Computer Hardware and Software (COCSC19)

Apache Spark

Introduction

Apache Spark, written in **Scala**, is a general-purpose distributed data processing engine. Or in other words: load big data, do computations on it in a distributed way, and then store it.

Apache Spark contains libraries for data analysis, machine learning, graph analysis, and streaming live data. Spark is generally faster than *Hadoop*.

This is because Hadoop writes intermediate results to disk (that is, lots of I/O operations) whereas Spark tries to keep intermediate results in memory(that is, in-memory computation) whenever possible. Moreover, Spark offers lazy evaluation of operations and optimizes them just before the final result.

Sparks maintains a series of transformations that are to be performed without actually performing those operations unless we try to obtain the results.

This way, Spark is able to find the best path looking at overall transformations required (for example, reducing two separate steps of adding number 5 and 20 to each element of the dataset into just a single step of adding 25 to each element of the dataset, or not actually doing operations on part of the dataset which will eventually be filtered out in the final result).

This makes Spark one of the most popular tools for big data analytics currently.

Our Datasets

We have used PySpark for implementing Apache spark in our project. PySpark is **an interface for Apache Spark in Python**. It not only allows you to write Spark applications using Python APIs, but also provides the PySpark shell for interactively analyzing your data in a distributed environment. We have taken 2 datasets, taken from Kaggle, for implementing Apache spark, creating 2 separate Jupyter notebook files.

First dataset used in our project performs various functionalities outlined in Apache Spark on different brands of cereal and their nutritional values.

name	mfr	type	calories	protein	fat	sodium	fiber	carbo	sugars	potass	vitamins	shelf	weight	cups	rating
100% Bran	N	C	70	4	1	130	10	5	6	280	25	3	1	0.33	68.402973
100% Natural Bran	Q	C	120	3	5	15	2	8	8	135	0	3	1	1	33.983679
All-Bran	K	C	70	4	1	260	9	7	5	320	25	3	1	0.33	59.425505
All-Bran with Extra Fiber	K	C	50	4	0	140	14	8	0	330	25	3	1	0.5	93.704912
Almond Delight	R	C	110	2	2	200	1	14	8	-1	25	3	1	0.75	34.384843
Apple Cinnamon Cheerios	G	C	110	2	2	180	1.5	10.5	10	70	25	1	1	0.75	29.509541
Apple Jacks	K	C	110	2	0	125	1	11	14	30	25	2	1	1	33.174094
Basic 4	G	C	130	3	2	210	2	18	8	100	25	3	1.33	0.75	37.038562
Bran Chex	R	C	90	2	1	200	4	15	6	125	25	1	1	0.67	49.120253
Bran Flakes	P	C	90	3	0	210	5	13	5	190	25	3	1	0.67	53.313813
Cap'n'Crunch	Q	C	120	1	2	220	0	12	12	35	25	2	1	0.75	18.042851
Cheerios	G	C	110	6	2	290	2	17	1	105	25	1	1	1.25	50.764999
Cinnamon Toast Crunch	G	C	120	1	3	210	0	13	9	45	25	2	1	0.75	19.823573
Clusters	G	C	110	3	2	140	2	13	7	105	25	3	1	0.5	40.400208

Second dataset utilizes a description of E-Commerce customers to carry out analysis on the basis of various factors such as Time on App, Average Session Length etc.

Email	Address	Avg Session Length	Time on App	Time on Website	Length of Membership	Yearly Amount Spent
mstephenson@fernandez.com	835 Frank TunnelWrightmouth, MI 82180-9605	34.49726773	12.65565115	39.57766802	4.082620633	587.951054
hduke@hotmail.com	4547 Archer CommonDiazchester, CA 06566-8576	31.92627203	11.10946073	37.26895887	2.664034182	392.2049334
pallen@yahoo.com	24645 Valerie Unions Suite 582Cobbborough, DC 99414-7564	33.00091476	11.33027806	37.11059744	4.104543202	487.5475049
riverarebecca@gmail.com	1414 David ThroughwayPort Jason, OH 22070-1220	34.30555663	13.71751367	36.72128268	3.120178783	581.852344
mstephens@davidson-herman.com	14023 Rodriguez PassagePort Jacobville, PR 37242-1057	33.33067252	12.79518855	37.5366533	4.446308318	599.406092
alvareznancy@lucas.biz	645 Martha Park Apt. 611Jeffreychester, MN 67218-7250	33.87103788	12.02692534	34.47687763	5.493507201	637.1024479
katherine20@yahoo.com	68388 Reyes Lights Suite 692Josephbury, WV 92213-0247	32.0215955	11.36634831	36.68377615	4.685017247	521.5721748
awatkins@yahoo.com	Unit 6538 Box 8980DPO AP 09026-4941	32.73914294	12.35195897	37.37335886	4.434273435	549.9041461
vchurch@walter-martinez.com	860 Lee KeyWest Debra, SD 97450-0495	33.9877729	13.38623528	37.53449734	3.273433578	570.200409
bonnie69@lin.biz	PSC 2734, Box 5255APO AA 98456-7482	31.93654862	11.81412829	37.14516822	3.202806072	427.1993849
andrew06@peterson.com	26104 Alexander GrovesAlexandriaport, WY 28244-9149	33.99257277	13.33897545	37.22580613	2.482607771	492.6060127
ryanwerner@freeman.biz	Unit 2413 Box 0347DPO AA 07580-2652	33.87936082	11.584783	37.08792607	3.713209203	522.3374046
knelson@gmail.com	6705 Miller Orchard Suite 186Lake Shanestad, MO 75696-5051	29.53242897	10.9612984	37.42021558	4.046423164	408.6403511
wrightpeter@yahoo.com	05302 Dunlap FerryNew Stephaniehaven, MP 42268	33.19033404	12.95922609	36.1446667	3.918541839	573.4158673
taylormason@gmail.com	7773 Powell Springs Suite 190Samanthaland, ND 44358	32.38797585	13.14872569	36.61995708	2.494543647	470.4527333
jstark@anderson.com	49558 Ramirez Road Suite 399Phillipstad, OH 35641-3238	30.73772037	12.63660605	36.21376309	3.357846842	461.7807422
wjennings@gmail.com	6362 Wilson Mountain Johnsonfurt, GA 15169	32.1253869	11.73386169	34.89409275	3.136132716	457.8476959

Installing PySpark and FindSpark using pip

```
!pip install pyspark
!pip install findspark
```

```
Collecting pyspark
  Downloading pyspark-3.2.1.tar.gz (281.4 MB)
     |████████████████████████████████| 281.4 MB 28 kB/s
Collecting py4j==0.10.9.3
  Downloading py4j-0.10.9.3-py2.py3-none-any.whl (198 kB)
     |████████████████████████████████| 198 kB 39.9 MB/s
Building wheels for collected packages: pyspark
  Building wheel for pyspark (setup.py) ... done
  Created wheel for pyspark: filename=pyspark-3.2.1-py2.py3-none-any.whl size=281853642 sh
  Stored in directory: /root/.cache/pip/wheels/9f/f5/07/7cd8017084dce4e93e84e92efd1e1d5334
Successfully built pyspark
Installing collected packages: py4j, pyspark
Successfully installed py4j-0.10.9.3 pyspark-3.2.1
Collecting findspark
  Downloading findspark-2.0.1-py2.py3-none-any.whl (4.4 kB)
Installing collected packages: findspark
Successfully installed findspark-2.0.1
```

Analysis

```
df.show(10)
```

name	mfr	type	calories	protein	fat	sodium	fiber	carbo	sugars	potass	vitamins	shelf	weight	cups	rating
100% Bran	N	C	70	4	1	130	10	5	6	280	25	3	1	0.33	68.402973
100% Natural Bran	Q	C	120	3	5	15	2	8	8	135	0	3	1	1	33.983679
All-Bran	K	C	70	4	1	260	9	7	5	320	25	3	1	0.33	59.425505
All-Bran with Ext...	K	C	50	4	0	140	14	8	0	330	25	3	1	0.5	93.704912
Almond Delight	R	C	110	2	2	200	1	14	8	-1	25	3	1	0.75	34.384843
Apple Cinnamon Ch...	G	C	110	2	2	180	1.5	10.5	10	70	25	1	1	0.75	29.509541
Apple Jacks	K	C	110	2	0	125	1	11	14	30	25	2	1	1	33.174094
Basic 4	G	C	130	3	2	210	2	18	8	100	25	3	1.33	0.75	37.038562
Bran Chex	R	C	90	2	1	200	4	15	6	125	25	1	1	0.67	49.120253
Bran Flakes	P	C	90	3	0	210	5	13	5	190	25	3	1	0.67	53.313813

```
only showing top 10 rows
```

df.show(argument) is used to show the first n number of lines based on the argument passed in the order they are found in the original data file.

```
df.printSchema()

root
 |-- name: string (nullable = true)
 |-- mfr: string (nullable = true)
 |-- type: string (nullable = true)
 |-- calories: string (nullable = true)
 |-- protein: string (nullable = true)
 |-- fat: string (nullable = true)
 |-- sodium: string (nullable = true)
 |-- fiber: string (nullable = true)
 |-- carbo: string (nullable = true)
 |-- sugars: string (nullable = true)
 |-- potass: string (nullable = true)
 |-- vitamins: string (nullable = true)
 |-- shelf: string (nullable = true)
 |-- weight: string (nullable = true)
 |-- cups: string (nullable = true)
 |-- rating: string (nullable = true)
```

df.printSchema() works in the same way **DESCRIBE** works in MySQL and prints the description of data types used in the database.

```
df.select('name', 'mfr', 'rating').show(10)

+--------------------+---+---------+
|                name|mfr|   rating|
+--------------------+---+---------+
|           100% Bran|  N|68.402973|
|   100% Natural Bran|  Q|33.983679|
|            All-Bran|  K|59.425505|
|All-Bran with Ext...|  K|93.704912|
|      Almond Delight|  R|34.384843|
|Apple Cinnamon Ch...|  G|29.509541|
|         Apple Jacks|  K|33.174094|
|             Basic 4|  G|37.038562|
|           Bran Chex|  R|49.120253|
|         Bran Flakes|  P|53.313813|
+--------------------+---+---------+
only showing top 10 rows
```

df.select() works in the same way as **SELECT FROM** clause works in MySQL.

```
df.select("name", when(df.vitamins >= "25", "rich in vitamins")).show()
```

```
+--------------------+-------------------------------------------------+
|                name|CASE WHEN (vitamins >= 25) THEN rich in vitamins END|
+--------------------+-------------------------------------------------+
|           100% Bran|                                 rich in vitamins|
|   100% Natural Bran|                                             null|
|            All-Bran|                                 rich in vitamins|
|All-Bran with Ext...|                                 rich in vitamins|
|      Almond Delight|                                 rich in vitamins|
|Apple Cinnamon Ch...|                                 rich in vitamins|
|         Apple Jacks|                                 rich in vitamins|
|             Basic 4|                                 rich in vitamins|
|           Bran Chex|                                 rich in vitamins|
|         Bran Flakes|                                 rich in vitamins|
|        Cap'n'Crunch|                                 rich in vitamins|
|            Cheerios|                                 rich in vitamins|
|Cinnamon Toast Cr...|                                 rich in vitamins|
|            Clusters|                                 rich in vitamins|
|         Cocoa Puffs|                                 rich in vitamins|
|           Corn Chex|                                 rich in vitamins|
|         Corn Flakes|                                 rich in vitamins|
|           Corn Pops|                                 rich in vitamins|
|       Count Chocula|                                 rich in vitamins|
|  Cracklin' Oat Bran|                                 rich in vitamins|
+--------------------+-------------------------------------------------+
only showing top 20 rows
```

when conditional argument can be passed through **df.select()** that applies a filter to the results (Ex: Name of brands that have more than or equal to 25 vitamins are shown as rich in vitamins).

```
df.filter(df.calories == "100").show()
```

name	mfr	type	calories	protein	fat	sodium	fiber	carbo	sugars	potass	vitamins	shelf	weight	cups	rating
Corn Flakes	K	C	100	2	0	290	1	21	2	35	25	1	1	1	45.863324
Cream of Wheat (Q...	N	H	100	3	0	80	1	21	0	-1	0	2	1	1	64.533816
Crispy Wheat & Ra...	G	C	100	2	1	140	2	11	10	120	25	3	1	0.75	36.176196
Double Chex	R	C	100	2	0	190	1	18	5	80	25	3	1	0.75	44.330856
Frosted Mini-Wheats	K	C	100	3	0	0	3	14	7	100	25	2	1	0.8	58.345141
Golden Crisp	P	C	100	2	0	45	0	11	15	40	25	1	1	0.88	35.252444
Grape Nuts Flakes	P	C	100	3	1	140	3	15	5	85	25	3	1	0.88	52.076897
Life	Q	C	100	4	2	150	2	12	6	95	25	2	1	0.67	45.328074
Maypo	A	H	100	4	1	0	0	16	3	95	25	2	1	1	54.850917
Multi-Grain Cheerios	G	C	100	2	1	220	2	15	6	90	25	1	1	1	40.105965
Product 19	K	C	100	3	0	320	1	20	3	45	100	3	1	1	41.503540
Quaker Oat Squares	Q	C	100	4	1	135	2	14	6	110	25	3	1	0.5	49.511874
Quaker Oatmeal	Q	H	100	5	2	0	2.7	-1	-1	110	0	1	1	0.67	50.828392
Raisin Nut Bran	G	C	100	3	2	140	2.5	10.5	8	140	25	3	1	0.5	39.703400
Total Whole Grain	G	C	100	3	1	200	3	16	3	110	100	3	1	1	46.658844
Wheat Chex	R	C	100	3	1	230	3	17	3	115	25	1	1	0.67	49.787445
Wheaties	G	C	100	3	1	200	3	17	3	110	25	1	1	1	51.592193

In case of using **when**, only the columns mentioned for a particular conditions are shown whereas in **filter**, the whole tuple is shown for a condition passed as an argument.

```
dataset.orderBy('Yearly Amount Spent', ascending=False).limit(5).toPandas()
```

	Email	Address	Avg Session Length	Time on App	Time on Website	Length of Membership	Yearly Amount Spent
0	kyang@diaz.org	223 Love Trail Suite 831Port Jeffrey, IN 46849	34.374258	15.126994	37.157624	5.377594	765.518462
1	asilva@yahoo.com	USNV JohnsonFPO AP 19026	34.603311	12.207298	33.913847	6.922689	744.221867
2	william82@gmail.com	11143 Park SquaresSamanthatown, UT 97073	33.256335	13.858062	37.780265	5.976768	725.584814
3	jeffrey54@mcdonald-williams.com	297 Francis ValleySouth Lindsey, NY 13669-5367	34.967610	13.919494	37.952013	5.066697	712.396327
4	rhonda01@gmail.com	939 Watson RunStaceyberg, VT 58376-0454	34.385820	12.729720	36.232110	5.705941	708.935185

By applying **orderBy()**, we can order the whole dataset in ascending or descending order on the basis of values provided in that column.

NOTE: It is not only used to present numerical data in ascending or descending order, but character and string data as well based on the ASCII values and Lexicographic order respectively.

```
dataset.summary().toPandas()
```

	summary	Email	Address	Avg Session Length	Time on App	Time on Website	Length of Membership	Yearly Amount Spent
0	count	500	500	500	500	500	500	500
1	mean	None	None	33.05319351824	12.052487936928012	37.060445421080004	3.5334615559300007	499.3140382608002
2	stddev	None	None	0.9925631111602911	0.9942156084624618	1.0104889068105993	0.9992775024367542	79.31478155115914
3	min	aaron04@yahoo.com	0001 Mack MillNorth Jennifer, NE 42021-5936	29.53242897	8.508152176	33.91384725	0.26990109	256.6705823
4	25%	None	None	32.33889932	11.38677555	36.33952101	2.926940235	444.9665517
5	50%	None	None	33.07871721	11.98204499	37.06708997	3.53286158	498.6355985
6	75%	None	None	33.71065306	12.75207661	37.71598618	4.125584363	549.1315733
7	max	zscott@wright.com	Unit 7502 Box 8345DPO AE 53747	36.13966249	15.12699429	40.00518164	6.922689335	765.5184619

dataset.summary() provides us with different stochastic and statistical measures as can be seen in the image above.

PySpark is also capable of providing a very big dataset with summary values such as count, mean, standard deviation, quartiles which are essential in carrying out data analysis tasks. Due to such functions, Pyspark proves to be very essential for Online Analysis Processing(OLAP) which is also known as Data Warehouse mining that is ultimately used in Data mining. Thus, Apache Spark is a very powerful tool in analysis and research related work for a particular data.

```
from pyspark.ml.regression import LinearRegression

from pyspark.ml.linalg import Vectors
from pyspark.ml.feature import VectorAssembler

featureassembler=VectorAssembler(inputCols=["Avg Session Length","Time on App","Time on Website","Length of Membership"],outputCol="Independer

output=featureassembler.transform(dataset)

output.toPandas()
```

	Email	Address	Avg Session Length	Time on App	Time on Website	Length of Membership	Yearly Amount Spent	Independent Features
0	mstephenson@fernandez.com	835 Frank TunnelWrightmouth, MI 82180-9605	34.497268	12.655651	39.577668	4.082621	587.951054	[34.49726773, 12.65565115, 39.57766802, 4.0826...
1	hduke@hotmail.com	4547 Archer CommonDiazchester, CA 06566-8576	31.926272	11.109461	37.268959	2.664034	392.204933	[31.92627203, 11.10946073, 37.26895887, 2.6640...
2	pallen@yahoo.com	24645 Valerie Unions Suite 582Cobbborough, DC ...	33.000915	11.330278	37.110597	4.104543	487.547505	[33.00091476, 11.33027806, 37.11059744, 4.1045...
3	riverarebecca@gmail.com	1414 David ThroughwayPort Jason, OH 22070-1220	34.305557	13.717514	36.721283	3.120179	581.852344	[34.30555663, 13.71751367, 36.72128268, 3.1201...
4	mstephens@davidson-herman.com	14023 Rodriguez PassagePort Jacobville, PR 372...	33.330673	12.795189	37.536653	4.446308	599.406092	[33.33067252, 12.79518855, 37.5366533, 4.44630...

VectorAssembler is **a transformer that combines a given list of columns into a single vector column**. It is useful for combining raw features and features generated by different feature transformers into a single feature vector, in order to train ML models like logistic regression and decision trees.

VectorAssembler, here, is capable of extracting, from raw data, Independent Variables for analysis work.

Resilient Distributed Datasets

Resilient Distributed Datasets (RDD) is a fundamental data structure of Spark. It is an immutable distributed collection of objects. Each dataset in RDD is divided into logical partitions, which may be computed on different nodes of the cluster. RDDs can contain any type of Python, Java, or Scala objects, including user-defined classes.

Formally, an RDD is a read-only, partitioned collection of records. RDDs can be created through deterministic operations on either data on stable storage or other RDDs. RDD is a fault-tolerant collection of elements that can be operated on in parallel.

There are two ways to create RDDs – parallelizing an existing collection in your driver program, or referencing a dataset in an external storage system, such as a shared file system, HDFS, HBase, or any data source offering a Hadoop Input Format.

Spark makes use of the concept of RDD to achieve faster and efficient MapReduce operations. Let us first discuss how MapReduce operations take place and why they are not so efficient.

Data Sharing is Slow in MapReduce

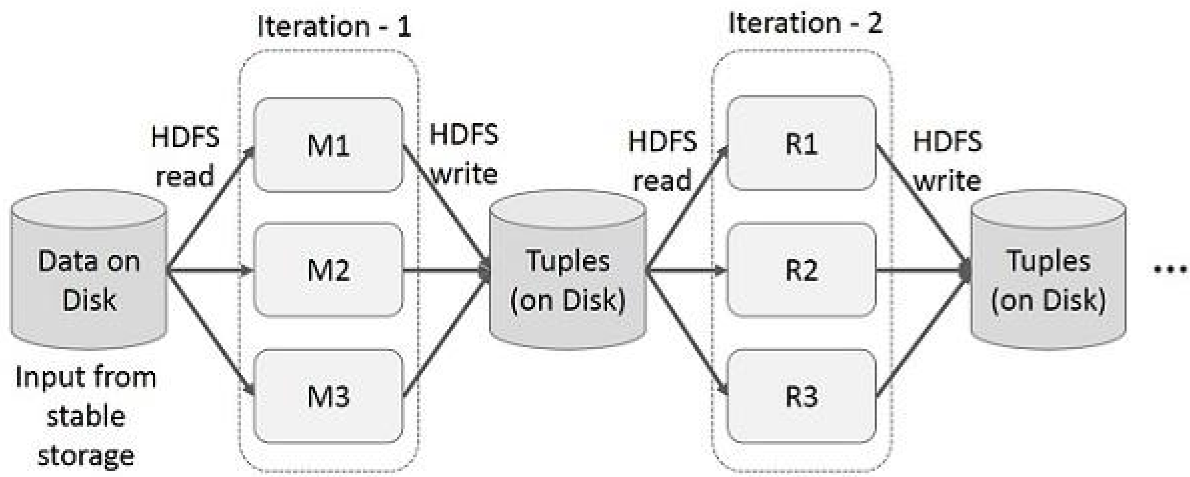
MapReduce is widely adopted for processing and generating large datasets with a parallel, distributed algorithm on a cluster. It allows users to write parallel computations, using a set of high-level operators, without having to worry about work distribution and fault tolerance.

Unfortunately, in most current frameworks, the only way to reuse data between computations (Ex – between two MapReduce jobs) is to write it to an external stable storage system (Ex – HDFS). Although this framework provides numerous abstractions for accessing a cluster's computational resources, users still want more.

Both Iterative and Interactive applications require faster data sharing across parallel jobs. Data sharing is slow in MapReduce due to replication, serialization, and disk IO. Regarding storage system, most of the Hadoop applications, they spend more than 90% of the time doing HDFS read-write operations.

Iterative Operations on MapReduce

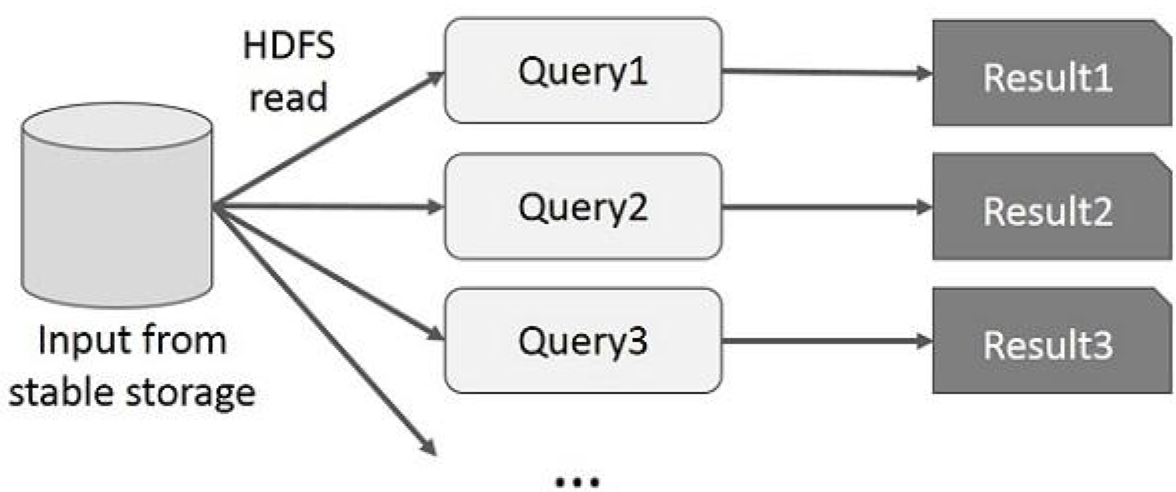
Reuse intermediate results across multiple computations in multi-stage applications. The following illustration explains how the current framework works, while doing the iterative operations on MapReduce. This incurs substantial overheads due to data replication, disk I/O, and serialization, which makes the system slow.



Interactive Operations on MapReduce

User runs ad-hoc queries on the same subset of data. Each query will do the disk I/O on the stable storage, which can dominate application execution time.

The following illustration explains how the current framework works while doing the interactive queries on MapReduce.



Data Sharing using Spark RDD

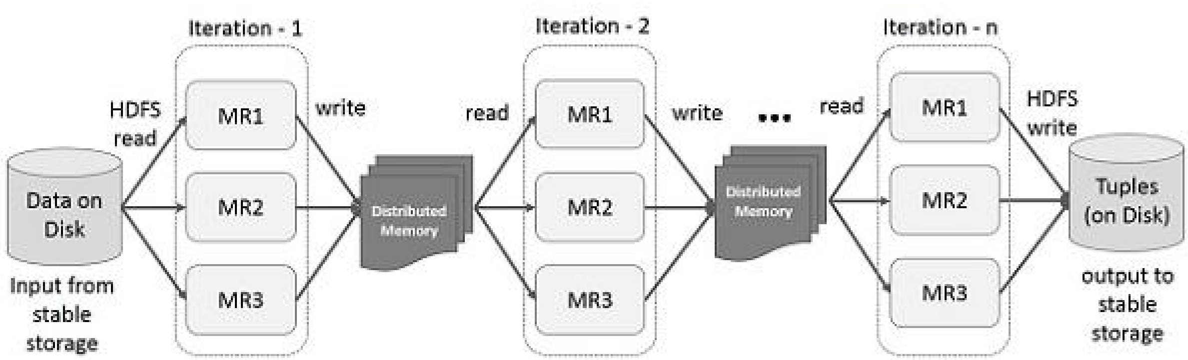
Data sharing is slow in MapReduce due to replication, serialization, and disk IO. Most of the Hadoop applications, they spend more than 90% of the time doing HDFS read-write operations.

Recognizing this problem, researchers developed a specialized framework called Apache Spark. The key idea of spark is Resilient Distributed Datasets (RDD); it supports in-memory processing computation. This means, it stores the state of memory as an object across the jobs and the object is sharable between those jobs. Data sharing in memory is 10 to 100 times faster than network and Disk.

Iterative Operations on Spark RDD

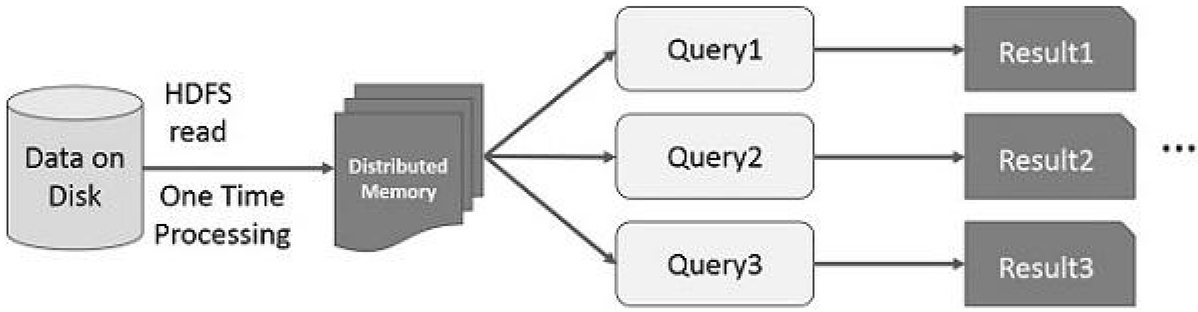
The illustration given below shows the iterative operations on Spark RDD. It will store intermediate results in a distributed memory instead of Stable storage (Disk) and make the system faster.

Note – If the Distributed memory (RAM) is not sufficient to store intermediate results (State of the JOB), then it will store those results on the disk.



Interactive Operations on Spark RDD

This illustration shows interactive operations on Spark RDD. If different queries are run on the same set of data repeatedly, this particular data can be kept in memory for better execution times.



By default, each transformed RDD may be recomputed each time you run an action on it. However, you may also persist an RDD in memory, in which case Spark will keep the elements around on the cluster for much faster access, the next time you query it. There is also support for persisting RDDs on disk, or replicated across multiple nodes.