1. **Spark Sql**
   1. **Left join**

Every column in the left table will be present in output irrespective of joining key matched or not

* 1. **Outer join**

Left outer join and left join are same. Full outer join will include all rows from both tables

* 1. **Inner join**

Only rows with matching joining values will be included

* 1. **There are two tables, if you want to find uncommon values from both the tables, what will you do?**

Select list from table A full outer join table B where A.value = B.value and A.value is null or B.value is Null

select id

from tablea

where a.id not in (select id from tableb)

union all

select id

from tableb

where b.id not in (select id from tablea);

* 1. **There are multiple records in a table, you want to fetch only a single record based on the date column, what is the approach?**

Select \* from table where date = ‘2020-01-01’ limit 1;

**denserank,rank**

**select company ,concat(name ,”-” ,organization) as name , rank() over (Partittioned by company order by power desc) as power rank from cars .**

* All of them require an order by clause.
* All of them return an increasing integer with a base value of 1.
* When combined with a PARTITION BY clause, all of these functions reset the returned integer value to 1 as we have seen.
* If there are no duplicated values in the column used by the ORDER BY clause, these functions return the same output.

* The RANK() function skips the next N-1 ranks if there is a tie between N previous ranks.
* DENSE\_RANK() function does not skip ranks if there is a tie between ranks
* ROW\_NUMBER() function has no concern with ranking. It simply returns the row number of the sorted records. Even if there are duplicate records in the column used in the ORDER BY clause, the ROW\_NUMBER function will not return duplicate values. Instead, it will continue to increment irrespective of the duplicate values.
  1. **What is indexing**

As the name suggest, indexing is just like the index of entire data. Our data is stored in hdfs which is very slow in terms of I/O operations, suppose there are 1000 records in our database and each block is holding 10 records, if we do a select \* from data with where condition it is going to take each block in memory, perform operation and based on if the required data is found or now it either returns output or goes to next block, with indexing we are pointing out a unique column on which most of our where condition is to be based upon. In this case there are two options, either it will store one column value of each record in index and load them in ram and figure out required data or it can have a column value of first record in each block which would further reduce latency time, the later one is only possible in case of sorted data. In case of non sorted data it is dense indexing and in terms of sorted data it is sparse indexing

* 1. **How do you load data incrementally in a table or edw?**

Based on delta column. It is done mostly based upon the value of datetime column that is used commonly named as recon created date or recon updated date, we can filter records based upon that column using greater then where clause on data.

Create another table with batch no and loaded until date column along with status, everytime we can select max(loaded until date)column from this table to fetch most recent records along with records that have status as failure

1. **What all parameters are passed in spark submit command**

--verbose. Yarn-master

--deploy-mode. Cluster/local/client

--jars. Jars to be deployed

--driver-class-path

--class. Main class

--files. Files to pass

--conf

>abc.log. Log file to write logs

--master

--application-jars

--application-arguments

--num-executors

--driver-memory

--no-of-cores

--executor-memory

* 1. **How do u calculate driver memory, nodes in a cluster**

10 nodes, 16 cores, 64 gb

Total cores - 160

Recommended partitions can be total no of cores, total no of cores\*2, total no of cores\*3

No of task to be running in parallel lets say 5

No of executors per node will be (16-1)/5 = 3 . 1gb for memory overhead

Memory with each executor will be 21gb

Memory for yarn overhead will be 10% of memory with executor = ~3gb

Memory for each executor will be 18gb

Driver memory = executor memory

Driver cores = executor cores

16 nodes, 16 cores, 64 gb

Total cores - 256

Recommended partitions can be total no of cores, total no of cores\*2, total no of cores\*3

No of task to be running in parallel lets say 5

No of executors per node will be (16-1)/5 = 3 . 1gb for memory overhead

Memory with each executor will be 21gb

Memory for yarn overhead will be 10% of memory with executor = ~3gb

Memory for each executor will be 18gb

Driver memory = executor memory

Driver cores = executor cores

https://blogs.perficient.com/2020/08/25/key-components-calculations-for-spark-memory-management/

1. **Spark optimization techniques**

Groupbykey, reducebykey,join,repartition,collease all this cause shuffle

**Avoid data skewness**

Joins optimization

Sort Merge when both are large datasets

If one is large and other is small them broadcast join

Count,distinct count and repartitions are heavy operations

Use approxcountdistinct

Drop duplicate before join

Persist and cache is basically same, caching is part of persist

Garbage collection might increase with too much of caching

Broadcast variables, read only, pass it across nodes for faster access

Accumulators are often used for counters, for eg: no of failed records, how many times api was called etc. these are shared variables which are editable.

Instead of foreach use seq.par.foreach

Try avoiding spark udf’s

Visit spark ui to see dag and other tabs to understand things better

Reducing data scans by using partitions, filter, bucketing

Hardware, type of disk, memory also plays a vital role

Reutilize transformed data using caching

**persist and cache**

Cache by default happens in memory where else in persist we can define the storage level

In persist storage level bydefault is memory and disk

df.persist(StorageLevel.MEMORY\_ONLY)

Storage Level

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Storage Level | Space used | CPU time | In memory | on-disk | Serialized | Recompute some partitions |
| Memory Only | HIGH | LOW | Y | N | N | Y |
| Memory\_only\_SER | LOW | HIGH | Y | N | Y | Y |
| MEMORY\_AND\_DISK | HIGH | MEDIUM | SOME | SOME | SOME | N |
| MEMORY\_AND\_DISK\_SER | LOW | HIGH | SOME | SOME | Y | N |
| DISK\_ONLY | LOW | HIGH | N | Y | Y | N |

**coalesce and repartitioning**

Both are expensive, repartition shuffles all partitions where else coalesce only shuffles reduced partitions. Using repartitions we can increase or decrease partitions bt in coalesce we only reduce the partitions

df.repartition(4)

df.coalesce(4)

Default no of repartitions is 200 in shuffle and joins

**Rdd**

Resilient distributed dataset

Rdd is an immutable distributed collection of elements of data, partitioned across nodes

When to use rdd:

Data is unstructured

Need to use low level transformation and actions and have control on data

Can compromise on performance

Imposing a schema is not required

Data frames and datasets are built on top of rdd and can move from one to another by calling api

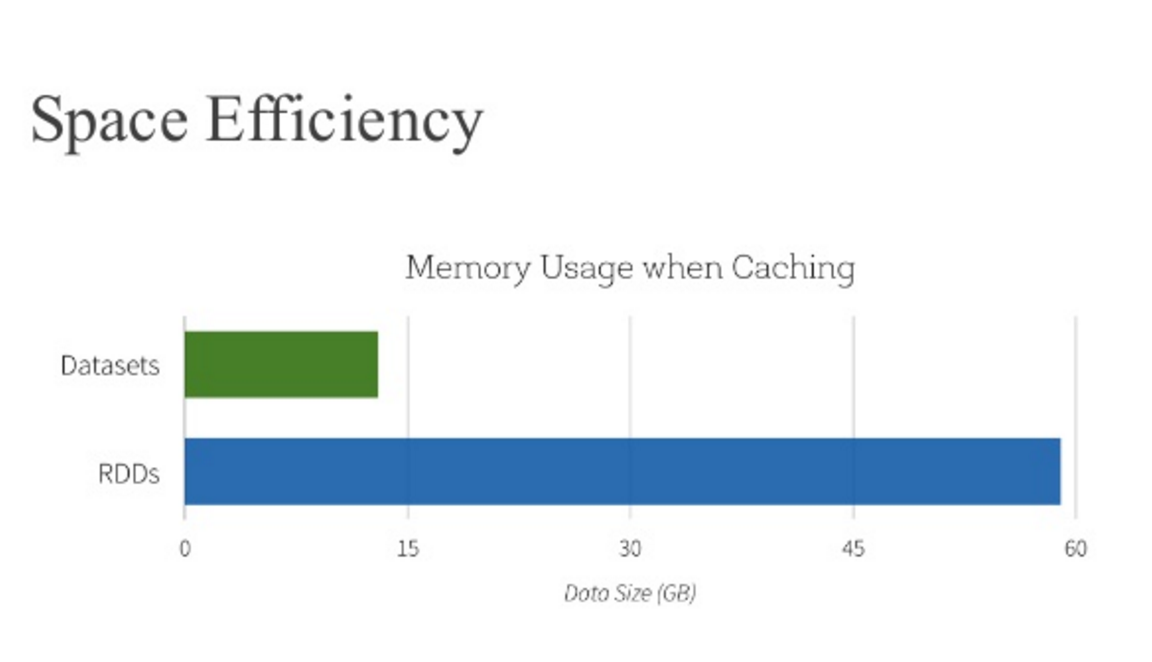
**df and datasets**

Df are immutable distributed collection of data, unlike rdd it is organised into named columns just like a table in rdbms, imposing structure makes it easier to handle large datasets

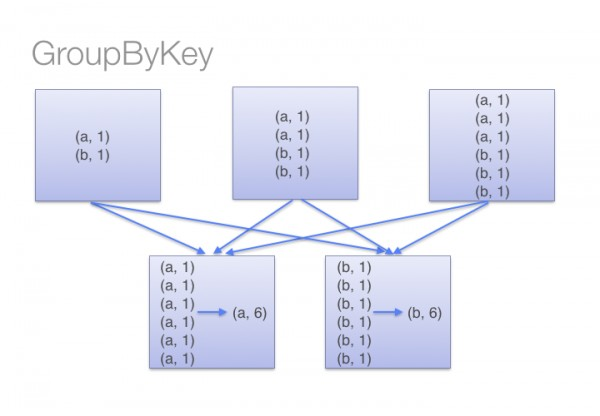
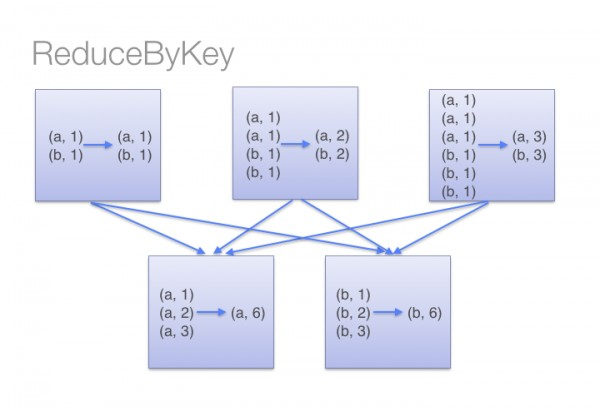
To be used when data is structured or semi structured

When there is a large set of data and need to use filter aggregation and stuff

Type safety at compile time



**groupby and reduceby**



Groupby key sends all data over the network and may cause out of space issues, reducebykey does that on each partition and sends data output of each partition over network to sort

**reduceby and reduce**

Basically, reduce must pull the entire dataset down into a single location because it is reducing to **one** final value. reduceByKey on the other hand is one value for **each** key. And since this action can be run on each machine locally first then it can remain an RDD and have further transformations done on its dataset.

**partitioning and bucketing**

Partitioning is required to process data parallelly and reduce latency time for queries. One can store data in folders based on date and time to segregate the records

Bucketing can be done with partitioning or without partitioning. If there are no values to build partitions on then bucketing can be used. It helps in even distribution of data. It does this based on hashing of key

No of buckets and column needs to be passed explicitly

Bucketing is helped in sampling and map side joins are performed better on bucketed tables. Partitions with bucketing often provide best results

No of buckets can be equal to no of executors for best results

In bucketed tables shuffle happens only once at start and thereafter joins, reduceby dont shuffle the data again

Transformations will be quicker on bucketed table and queries on bucketed column will be faster

**Limitations of bucketing in spark sql**

We don’t require bucketing if we do not have the same column in joins/transformations and buckets.:

Spark SQL bucketing requires sorting on reading time which greatly degrades the performance.

When Spark writes data to a bucketing table, it can generate tens of millions of small files that are not supported by HDFS.

Bucket joins are triggered only when the two tables have the same number of buckets.

It needs the bucket key set to be similar to the join key set or grouping key set.

The bucketing technique in Spark SQL is different from Hive which gives way to an expensive migration process from Hive to Spark SQL.

1. **What do u mean by broadcast variables and accumulators**

Boardcast variables are spread across nodes that can be used as lookup. These are non editable variables just like final in java and makes processing faster

Accumulators are like counters which are editable and are shared across cluster for faster access

1. If you want to move a data from one cluster to another, what command will you use

hadoop distcp hdfs://nn1:8020/foo/bar \hdfs://nn2:8020/bar/foo

1. Default no of repartitions

200, if we use groupby/shuffle it will change no of partitions to default 200 if not specified by no

**park.sql.shuffle.partitions** stores this no

1. How to calculate partitions

No of partitions can ideally be set to no of cores or 2X to no of cores

1. When spark job is running where does it hold the data

If we use persist and mention the storage type it holds data in memory or disk as specified

1. Types of persist

5 types

Disk only

Memory only

Disk and memory

Memory only ser

Memory and disk ser

1. Types of files and when to use which type

The plain text format or CSV would only be recommended in case of extractions of data from Hadoop or a massive data load from a file.

The SequenceFile format is recommended in case of storing intermediate data in MapReduce jobs.

Avro is a good choice in case the data scheme can evolve over time.

Parquet and ORC are recommended when query performance is important.

1. Schema rdd

SchemaRDDs are composed [Row](https://spark.apache.org/docs/1.0.2/api/scala/index.html#org.apache.spark.sql.catalyst.expressions.Row) objects along with a schema that describes the data types of each column in the row. A SchemaRDD is similar to a table in a traditional relational database. A SchemaRDD can be created from an existing RDD, [Parquet](http://parquet.io/) file, a JSON dataset, or by running HiveQL against data stored in [Apache Hive](http://hive.apache.org/).

1. Dstream

Continuous stream of data, not used

1. Dag interpretation

Helps to understand the flow and improve performance, no of executors, garbage collection, no of stages and task and time taken can be identified, we can see the skipped stages and the sequence of flow

1. No of task and partition

No of partitions is equal to no of blocks/input files. No of task is ideally equal to no of partitions

1. How to create rdd,df,ds

Val rdd = spark.sparkContext.parallelize(data)

Val df = rdd.toDF()

To mention schema we can provide column names, by default it will be \_1,\_2,...

Val df = rdd.toDF(“language”,”user”)

1. Spark.range

spark.range(1,7,2,3)

1,7 are start and end points, 2 is the step, 3 is no of partitions

Output will be 1,3,5 and distributed in 3 partitions

1. Why repartitioning is required

If the data is filtered out we can use repartitioning to distribute it across the cluster

<https://towardsdatascience.com/should-i-repartition-836f7842298c>

To avoid reduced joins in some cases

1. Where repartitioning is required
2. How to decide no. Of repartition

Should be equal to no of cores

1. Default no of repartition

200

1. How to use spark standalone without hdfs

https://www.edureka.co/blog/interview-questions/top-apache-spark-interview-questions-2016/

Word count program

Val df = sc.textFile(“/pathOfFile”).flatMap(line=>line.split(“ “)).map(word => (word,1)).reduceByKey(\_+\_).saveAsTextFile(“hdfs://”)

(“file://”) if file is on local

Map flatmap and reduceby

Map takes one rdd and returns one rdd

Flatmap takes one rdd and returns 0,1 or more rdds

Val rdd = data.map(line=>line.toUpperCase())

Val rdd = data.flatMap(line=>line.split(“ “))

What is sc

sparkContext, used to initialize sparkcontext on console

Collect

Collect or collectAsList is a heavy operation and it results in retrieving all the elements of rdd/df/ds from all nodes to the driver node. It must only be performed on a smaller data set else it may result in out of memory

Results data as an array of list instead of df

Driver program

Core program, supervise

Responsible for launching parallel operations on cluster

Holds main program

Driver program access spark using spark context object

Driver program is responsible for creating a task from dag

It is a process which is running the user code, creating rdds and carrying out transformations

Converts user program into unit of physical execution called task

It defines distributed datasets on the cluster and we apply transformations and actions on it

Metadata in standalone

Uses hive metadata itself.

Yarn

1. Client submits an application
2. The Resource Manager allocates a container to start the Application Manager
3. The Application Manager registers itself with the Resource Manager
4. The Application Manager negotiates containers from the Resource Manager
5. The Application Manager notifies the Node Manager to launch containers
6. Application code is executed in the container
7. Client contacts Resource Manager/Application Manager to monitor application’s status
8. Once the processing is complete, the Application Manager un-registers with the Resource Manager

Driver program interacts with cluster manager to fetch resources. Cluster manager allocates executors on nodes

Data locality

Spark stores data in persist for intermediate output

Ways to create rdd:

Parallize

Spark.read.textfile

Rdd from different rdd ( using transformations on rdd)

Collect works only on rdd

Collect and take are heavy operations since they bring entire data on driver memory

Spark parameters calculations

Num executors

Num executors memory

Driver memory

No of cores

No of blocks = no of task

No of input task = no of output task in spark

In mapreduce no of output task is 1 by default

Custom partition to decide which data goes where - getpartition method override, extends practitioner

Spark-shell --master local[2]

rdd.getNumPartitions

In local default No of partitions = no of cores allotted to application

In cluster mode in hdfs no of partitions = no of blocks/no of input files

minimum no of partitions = 2 for parallel processing

No of executors = no of partitions = no of task

Wide transformations are one where shuffling takes place

After shuffling new stage is created

Executor is program that process partitions

Task is a partition under execution

Driver monitors the state of executor