Spark joins:

Basically when dealing with data in spark ,Spark internally performs some join operations on distributed engine in order to give massive parallelism and throughput.

So intelligently spark changes it action of course depending upon the Dataset and Volume consumption.

There are three types of joins in spark:-

1)Broadcast hash join

2)Shuffle hash join

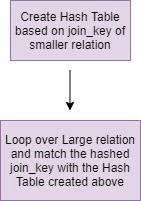
3)Sort merge join

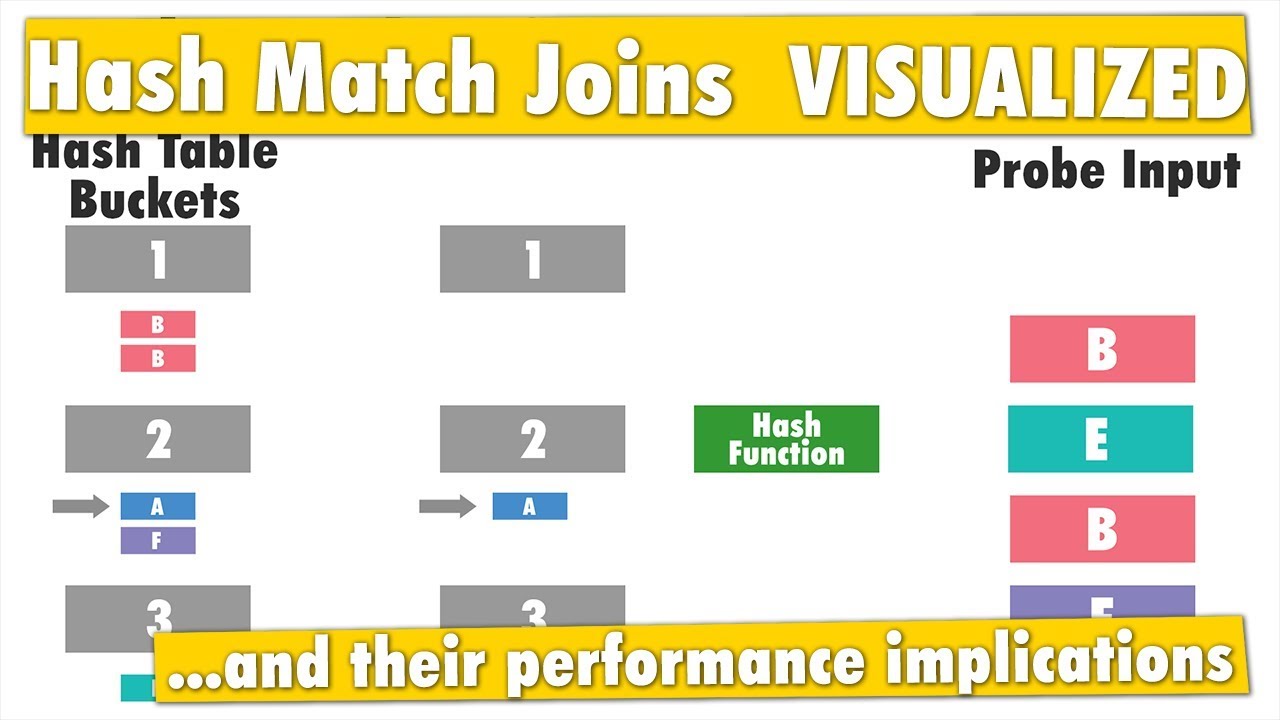
First of all Before learning Joins lets understand Hash Table-join

Hash join :-

The Hash Join algorithm is used to perform the natural join or equi join operations. The concept behind the Hash join algorithm is to partition the tuples of each given relation into sets. The partition is done on the basis of the same hash value on the join attributes. The hash function provides the hash value. The main goal of using the hash function in the algorithm is to reduce the number of comparisons and increase the efficiency to complete the join operation on the relations.

So extending the power of hash join 3 new spark joins above discussed are come in to existence with Spark



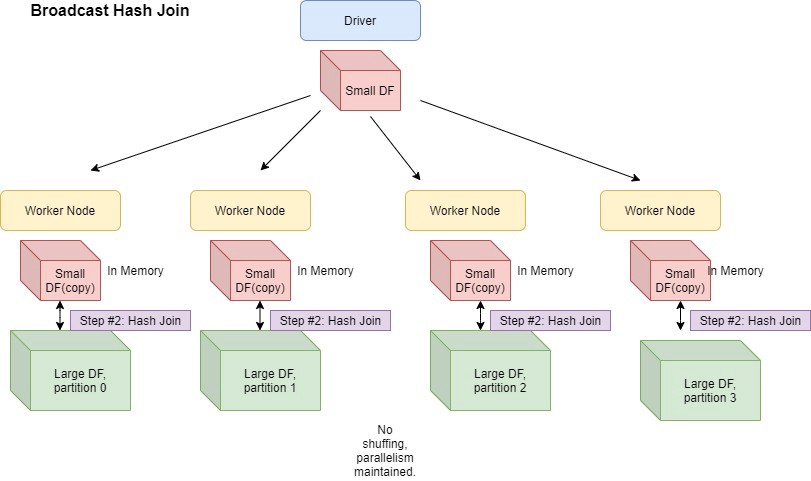


BroadCast Join:-

BroadCast join **performs a join on two relations by first broadcasting the smaller one to all Spark executors, then evaluating the join criteria with each executor's partitions of the other relation**.

In broadcast hash join, copy of one of the join relations are being sent to all the worker nodes and **it saves shuffling cost**. This is useful when you are joining a large relation with a smaller one. It is also known as map-side join(associating worker nodes with mappers).

Spark deploys this join strategy when the size of one of the join relations is less than the threshold values(default 10 M). The spark property which defines this threshold is spark.sql.autoBroadcastJoinThreshold(configurable).



Shuffle Hash join:-

There are two phases

a)Shuffle Phase

b) Hash phase

Shuffle Phase:-

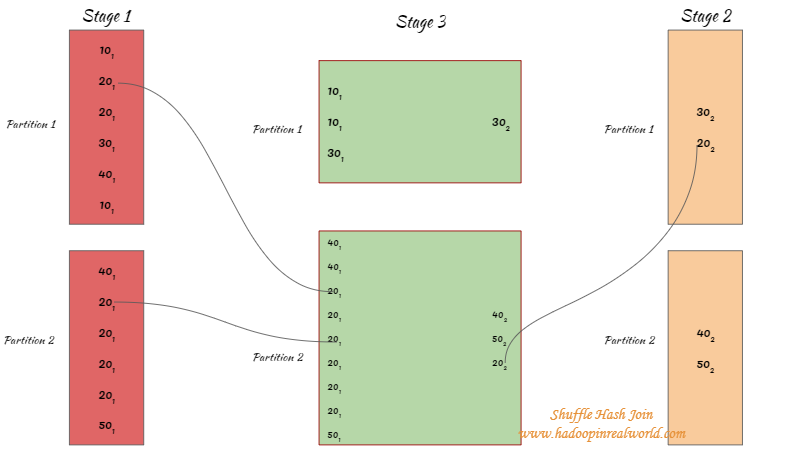
Second type of Join in Spark Joins is data gets shuffled o join key and move across the partitions.

Two tables have the same keys that end up with same partition so that join required on those data will be available in same partition

Hash Phase:-

The data on each partition performs a classic single nod hash join algorithm.

It reduces lookup time and join faster.



When Used:-

1. Any partition of build side could fit in memory.
2. One table is much smaller than other, the cost to build a hash table on smaller table is

Smaller than sorting larger table.

Issues:-

1. It breaks one big join of 2 smaller chunks of localized relatively smaller branch.
2. Shuffle is very expensive operation as it involves a lot of n/w intensive movement of data across the nodes of cluster.

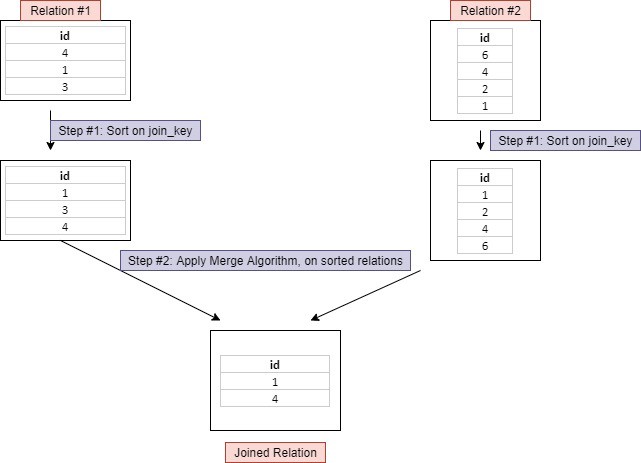
Sort Merge Join:-

1)If the matching join keys are sortable and not eliglible for bradcast join or shuffle hash join.

2)It is very scalable approach and performs better than other joins most of the time

3)it can spill the data to disk and doesn’t require the entire data fit into memory.

It traits from Map-reduce Program.



It has 3 phases:-

Shuffle phase:- two large tables are repartitioned as per the join keys across the partitions parallel.

Sort Phase:- Sort the data within each partition parallel.

Merge Phase:- Join the two sorted partitioned data This is basically merging of dataset by interating over the elements by joining the rows having the same value for the join key.

Useful:-

Unwanted data shuffle.

Repartiton and Coalese:-

Coalesce method reduces the number of partitions in a DataFrame.

Coalesce avoids full shuffle, instead of creating new partitions, it shuffles the data using Hash Partitioner (Default) and adjusts into existing partitions, this means it can only decrease the number of partitions.

Hence, some of the executors which have less amount of data to process will sit idle after completing the task assigned to them, and others with more data will be working.

Once all the transformations are applied and you want to save all the data into fewer files(no. of files = no.of partitions) instead of many files, use coalesce.

ex. df.coalesce(1).write.format('json').save('myfile.json')

Repartition:-

Repartition method can be used to either increase or decrease the number of partitions in a DataFrame.

Repartition is a full Shuffle operation, whole data is taken out from existing partitions and equally distributed into newly formed partitions.

Hence the amount of work is equally distributed among all the executors and better parallelism can be achieved.

If you have loaded a dataset, that includes huge data, and a lot of transformations that need an equal distribution of load on executors, you need to use Repartition. Before using repartition first check how many partitions are there

Ex. df.rdd.getNumPartitions()

Group by and Reduce by:-

On applying groupByKey() on a dataset of (K, V) pairs, the data shuffle according to the key value K in another RDD. In this transformation, lots of unnecessary data transfer over the network.

Spark provides the provision to save data to disk when there is more data shuffling onto a single executor machine than can fit in memory.

Example:

val data = spark.sparkContext.parallelize(Array(('k',5),('s',3),('s',4),('p',7),('p',5),('t',8),('k',6)),3)

val group = data.groupByKey().collect()

group.foreach(println)

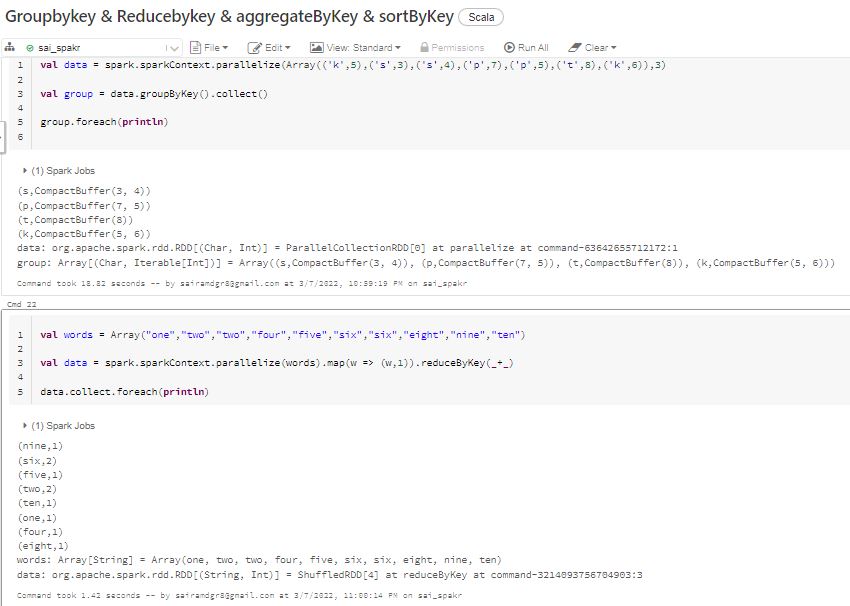
On applying reduceByKey on a dataset (K, V), before shuffling of data the pairs on the same machine with the same key are combined.

Example:

val words = Array("one","two","two","four","five","six","six","eight","nine","ten")

val data = spark.sparkContext.parallelize(words).map(w => (w,1)).reduceByKey(\_+\_)

data.collect.foreach(println)



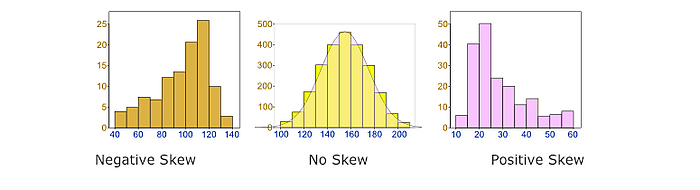
Spark collect v/s take

Spark dataframe: collect () vs select () Calling collect() on an RDD will return the entire dataset to the driver which can cause out of memory and we should avoid that.

Spark Take Function. In Spark, the take function behaves like an array. It receives an integer value (let say, n) as a parameter and returns an array of first n elements of the dataset.

Spark Skewness:-

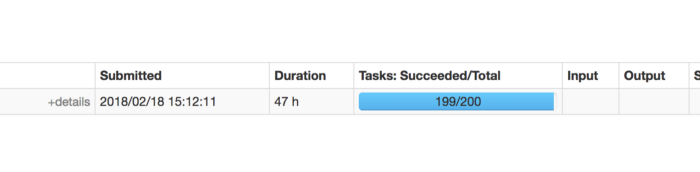
 Skewness is the statistical term, which refers to the value distribution in a given dataset. When we say that there is highly skewed data, it means that some column values have more rows and some very few, i.e., the data is not properly/evenly distributed. Data skewness affects the performance and parallelism in any distributed system. You can learn more about the use cases related to skewed data statistics [here](https://www.clairvoyant.ai/cloud-services).



Joining two or more large tables having skewed data in Spark

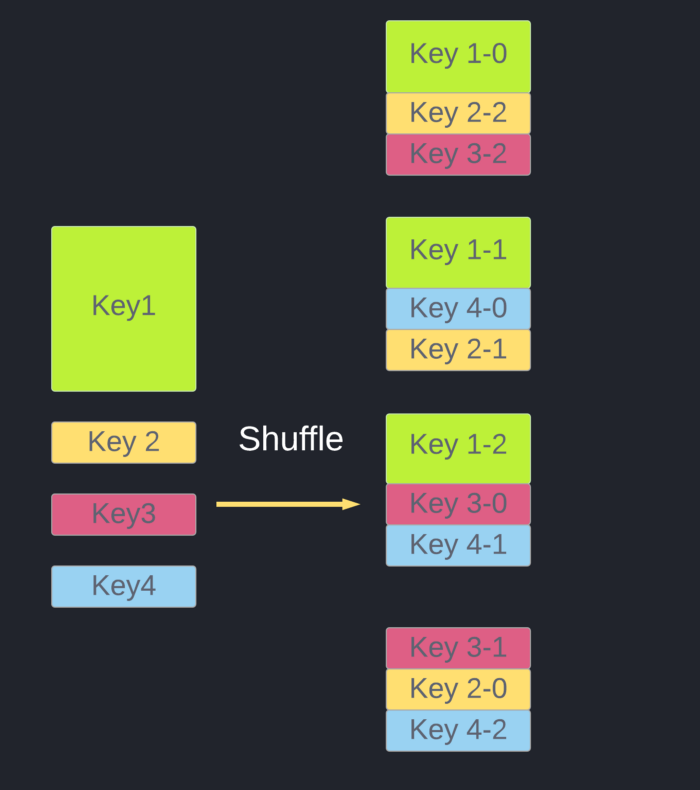
**Joining two or more large tables having skew data in Spark**

While using Spark for our pipelines, we were faced with a use-case where we were required to join a large (driving) table on multiple columns with another large table on a different joining column and condition. The Spark join column was highly skewed, and the other table was an evenly distributed data frame. Both of these data frames were fairly large (millions of records). The job was getting stuck at the last stage (say at 399/400 steps) and stayed that way for 3 to 4 hours post, which threw an error that read- **Caused by: org.apache.spark.shuffle.FetchFailedException: Too large frame: 7498008366**. On the Spark Web App UI, we saw this-



In Spark, SALT is a technique that adds random values to push Spark partition data evenly. It’s usually good to adopt for wide transformation requires shuffling like join operation.

The following image visualizes how SALT is going to change the key distribution. Key 1(light green) is the hot key that causes skewed data in a single partition. After applying SALT, the original key is split into 3 parts and driving the new keys to shuffle to different partitions than before. In this case, Key 1 goes to 3 different partitions, and the original partition can be processed in parallel among those 3 partitions.



# How to use SALT in Spark

The process of using SALT in Spark can be breakdown into:

1. Add a new field and populate it with random numbers.
2. Combine this new field and the existing keys as a composite key, perform any transformation.
3. Once the processing is done, combine the final result.

We can write a line of spark code like:

df.withColumn("salt\_random\_column", (rand \* n).cast(IntegerType)) // n is the size of partition you'd like to have

.groupBy(groupByFields, "salt\_random\_column")

.agg(aggFields)

.groupBy(groupByFields)

.agg(aggFields)

|  |
| --- |
|  |
|  |  |
|  |  |
|  |  |
|  |  |

Spark 3.0 Features:-

* [Adaptive Query execution also called AQE](https://sparkbyexamples.com/spark/spark-3-0-features-with-examples-part-i/#adq)
* [Language Version Upgrades](https://sparkbyexamples.com/spark/spark-3-0-features-with-examples-part-i/#language-version)
* [New UI for Structure Streaming](https://sparkbyexamples.com/spark/spark-3-0-features-with-examples-part-i/#new-ui)
* [Datasource to Read Binary Files](https://sparkbyexamples.com/spark/spark-3-0-features-with-examples-part-i/#read-binary-files)
* [Support Recursive folders](https://sparkbyexamples.com/spark/spark-3-0-features-with-examples-part-i/#recursive)
* [Support Multi char delimiter (||)](https://sparkbyexamples.com/spark/spark-3-0-features-with-examples-part-i/#multi-char)
* [New Spark build-in functions](https://sparkbyexamples.com/spark/spark-3-0-features-with-examples-part-i/#spark-functions)
* [Switch to Proleptic Gregorian calendar](https://sparkbyexamples.com/spark/spark-3-0-features-with-examples-part-i/#proleptic)
* [DataFrame Tail](https://sparkbyexamples.com/spark/spark-3-0-features-with-examples-part-i/#tail)
* [Added repartition to SQL queries](https://sparkbyexamples.com/spark/spark-3-0-features-with-examples-part-i/#partitionby)
* [Better ANSI SQL compatible](https://sparkbyexamples.com/spark/spark-3-0-features-with-examples-part-i/#ansi-sql)