

**Sqoop** jobs use 4 **map tasks** by **default**. It can be modified by passing either -m or --num-**mappers** argument to the job. There is no **maximum limit** on the number **of mappers set** by **Sqoop**, but the total **number** of concurrent connections to the database is a factor to consider.

Sqoop imports data in parallel from most database sources. We can specify the number of map tasks (parallel processes) to use to perform the import by using the -m or --num-mappers argument. Each of these arguments takes an integer value which corresponds to the degree of parallelism to employ. By default, four tasks are used.

We can control the number of mappers independently from the number of files present in the directory. Export performance depends on the degree of parallelism. By default, Sqoop will use four tasks in parallel for the export process.

<https://stackoverflow.com/questions/16618753/how-to-find-optimal-number-of-mappers-when-running-sqoop-import-and-export>

**4. Sqooping Process**

Prerequisite: Hadoop Environment with Sqoop and Hive installed and working.

First we need to Create Table in MySQL

mysql> create database sqoop;

mysql> use sqoop;

mysql> create table customer(id varchar(3), name varchar(20), age           varchar(3), salary integer(10));

Query OK, 0 rows affected (0.09 sec)

mysql> desc customer;

mysql> select \* from customer;

The **customer** table **does not have any primary key**.  By default, Sqoop will identify the primary key column (if present) in a table and use it as the splitting column. If the actual values for the primary key are not uniformly distributed across its range, then this can result in unbalanced tasks. We should explicitly choose a different column with the --split-by argument. For example, --split-by id.

Since I want to import this table directly into Hive I am adding –hive-import to my Sqoop command:

sqoop import

--connect jdbc:mysql://localhost:3306/sqoop

--username root

-P

--split-by id

--columns id,name

--table customer

--target-dir /user/cloudera/ingest/raw/customers

--fields-terminated-by ","

--hive-import

--create-hive-table

--hive-table sqoop\_workspace.customers

Finally, let’s verify the output in Hive:

hive> show tables;

customers

hive> select \* from customers;

**5. Performance Tuning:-**

<https://community.cloudera.com/t5/Community-Articles/SQOOP-Performance-tuning/ta-p/248260>

1) Inserting Data in Batches--Specifies that you can group the related SQL statements into a batch when you export data.

2) Custom Boundary Queries--Specifies the range of values that we can import. We can use boundary-query if we do not get the desired results by using the split-by argument alone. When we configure the boundary-query argument, we must specify the min(id) and max(id) along with the table name.

3) Importing Data Directly into Hive--Specifies the direct import fast path when you import data from RDBMS.

4) Importing Data using Fetch-size--Specifies the number of entries that Sqoop can import at a time. Where <n> represents the number of entries that Sqoop must fetch at a time. Default is 1000.

5) Controlling Parallelism

Specifies number of map tasks that can run in parallel. Default is 4. To optimize performance, set the number of map tasks to a value lower than the maximum number of connections that the database supports.

6) Using Split-By command

Specifies the column name based on which Sqoop must split the work units.

**5. Parallelism control:-**

Sqoop imports data in parallel from most database sources. We can specify the number of map tasks (parallel processes) to use to perform the import by using the -m or --num-mappers argument. Each of these arguments takes an integer value which corresponds to the degree of parallelism to employ. By default, four tasks (4 mappers by default) are used. Some databases may see improved performance by increasing this value to 8 or 16.

[ Do not increase the degree of parallelism greater than that available within our MapReduce cluster; tasks will run serially and will likely increase the amount of time required to perform the import. Likewise, do not increase the degree of parallelism higher than that which our database can reasonably support. Connecting 100 concurrent clients to our database may increase the load on the database server to a point where performance suffers as a result.]

When performing parallel imports, Sqoop needs a criterion by which it can split the workload. Sqoop uses a *splitting column* to split the workload. By default, Sqoop will identify the primary key column (if present) in a table and use it as the splitting column.

**5. NO PRIMARY key in the Table ?**

If our table has no primary key defined then we have to give -m 1 option for importing the data or we have to provide --split-by argument with some column name, otherwise it gives an error.

Then our sqoop command will look like this:

sqoop import \

 --connect jdbc:mysql://localhost/test\_db \

 --username root \

 --password \*\*\*\* \

 --table user \

 --target-dir /user/root/user\_data \

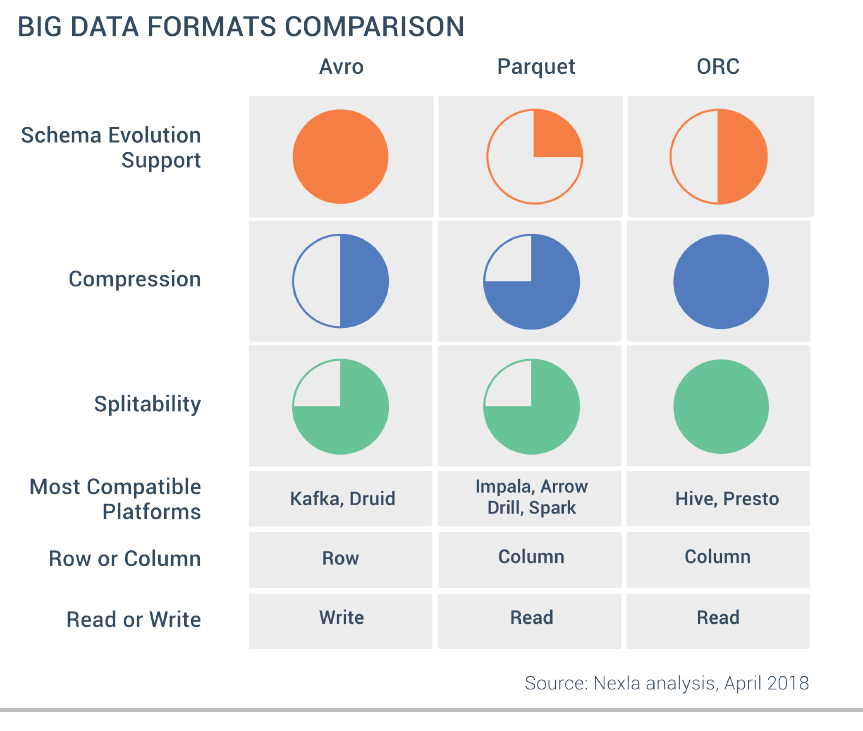
 --columns "first\_name, last\_name, created\_date"

 - m 1 or --split-by created\_date

**6. File Format list that you use?**

-Parquet , Avro, ORC, CSV

**7. AVRO vs PARQUET vs ORC file format:**



<https://blog.clairvoyantsoft.com/big-data-file-formats-3fb659903271>

**8. Copy (files / tables) into HDFS?**

To manually copy data from Oracle Database to Hadoop using Copy to Hadoop we have to take the following steps:

1. On the Oracle Database server, we have to connect to Oracle Database and generate Data Pump format files containing the table data and metadata.  
   See "[Generating the Data Pump Files](https://docs.oracle.com/bigdata/bds3212/BDSUG/generate_data_pump_file.htm#GUID-9A6D9F3F-BF74-4CD0-B237-5EAE85159267)". Data Pump files are typically used to move data and metadata from one database to another.
2. Then we have to Copy the files to HDFS on the Hadoop cluster.  
   See "[Copying the Files to HDFS](https://docs.oracle.com/bigdata/bds3212/BDSUG/generate_data_pump_file.htm#GUID-F4F247BE-6081-469F-84EC-8DA5A5EDB539)".
3. Then we need Connect to Hive and create an external table from the files.  
   See "[Creating a Hive Table](https://docs.oracle.com/bigdata/bds3212/BDSUG/generate_data_pump_file.htm#GUID-D6014CA4-B53A-4DE6-80E4-D22656FC1FDA)". We need to provide hdfs dir for external tables.
4. Query this Hive table the same as we would any other Hive table.

**9. Sqoop Features:-**

1. *Full Load*: Apache Sqoop can load the whole table by a single command. You can also load all the tables from a database using a single command.
2. *Incremental* *Load*: Apache Sqoop also provides the facility of incremental load where you can load parts of the table whenever it is updated.
3. *Parallel* *import/export*: Sqoop uses YARN framework to import and export the data, which provides fault tolerance on top of parallelism.
4. *Import* *results* *of* *SQL* *query*: You can also import the result returned from an SQL query in HDFS.
5. *Compression*: We can compress your data by using deflate(gzip) algorithm with –compress argument, or by specifying –compression-codec argument. We can also load a compressed table in *Apache Hive*.
6. *Connectors* *for* *all* *major* *RDBMS* *Databases*: Apache Sqoop provides connectors for multiple RDBMS databases, covering almost the entire circumference.
7. *Kerberos* *Security* *Integration*: Kerberos is a computer network authentication protocol which works on the basis of ‘tickets’ to allow nodes communicating over a non-secure network to prove their identity to one another in a secure manner. Sqoop supports Kerberos authentication.
8. *Load* *data* *directly* *into* *HIVE/HBase*: You can load data directly into *Apache Hive* for analysis and also dump your data in HBase, which is a NoSQL database.
9. *Support* *for* *Accumulo*: You can also instruct Sqoop to import the table in Accumulo rather than a directory in HDFS.

**12. Insert Overwrite Mechanism:**

1. INSERT OVERWRITE will overwrite any existing data in the table or partition. unless IF NOT EXISTS is provided for a partition ...In addition, INSERT OVERWRITE statement **overwrites** table data as well as a specific directory. INSERT OVERWRITE statement follows INSERT INTO SELECT statement of SQL.
2. INSERT INTO will append to the table or partition, keeping the existing data intact. (Note: INSERT INTO syntax is only available starting in version 0.8.)

**13.CDC (change data capture)**

**Change data capture** (**CDC**) is the process of capturing changes made at the **data** source and applying them throughout the enterprise. **CDC** minimizes the resources required for ETL ( extract, transform, load ) processes because it only deals with **data** changes. The goal of **CDC** is to ensure **data** synchronicity

**14.Cost-Based optimization (cbo) ?**

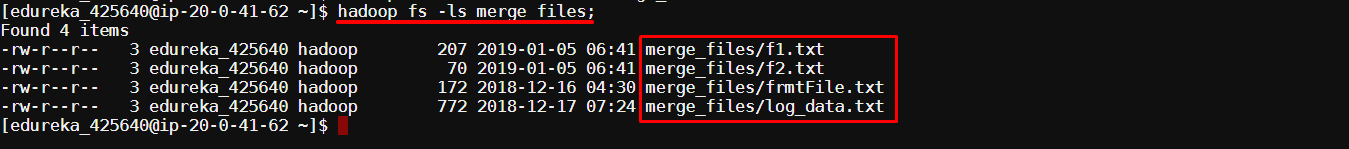
Optimization method that reduces the run time for mappings that perform join operations. With cost-based optimization, the Data Integration Service creates different plans to run a mapping and calculates a cost for each plan. The Data Integration Service runs the plan with the smallest cost. The Data Integration Service calculates cost based on database statistics, I/O, CPU, network, and memory.

**16**.**Small file with a Big file?**

There are different solutions to this subject but I/we choose to merge small files into big files. This solution provides us with cleaner hdfs. We don't want to see thousands of files on every check of hdfs.

In order to merge two or more files into one single file and store it in hdfs, we need to have a folder in the hdfs path containing the files that we want to merge.

Here, I am having a folder namely merge\_files which contains the following files that I want to merge



Then we can execute the following command to the merge the files and store it in hdfs:

hadoop fs -cat /user/edureka\_425640/merge\_files/\* | hadoop fs -put - /user/edureka\_425640/merged\_file s

The merged\_files folder need not be created manually. It is going to be created automatically to store our output when we are using the above command. We can view our output using the following command.

Here my merged\_files is storing my output result.

hadoop fs -cat merged\_files

**17. Append & Last-Modified**

There are two variants of —incremental option :-

* **append** :- it is used when rows in a source table in DB get *inserted regularly* and the table must have a numeric primary key, if not then a numeric –split-by column that is used in absence of numeric primary key. For e.g

$sqoop import –connect jdbc://mysql:/localhost/DB\_name –user username –password pasword –table tablename –incremental append –check-column colname –last-value 100

* **lastmodified** :- it is used when rows in a source table in DB get *updated regularly* and the table must have a numeric primary key, if not then a numeric –split-by column that is used in absence of numeric primary key. For e.g

$sqoop import –connect jdbc://mysql:/localhost/DB\_name –username username –password pasword –table tablename –incremental lastmodified –check-column colname –last-value “yyyy-mm–dd 0000:00:0”

**18. Data Compression**

The purpose of **compression** is to make a file, message, or any other chunk of data smaller. Data compression can significantly decrease the amount of **storage** space a file takes up.

We can enable data compression from the command line itself. All we have to do is send the compress switch to our sqoop import command. We can Try:---

$ sqoop import <switches> --compress

**Codec snappy** is a best **Sqoop data compression** technique used in the big data hadoop to reduce the storage size. There are two main **types of compression**: lossy and lossless.

**19. Boundary Query**

**Sqoop** uses a query to select minimum value for splitting, maximum value for splitting to find out **boundaries** for creating splits. This **Sqoop** operation is known as **Boundary** Value **Query**. Import data from mysql to hdfs using **Boundary query**.

**20. Split--By in Sqoop**

The command --**split**-by is **used** to specify the column of the table **used** to generate **splits** for imports. This means that it specifies which column will be **used** to create the **split** while importing the data into the cluster. Basically it is **used** to improve the import performance to achieve faster parallelism.

**21. Split--By vs Boundary Query**

--split-by id will split our data uniformly on the basis of the number of mappers

 (default 4).

Now boundary query by default is something like:--

--boundary-query "SELECT min(id), max(id) from some\_table"

But if we already know that, id starts from val1 and ends with val2. Then there is no point to calculate min() and max() operations. This will make sqoop command execution faster.

**22. Free form Query**

**Free form query** import in Apache **Sqoop** means we can import the data from the relational database using SQL **queries** instead of performing the direct imports of tables from these relational databases.

48.

49.

**50**.**What is Type Safety in a programming language?**

**Ans:**Type safety is prevention of “typed errors” in a programming language. "Type safe" usually refers to languages that ensure that an operation is working on the right kind of data at some point before the operation is actually performed. This may be at compile time or at run time.

ETL (extract, transform,load)

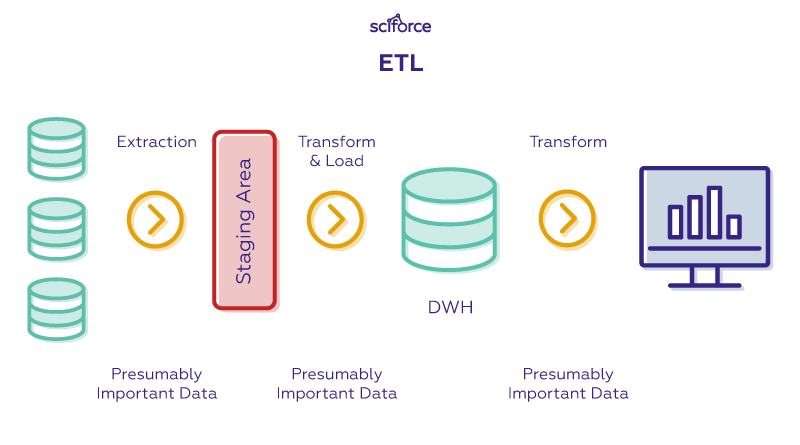
\*\*We use sqoop/flume with hadoop/spark\*\*

[ETL (Extract, Transform, and Load) Process](https://www.guru99.com/etl-extract-load-process.html)

[ETL Architecture — ETL Database](https://www.stitchdata.com/etldatabase/etl-architecture/)

**ETL** is a process in **Data Warehousing** and it stands for Extract, Transform and Load. It is a process in which an **ETL** tool..like(stitch, talend).. extracts the **data** from various **data** source systems, transforms it in the staging area and then finally, loads it into the **DataWarehouse**system.

Apache **Flume** is a tool for data ingestion in **HDFS**. It collects, aggregates and transports large amounts of streaming data such as log files, events from various sources like network traffic, social media, email messages etc. to **HDFS**. **Flume** is highly reliable & distributed.



Apache Flume is a tool for data ingestion in HDFS. It collects, aggregates and transports large amounts of streaming data such as log files, events from various sources like network traffic, social media, email messages etc. to HDFS. Flume is highly reliable & distributed.

Data is extracted from online transaction processing (OLTP) databases, today more commonly known just as 'transactional databases', and other data sources. OLTP applications have high throughput, with large numbers of read and write requests.

**Sqoop** uses **export** and **import** commands for transferring datasets from other databases to HDFS. Internally, **Sqoop** uses a map reduce program for storing datasets to HDFS. **Sqoop** provides automation for transferring data from various databases and offers parallel processing as well as fault tolerance.

**Extraction**

[ETL Extract — ETL Database](https://www.stitchdata.com/etldatabase/etl-extract/)

In the first step, data is extracted from a source ([*Salesforce*](https://www.xplenty.com/integrations/salesforce/)*, Google AdWords, etc.*) into a staging area. The staging area acts as a buffer between the data warehouse and the source data. Since data may be coming from multiple different sources, it's likely in various formats, and directly transferring the data to the warehouse may result in corrupted data. The staging area is used for data cleansing and organization.

A big challenge during the extraction phase is how your ETL tool handles structured and [unstructured data](https://www.xplenty.com/blog/processing-unstructured-data-101/). All of those unstructured items (e.g., emails, web pages, etc.) can be difficult to extract without the right tool, and we may have to create a custom solution to assisting in transferring unstructured data if we chose a tool with poor unstructured data capabilities.

**Transformation**

[ETL Transform — ETL Database](https://www.stitchdata.com/etldatabase/etl-transform/)

The data cleaning and organization stage is the transformation stage. All of that data from multiple source systems will be normalized and converted to a single system format — improving data quality and compliance. During the transformation stage, data may undergo any of the following:

* Cleaning
* Filtering
* Joining
* Sorting
* Splitting
* Deduplication  & Summarization

**Loading**

[ETL Load — ETL Database](https://www.stitchdata.com/etldatabase/etl-load)

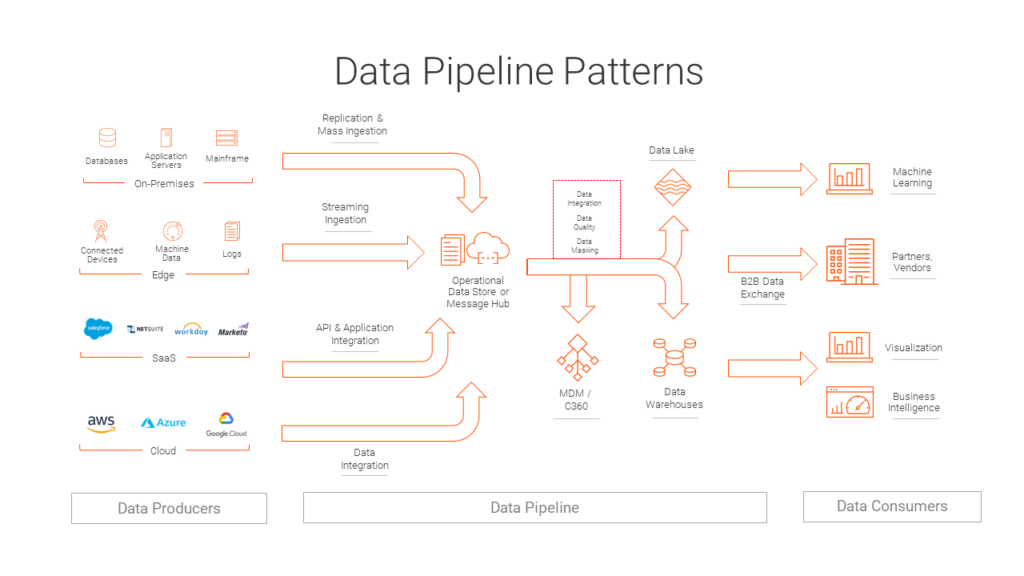
Finally, data that has been extracted to a staging area and transformed is loaded into our data warehouse. Depending upon our business needs, data can be loaded in batches or all at once. The exact nature of the loading will depend upon the data source, ETL tools, and various other factors.

DATA PIPELINE

It refers to a system for moving **data** from one system to another. The **data** may or may not be transformed, and it may be processed in real time (streaming) instead of batches.

A **data pipeline** is a series of data processing steps. Data pipelines consist of three key elements: a source, a processing step or steps, and a destination. In some data pipelines, the destination may be called a sink. Data pipelines enable the flow of data from an application to a data warehouse, from a data lake to an analytics database, or into a [payment processing](https://hazelcast.com/use-cases/payment-processing/) system.

<https://www.whizlabs.com/blog/real-time-big-data-pipeline/>



**Building data pipelines using Kafka Connect and Spark**

To copy data from a source to a destination file using Kafka, users mainly opt to choose these Kafka Connectors. For doing this, many types of source connectors and sink connectors are available for Kafka.

1st, we need to move into our Kafka’s installed directory, $*KAFKA\_HOME/config*, and check for the file: **connect-file-source.properties.**

name=local-file-source //name of your file source

connector.class=FileStreamSource //Connector class – default for FileStream

tasks.max=1 //Number of tasks to run in parallel

file=test.txt //file location - Need to change accordingly

topic=kafka\_connect-test //Name of the topic

2nd, now we need to check for the Kafka brokers’ port numbers.

By default, the port number is 9092; If we want to change it, we need to set it in the **connect-standalone.properties file.**

Bootstrap.servers = localhost:9092

Key.converter = org.apache.kafka.connect.json.JsonConverter

Value.converter = org.apache.kafka.connect.json.JsonConverter

Key.converter.schemas.enable = true

Value.converter.schemas.enable = true

3rd, Now, we have to start the Kafka servers, sources, and the zookeeper servers to populate the data into our file and let it get consumed by a Spark application.

Firstly, we have to start the zookeeper server by using the zookeeper properties as shown in the command below:

zookeeper-server-start.sh kafka\_2.11-0.10.2.1/config/zookeeper.properties

We need to keep the terminal running, then need to open another terminal, and have to start the Kafka server using the kafka server.properties as shown in the command below:

kafka-server-start.sh kafka\_2.11-0.10.2.1/config/server.properties

Now, whatever data that we enter into the file will be converted into a string and will be stored in the topics on the brokers. The data will be stored in the JSON format.

So, in our Spark application, we need to make a change to our program in order to pull out the actual data. For parsing the JSON string, we can use Scala’s JSON parser present in:

scala.util.parsing.json.JSON.parseFull

Now, we will run this application and provide some inputs to the file in real-time. Then, we can  push that data into the file.

For whatever data that we enter into the file, Kafka Connect will push this data into its topics (this typically happens whenever an event occurs, which means, whenever a new entry is made into the file).

The Spark streaming job will continuously run on the subscribed Kafka topics. We have given the time frame in seconds, so whatever data that was entered into the topics in those seconds will be taken and processed in real time and a stateful processing will be performed on it.

DATA WAREHOUSE

[**https://www.geeksforgeeks.org/data-warehouse-architecture/**](https://www.geeksforgeeks.org/data-warehouse-architecture/)

A data-warehouse is a heterogeneous collection of different data sources organised under a unified schema. There are 2 approaches for constructing data-warehouse:

**Top-down approach and Bottom-up approach**.

The data warehouse holds atomic or transaction data that is extracted from one or more source systems and integrated within a normalized, enterprise **data** model. ... The major benefit of a “top-down” **approach** is that it provides an integrated, flexible **architecture** to support downstream analytic **data** structures.

After cleansing of data, it is stored in the data warehouse as a central repository. It actually stores the metadata and the actual data gets stored in the data marts. **Note** that data warehouse stores the data in its purest form in this top-down approach.

Data mart is also a part of the storage component. There can be as many number of data marts in an organisation depending upon the functions. We can also say that a data mart contains a subset of the data stored in a data warehouse.

There are 3 approaches for constructing data-warehouse: Single Tier, Two tier and Three tier

[**https://www.guru99.com/data-warehouse-architecture.html**](https://www.guru99.com/data-warehouse-architecture.html)

**1. Design Approaches**:--

[**https://www.folkstalk.com/2011/04/data-warehouse-design-approaches.html**](https://www.folkstalk.com/2011/04/data-warehouse-design-approaches.html)

**2. Slow changing dimension (SCD)**

[Slowly Changing Dimensions](https://www.1keydata.com/datawarehousing/slowly-changing-dimensions.html)

**3. Data Lake Building:**

[Building a Data Lake: Step by Step](https://www.unicon.net/insights/articles/building-a-data-lake)

SQL

**1.Identify Duplicate value**

[Finding Duplicate Rows in SQL Server](https://www.sqlservertutorial.net/sql-server-basics/sql-server-find-duplicates/)

We can use the GROUP BY clause or ROW\_NUMBER() function to find duplicate values in a table.

Technically, I can use the [UNIQUE](https://www.sqlservertutorial.net/sql-server-basics/sql-server-unique-constraint/) constraints to enforce the uniqueness of rows in one or more columns of a table. To find the duplicate values in a table, I follow these steps:

1) First, define criteria for duplicates: values in a single column or multiple columns.

2) Second, write a query to search for duplicates.

To DELETE

To delete the duplicate rows from the table in SQL Server, you follow these steps:

1) Find duplicate rows using [GROUP BY](https://www.sqlservertutorial.net/sql-server-basics/sql-server-group-by/) clause or [ROW\_NUMBER()](https://www.sqlservertutorial.net/sql-server-window-functions/sql-server-row_number-function/) function.

2) Use [DELETE](https://www.sqlservertutorial.net/sql-server-basics/sql-server-delete/) statement to remove the duplicate rows.

* First, the CTE uses the [ROW\_NUMBER()](https://www.sqlservertutorial.net/sql-server-window-functions/sql-server-row_number-function/) function to find the duplicate rows specified by values in the first\_name, last\_name, and email columns.
* Then, the DELETE statement deletes all the duplicate rows but keeps only one occurrence of each duplicate group.

**2.RANK, DENSE\_RANK, ROW\_NUMBER**

[Similarities and Differences among RANK, DENSE\_RANK and ROW\_NUMBER Functions](https://codingsight.com/similarities-and-differences-among-rank-dense_rank-and-row_number-functions/)

The RANK, DENSE\_RANK and ROW\_NUMBER functions are used to retrieve an increasing integer value. They start with a value based on the condition imposed by the ORDER BY clause. All of these functions require the ORDER BY clause to function properly.

The RANK function is used to retrieve ranked rows based on the condition of the ORDER BY clause. For example, if you want to find the name of the car with third highest power, you can use RANK Function.

SELECT name,company, power,

RANK() OVER(ORDER BY power DESC) AS PowerRank

FROM Cars

**DROP VS TRUNCATE:**

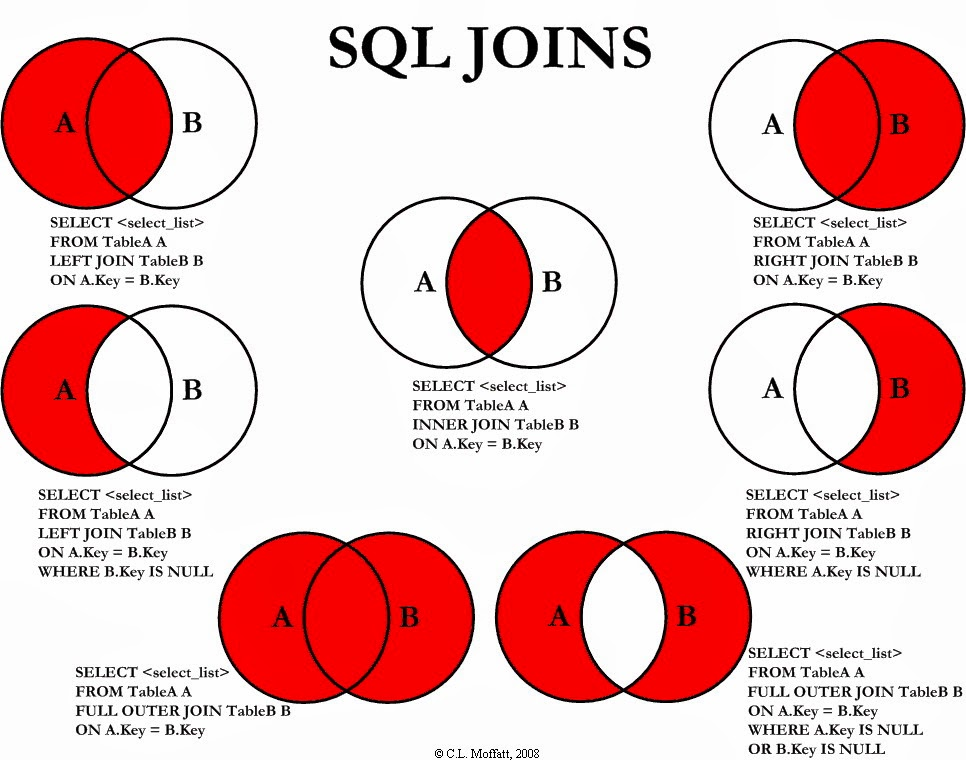
DROP is used to delete a whole database or just a table.The DROP statement destroys the objects like an existing database, table, index, or view.

A DROP statement in SQL removes a component from a relational database management system (RDBMS).

TRUNCATE statement is a Data Definition Language (DDL) operation that is used to mark the extents of a table for deallocation (empty for reuse).

DROP vs TRUNCATE

* Truncate is normally ultra-fast and it's ideal for deleting data from a temporary table.
* Truncate preserves the structure of the table for future use, unlike drop tables where the table is deleted with its full structure.
* Table or Database deletion using DROP statements **cannot** be rolled back, so it must be used wisely.

****

SQL Joins (Inner, Left, Right and Full Joins)

**INNER JOIN:** The INNER JOIN keyword selects all rows from both the tables as long as the condition satisfies. This keyword will create the result-set by combining all rows from both the tables where the condition satisfies i.e value of the common field will be the same.

Syntax:

SELECT table1.column1,table1.column2,table2.column1,....

FROM table1

INNER JOIN table2

ON table1.matching\_column = table2.matching\_column;

Example:

SELECT Orders.OrderID, Customers.CustomerName

FROM Orders

INNER JOIN Customers

ON Orders.CustomerID = Customers.CustomerID;

**LEFT JOIN:** This join returns all the rows of the table on the left side of the join and matching rows for the table on the right side of the join. The rows for which there is no matching row on the right side, the result-set will contain *null*. LEFT JOIN is also known as LEFT OUTER JOIN. **Syntax:--**

SELECT table1.column1,table1.column2,table2.column1,....

FROM table1

LEFT JOIN table2

ON table1.matching\_column = table2.matching\_column;

Example:--

Left Table: Customers                                             Right Table: Orders

SELECT Customers.CustomerName, Orders.OrderID

FROM Customers

LEFT JOIN Orders

ON Customers.CustomerID = Orders.CustomerID

ORDER BY Customers.CustomerName;

**RIGHT JOIN:** RIGHT JOIN is similar to LEFT JOIN. This join returns all the rows of the table on the right side of the join and matching rows for the table on the left side of the join. The rows for which there is no matching row on the left side, the result-set will contain *null*. RIGHT JOIN is also known as RIGHT OUTER JOIN.

**Syntax:**

SELECT table1.column1,table1.column2,table2.column1,....

FROM table1

RIGHT JOIN table2

ON table1.matching\_column = table2.matching\_column;

Example:-……Left: Orders…………Right: Employees………………………

SELECT Orders.OrderID, Employees.LastName, Employees.FirstName

FROM Orders

RIGHT JOIN Employees

ON Orders.EmployeeID = Employees.EmployeeID

ORDER BY Orders.OrderID;

**FULL JOIN:** FULL JOIN creates the result-set by combining the result of both LEFT JOIN and RIGHT JOIN. The result-set will contain all the rows from both the tables. The rows for which there is no matching, the result-set will contain *NULL* values.

**Syntax:**

SELECT table1.column1, table1.column2, table2.column1,....

FROM table1

FULL JOIN table2

ON table1.matching\_column = table2.matching\_column;

Example:

SELECT Customers.CustomerName, Orders.OrderID

FROM Customers

FULL OUTER JOIN Orders

ON Customers.CustomerID=Orders.CustomerID

ORDER BY Customers.CustomerName;

**CARTESIAN JOIN:** The CARTESIAN JOIN is also known as CROSS JOIN. In a CARTESIAN JOIN there is a join for each row of one table to every row of another table. This usually happens when the matching column or WHERE condition is not specified.

* In the absence of a WHERE condition the CARTESIAN JOIN will behave like a CARTESIAN PRODUCT . i.e., the number of rows in the result-set is the product of the number of rows of the two tables.
* In the presence of WHERE condition this JOIN will function like a INNER JOIN.
* Generally speaking, Cross join is similar to an inner join where the join-condition will always evaluate to True

**Syntax:**

SELECT table1.column1 , table1.column2, table2.column1...

FROM table1

CROSS JOIN table2;

**SELF JOIN:** A self JOIN is a regular join, but the table is joined with itself.

**Syntax:**

SELECT *column\_name(s)*

FROM *table1 T1, table1 T2*

WHERE *condition*;

Example: The following SQL statement matches customers that are from the same city:

SELECT A.CustomerName AS CustomerName1, B.CustomerName AS CustomerName2, A.City

FROM Customers A, Customers B

WHERE A.CustomerID <> B.CustomerID

AND A.City = B.City

ORDER BY A.City;

**UNION**

As you've seen, **JOINs** horizontally combine results from different tables. If you instead would like to vertically concatenate columns, you can do so with a **UNION**. The example query below combines the Age columns from both tables.



Note that with a **UNION**, the data types of both columns must be the same, but the column names can be different. (So, for instance, we cannot take the **UNION** of the Age column from the owners table and the Pet\_Name column from the pets table.)

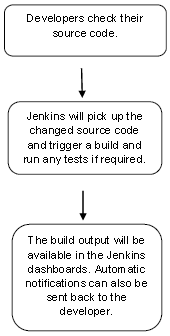
We use **UNION ALL** to include duplicate values - you'll notice that 9 appears in both the owners table and the pets table, and shows up twice in the concatenated results. If you'd like to drop duplicate values, you need only change **UNION ALL** in the query to **UNION DISTINCT**.

CI/CD JENKINS

<https://www.edureka.co/blog/ci-cd-pipeline/>

CI CD stands for continuous integration and continuous delivery/deployment.

Jenkins is a software that allows continuous integration. Jenkins will be installed on a server where the central build will take place.



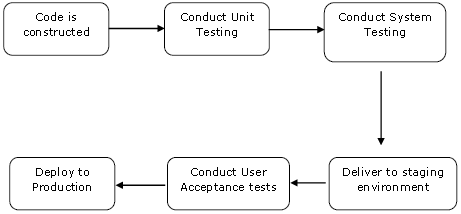
**Continuous Integration:**

Continuous Integration is a development practice that requires developers to integrate code into a shared repository at regular intervals. This concept was meant to remove the problem of finding later occurrence of issues in the build lifecycle. Continuous integration requires the developers to have frequent builds. The common practice is that whenever a code commit occurs, a build should be triggered.

Use Selenium to run automated web tests.

**Continuous Deployment:**

Jenkins provides good support for providing continuous deployment and delivery. If we look at the flow of any software development through deployment, it will be as shown like:



The main part of Continuous deployment is to ensure that the entire process which is shown above is automated. Jenkins achieves all of this via various plugins, one of them being the “Deploy to container Plugin”.

CONNECTIONS

**1.JDBC Connection in Java**

JDBC is an acronym for Java Database Connectivity. It’s an advancement for ODBC ( Open Database Connectivity ). JDBC is a standard API specification developed in order to move data from frontend to backend. This API consists of classes and interfaces written in Java. It basically acts as an interface (not the one we use in Java) or channel between our Java program and databases.

1. Import JDBC packages.

*import java.sql.\*;*

*import oracle.jdbc.driver.\*;*

*import oracle.sql.\*;*

2. Load and register the JDBC driver.

This is done by using the static registerDriver() method of the DriverManager class.

*DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());*

Alternatively, the forName() method of the java.lang.Class class can be used to load and register the JDBC driver: *Class.forName("oracle.jdbc.driver.OracleDriver");*

3. Open a connection to the database.

This is done by using the getConnection() method of the DriverManager class. A call to this method creates an object instance of the java.sql.Connection class. The getConnection() requires three input parameters, namely, a connect string, a username, and a password.

*Connection conn = DriverManager.getConnection(URL, username, passwd);*

*Connection conn = DriverManager.getConnection(URL);*

4. Create a statement object to perform a query.

This is to instantiate objects that run the query against the database connected to. This is done by the createStatement() method of the conn Connection object.

*Statement sql\_stmt = conn.createStatement();*

5. Execute the statement object and return a query resultset.

Once a Statement object has been constructed, the next step is to execute the query. This is done by using the executeQuery() method of the Statement object.

*ResultSet rset = sql\_stmt.executeQuery*

*("SELECT empno, ename, sal, deptno FROM emp ORDER BY ename");*

6. Process the resultset.

7. Close the resultset and statement objects.

8. Close the connection.

The last step is to close the database connection opened in the beginning after importing the packages and loading the JDBC drivers. This is done by a call to the close() method of the Connection class.

*conn.close();*

**2. HIVE & SPARK SQL**

Step 1: Move hive-site.xml from *$HIVE\_HOME/conf/hive-site.xml* to *$SPARK\_HOME/conf/hive-site.xml*. Make an entry regarding hive metastore uris in this file.

Step 2: Extract all the dependencies for required Spark components (in this case Spark SQL and Hive) in the build.sbt file.

Step 3: Start all Hadoop processes in the cluster with the JPS command.

Step 4: Start MySQL because Hive needs it to connect to the metastore and because Spark SQL will also need it when it connects to Hive.

Step 5: Run the Hive metastore process so that when Spark SQL runs, it can connect to metastore uris and take from it the hive-site.xml file.

**3.** [Sqoop IMPORT data from MySQL to Hive](https://stackoverflow.com/questions/22404641/using-sqoop-to-import-data-from-mysql-to-hive)

It would involve two steps.

1. Create an external hive table.

2. Import data using Sqoop.

Creation of External table : External tables in hive are kind of permanent tables and stays there even if the hive is stopped or the server goes down. "EXTERNAL" keyword is used to specify table type.

CREATE EXTERNAL TABLE IF NOT EXISTS HIVEDB.HIVE\_TABLE1 (DATE\_COL DATE, BIG\_INT\_COL BIGINT, INT\_COL INT, VARCHAR\_COL VARCHAR(221), FLOAT\_COL FLOAT);

Import the data using Sqoop : Specify the created table name while importing the data, instead of using "--hive-create" option.

sqoop import

--connect jdbc:mysql://mysqlhost/mysqldb

--username user

--password passwd

--query "SELECT table1.date\_col, table1.big\_int\_col, table1.int\_col, table1.varchar\_col, table1.float\_col FROM MYSQL\_TABLE1 AS table1 WHERE \$CONDITIONS"

--split-by table1.date\_col

--hive-import

--hive-table hivedb.hive\_table1

--target-dir hive\_table1\_data`

**4. Sqoop Export HDFS to RDBMS**

We can export data back from the HDFS to the RDBMS database. The target table must exist in the target database. It is mandatory that the table to be exported is created manually and is present in the database from where it has to be exported.

mysql> USE db;

mysql> CREATE TABLE employee (

   id INT NOT NULL PRIMARY KEY…….

This command is used to export the table data (which is in emp\_data file on HDFS) to the employee table in db database of Mysql database server.

$ sqoop export \

--connect jdbc:mysql://localhost/db \

--username root \

--table employee \

--export-dir /emp/emp\_data

The following command is used to verify the table in mysql command line.

mysql>select \* from employee;

**5. S3 through Spark**

To access data stored in Amazon S3 from Spark applications, we need to use Hadoop file APIs (SparkContext.hadoopFile, JavaHadoopRDD.saveAsHadoopFile, SparkContext.newAPIHadoopRDD, and JavaHadoopRDD.saveAsNewAPIHadoopFile) for reading and writing RDDs, providing URLs of the form s3a://bucket\_name/path/to/file. You can read and write Spark SQL DataFrames using the Data Source API.

We can access Amazon S3 by the following methods:

Without credentials:

We can Run EC2 instances with instance profiles associated with IAM roles that have the permissions we want. Requests from a machine with such a profile authenticate without credentials.

With credentials:

Specify the credentials in a configuration file, such as core-site.xml:  
<property>

    <name>fs.s3a.access.key</name>

    <value>...</value>

</property>

<property>

    <name>fs.s3a.secret.key</name>

    <value>...</value>

</property>

**6. DATA HDFS TO AMAZON S3**

AWS

**1. EC2 Instance:**

An **EC2 instance** is nothing but a virtual server in Amazon [Web services](https://www.guru99.com/web-services-tutorial.html) terminology. It stands for **Elastic Compute Cloud.** It is a web service where an AWS subscriber can request and provision a computer server in AWS cloud. We will choose t2.micro instance type, which is a 1vCPU and 1GB memory server offered by AWS.

**Create & Launch an AWS EC2 Instance:**

[**Step 2: Create Your EC2 Resources and Launch Your EC2 Instance - Amazon Elastic File System**](https://docs.aws.amazon.com/efs/latest/ug/gs-step-one-create-ec2-resources.html)

**2. Route 53:**

[Route53](http://aws.amazon.com/route53/) is a highly available, scalable, and feature rich domain name service (DNS) web service. What a DNS service does is translate a domain name like “setfive.com” into an IP address like 64.22.80.79 which allows a client’s computer to “find” the correct server for a given domain name.

**3. IAM - Identity and Access Management:**

[Identity and access management (IAM)](http://aws.amazon.com/iam/) provides enhanced security and identity management for our AWS account. In addition, it allows us to enable “multi factor” authentication to enhance the security of our AWS account.

**4. S3 Simple Storage Service:**

Simple storage service (S3) is a flexible, scalable, and highly available storage web service. S3 is like having an infinitely large hard drive where we can store files which are then accessible via a unique URL. S3 also supports access control, expiration times, and several other useful features. Additionally, the payment model for S3 is “pay as you go” so we’ll only be billed for the amount of data you store and how much bandwidth we use to transfer it in and out.

**5. Elastic Compute Cloud (EC2):**

[Elastic Compute Cloud (EC2)](http://aws.amazon.com/ec2/) is the central piece of the AWS ecosystem. EC2 provides flexible, on-demand computing resources with a “pay as you go” pricing model. Concretely, what this means is that we can “rent” computing resources for as long as we need them and process any workload on the machines we’ve provisioned. Because of its flexibility, EC2 is an attractive alternative to buying traditional servers for unpredictable workloads.

**6. Cloud Watch:**

[CloudWatch](http://aws.amazon.com/cloudwatch/) provides monitoring for AWS resources including EC2 and EBS. CloudWatch enables administrators to view and collect key metrics and also set a series of alarms to be notified in case of trouble. In addition, CloudWatch can aggregate metrics across EC2 instances which provides useful insight into how your entire stack is operating.

**7. AWS LAMBDA:**

AWS Lambda lets us run code without provisioning or managing servers. We pay only for the compute time we consume. With Lambda, we can run code for virtually any type of application or backend service - all with zero administration. Just need to upload our code and Lambda takes care of everything required to run and scale our code with high availability. We can set up our code to automatically trigger from other AWS services or call it directly from any web or mobile app.

**8. Amazon Aurora:**

Amazon Aurora is a MySQL and PostgreSQL compatible [relational database](https://aws.amazon.com/relational-database/) built for the cloud, that combines the performance and availability of traditional enterprise databases with the simplicity and cost-effectiveness of open source databases.

Amazon Aurora is up to five times faster than standard [MySQL](https://aws.amazon.com/rds/mysql/what-is-mysql/) databases and three times faster than standard PostgreSQL databases. It provides the security, availability, and reliability of commercial databases at 1/10th the cost. Amazon Aurora is fully managed by [Amazon Relational Database Service (RDS)](https://aws.amazon.com/rds/), which automates time-consuming administration tasks like hardware provisioning, database setup, patching, and backups.

Amazon Aurora features a distributed, fault-tolerant, self-healing storage system that auto-scales up to 64TB per database instance. It delivers high performance and availability with up to 15 low-latency read replicas, point-in-time recovery, continuous backup to Amazon S3, and replication across three Availability Zones (AZs).

**9. Amazon DynamoDB:**

Amazon DynamoDB is a key-value and document database that delivers single-digit millisecond performance at any scale. It's a fully managed, multiregion, multimaster, durable database with built-in security, backup and restore, and in-memory caching for internet-scale applications. DynamoDB can handle more than 10 trillion requests per day and can support peaks of more than 20 million requests per second.

Many big and fast growing companies like: Lyft, Airbnb, and Redfin as well as enterprises such as Samsung, Toyota, and Capital One depend on the scale and performance of DynamoDB to support their mission-critical workloads.

**10. Amazon RDS:**

Amazon Relational Database Service (Amazon RDS) makes it easy to set up, operate, and scale a relational database in the cloud. It provides cost-efficient and resizable capacity while automating time-consuming administration tasks such as hardware provisioning, database setup, patching and backups. It frees us to focus on our applications so we can give them the fast performance, high availability, security and compatibility they need.

Amazon RDS is available on several database instance types - optimized for memory, performance or I/O - and provides us with six familiar database engines to choose from, including [Amazon Aurora](https://aws.amazon.com/rds/aurora/), [PostgreSQL](https://aws.amazon.com/rds/postgresql/), [MySQL](https://aws.amazon.com/rds/mysql/), [MariaDB](https://aws.amazon.com/rds/mariadb/), [Oracle Database](https://aws.amazon.com/rds/oracle/), and [SQL Server](https://aws.amazon.com/rds/sqlserver/). We can use the [AWS Database Migration Service](https://aws.amazon.com/dms/) to easily migrate or replicate our existing databases to Amazon RDS.

**11. Amazon Athena:**

Amazon Athena is an interactive query service that makes it easy to analyze data in Amazon S3 using standard SQL. Athena is serverless, so there is no infrastructure to manage, and you pay only for the queries that you run.

Athena is easy to use. Simply point to your data in Amazon S3, define the schema, and start querying using standard SQL. Most results are delivered within seconds. With Athena, there’s no need for complex ETL jobs to prepare your data for analysis. This makes it easy for anyone with SQL skills to quickly analyze large-scale datasets.

**12. Amazon EMR:**

[Getting Started: Analyzing Big Data with Amazon EMR - Amazon EMR](https://docs.aws.amazon.com/emr/latest/ManagementGuide/emr-gs.html)

Amazon EMR is the industry leading cloud-native big data platform for processing vast amounts of data quickly and cost-effectively at scale. Using open source tools such as [Apache Spark](https://aws.amazon.com/emr/features/spark/), [Apache Hive](https://aws.amazon.com/emr/features/hive/), [Apache HBase](https://aws.amazon.com/emr/features/hbase/), [Apache Flink](https://aws.amazon.com/blogs/big-data/use-apache-flink-on-amazon-emr/), [Apache Hudi (Incubating)](https://aws.amazon.com/emr/features/hudi/), and [Presto](https://aws.amazon.com/emr/features/presto/), coupled with the dynamic scalability of [Amazon EC2](https://aws.amazon.com/ec2/) and scalable storage of [Amazon S3](https://aws.amazon.com/s3/), EMR gives analytical teams the engines and elasticity to run Petabyte-scale analysis for a fraction of the cost of traditional on-premises clusters.

**13. Amazon Kinesis:**

Amazon Kinesis makes it easy to collect, process, and analyze real-time, streaming data so we can get timely insights and react quickly to new information. Amazon Kinesis offers key capabilities to cost-effectively process streaming data at any scale, along with the flexibility to choose the tools that best suit the requirements of your application. With Amazon Kinesis, we can ingest real-time data such as video, audio, application logs, website clickstreams, and IoT telemetry data for machine learning, analytics, and other applications. Amazon Kinesis enables us to process and analyze data as it arrives and respond instantly instead of having to wait until all your data is collected before the processing can begin.

**14. Amazon Redshift:**

**Amazon Redshift** is a fully-managed petabyte-scale cloud based data warehouse product designed for large scale data set storage and analysis. It is also **used to** perform large scale database migrations. **Data** in Amazon **Redshift data** warehouse is **stored** in a columnar fashion which drastically reduces the I/O on disks. Columnar **storage** reduces the number of disk I/O requests and minimizes the amount of **data** loaded into the memory to execute a query.

**15. Amazon Glue:**

AWS Glue is a fully managed extract, transform, and load (ETL) service that makes it easy for customers to prepare and load their data for analytics. We can create and run an ETL job with a few clicks in the AWS Management Console. We can simply point AWS Glue to our data stored on AWS, and AWS Glue discovers our data and stores the associated metadata (e.g. table definition and schema) in the AWS Glue Data Catalog. Once cataloged, our data is immediately searchable, queryable, and available for ETL.

**16. AWS Data Pipeline:**

AWS Data Pipeline is a web service that helps us reliably process and move data between different AWS compute and storage services, as well as on-premises data sources, at specified intervals. With AWS Data Pipeline, we can regularly access our data where it’s stored, transform and process it at scale, and efficiently transfer the results to AWS services such as Amazon S3, Amazon RDS, Amazon DynamoDB, and Amazon EMR.

AWS Data Pipeline helps us easily create complex data processing workloads that are fault tolerant, repeatable, and highly available. We don’t have to worry about ensuring resource availability, managing inter-task dependencies, retrying transient failures or timeouts in individual tasks, or creating a failure notification system. AWS Data Pipeline also allows us to move and process data that was previously locked up in on-premises data silos.

**17. Create a Data Lake step-by-step:**

<https://docs.aws.amazon.com/lake-formation/latest/dg/getting-started-tutorial.html>

SCALA

**1.Tuples:**

A **tuple** is a sequence of immutable Python objects. **Tuples** are sequences, just like lists. The differences between **tuples** and lists are, the **tuples** cannot be changed unlike lists and **tuples** use parentheses, whereas lists use square brackets. Creating a **tuple** is as simple as putting different comma-separated values.      val t = (1, "hello", Console)

**2.Traits:**

A trait encapsulates method and field definitions, which can then be reused by mixing them into classes. Unlike class inheritance, in which each class must inherit from just one superclass, a class can mix in any number of traits.

**3. Closures:**

A closure is a function, whose return value depends on the value of one or more variables declared outside this function.

**4. Monad in Scala:**

Monads are containers. That means they contain some sort of element. Instead of allowing us to operate on these elements directly, the container itself has certain properties. We can then work with the container and the container works with the element within. This will help us to create more concise code; as mentioned, Scala collections use monads internally for this reason, such as Option[T].

**4. Singleton Object**

Singleton object is an object which is declared by using object keyword instead of class. No object is required to call **methods** declared inside a singleton object. In scala, there is no static concept. So scala creates a singleton object to provide entry point for your program execution.

The **purpose** of the **singleton** class is to control object creation, limiting the number of objects to only one. The **singleton** allows only one entry **point** to create the new instance of the class. **The disadvantage** of singletons is that they make unit testing very hard.

**5. Singleton object VS companion object?**

A **companion object** is an **object** that's declared in the same file as a class , and has the same name as the class. A **companion object** and its class can access each other's private members. A **companion object's** apply method lets ua create new instances of a class without using the new keyword.

Well to put it simply. A **singleton object** named the same as a class is called a **companion object**. Also a **companion object** must be defined inside the same source file as the class. ... As the name reflects, **Singleton** in Scala has one instance throughout the application life-cycle.

**5. Higher Order Function:**

a higher-order function is a function that does at least one of the following: takes one or more functions as arguments, returns a function as its result. All other functions are first-order functions. In mathematics higher-order functions are also termed operators or functionals.

For **example**, the map **function** on arrays is a **higher order function**. The map **function** takes a **function** as an argument. The map **function** is one of the many **higher**-**order functions** built into the language. **sort**, reduce, filter, forEach are other **examples** of **higher**-**order functions** built into the language.

**6. ofDim in Scala:**

ofDim() is a method in Scala that lets us create multidimensional arrays. Since these let us store data in more than one dimension, we can store data like in a matrix.

**7.Exception handling in Scala**

Exceptions are the events that can change the flow of control through a program. When we want to handle exceptions, we should use a try{…} & catch{…} block as we would in [Java](https://intellipaat.com/java-training/) except that the catch block uses matching to identify and handle the exceptions.

Throwing Exceptions

Throwing an exception looks the same as in Java. You create an exception object and then you throw it with the throw keyword:

throw new IllegalArgumentException

Catching Expressions

**8. Regular Expression (reg-ex):**

Regular expressions are patterns that permit us to “match” various string values in a variety of ways. Scala uses import scala.util.matching.Regex to implement regular expression concepts.

import scala.util.matching.Regex

object Demo {

   def main(args: Array[String]) {

      val pattern = "Scala".r

      val str = "Scala is Scalable and cool"

      println(pattern findFirstIn str)

   }

}

**9. vector in Scala:**

A vector is a general-purpose data structure that is immutable. We can use it when we want to hold a huge number of elements and want random access to them. This data structure extends the trait IndexedSeq and the abstract class AbstractSeq.

1. scala> var v1=**Vector**(7,2,4,3,1)
2. v1: scala.collection.immutable.Vector[Int] = **Vector**(7, 2, 4, 3, 1)

**10.Null, Nil, None, and Nothing differ in Scala?**

* Null represents the absence of value. It depicts the absence of type information for complex types inherited from AnyRef.
* Nil denotes the end of a List.
* None is the value of an Option with no value in it.
* Nothing is the lowest type in the entire type system. All values under AnyVal and AnyRef fall under this. A method that throws an exception uses Nothing as a return type.

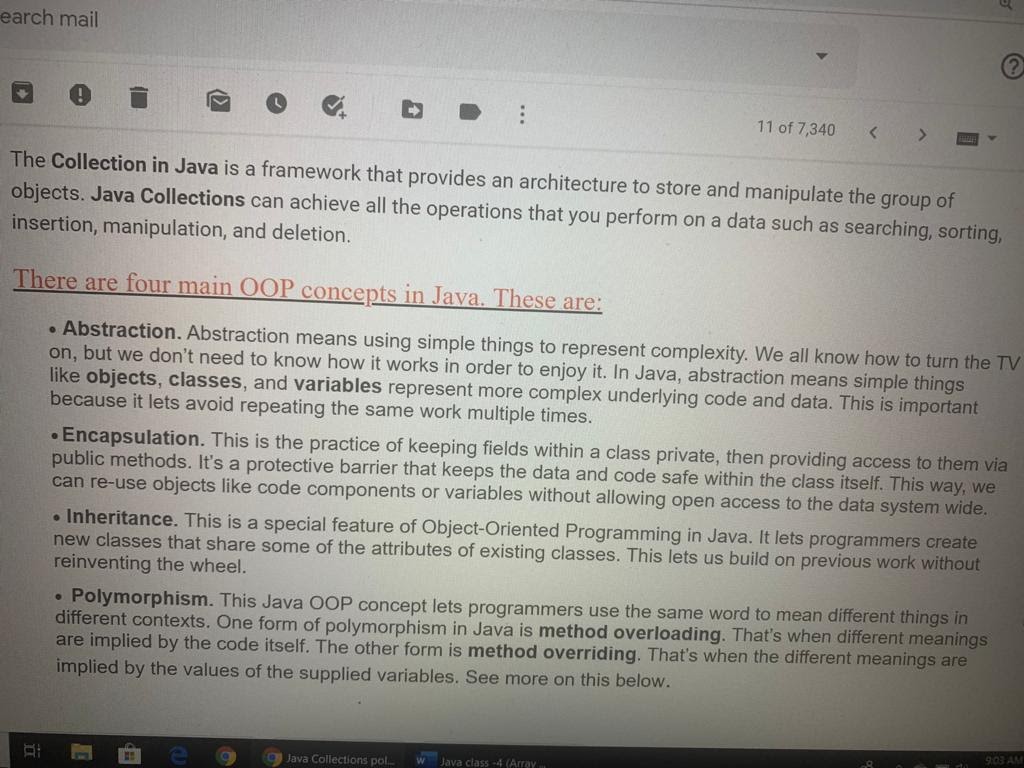
**11.Final, Finally & Finalize:**

Final class can't be inherited, final method can't be overridden and final variable value can't be changed.

Finally is used to place important code, it will be executed whether an exception is handled or not.

Finalize is used to perform clean up processing just before the object is garbage collected.

**12. Abstraction, Encapsulation, Inheritance, Polymorphism:**



**13.Build & Deploy a Scala project/App**

Almost all Scala projects start from a **build.sbt** file. As a rule, it contains meta information about the project: name, version, dependencies…

name := "sbt-build-example"

version := "1.0"

scalaVersion := "2.12.2"

libraryDependencies ++= Seq(

 "com.typesafe.akka" %% "akka-http" % "10.0.9"

)

 To be more precise, here we have a single dependency. For sure, other projects contain more than one. But for this demonstration, it’s good enough.

Then we have a Main class where all “business logic” happens.

object Main extends App {

    implicit val system = ActorSystem()

  implicit val ec = system.dispatcher

  implicit val materializer = ActorMaterializer()

  val routes = pathPrefix("ping") {

    get {

      complete("pong")

    }

  }

 val bindingFuture: Future[ServerBinding] = Http().bindAndHandle(routes, "127.0.0.1", 5000)

 }

So the demo app has only one endpoint which returns “pong” when we make a GET request to ping.

**So how to build this app? How to make it deployable?** In order to perform this, we need to add [sbt-native-packager](http://sbt-native-packager.readthedocs.io/) plugin to the project. we can do it by adding a single line of code to the plugins.sbt file.

|  |  |
| --- | --- |
| 1 | addSbtPlugin("com.typesafe.sbt" % "sbt-native-packager" % "1.2.0") |

Then we need to add the following line of code to the build.sbt file as well:

|  |  |
| --- | --- |
| enablePlugins(JavaAppPackaging) | enablePlugins(JavaAppPackaging) |

Then need to Open a new terminal, need to navigate to the root folder of the project and execute the following command:

|  |  |
| --- | --- |
| 1 | sbt stage |

This command generates a new folders in our project directory:

target/universal/stage/

There are two folders in the stage directory: bin & lib.

The first one contains launchers (Linux / Mac / Windows). The second one contains all dependencies and a jar with the application classes.

So What’s next? As we may guess, the content of the stage directory can be moved to our server and launched there by executing following command in the console:

|  |  |
| --- | --- |
| 1 | ./bin/app-name |

This command must be run from the stage folder. By the way, app-name may vary depending on a project name (package name) which we can specify in our build.sbt file.

Ni-Fi

(<https://nifi.apache.org/docs.html>)

NiFi was built to automate the flow of data between systems.While the term 'dataflow' is used here to mean the automated and managed flow of information between systems. It allows us to easily design data flows and monitor the execution and queues, evaluate what data went. It provides a web-based User Interface for creating, monitoring, & controlling data flows.where and make changes in realtime to the flow

NiFi is not only an ingestion tool. It's a [data logistics platform]. This means that NiFi enables easy collection, curation, analysis and action on any data anywhere (edge, cloud, data center) with built-in end-to-end security and provenance.

[Information Logistics deals with the flow of information between human and / or machine actors within or between any number of organizations that in turn form a value creating network.]

Apache NiFi provides users the ability to build very large and complex DataFlows using NiFi. This is achieved by using the basic components: Processor, Funnel, Input/Output Port, Process Group, and Remote Process Group. These can be thought of as the most basic building blocks for constructing a DataFlow.

[<http://nifi.apache.org/docs/nifi-docs/html/user-guide.html#building-dataflow>]

**SAMPLE work**

<https://datamelt.weebly.com/blog/apache-nifi-starter-part-2>

**What happens when we send more data than the connection can handle?**

If the number of FlowFiles or the quantity of data goes above the defined threshold, *backpressure* is applied. The Flow Controller won’t schedule the previous processor to run again until there is room in the queue.

Let’s say we have a limit of 10,000 FlowFiles between two processors. At some point, the connection has 7,000 elements in it. It is ok since the limit is 10,000. P1 can still send data through the connection to P2.

Now let’s say that processor one sends 4,000 new FlowFiles to the connection.

7,0000 + 4,000 = 11000 → We go above the connection threshold of 10,000 FlowFiles.

The limits are soft limits, meaning they can be exceeded. However, once they are, the previous processor, P1 won’t be scheduled until the connector goes back below its threshold value — 10 000 FlowFiles.

This simplified example gives the big picture of how [backpressure](https://en.wikipedia.org/wiki/Back_pressure) works.

The idea of exceeding a limit may sound odd. When the number of FlowFiles or the associated data go beyond the threshold, a [swap mechanism](https://community.hortonworks.com/articles/184990/dissecting-the-nifi-connection-heap-usage-and-perf.html) is triggered.

Docker

**1.What is Docker?**

There are two ways of looking at Docker. The first approach involves seeing Docker containers as really lightweight virtual machines, while the second approach is to see Docker as a software packaging and delivery platform.

**Docker** provides the ability to package and run an application in a loosely isolated environment called a **container**. The isolation and security allow us to run many **containers** simultaneously on a given host.

Docker enables more efficient use of system resources.

Instances of containerized apps use far less memory than virtual machines, they start up and stop more quickly, and they can be packed far more densely on their host hardware.

**2.Why is it used for ?**

A **Docker container** is an open source software development platform. Its main benefit is to package applications in **containers**, allowing them to be portable to any system running a Linux or Windows operating system (OS). A Windows machine can run Linux **containers** by using a virtual machine (VM). **Docker**, a **container** management tool, is used in **DevOps** to manage software parts as isolated, self-sufficient **containers**, which can be deployed and run in any environment. **Docker** reduces back and forth between Dev and Ops in Continuous Deployment, which eliminates overheads and cuts operational costs.

**3.When To Use Docker?**

Docker is a basic tool, like git or java, that you should start incorporating into your daily development and ops practices.

* Use Docker as version control system for your entire app's operating system
* Use Docker when you want to distribute/collaborate on your app's operating system with a team
* Use Docker to run your code on your laptop in the same environment as you have on your server (try the [building tool](https://github.com/centurylinklabs/building))
* Use Docker whenever your app needs to go through multiple phases of development (dev/test/qa/prod, try [Drone](https://drone.io/) or [Shippable](https://app.shippable.com/), both do Docker CI/CD)
* Use Docker with your Chef Cookbooks and [Puppet Manifests](https://puppetlabs.com/blog/building-puppet-based-applications-inside-docker) (remember, Docker doesn't do configuration management)

**4.Docker Image**

A **Docker image** is a read-only template that contains a set of instructions for creating a container that can run on the **Docker** platform. It provides a convenient way to package up applications and pre-configured server environments, which we can use for our own private use or share publicly with other **Docker** users.

**Images** can exist without **containers**, whereas a **container** needs to run an **image** to exist. Therefore, **containers** are dependent on **images** and use them to construct a run-time environment and run an application. The two concepts exist as essential components (or rather phases) in the process of running a **Docker container**.

**5.How does a docker image work?**

**Docker** uses a client-server architecture. The **Docker** client talks to the **Docker** daemon, which **does** the heavy lifting of building, running, and distributing your **Docker** containers. The **Docker** client and daemon can run on the same system, or you can connect a **Docker** client to a remote **Docker** daemon.

**5. Create a Docker Image**

<https://www.scalyr.com/blog/create-docker-image/>

1. Step 1: **Create** a Base **Container**. Let's get started by **creating** a running **container**. ...
2. Step 2: Inspect **Images**. ...
3. Step 3: Inspect Containers. ...
4. Step 4: Start the **Container**. ...
5. Step 5: Modify the Running **Container**. ...
6. Step 6: **Create** an **Image** From a **Container**. ...
7. Step 7: Tag the **Image**. ...
8. Step 8: **Create Images** With Tags.

**6.Inside a docker image?**

A **Docker image** includes the elements needed to run an application as a **container** -- such as code, config files, environment variables, libraries and run time. If the **image** is deployed to a **Docker** environment it can then be executed as a **Docker container**.

**7.Dockerizing an Application**

**Dockerizing** an **application** is the process of converting an **application** to run within a Docker container. ... Making an **application** use environment variables when it relies on configuration files. Sending **application** logs to STDOUT/STDERR when it defaults to files in the container's file system.

**8.Docker layers**

Each container is an image with a readable/writeable **layer** on top of a bunch of read-only **layers**. These **layers** (also called intermediate images) are generated when the commands in the **Dockerfile** are executed during the **Docker** image build. ... And each **layer** is made up of the file generated from running that command.

**9.Docker Swarm**

A Docker Swarm is a group of either physical or virtual machines that are running the Docker application and that have been configured to join together in a cluster. ... Docker swarm is a container orchestration tool, meaning that it allows the user to manage multiple containers deployed across multiple host machines. A **swarm** consists of multiple **Docker** hosts which run in **swarm** mode and act as managers.  One of the key benefits associated with the operation of a **docker swarm is** the high level of availability offered for applications.

Kubernetes

**Kubernetes** is a system for managing containerized applications across a cluster of nodes. In **simple terms**, suppose we have a group of machines (e.g. VMs) and containerized applications (e.g. Dockerized applications), and **Kubernetes** will help us to easily manage those apps across those machines.

**Kubernetes** is a vendor-agnostic cluster and container management tool, open-sourced by Google in 2014. It provides a “platform for automating deployment, scaling, and operations of application containers across clusters of hosts”.

Developers love to use **Kubernetes** as it has solved their big problem- taking any application from its development environment to production environment. .

**1. Kubernetes VS Docker?**

**Docker** is a platform and tool for building, distributing, and running **Docker** containers. ... **Kubernetes** is a container orchestration system for **Docker** containers that is more extensive than **Docker** Swarm and is meant to coordinate clusters of nodes at scale in production in an efficient manner.

A fundamental difference between **Kubernetes** and **Docker** is that **Kubernetes** is meant to run across a cluster while **Docker** runs on a single node. **Kubernetes** is more extensive than **Docker** Swarm and is meant to coordinate clusters of nodes at scale in production in an efficient manner.

**2.Can Kubernetes run without Docker?**

**Kubernetes can run without Docker** and **Docker can function without Kubernetes**. But **Kubernetes can** (and **does**) benefit greatly from **Docker** and vice versa. **Docker** is a standalone software that **can** be installed on any computer to **run** containerized applications. ... **Kubernetes** turns it up to 11, so to speak.

**3.When should I use Kubernetes?**

When we are looking for Zero-downtime deployments, fault tolerance, high availability, scaling, scheduling, and self-healing add significant value in **Kubernetes**. We can **use** it to mount volumes for stateful applications. It allows us to store confidential information as secrets. We can **use** it to validate the health of our services.

**Kubernetes** allows us to deploy cloud-native applications anywhere and manage them exactly as we like everywhere. As most modern software developers can attest, containers have provided us with dramatically more flexibility for running cloud-native applications on physical and virtual infrastructure.

**4.Advantages of Kubernetes**

Kubernetes leverages the power of containers while simplifying the management of services and machines in a cluster. Kubernetes Clusters abstract their underlying computing resources, allowing users to deploy workloads to the entire cluster as opposed to a particular server.

**5.Features of Kubernetes?**

Here are the essential features of Kubernetes:

* Automated Scheduling.
* Self-Healing Capabilities.
* Automated rollouts & rollback.
* Horizontal Scaling & Load Balancing.
* Offers environment consistency for development, testing, and production.
* Infrastructure is loosely coupled to each component and can act as a separate unit.

**6. Kubernetes Pod**

Pods are the smallest deployable units of computing that we can create and manage in Kubernetes. A Pod is a group of one or more [containers](https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/#why-containers), with shared storage/network resources, and a specification for how to run the containers.

A **Kubernetes pod** is a group of containers that are deployed together on the same host.

Kubernetes doesn't run **containers** directly; instead it wraps one or more **containers** into a higher-level structure called a **pod**. Any **containers in the** same **pod** will share the same resources and local network. **Pods** are used as the unit of replication in Kubernetes.