MITS6005

Big Data

Session 9

Hadoop & Spark

Apache Hadoop: Purpose

- “Framework that allows distributed processing of large data sets across clusters of computers…
- using simple programming models.
- It is designed to scale up from single servers to thousands of machines, each offering local computation and storage.
- Rather than rely on hardware to deliver high-availability, the library itself is designed to detect and handle failures at the application layer, so delivering a highly-available service on top of a cluster of computers, each of which may be prone to failures.”

Source: https://hadoop.apache.org/

Apache Hadoop – key components

- Hadoop Common: Common utilities
- (Storage Component) Hadoop Distributed File System (HDFS): A distributed file system that provides high-throughput access
  - Many other data storage approaches also in use
  - E.G., Apache Cassandra, Apache Hbase, Apache Accumulo (NSA-contributed)
- (Scheduling) Hadoop **YARN**: A framework for job scheduling and cluster resource management
- (Processing) Hadoop **MapReduce** (MR2): A YARN-based system for parallel processing of large data sets
  - Other execution engines increasingly in use, e.g., Spark
- Note:
  - All of these key components are OSS under Apache 2.0 license
  - Related: Ambari, Avro, Cassandra, Chukwa, Hbase, Hive, Mahout, Pig, Tez, Zookeeper

Hadoop Distributed File System (HDFS) (1)

- Inspired by “Google File System”
- Stores large files (typically gigabytes-terabytes) across multiple machines, replicating across multiple hosts
  - Breaks up files into fixed-size blocks (typically 64 MiB), distributes blocks
  - The blocks of a file are replicated for fault tolerance
  - The block size and replication factor are configurable per file
  - Default replication value (3) - data is stored on three nodes: two on the same rack, and one on a different rack
- File system intentionally not fully POSIX-compliant
  - Write-once-read-many access model for files. A file once created, written, and closed cannot be changed. This assumption simplifies data coherency issues and enables high throughput data access
  - Intend to add support for appending-writes in the future
  - Can rename & remove files
- “Namenode” tracks names and where the blocks are

Hadoop Distributed File System (HDFS) (2)

- Hadoop can work with any distributed file system but this loses locality
- To reduce network traffic, Hadoop must know which servers are closest to the data; **HDFS** does this
- Hadoop job tracker schedules jobs to task trackers with an awareness of the data location
  - For example, if node A contains data (x,y,z) and node B contains data (a,b,c), the job tracker schedules node B to perform tasks on (a,b,c) and node A would be scheduled to perform tasks on (x,y,z)
  - This reduces the amount of traffic that goes over the network and prevents unnecessary data transfer
  - Location awareness can significantly reduce job-completion times when running data-intensive jobs

Source: http://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html

Handling structured data

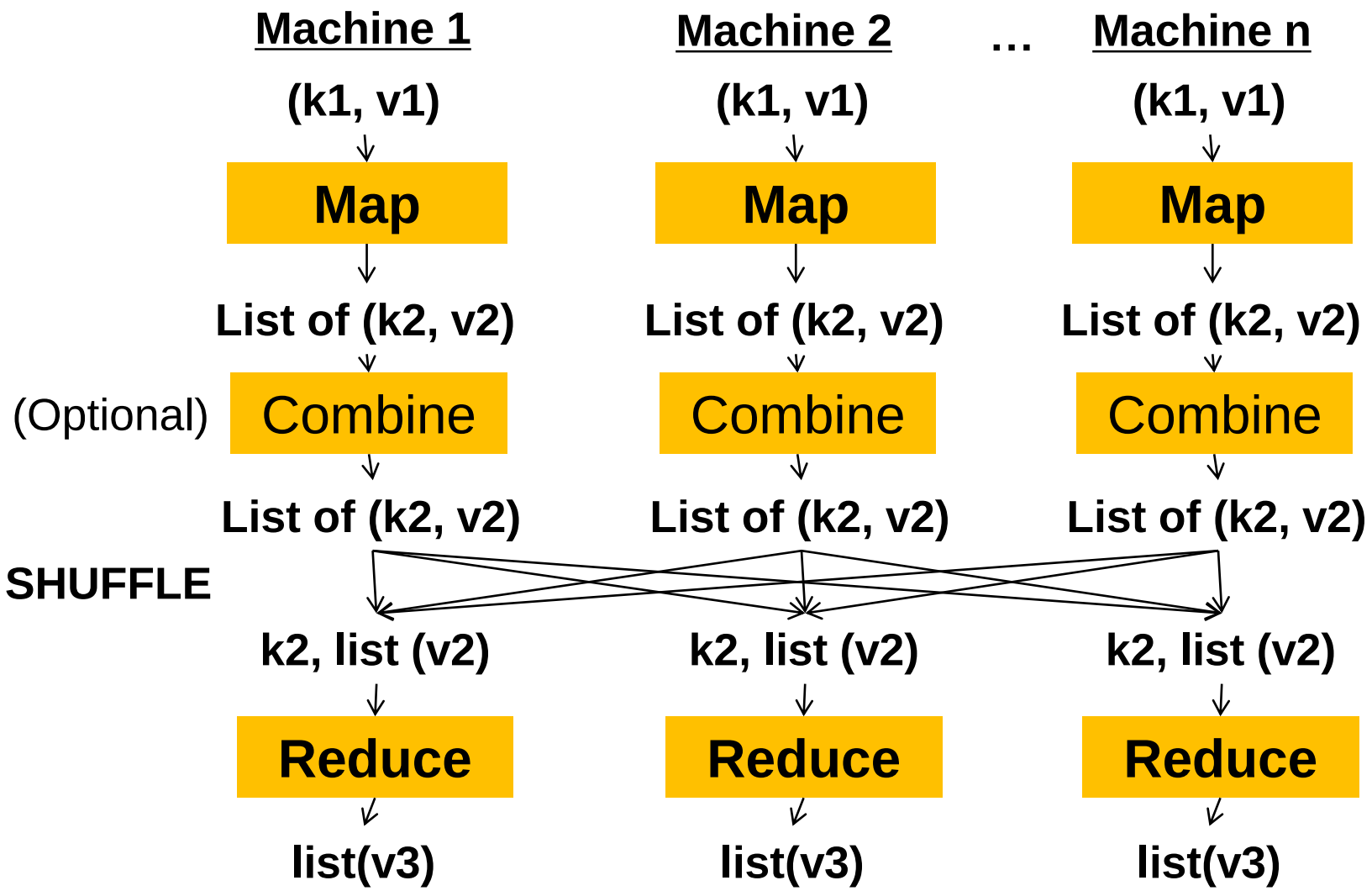
- Data often more structured
- Google **BigTable** (2006 paper)
  - Designed to scale to Petabytes, 1000s of machines
  - Maps two arbitrary string values (row key and column key) and timestamp into an associated arbitrary byte array
  - Tables split into multiple "tablets" along row chosen so tablet will be ~200 megabytes in size (compressed when necessary)
  - Data maintained in lexicographic order by row key; clients can exploit this by selecting row keys for good locality (e.g., reversing hostname in URL)
  - Not a relational database; really a sparse, distributed multi-dimensional sorted map
- Implementations of approach include: Apache Accumulo (from NSA; cell-level access labels), Apache Cassandra, Apache Hbase, Google Cloud BigTable (released 2005)

Sources: https://en.wikipedia.org/wiki/BigTable; "Bigtable…" by Chang

(Parallel) MapReduce

- **MapReduce** is a programming model for processing and generating large data sets with a parallel, distributed algorithm on a cluster
- Programmer defines two functions, **map & reduce**
  - Map(k1,v1) → list(k2,v2). Takes a series of key/value pairs, processes each, generates zero or more output key/value pairs
  - Reduce(k2, list (v2)) → list(v3). Executed once for each unique key k2 in the sorted order; iterate through the values associated with that key and produce zero or more outputs
- System "shuffles" data between map and reduce (so "reduce" function has whole set of data for its given keys) & automatically handles system failures, etc.

MapReduce process figure



MapReduce: Word Count Example (Pseudocode)

```
map(String input_key, String input_value):
    // input_key: document name
    // input_value: document contents
    for each word w in input_value:
        EmitIntermediate(w, "1");

reduce(String output_key, Iterator intermediate_values):
    // output_key: a word
    // output_values: a list of counts
    int result = 0;
    for each v in intermediate_values:
        result += ParseInt(v);
    Emit(AsString(result));
```

Any 12
Ball 1
Computer 3
…

http://research.google.com/archive/mapreduce-osdi04-slides/index-auto-0004.html

MapReduce Combiner

- Can also define an option function “Combiner” (to optimize bandwidth)
  - If defined, runs after Mapper & before Reducer on every node that has run a map task
  - Combiner receives as input all data emitted by the Mapper instances on a given node
  - Combiner output sent to the Reducers, instead of the output from the Mappers
  - Is a "mini-reduce" process which operates only on data generated by one machine
- If a reduce function is both commutative and associative, then it can be used as a Combiner as well
- Useful for word count – combine local counts

Source: https://developer.yahoo.com/hadoop/tutorial/module4.html

MapReduce problems & potential solution

- MapReduce problems:
  - Many problems aren't easily described as map-reduce
  - Persistence to disk typically slower than in-memory work
- Alternative: Apache Spark
  - a general-purpose processing engine that can be used instead of MapReduce

Apache Spark

- Processing engine; instead of just “map” and “reduce”, defines a large set of *operations* (transformations & actions)
  - Operations can be arbitrarily combined in any order
- Open source software
- **Supports** Java, Scala and Python
- Original key construct: Resilient Distributed Dataset (RDD)*
  - Original construct, so we’ll focus on that first
- More recently added: DataFrames & DataSets
  - Different APIs for aggregate data

*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.

Resilient Distributed Dataset (RDD) – key Spark construct

- RDDs represent data or **transformations** on data
- RDDs can be created from Hadoop Input Formats (such as HDFS files), "parallelize()" datasets, or by transforming other RDDs (you can stack RDDs)
- **Actions** can be applied to RDDs; actions force calculations and return values
- **Lazy evaluation**: Nothing computed until an action requires it
- RDDs are best suited for applications that apply the same operation to all elements of a dataset
  - Less suitable for applications that make asynchronous fine-grained updates to shared state

Spark example (Python)

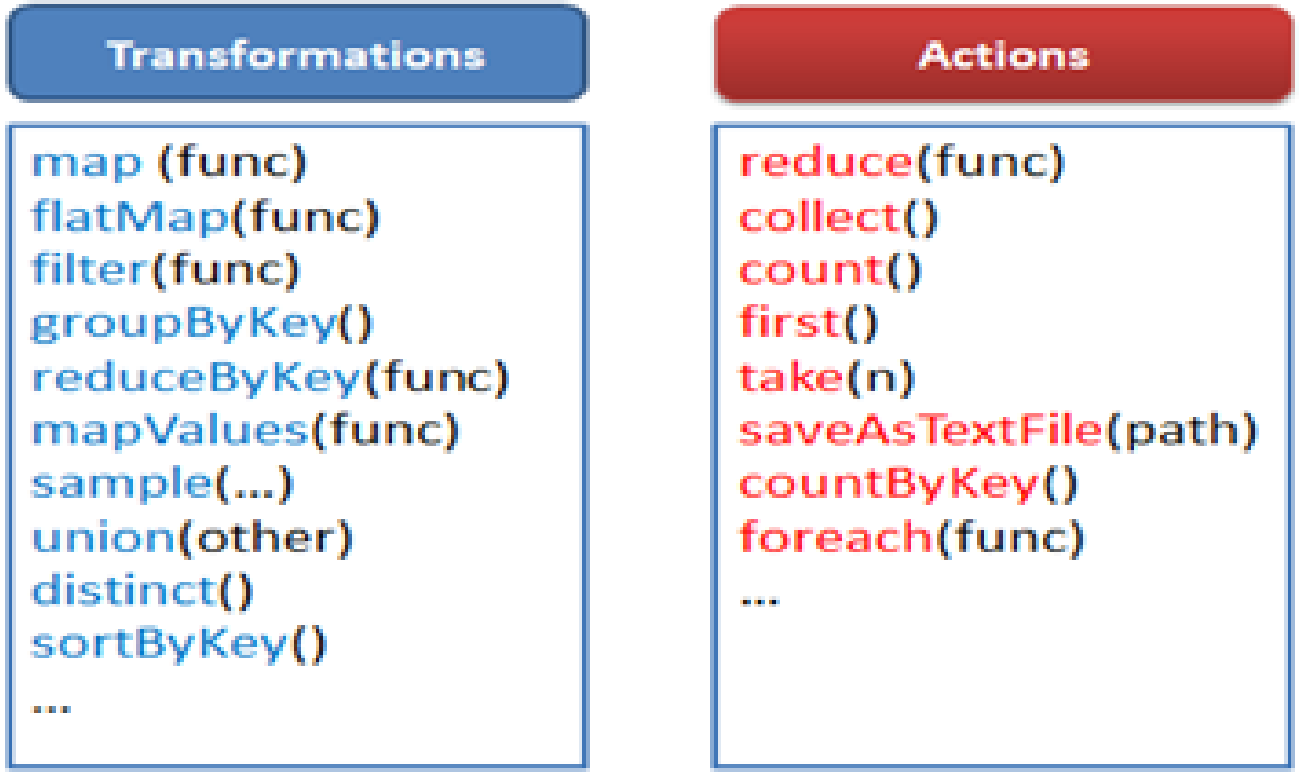
```
#  Estimate π (compute-intensive task).
# Pick  random points in the unit square ((0, 0) to (1,1)),
# See how many fall in the unit circle. The fraction should be π / 4
# Note that “parallelize” method creates an RDD
def sample(p):
    x, y = random(), random()
    return 1 if x*x + y*y < 1 else 0

count = spark.parallelize(xrange(0, NUM_SAMPLES)).map(sample) \
            .reduce(lambda a, b: a + b)
print "Pi is roughly %f" % (4.0 * count / NUM_SAMPLES)
```

Source: https://spark.apache.org/docs/latest/quick-start.html

Transformations and Actions in RDDs

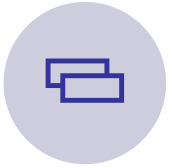
- Some of functions in RDDs



Some Spark transformations

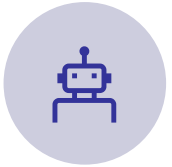
map(func): Return a new distributed dataset formed by passing each element of the source through a function func.

filter(func): Return a new dataset formed by selecting those elements of the source on which func returns true



union(otherDataset): Return a new dataset that contains the union of the elements in the source dataset and the argument.

intersection(otherDataset): Return a new RDD that contains the intersection of elements in the source dataset and the argument.



distinct([numTasks])): Return a new dataset that contains the distinct elements of the source dataset

join(otherDataset, [numTasks]): When called on datasets of type (K, V) and (K, W), returns a dataset of (K, (V, W)) pairs with all pairs of elements for each key. Outer joins are supported through leftOuterJoin, rightOuterJoin, and fullOuterJoin.

Source: https://spark.apache.org/docs/latest/programming-guide.html

Some Spark Actions

- reduce(func): Aggregate the elements of the dataset using a function func (which takes two arguments and returns one). The function should be commutative and associative so that it can be computed correctly in parallel.
- collect(): Return all the elements of the dataset as an array at the driver program. This is usually useful after a filter or other operation that returns a sufficiently small subset of the data.
- count(): Return the number of elements in the dataset.

Remember: *Actions* cause calculations to be performed; *transformations* just set things up (lazy evaluation)

Source: https://spark.apache.org/docs/latest/programming-guide.html

Transformations and Actions in RDDs – pySpark

VICTORIAN INSTITUTE OF TECHNOLOGY
www.vit.edu.au

pySpark: Read data and take 5

How you can do Spark things in Python

```
rdd = sc.textFile("file:///home/edureka/Desktop/Sample")
```

```
rdd.take(5)
```

```
def Func(lines):
      lines = lines.lower()
      lines = lines.split()
      return lines
rdd1 = rdd.map(Func)
```

```
rdd2 = rdd.flatMap(Func)
rdd2.take(5)
```

```
[u'contracts,', u'transactions,', u'and', u'the', u'records']
```

```
stopwords = ['a','all','the','as','is','am','an','and','be','been','from','had','I','I'd','why','with']
rdd3 = rdd2.filter(lambda x: x not in stopwords)
rdd3.take(10)
```

```
rdd4 = rdd3.groupBy(lambda w: w[0:3])
print [(k, list(v)) for (k, v) in rdd4.take(1)]
```

```
[(u'edg', [u'edges'])]
```

```
sqlContext.sql('select * from NYC_Flights where air_time IN (select min(air_time) from NYC_Flights)').show()
```

```
+----+-----+---+--------+---------+--------+---------+-------+-------+------+------+------+--------+----+------+
|year|month|day|dep_time|dep_delay|arr_time|arr_delay|carrier|tailnum|flight|origin|dest|air_time|distance|hour|minute|
+----+-----+---+--------+---------+--------+---------+-------+-------+------+------+----+--------+--------+----+------+
|2013|    1| 16|    1355|       40|    1442|       31|     EV| N16911|  4368|   EWR| BDL|      20|     116|  13|    55|
|2013|    4| 13|     537|       10|     622|       -6|     EV| N12167|  4631|   EWR| BDL|      20|     116|   5|    37|
+----+-----+---+--------+---------+--------+---------+-------+-------+------+------+----+--------+--------+----+------+
```

Spark – RDD Persistence

- You can persist (cache) an RDD
- When you persist an RDD, each node stores any partitions of it that it computes in memory and reuses them in other actions on that dataset (or datasets derived from it)
- Allows future actions to be much faster (often >10x).
- Mark RDD to be persisted using the persist() or cache() methods on it. The first time it is computed in an action, it will be kept in memory on the nodes.
- Cache is fault-tolerant – if any partition of an RDD is lost, it will automatically be recomputed using the transformations that originally created it
- Can choose storage level (MEMORY_ONLY, DISK_ONLY, MEMORY_AND_DISK, etc.)
- Can manually call unpersist()

Spark Example (Python)

```
# Logistic Regression - iterative machine learning algorithm
# Find best hyperplane that separates two sets of points in a
# multi-dimensional feature space.  Applies MapReduce operation
# repeatedly to the same dataset, so it benefits greatly
# from caching the input in RAM


points = spark.textFile(...).map(parsePoint).cache()
w = numpy.random.ranf(size = D) # current separating plane
for i in range(ITERATIONS):
    gradient = points.map(
        lambda p: (1 / (1 + exp(-p.y*(w.dot(p.x)))) - 1) * p.y * p.x
    ).reduce(lambda a, b: a + b)
    w -= gradient
print "Final separating plane: %s" % w
```

Source: https://spark.apache.org/docs/latest/quick-start.html

A tale of multiple APIs

- For simplicity I've emphasized a single API set – the original one, RDD
- Spark now has three sets of APIs—RDDs, DataFrames, and Datasets
  - RDD – In Spark 1.0 release, "lower level"
  - DataFrames – Introduced in Spark 1.3 release
  - Dataset – Introduced in Spark 1.6 release
- Each with pros/cons/limitations

DataFrame & Dataset

- DataFrame:
  - Unlike an RDD, data organized into named columns, e.g. a table in a relational database.
  - Imposes a structure onto a distributed collection of data, allowing higher-level abstraction
- Dataset:
  - Extension of DataFrame API which provides type-safe, object-oriented programming interface (compile-time error detection)

Both built on Spark SQL engine & use Catalyst to generate optimized logical and physical query plan; both can be converted to an RDD

https://data-flair.training/blogs/apache-spark-rdd-vs-dataframe-vs-dataset/
https://databricks.com/blog/2016/07/14/a-tale-of-three-apache-spark-apis-rdds-dataframes-and-datasets.html

API distinction: typing

- **Python & R** don't have compile-time type safety checks, so *only* support DataFrame
  - Error detection only at runtime
- **Java & Scala** support compile-time type safety checks, so support *both* DataSet and DataFrame
  - Dataset APIs are all expressed as lambda functions and JVM typed objects
  - any mismatch of typed-parameters will be detected at compile time.

https://databricks.com/blog/2016/07/14/a-tale-of-three-apache-spark-apis-rdds-dataframes-and-datasets.html

Apache Spark: Libraries “on top” of core that come with it

- Spark SQL
- Spark Streaming – stream processing of live datastreams
- MLlib - machine learning
- GraphX – graph manipulation
  - extends Spark RDD with Graph abstraction: a directed multigraph with properties attached to each vertex and edge.

Gray sort competition: Winner Spark-based (previously MR)

	Hadoop MR Record	Spark Record (2014)
Data Size	102.5 TB	100 TB
Elapsed Time	72 mins	23 mins
# Nodes	2100	206
# Cores	50400 physical	6592 virtualized
Cluster disk throughput	3150 GB/s (est.)	618 GB/s
Network	dedicated data center, 10Gbps	virtualized (EC2) 10Gbps network
Sort rate	**1.42 TB/min**	**4.27 TB/min**
Sort rate/node	**0.67 GB/min**	**20.7 GB/min**

Spark-based System 3x faster with 1/10 # of nodes

Sort benchmark, Daytona Gray: sort of 100 TB of data (1 trillion records)

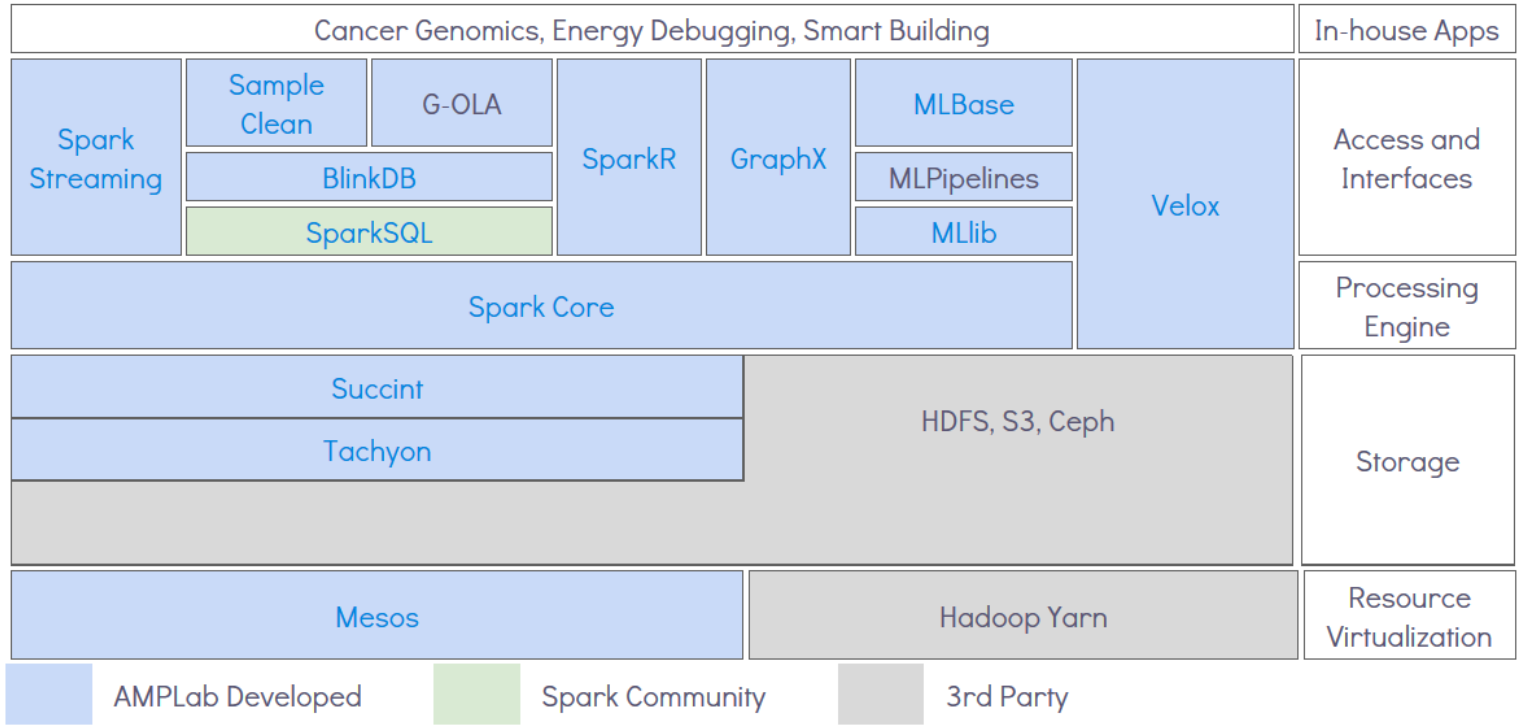
http://databricks.com/blog/2014/11/05/spark-officially-sets-a-new-record-in-large-scale-sorting.html

Spark vs. Hadoop MapReduce

- Performance: Spark normally faster but with caveats
  - Spark can process data in-memory; Hadoop MapReduce persists back to the disk after a map or reduce action
  - Spark generally outperforms MapReduce, but it often needs lots of memory to do well; if there are other resource-demanding services or can't fit in memory, Spark degrades
  - MapReduce easily runs alongside other services with minor performance differences, & works well with the 1-pass jobs it was designed for
- Ease of use: Spark is easier to program
- Data processing: Spark more general
- Maturity: Spark maturing, Hadoop MapReduce mature

"Spark vs. Hadoop MapReduce" by Saggi Neumann (November 24, 2014)
https://www.xplenty.com/blog/2014/11/apache-spark-vs-hadoop-mapreduce/

AMPLab's Berkeley Data Analytics Stack (BDAS)



Don't need to memorize this figure – the point is to know that components can be combined to solve big data problems

Source: https://amplab.cs.berkeley.edu/software/

For more information

- Spark tutorials
  - http://spark-summit.org/2014/training
- “Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing” by Matei Zaharia et al
  - https://www.cs.berkeley.edu/~matei/papers/2012/nsdi_spark.pdf