Apache Spark

In-Memory Cluster Computing for Iterative and Interactive Applications

amplab
UC Berkeley

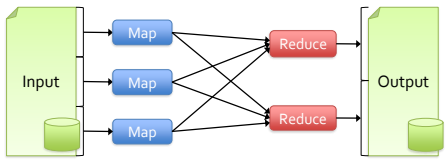
Background

- Commodity clusters have become an important computing platform for a variety of applications
  - **In industry:** search, machine translation, ad targeting, …
  - **In research:** bioinformatics, NLP, climate simulation, …
- High-level cluster programming models like MapReduce power many of these apps
- *Theme of this work: provide similarly powerful abstractions for a broader class of applications*

Motivation

Current popular programming models for clusters transform data flowing from stable storage to stable storage

e.g., MapReduce:



Motivation

- Acyclic data flow is a powerful abstraction, but is not efficient for applications that repeatedly reuse a *working set* of data:
  - **Iterative** algorithms (many in machine learning)
  - **Interactive** data mining tools (R, Excel, Python)
- Spark makes working sets a first-class concept to efficiently support these apps

Spark Goal

- Provide distributed memory abstractions for clusters to support apps with working sets
- Retain the attractive properties of MapReduce:
  - Fault tolerance (for crashes & stragglers)
  - Data locality
  - Scalability

Solution: augment data flow model with "resilient distributed datasets" (RDDs)

Programming Model

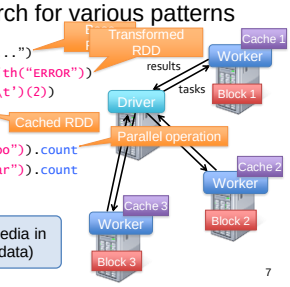
- Resilient distributed datasets (RDDs)
  - Immutable collections partitioned across cluster that can be rebuilt if a partition is lost
  - Created by transforming data in stable storage using data flow operators (map, filter, group-by, …)
  - Can be *cached* across parallel operations
- Parallel operations on RDDs
  - Reduce, collect, count, save, …
- Restricted shared variables
  - Accumulators, broadcast variables

Example 1: Log Mining

•Load error messages from a log into memory, then interactively search for various patterns

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(_.startsWith("ERROR"))
messages = errors.map(_.split('\t')(2))
cachedMsgs = messages.cache()

cachedMsgs.filter(_.contains("foo")).count
cachedMsgs.filter(_.contains("bar")).count
. . .
```



Result: full-text search of Wikipedia in <1 sec (vs 20 sec for on-disk data)

7

RDDs in More Detail

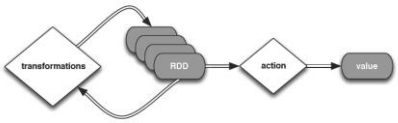
- An RDD is an immutable, partitioned, logical collection of records
  - Need not be materialized, but rather contains information to rebuild a dataset from stable storage
- Partitioning can be based on a key in each record (using hash or range partitioning)
- Built using bulk transformations on other RDDs
- Can be cached for future reuse

8

Spark Essentials: *RDD*

Spark can create RDDs from any file stored in HDFS or other storage systems supported by Hadoop, e.g., local file system, Amazon S3, Hypertable, HBase, etc.

Spark supports text files, SequenceFiles, and any other Hadoop `InputFormat`, and can also take a directory or a glob (e.g. `/data/201404*`)



9

RDD Operations

Transformations (define a new RDD)	Parallel operations (Actions) (return a result to driver)
map	reduce
filter	collect
sample	count
union	save
groupByKey	lookupKey
reduceByKey	…
join	
cache	
…	

10

Spark Essentials: *Transformations*

Transformations create a new dataset from an existing one

All transformations in Spark are *lazy*: they do not compute their results right away – instead they remember the transformations applied to some base dataset

- optimize the required calculations
- recover from lost data partitions

11

Spark Essentials: *Transformations*

transformation	description
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
flatMap(*func*)	similar to map, but each input item can be mapped to 0 or more output items (so *func* should return a Seq rather than a single item)
sample(*withReplacement*, *fraction*, *seed*)	sample a fraction *fraction* of the data, with or without replacement, using a given random number generator *seed*
union(*otherDataset*)	return a new dataset that contains the union of the elements in the source dataset and the argument
distinct([*numTasks*]))	return a new dataset that contains the distinct elements of the source dataset

12

Spark Essentials: *Transformations*

transformation	description
`groupByKey([numTasks])`	when called on a dataset of `(K, V)` pairs, returns a dataset of `(K, Seq[V])` pairs
`reduceByKey(func, [numTasks])`	when called on a dataset of `(K, V)` pairs, returns a dataset of `(K, V)` pairs where the values for each key are aggregated using the given reduce function
`sortByKey([ascending], [numTasks])`	when called on a dataset of `(K, V)` pairs where `K` implements `Ordered`, returns a dataset of `(K, V)` pairs sorted by keys in ascending or descending order, as specified in the boolean ascending argument
`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
`cogroup(otherDataset, [numTasks])`	when called on datasets of type `(K, V)` and `(K, W)`, returns a dataset of `(K, Seq[V], Seq[W])` tuples – also called `groupWith`
`cartesian(otherDataset)`	when called on datasets of types `T` and `U`, returns a dataset of `(T, U)` pairs (all pairs of elements)

13

Spark Essentials: *Actions*

action	description
`reduce(func)`	aggregate the elements of the dataset using a function *func* (which takes two arguments and returns one), and should also 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 – 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
`first()`	return the first element of the dataset – similar to *take(1)*
`take(n)`	return an array with the first *n* elements of the dataset – currently not executed in parallel, instead the driver program computes all the elements
`takeSample(withReplacement, fraction, seed)`	return an array with a random sample of *num* elements of the dataset, with or without replacement, using the given random number generator seed

14

Spark Essentials: *Actions*

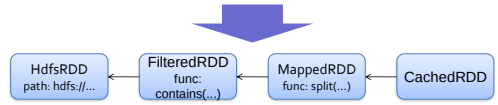
action	description
`saveAsTextFile(path)`	write the elements of the dataset as a text file (or set of text files) in a given directory in the local filesystem, HDFS or any other Hadoop-supported file system. Spark will call `toString` on each element to convert it to a line of text in the file
`saveAsSequenceFile(path)`	write the elements of the dataset as a Hadoop `SequenceFile` in a given path in the local filesystem, HDFS or any other Hadoop-supported file system. Only available on RDDs of key-value pairs that either implement Hadoop's `Writable` interface or are implicitly convertible to `Writable` (Spark includes conversions for basic types like `Int`, `Double`, `String`, etc).
`countByKey()`	only available on RDDs of type `(K, V)`. Returns a `Map` of `(K, Int)` pairs with the count of each key
`foreach(func)`	run a function *func* on each element of the dataset – usually done for side effects such as updating an accumulator variable or interacting with external storage systems

15

RDD Fault Tolerance

- RDDs maintain *lineage* information that can be used to reconstruct lost partitions
- e.g.:

```
cachedMsgs = textFile(...).filter(_.contains("error"))
                          .map(_.split('\t')(2))
                          .cache()
```



16

Example 2: Spark MapReduce for Word Counting

- MapReduce data flow can be expressed using standard RDD transformations

```
res = data.flatMap(rec => myMapFunc(rec))
          .groupByKey()
          .map((key, vals) => myReduceFunc(key, vals))
```

Or with combiners:

```
res = data.flatMap(rec => myMapFunc(rec))
          .reduceByKey(myCombiner)
          .map((key, val) => myReduceFunc(key, val))
```

17

Simple Spark Apps: *WordCount*

Definition:

count how often each word appears in a collection of text documents

This simple program provides a good test case for parallel processing, since it:

- requires a minimal amount of code
- demonstrates use of both symbolic and numeric values
- isn't many steps away from search indexing
- serves as a "Hello World" for Big Data apps

A distributed computing framework that can run WordCount **efficiently in parallel at scale** can likely handle much larger and more interesting compute problems

```
void map (String doc_id, String text):
  for each word w in segment(text):
    emit(w, "1");

void reduce (String word, Iterator group):
  int count = 0;

  for each pc in group:
    count += Int(pc);

  emit(word, String(count));
```

18

Simple Spark Apps: *WordCount*

```
val f = sc.textFile(inputPath)
val w = f.flatMap(l => l.split(" ")).map(word => (word, 1)).cache()
w.reduceByKey(_ + _).saveAsText(outputPath)
```

WordCount in 3 lines of Spark

WordCount in 50+ lines of Java MR

Simple Spark Apps: *WordCount*

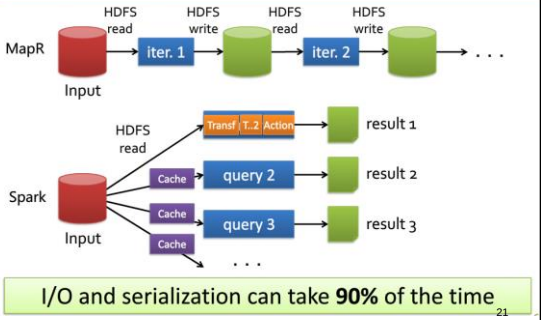
Scala:

```
val f = sc.textFile("README.md")
val wc = f.flatMap(l => l.split(" ")).map(word => (word, 1)).reduceByKey(_ + _)
wc.saveAsTextFile("wc_out")
```

Python:

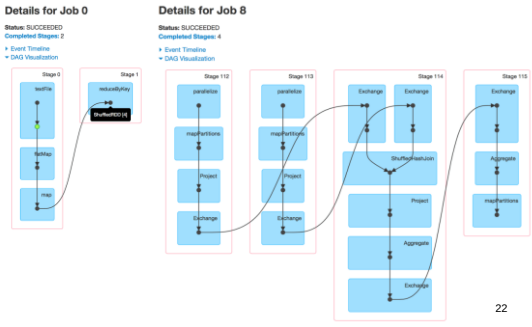
```
from operator import add
f = sc.textFile("README.md")
wc = f.flatMap(lambda x: x.split(' ')).map(lambda x: (x, 1)).reduceByKey(add)
wc.saveAsTextFile("wc_out")
```


Spark MapR faster than usual MapR?



I/O and serialization can take **90%** of the time

Spark RDD Computation DAG



RDD vs DataFrame

- New DataFrame API
  - Goal: enable wider audiences beyond "Big Data" engineers to leverage power of distributed processing
  - DataFrames are like database tables
    - Can load from csv's, json, etc.
    - Implemented underneath as RDDs
    - Have column meta-data that allows further optimization
  - DataFrames support a **subset of SQL queries** that return new DataFrames

DataFrame Supported Operators

map	reduce	sample
filter	count	take
groupBy	fold	first
sort	reduceByKey	partitionBy
union	groupByKey	save
join	cogroup	...
leftOuterJoin	cross	
rightOuterJoin	zip	

And even a parser for direct SQL query statements, i.e.

```
df2 = spark.sql("SELECT field1 AS f1, field2 as f2 from table1")
```

Java DataFrame Example

```
JavaSparkContext sc = ...; // An existing JavaSparkContext.
SQLContext sqlContext = new org.apache.spark.sql.SQLContext(sc);

// Load directly from json
DataFrame df = sqlContext.read().json("examples/src/main/resources/people.json");

// Displays the content of the DataFrame to stdout
df.show();
```

https://www.tutorialspoint.com/spark_sql/spark_sql_dataframes.htm

25

Java DataFrame Operations

```
// Print the schema in a tree format
df.printSchema();

// Select only the "name" column
df.select("name").show();

// Select everybody, but increment the age by 1
df.select(df.col("name"), df.col("age").plus(1)).show();

// Select people older than 21
df.filter(df.col("age").gt(21)).show();

// Count people by age
df.groupBy("age").count().show();
```

A Dataframe overlays columnar named schema on the RDD permitting direct SQL and SQL-like operations

```
// SQL can even be run over RDDs that
// have been registered as tables.
DataFrame teenagers =
sqlContext.sql("SELECT name FROM people
        WHERE age >= 13 AND age <= 19")
```

26

Conclusion

- By making distributed datasets a first-class primitive, Spark provides a simple, efficient programming model for stateful data analytics
- RDDs provide:
  - Lineage info for fault recovery and debugging
  - Adjustable in-memory caching
- DataFrames provide
  - Database (SQL) and Pandas-like functionality
- Spark is the industry standard for Big Data processing
  - Now with built-in support for
    - Machine learning via MIlib,
    - Graphs and social network analysis via GraphX,
    - Streaming data

27