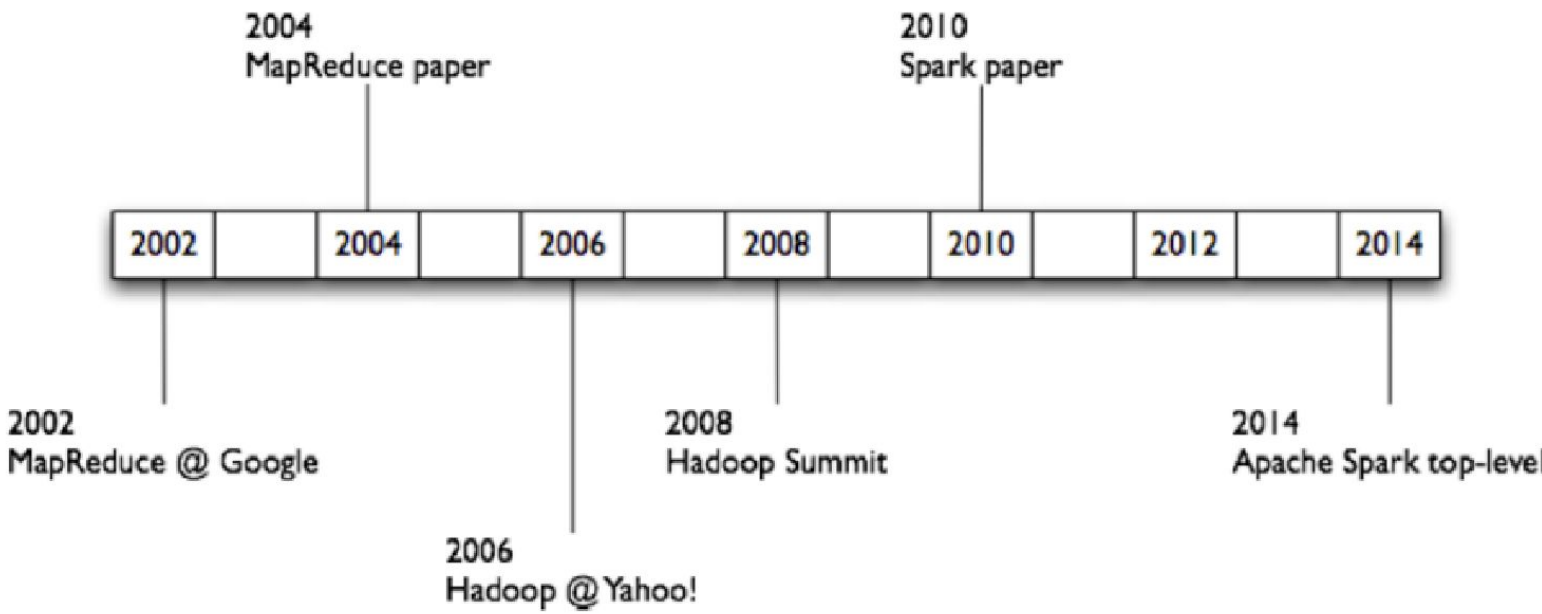
Apache Spark - Big Data Analytics

Matei Zaharia, Mosharaf Chowdhury, Tathagata Das,
Ankur Dave, Justin Ma, Murphy McCauley, Michael Franklin,
Scott Shenker, Ion Stoica - UC Berkeley.

The first part of this presentation is adapted from a talk by Zaharia.

History of Hadoop and Spark



Project Goals

Extend the MapReduce model to better support two common classes of analytics apps:

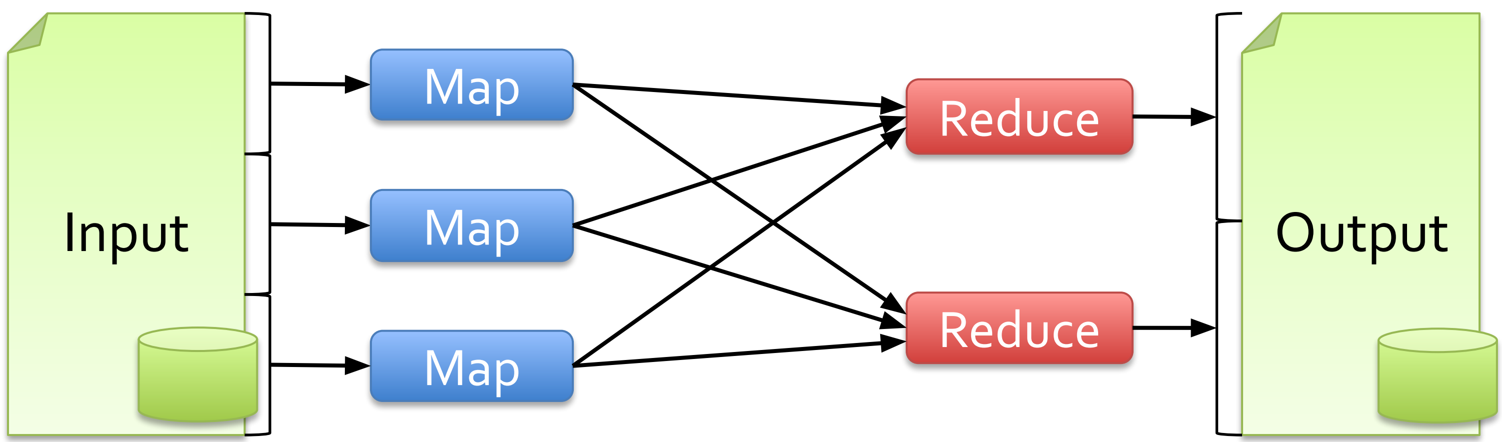
- **Iterative** algorithms (machine learning, graphs)
- **Interactive** data mining

Enhance programmability:

- Integrate into Scala programming language
- Allow interactive use from Scala interpreter
- Also: Java, Python, R

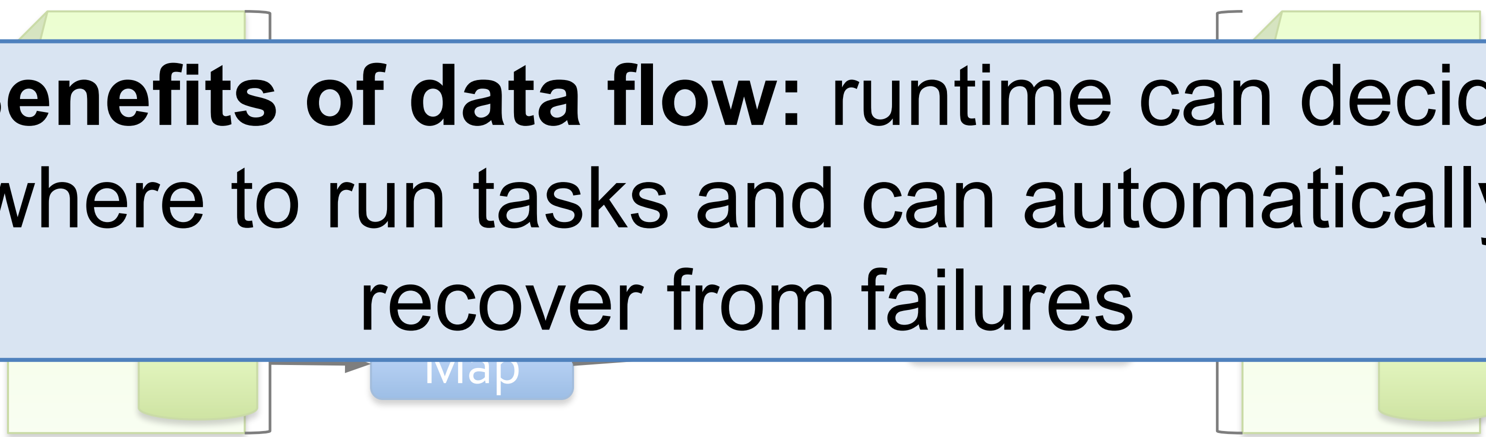
Motivation

Most current cluster programming models are based on *acyclic data flow* from stable storage to stable storage



Motivation

Most current cluster programming models are based on *acyclic data flow* from stable storage to stable storage



Benefits of data flow: runtime can decide where to run tasks and can automatically recover from failures

Motivation

Acyclic data flow is inefficient for applications that repeatedly reuse a *working set* of data:

- **Iterative** algorithms (machine learning, graphs)
- **Interactive** data mining tools (R, Excel, Python)

With current frameworks, apps reload data from stable storage on each query

Solution: Resilient Distributed Datasets (RDDs)

Allow apps to keep working sets in memory for efficient reuse

Retain the attractive properties of MapReduce

- Fault tolerance, data locality, scalability

Support a wide range of applications

Programming Model

- **Resilient distributed datasets (RDDs)**
  - Immutable, partitioned collections of objects
  - Created through parallel *transformations* (map, filter, groupBy, join, …) on data in stable storage
  - Can be *cached* for efficient reuse

- *Actions* on RDDs
  - Count, reduce, collect, save, …

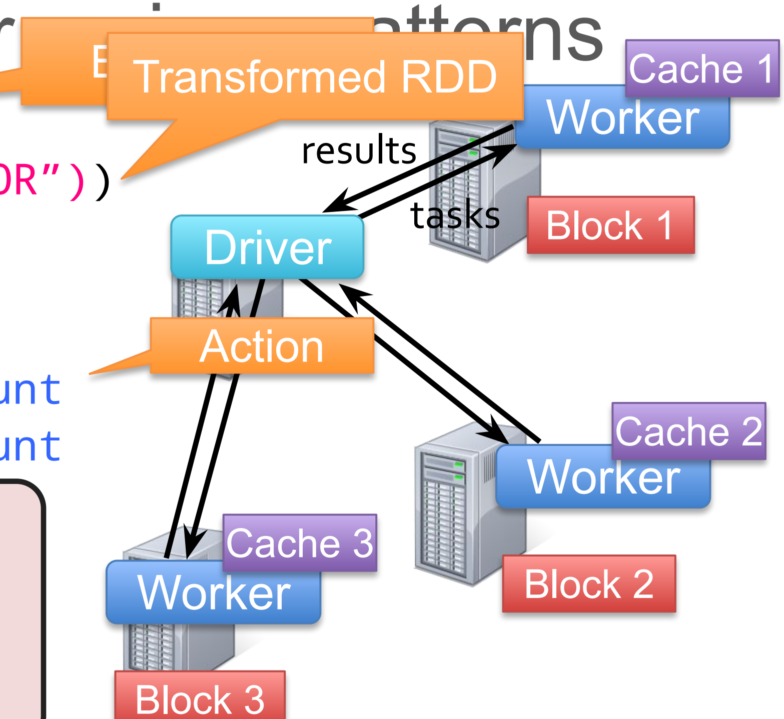
- More recently:
  - DataSet and DataFrame instead of RDD
  - Allows multiple columns

Example: Log Mining

Load error messages from a log into memory, then interactively search for 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
```



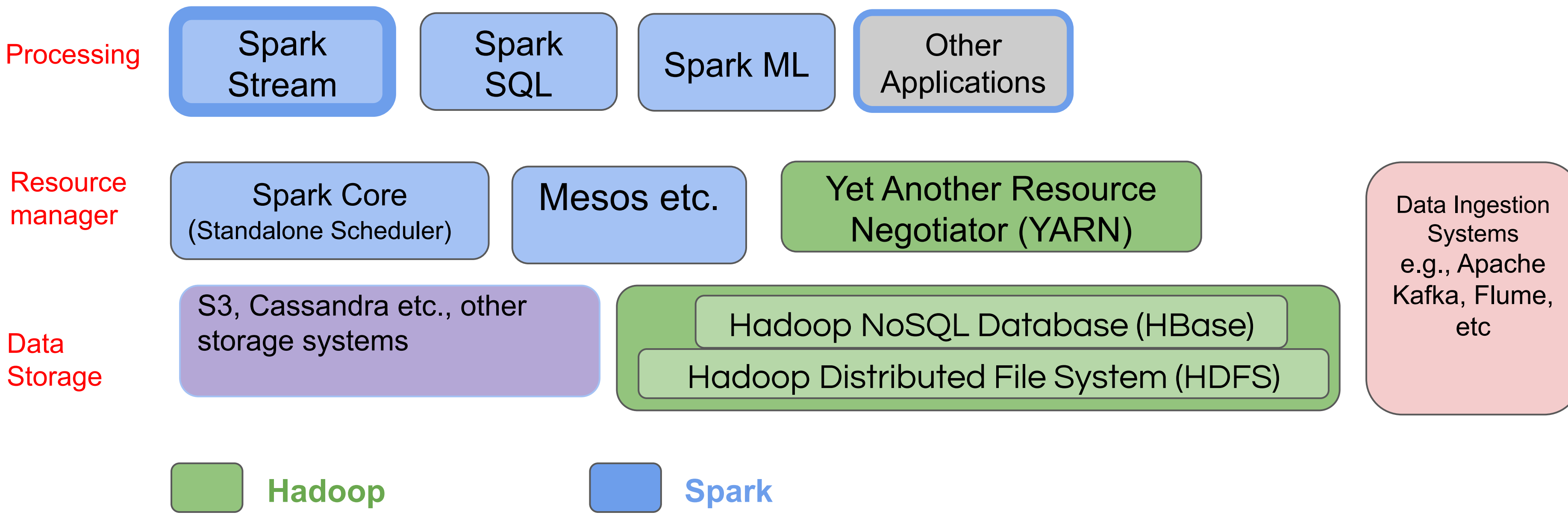
Result: scaled to 1 TB data in 5-7 sec
(vs 170 sec for on-disk data)

Operations on RDDs

Transformations (define a new RDD)	map filter sample groupByKey reduceByKey sortByKey	flatMap union join cogroup cross mapValues
Actions (return a non-RDD result to driver program)	collect reduce count save lookupKey	

Apache Spark

** Spark can connect to several types of *cluster managers* (either Spark's own standalone cluster manager, Mesos or YARN)



PySpark

Spark Python API - PySpark

PySpark installation with pip

```
>>> pip install pyspark
```

To run a Spark application app_name

```
>>> from pyspark.sql import SparkSession
```

instantiate your SparkSession object

```
>>> spark = SparkSession.builder.appName("app_name")\
... .getOrCreate()
```

stop your SparkSession

```
>>> spark.stop()
```

Interactive PySpark with RDDs

```
# SparkSession available as spark
# load the data directly into an RDD
>>> titanic = spark.sparkContext.textFile('titanic.csv')
# the file is of the format
# Survived | Class | Name | Sex | Age | Siblings/Spouses Aboard | Parents/ -
Children Aboard | Fare
# get the first n (2) objects in the RDD
>>> titanic.take(2)
['0,3,Mr. Owen Harris Braund,male,22,1,0,7.25',
'1,1,Mrs. John Bradley (Florence Briggs Thayer) Cumings,female,38,1,0,71.283']
# note that each element is a single string - not particularly useful
# one option is to first load the data into a numpy array
>>> np_titanic = np.loadtxt('titanic.csv', delimiter=',', dtype=list)
# use sparkContext to parallelize the data into 4 partitions
>>> titanic_parallelize = spark.sparkContext.parallelize(np_titanic, 4)
>>> titanic_parallelize.take(2)
[array(['0', '3', ..., 'male', '22', '1', '0', '7.25'], dtype=object),
array(['1', '1', ..., 'female', '38', '1', '0', '71.2833'], dtype=object)]
```

Transformations on RDDs

```
# use map() to format the data
>>> titanic = spark.sparkContext.textFile('titanic.csv')
>>> titanic.take(2)
['0,3,Mr. Owen Harris Braund,male,22,1,0,7.25',
'1,1,Mrs. John Bradley (Florence Briggs Thayer) Cumings,female,38,1,0,71.283']
# apply split(',') to each element of the RDD with map()
>>> titanic.map(lambda row: row.split(',')).take(2)
[['0', '3', 'Mr. Owen Harris Braund', 'male', '22', '1', '0', '7.25'],
['1', '1', ..., 'female', '38', '1', '0', '71.283']]
# compare to flatMap(), which flattens the results of each row
>>> titanic.flatMap(lambda row: row.split(',')).take(2)
['0', '3']
# create a new RDD containing only the female passengers
>>> titanic = titanic.map(lambda row: row.split(','))
>>> titanic_f = titanic.filter(lambda row: row[3] == 'female')
>>> titanic_f.take(3)
[['1', '1', ..., 'female', '38', '1', '0', '71.2833'],
['1', '3', ..., 'female', '26', '0', '0', '7.925'],
['1', '1', ..., 'female', '35', '1', '0', '53.1']]
```

RDD Transformations

Spark Command	Transformation
`map(f)`	Returns a new RDD by applying `f` to each element of this RDD
`flatMap(f)`	Same as `map(f)`, except the results are flattened
`filter(f)`	Returns a new RDD containing only the elements that satisfy `f`
`distinct()`	Returns a new RDD containing the distinct elements of the original
`reduceByKey(f)`	Takes an RDD of `(key, val)` pairs and merges the values for each `key` using an associative and commutative reduce function `f`
`sortBy(f)`	Sorts this RDD by the given function `f`
`sortByKey(f)`	Sorts an RDD assumed to consist of `(key, val)` pairs by the given function `f`
`groupBy(f)`	Returns a new RDD of groups of items based on `f`
`groupByKey()`	Takes an RDD of `(key, val)` pairs and returns a new RDD with `(key, (val1, val2, ...))` pairs

Actions

```
# create a new RDD containing only survival data
>>> survived = titanic.map(lambda row: int(row[0]))
>>> survived.take(5)
[0, 1, 1, 1, 0]
# find total number of survivors
>>> survived.reduce(lambda x, y: x + y)
500
```

Actions

Spark Command	Action
`take(n)`	returns the first `n` elements of an RDD
`collect()`	returns the entire contents of an RDD
`reduce(f)`	merges the values of an RDD using an associative and commutative operator `f`
`count()`	returns the number of elements in the RDD
`min(); max(); mean()`	returns the minimum, maximum, or mean of the RDD, respectively
`sum()`	adds the elements in the RDD and returns the result
`saveAsTextFile(path)`	saves the RDD as a collection of text files (one for each partition) in the directory specified
`foreach(f)`	immediately applies `f` to each element of the RDD; not to be confused with `map()`, `foreach()` is useful for saving data somewhere not natively supported by PySpark

Dataframes

DataFrames are immutable distributed collections of objects; however,unlike RDDs, DataFrames are organized into named (and typed) columns. In this way they are conceptually similar to a relational database (or a pandas DataFrame).

Dataframes in PySpark

```
# load the titanic dataset using default settings. inferSchema=True
>>> titanic = spark.read.csv('titanic.csv')
>>> titanic.show(2)
+---+---+--------------------+------+---+---+---+-------+
|_c0|_c1|                 _c2|   _c3|_c4|_c5|_c6|    _c7|
+---+---+--------------------+------+---+---+---+-------+
|  0|  3|Mr. Owen Harris B...|  male| 22|  1|  0|   7.25|
|  1|  1|Mrs. John Bradley...|female| 38|  1|  0|71.2833|
+---+---+--------------------+------+---+---+---+-------+
only showing top 2 rows
# load the titanic dataset specifying the schema
>>> schema = ('survived INT, pclass INT, name STRING, sex STRING, ', 'age
FLOAT, sibsp INT, parch INT, fare FLOAT')
>>> titanic = spark.read.csv('titanic.csv', schema=schema)
>>> titanic.show(2)
+--------+------+--------------------+------+---+-----+-----+-------+
|survived|pclass|                name|   sex|age|sibsp|parch|   fare|
+--------+------+--------------------+------+---+-----+-----+-------+
|       0|     3|Mr. Owen Harris B...|  male| 22|    1|    0|   7.25|
|       1|     1|Mrs. John Bradley...|female| 38|    1|    0|71.2833|
+--------+------+--------------------+------+---+-----+-----+-------+
only showing top 2 rows
# for files with headers, the following is convenient
>>> spark.read.csv('my_file.csv', header=True, inferSchema=True)
```

SQL with Dataframes

DataFrames can be easily updated, queried, and analyzed using SQL operations. Spark allows you to run queries directly on DataFrames similar to how you perform transformations on RDDs. Additionally, the `pyspark.sql.functions` module contains many additional functions for further analysis.

```
# select data from the survived column
>>> titanic.select(titanic.survived).show(3)
+--------+
|survived|
+--------+
| 0|
| 1|
| 1|
+--------+
only showing top 3 rows
# find all distinct ages of passengers (great for data exploration)
>>> titanic.select("age").distinct().show(3)
+----+
| age|
+----+
|18.0|
|64.0|
|0.42|
+----+
only showing top 3 rows
```

Aggregations and GroupBy

```
# filter the DataFrame for passengers between 20-30 years old (inclusive)
>>> titanic.filter(titanic.age.between(20, 30)).show(3)
+--------+------+--------------------+------+----+-----+-----+------+
|survived|pclass| name| sex| age|sibsp|parch| fare|
+--------+------+--------------------+------+----+-----+-----+------+
| 0| 3|Mr. Owen Harris B...| male|22.0| 1| 0| 7.25|
| 1| 3|Miss. Laina Heikk...|female|26.0| 0| 0| 7.925|
| 0| 3| Mr. James Moran| male|27.0| 0| 0|8.4583|
+--------+------+--------------------+------+----+-----+-----+------+
only showing top 3 rows
# find total fare by pclass (or use .avg('fare') for an average)
>>> titanic.groupBy('pclass').sum('fare').show()
+------+---------+
|pclass|sum(fare)|
+------+---------+
| 1| 18177.41|
| 3| 6675.65|
| 2| 3801.84|
+------+---------+
```

Spark SQL Commands

Spark SQL Command	SQLite Command
`select(*cols)`	`SELECT`
`groupBy(*cols)`	`GROUP BY`
`sort(*cols, **kwargs)`	`ORDER BY`
`filter(condition)`	`WHERE`
`when(condition, value)`	`WHEN`
`between(lowerBound, upperBound)`	`BETWEEN`
`drop(*cols)`	`DROP`
`join(other, on=None, how=None)`	`JOIN` (join type specified by `how`)
`count()`	`COUNT()`
`sum(*cols)`	`SUM()`
`avg(*cols)` or `mean(*cols)`	`AVG()`
`collect()`	`fetchall()`