Big Data Engineering and Data Science on Apache Spark

This Lecture

Big Data Problems: Distributing Work

Resilient Distributed Datasets (RDDs)

Creating an RDD

Spark RDD Transformations and Actions

Spark RDD Programming Model

Spark Shared Variables

What is Apache Spark?

Scalable, *efficient analysis* of *Big Data*

What is Apache Spark?

Scalable, *efficient analysis* of *Big Data*

This lecture

What's Hard About Cluster Computing?

How do we split work across machines?

Let's look at a simple task: word counting

How do you count the number of occurrences of each word in a document?

"I am Sam
I am Sam
Sam I am
Do you like
Green eggs and ham?"

I: 3
am: 3
Sam: 3
do: 1
you: 1
like: 1
…

One Approach: Use a Hash Table

"I am Sam
I am Sam
Sam I am
Do you like
Green eggs and ham?"

{}

One Approach: Use a Hash Table

“I am Sam
I am Sam
Sam I am
Do you like
Green eggs and ham?”

{I : 1}

One Approach: Use a Hash Table

"I $\boxed{\text{am}}$ Sam
I am Sam
Sam I am
Do you like
Green eggs and ham?"

{I: **1**,
am: **1**}

One Approach: Use a Hash Table

"I am Sam
I am Sam
Sam I am
Do you like
Green eggs and ham?"

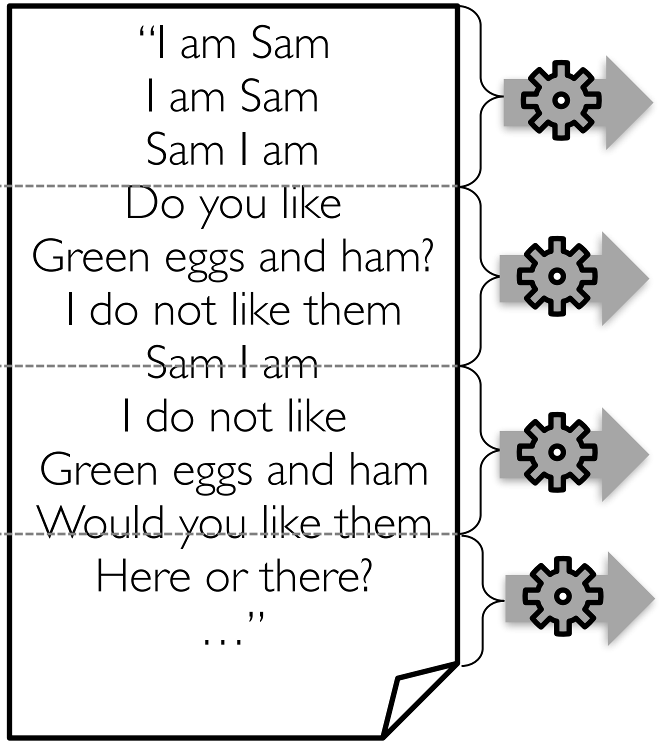
{I: **1**,
am: **1**,
Sam: **1**}

One Approach: Use a Hash Table

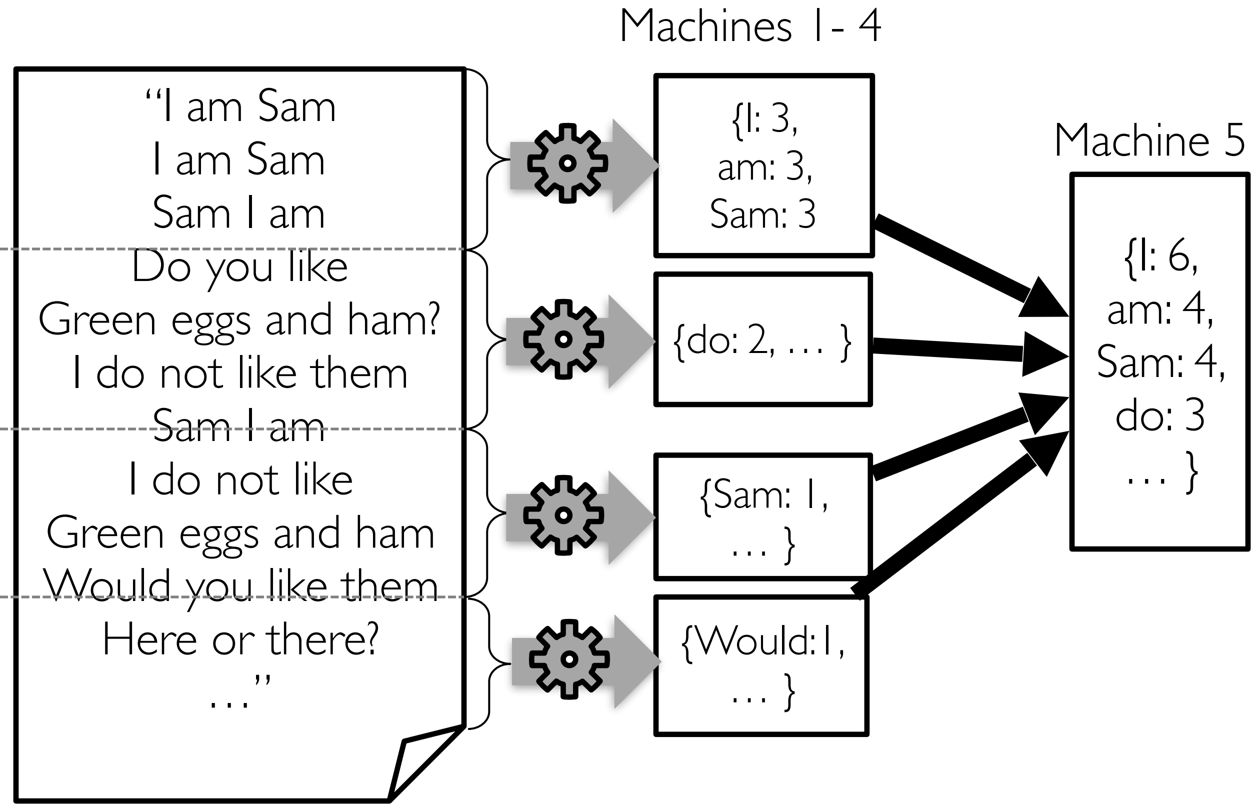
"I am Sam
$\boxed{\text{I}}$ am Sam
Sam I am
Do you like
Green eggs and ham?"

{I: 2,
am: 1,
Sam: 1}

What if the Document is Really Big?

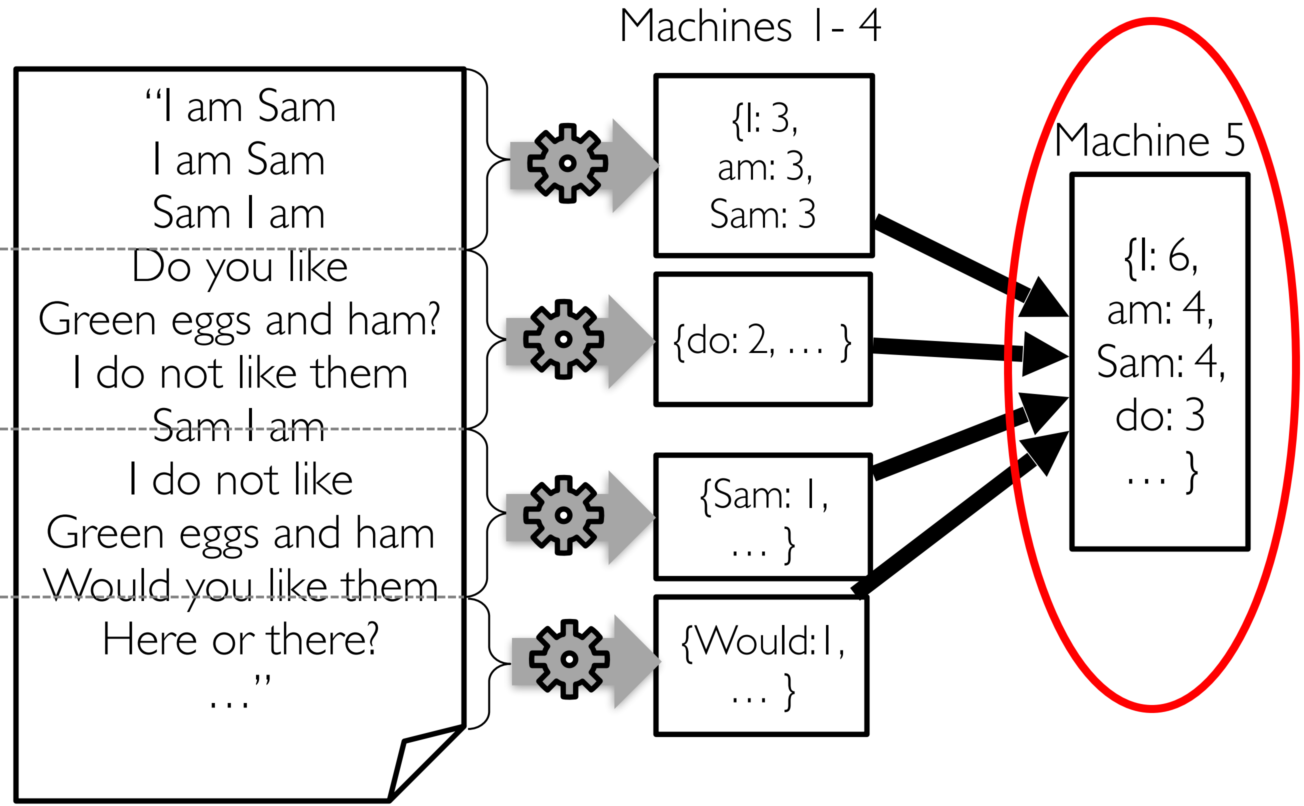


What if the Document is Really Big?



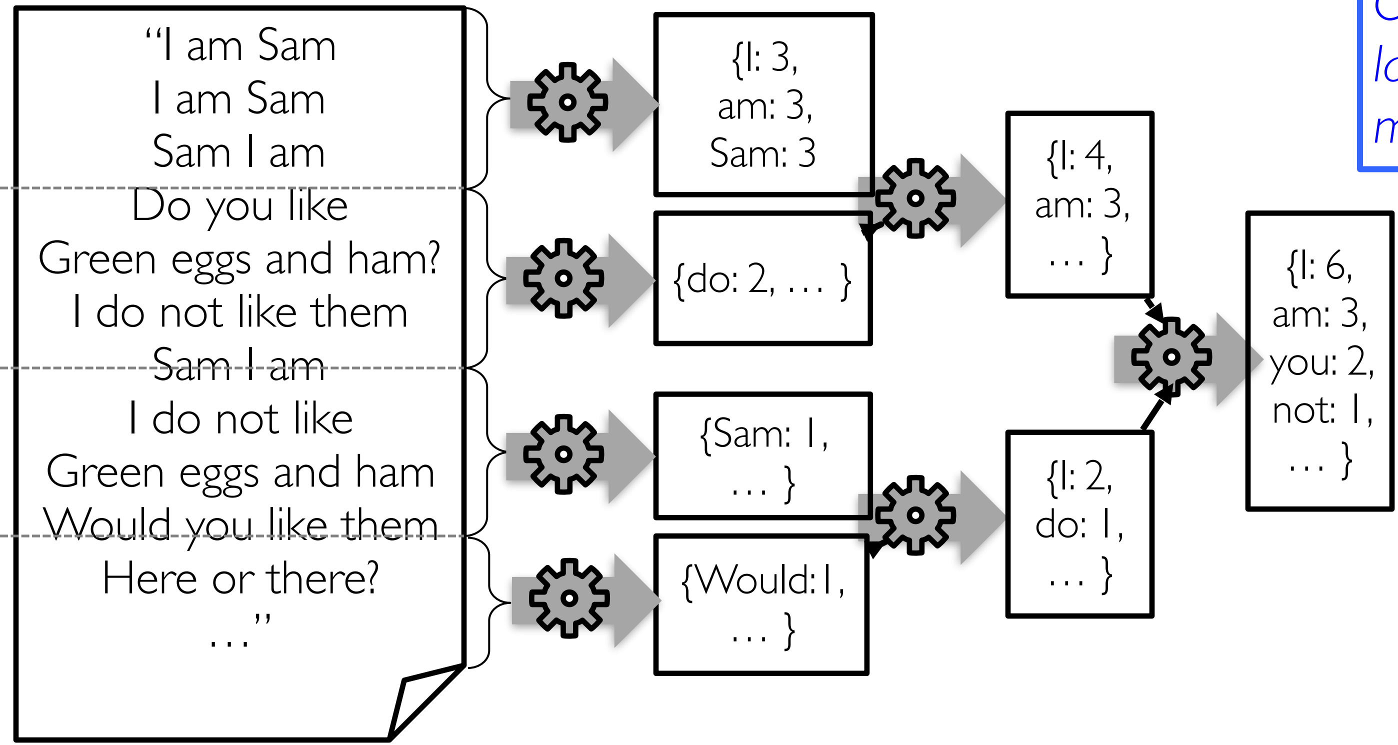
What's the problem with this approach?

What if the Document is Really Big?



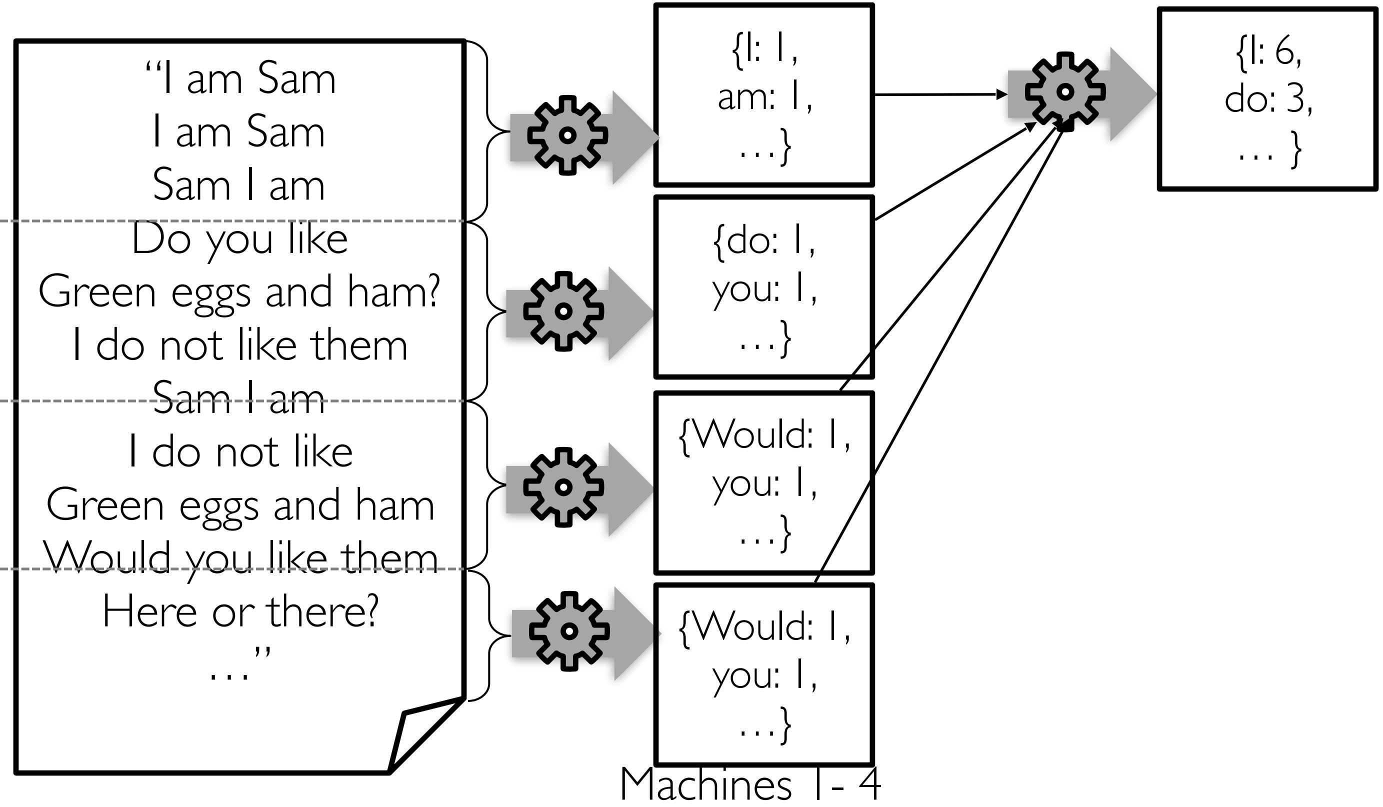
Results have to fit on one machine

What if the Document is Really Big?



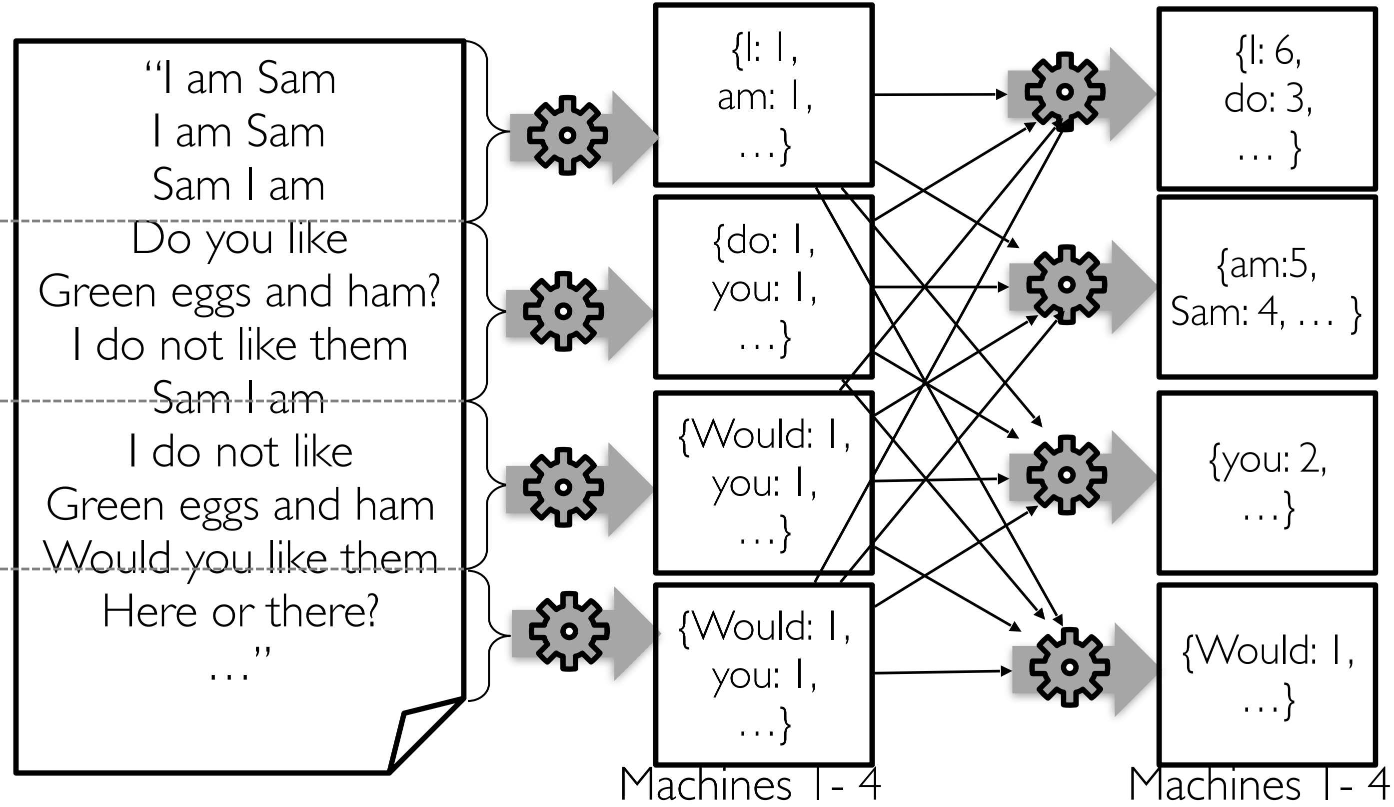
Can add aggregation layers but results still must fit on one machine

What if the Document is Really Big?



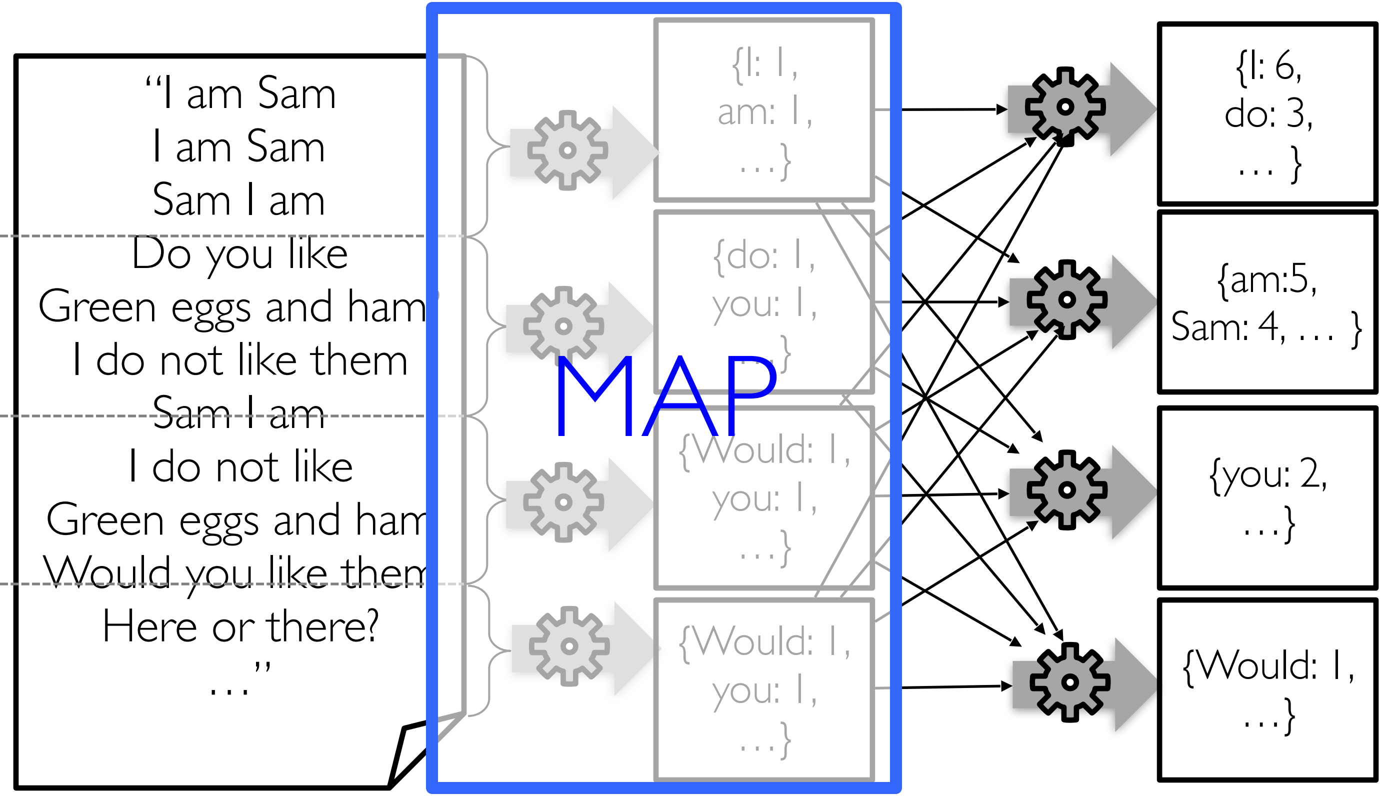
Use Divide and Conquer!!

What if the Document is Really Big?



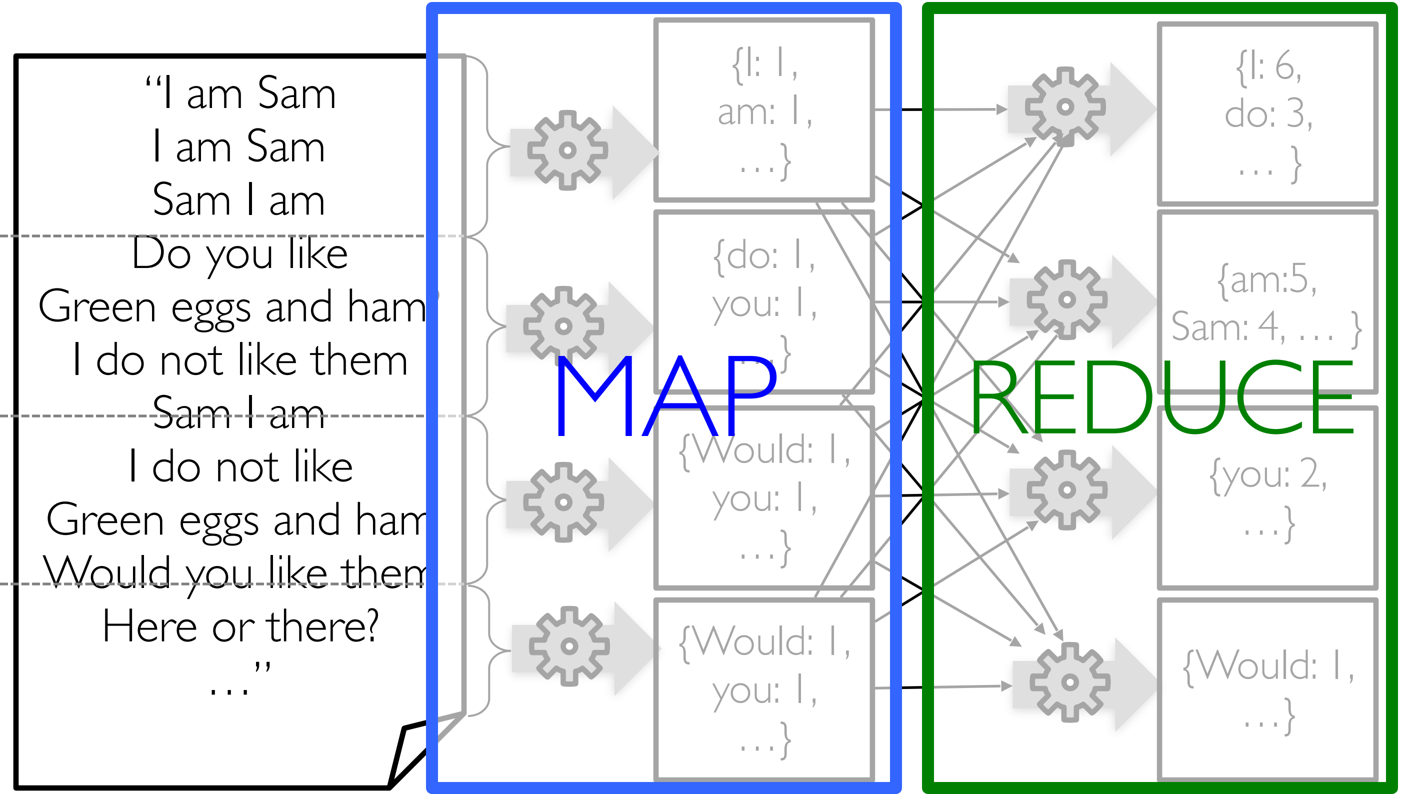
Use Divide and Conquer!!

What if the Document is Really Big?



Use Divide and Conquer!!

What if the Document is Really Big?

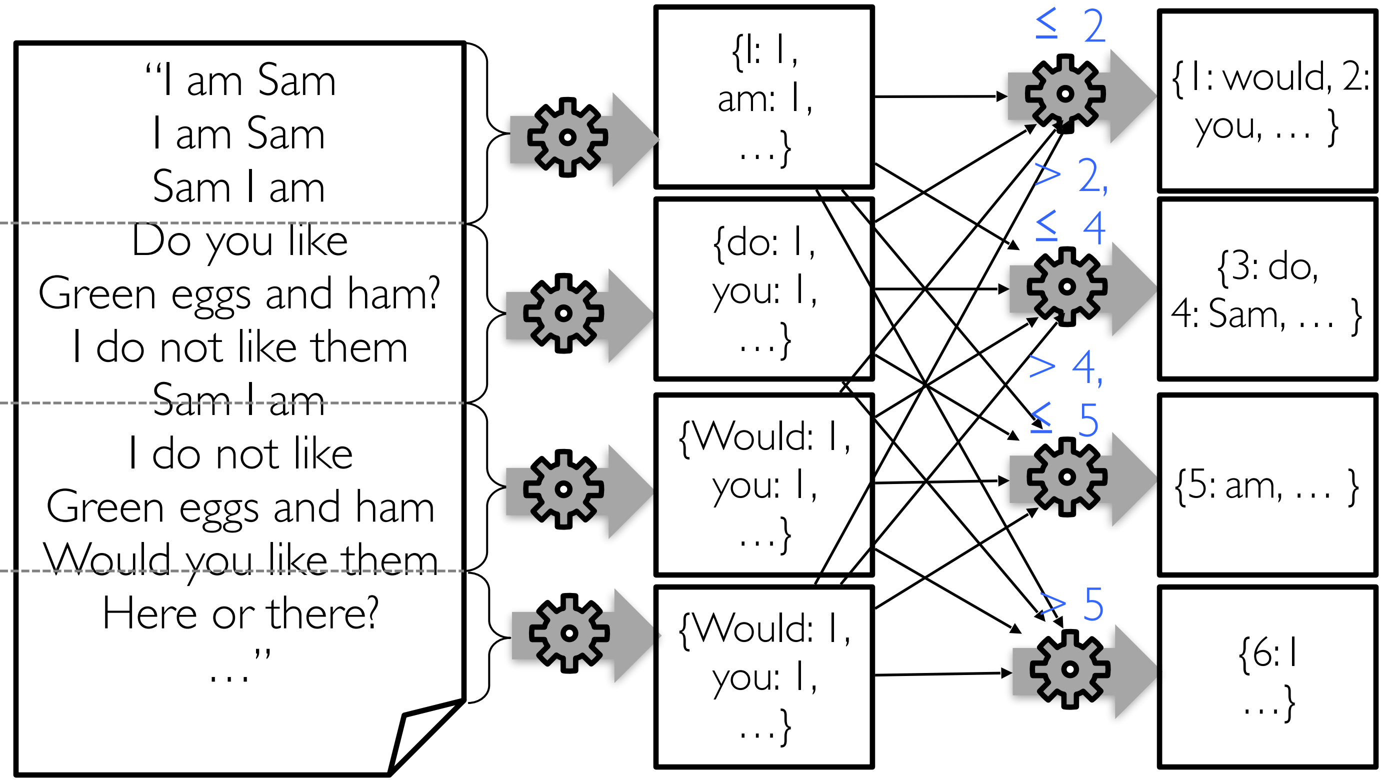


Google
Map Reduce 2004

Apache Hadoop

http://research.google.com/archive/mapreduce.html

Map Reduce for Sorting



"What word is used most?"

Review: Python Spark (pySpark)

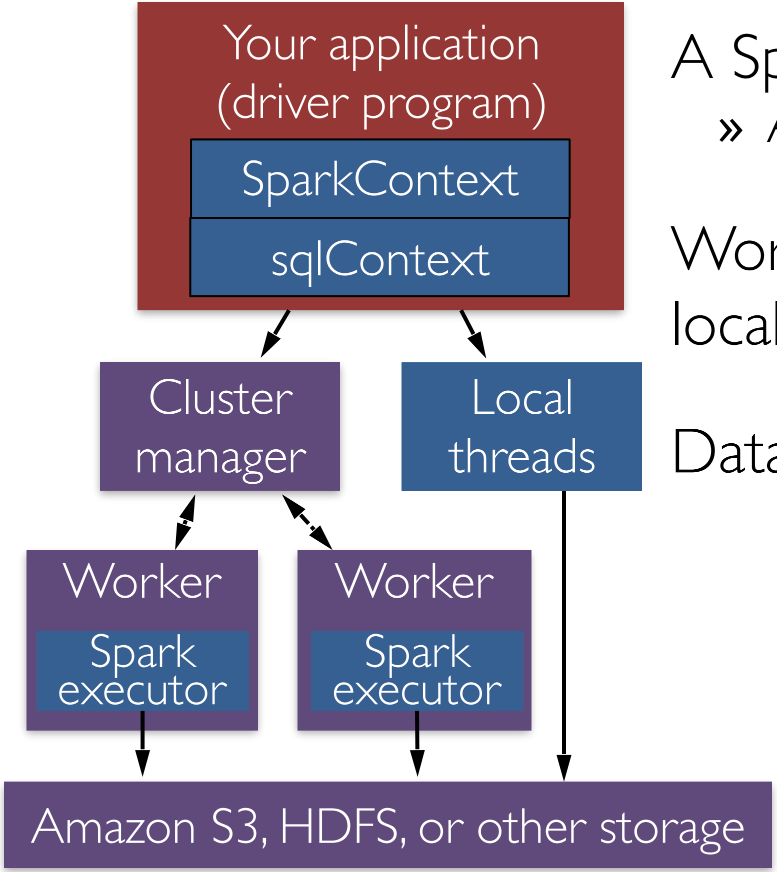
We are using the Python programming interface to Spark (pySpark)

pySpark provides an easy-to-use programming abstraction and parallel runtime:

» "Here's an operation, run it on all of the data"

DataFrames are the key concept

Review: Spark Driver and Workers



A Spark program is two programs:

» A **driver program** and a **workers program**

Worker programs run on cluster nodes or in local threads

DataFrames are distributed across workers

Review: Spark and SQL Contexts

A Spark program first creates a `SparkContext` object

- » `SparkContext` tells Spark how and where to access a cluster
- » pySpark shell, Databricks CE automatically create `SparkContext`
- » iPython and programs must create a new `SparkContext`

The program next creates a `sqlContext` object

Use `sqlContext` to create DataFrames

Review: DataFrames

The primary abstraction in Spark

» **Immutable once constructed**

» Track lineage information to efficiently recompute lost data

» Enable operations on collection of elements in parallel

You construct DataFrames

» by *parallelizing* existing Python collections (lists)

» by *transforming* an existing Spark or pandas DFs

» from *files* in HDFS or any other storage system

Review: DataFrames

Two types of operations: ***transformations*** and ***actions***

Transformations are lazy (*not computed immediately*)

Transformed DF is executed when action runs on it

Persist (cache) DFs in memory or disk

Resilient Distributed Datasets

Untyped Spark abstraction underneath DataFrames:

» **Immutable once constructed**
» Track lineage information to efficiently recompute lost data
» Enable operations on collection of elements in parallel

You construct RDDs

» by *parallelizing* existing Python collections (lists)
» by *transforming* an existing RDDs or DataFrame
» from *files* in HDFS or any other storage system

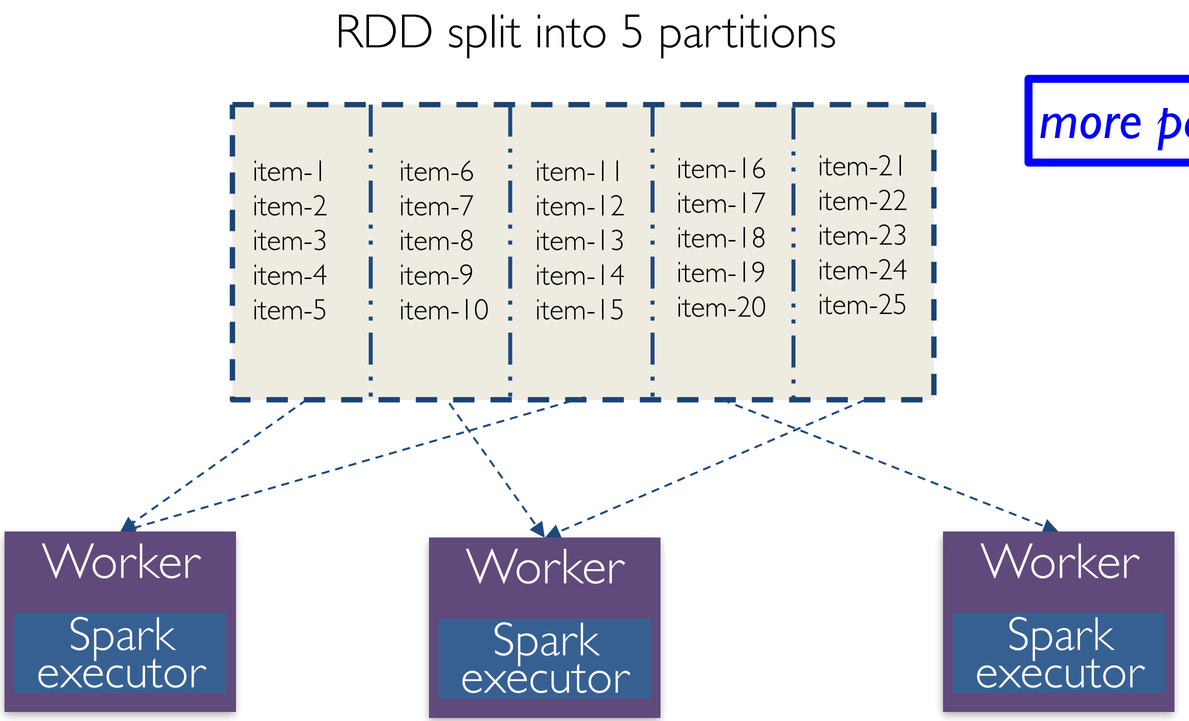
http://spark.apache.org/docs/latest/api/python/pyspark.html

RDDs

Programmer specifies number of partitions for an RDD

(Default value used if unspecified)

more partitions = more parallelism



RDDs

Two types of operations: *transformations* and *actions*

Transformations are lazy (*not computed immediately*)

Transformed RDD is executed when action runs on it

Persist (cache) RDDs in memory or disk

When to Use DataFrames?

Need high-level transformations and actions, and want high-level control over your dataset

Have typed (structured or semi-structured) data

You want DataFrame optimization and performance benefits

- » Catalyst Optimization Engine
- » Project Tungsten off-heap memory management

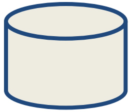
When to Use RDDs?

Need low-level transformations and actions, and want low-level control over your dataset

Have unstructured or schema-less data (e.g., media or text streams)

Want to manipulate your data with functional programming constructs other than domain specific expressions

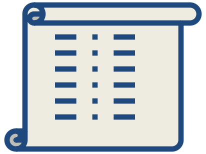
Working with RDDs

Create an RDD from a data source:  <list>

Apply transformations to an RDD: map filter

Apply actions to an RDD: collect count

`collect` action causes **`parallelize`**, **`filter`**, and **`map`** transforms to be executed



Result

Creating an RDD

Create RDDs from Python collections (lists)

No computation occurs with **`sc.parallelize()`**

- Spark only records how to create the RDD with four partitions

```
>>> data = [1, 2, 3, 4, 5]
>>> data
[1, 2, 3, 4, 5]

>>> rDD = sc.parallelize(data, 4)
>>> rDD
ParallelCollectionRDD[0] at parallelize at PythonRDD.scala:229
```

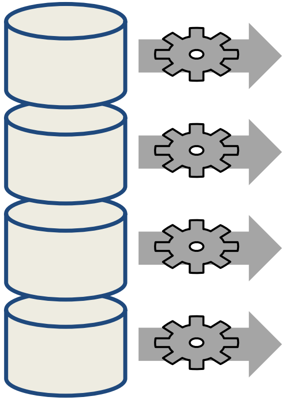
Creating RDDs

From HDFS, text files, [Hypertable](), [Amazon S3](), [Apache Hbase](), SequenceFiles, any other Hadoop `InputFormat`, and directory or glob wildcard: `/data/201404*`

```
>>> distFile = sc.textFile("README.md", 4)
>>> distFile
MappedRDD[2] at textFile at
   NativeMethodAccessorImpl.java:-2
```

Creating an RDD from a File

```
distFile = sc.textFile("...", 4)
```



RDD distributed in 4 partitions

Elements are lines of input

Lazy evaluation means
no execution happens now

Spark Transformations

Create new datasets from an existing one

Use ***lazy evaluation***: results not computed right away – instead Spark remembers set of transformations applied to base dataset

- » Spark optimizes the required calculations
- » Spark recovers from failures and slow workers

Think of this as a recipe for creating result

Some 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
`distinct([`*numTasks*`]))`	return a new dataset that contains the distinct elements of the source dataset
`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)

Transformations

```
>>> rdd = sc.parallelize([1, 2, 3, 4])
>>> rdd.map(lambda x: x * 2)
RDD: [1, 2, 3, 4] → [2, 4, 6, 8]

>>> rdd.filter(lambda x: x % 2 == 0)
RDD: [1, 2, 3, 4] → [2, 4]

>>> rdd2 = sc.parallelize([1, 4, 2, 2, 3])
>>> rdd2.distinct()
RDD: [1, 4, 2, 2, 3] → [1, 4, 2, 3]
```

Function literals (green) are closures automatically passed to workers

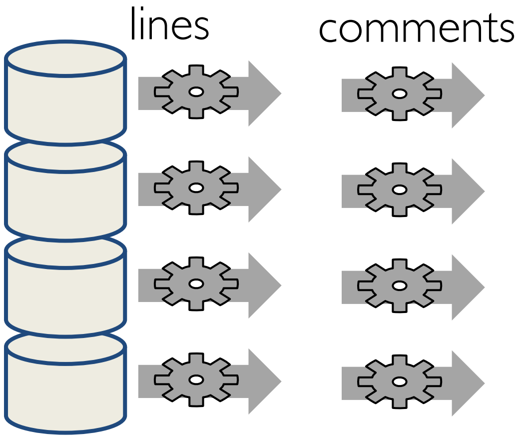
Transformations

```
>>> rdd = sc.parallelize([1, 2, 3])
>>> rdd.Map(lambda x: [x, x+5])
RDD: [1, 2, 3] → [[1, 6], [2, 7], [3, 8]]

>>> rdd.flatMap(lambda x: [x, x+5])
RDD: [1, 2, 3] → [1, 6, 2, 7, 3, 8]
```

Function literals (green) are closures automatically passed to workers

Transforming an RDD

```
lines = sc.textFile("...", 4)

comments = lines.filter(isComment)
```



Lazy evaluation means nothing executes – Spark saves recipe for transforming source

Spark Actions

Cause Spark to execute recipe to transform source

Mechanism for getting results out of Spark

Some Actions

Action	Description
`reduce(`*func*`)`	aggregate dataset's elements using function *func*. *func* takes two arguments and returns one, and is commutative and associative so that it can be computed correctly in parallel
`take(`*n*`)`	return an array with the first *n* elements
`collect()`	return all the elements as an array WARNING: make sure will fit in driver program
`takeOrdered(`*n*`, `*key=func*`)`	return n elements ordered in ascending order or as specified by the optional key function

Getting Data Out of RDDs

```
>>> rdd = sc.parallelize([1, 2, 3])
>>> rdd.reduce(lambda a, b: a * b)
Value: 6

>>> rdd.take(2)
Value: [1,2] # as list

>>> rdd.collect()
Value: [1,2,3] # as list
```

Getting Data Out of RDDs

```
>>> rdd = sc.parallelize([5,3,1,2])
>>> rdd.takeOrdered(3, lambda s: -1 * s)
Value: [5,3,2] # as list
```

Spark Key-Value RDDs

Similar to Map Reduce, Spark supports Key-Value pairs

Each element of a Pair RDD is a pair tuple

```
>>> rdd = sc.parallelize([(1, 2), (3, 4)])
RDD: [(1, 2), (3, 4)]
```

Some Key-Value Transformations

Key-Value Transformation	Description
`reduceByKey(`*func*`)`	return a new distributed dataset of (K,V) pairs where the values for each key are aggregated using the given reduce function *func*, which must be of type (V,V) ➔ V
`sortByKey()`	return a new dataset (K,V) pairs sorted by keys in ascending order
`groupByKey()`	return a new dataset of (K, Iterable<V>) pairs

Key-Value Transformations

```
>>> rdd = sc.parallelize([(1,2), (3,4), (3,6)])
>>> rdd.reduceByKey(lambda a, b: a + b)
RDD: [(1,2), (3,4), (3,6)] → [(1,2), (3,10)]

>>> rdd2 = sc.parallelize([(1,'a'), (2,'c'), (1,'b')])
>>> rdd2.sortByKey()
RDD: [(1,'a'), (2,'c'), (1,'b')] →
             [(1,'a'), (1,'b'), (2,'c')]
```

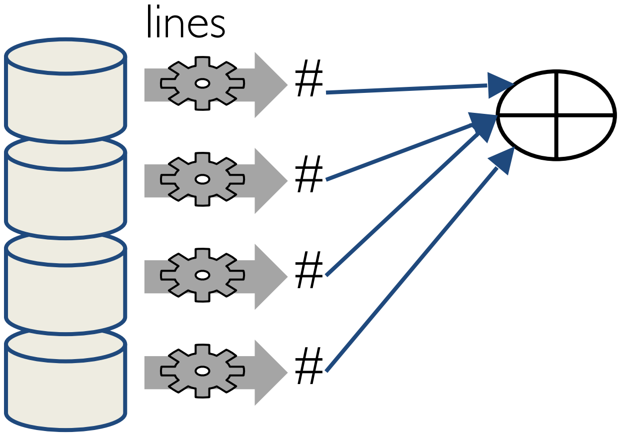
Key-Value Transformations

```
>>> rdd2 = sc.parallelize([(1,'a'), (2,'c'), (1,'b')])
>>> rdd2.groupByKey()
RDD: [(1,'a'), (1,'b'), (2,'c')] →
               [(1,['a','b']), (2,['c'])]
```

Be careful using **groupByKey()** as it can cause a lot of data movement across the network and create large Iterables at workers

Spark Programming Model

```
lines = sc.textFile("...", 4)

print lines.count()
```

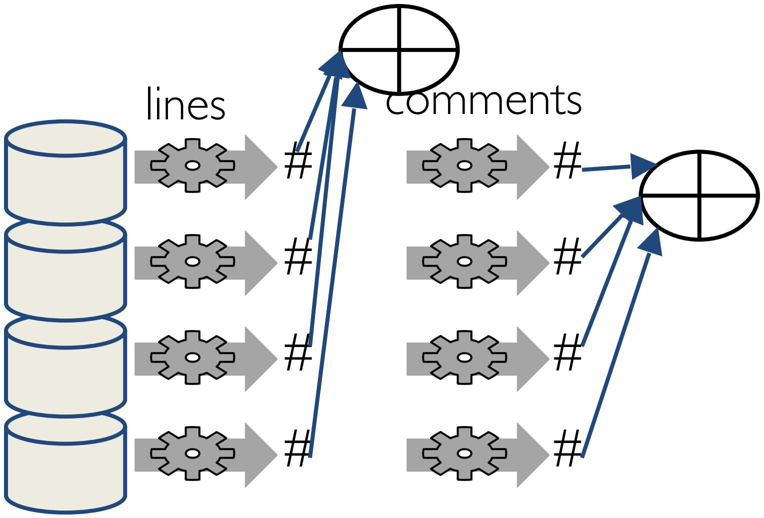


`count()` causes Spark to:

- read data
- sum within partitions
- combine sums in driver

Spark Programming Model

```
lines = sc.textFile("...", 4)
comments = lines.filter(isComment)
print lines.count(), comments.count()
```

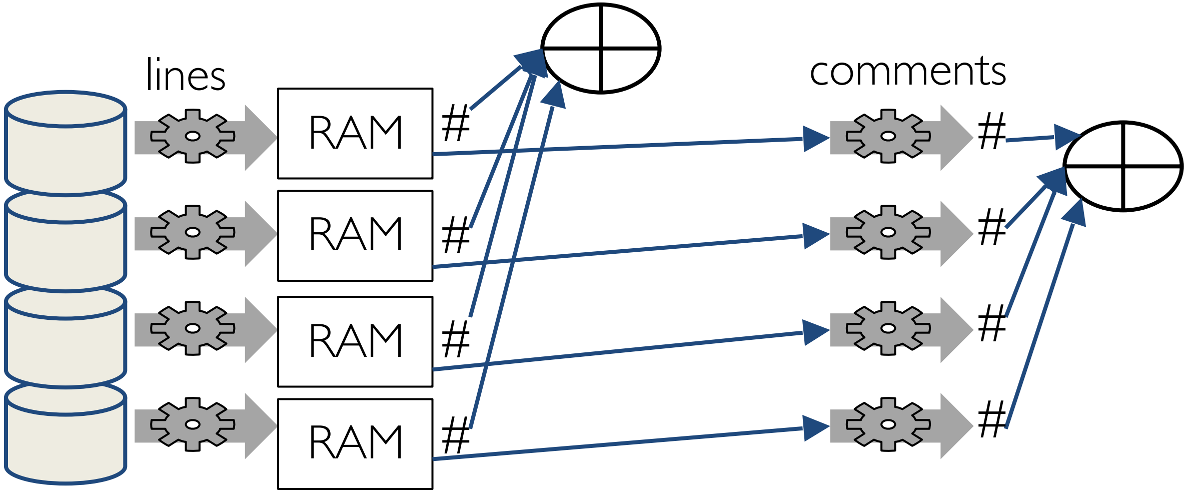


Spark recomputes **lines**:

- read data (again)
- sum within partitions
- combine sums in driver

Caching RDDs

```
lines = sc.textFile("...", 4)
lines.cache() # save, don't recompute!
comments = lines.filter(isComment)
print lines.count(),comments.count()
```



Spark Program Lifecycle with RDDs

1. Create RDDs from external data or <u>parallelize</u> a collection in your driver program

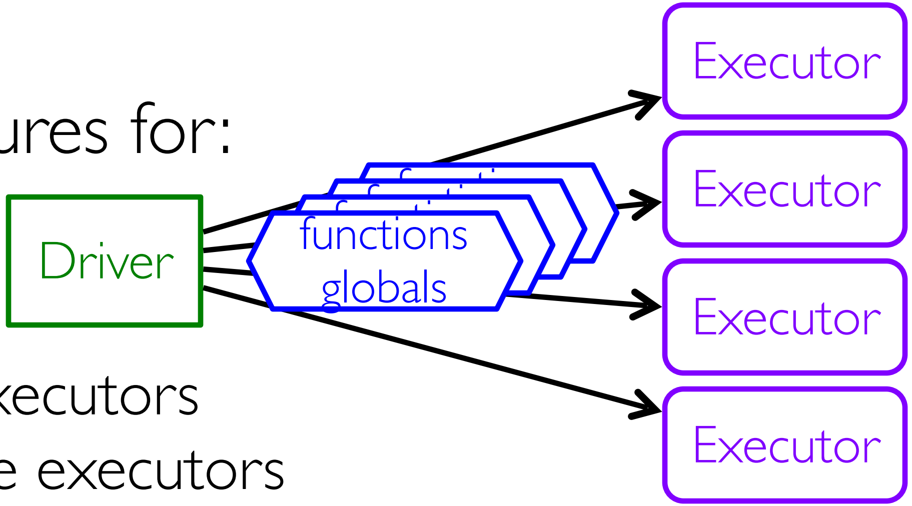
2. Lazily <u>transform</u> them into new RDDs

3. **`cache()`** some RDDs for reuse

4. Perform <u>actions</u> to execute parallel computation and produce results

pySpark Closures

Spark automatically creates closures for:



- » Functions that run on RDDs at executors
- » Any global variables used by those executors

One closure per executor

- » Sent for ***every*** task
- » No communication between executors
- » Changes to global variables at executors are not sent to driver

Consider These Use Cases

Iterative or single jobs with large global variables

- » Sending large read-only lookup table to executors
- » Sending large feature vector in a ML algorithm to executors

Counting events that occur during job execution

- » How many input lines were blank?
- » How many input records were corrupt?

Consider These Use Cases

Iterative or single jobs with large global variables

» Sending large read-only lookup table to executors

» Sending large feature vector in a ML algorithm to executors

Counting events that occur during job execution

» How many input lines were blank?

» How many input records were corrupt?

Problems:

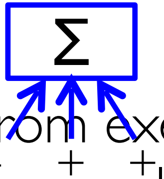
- Closures are (re-)sent with ***every*** job
- Inefficient to send large data to each worker
- Closures are one way: driver → worker

pySpark Shared Variables

Broadcast Variables

» Efficiently send large, ***read-only*** value to all executors
» Saved at workers for use in one or more Spark operations
» Like sending a large, read-only lookup table to all the nodes

Accumulators



» Aggregate values from executors back to driver
» Only driver can access value of accumulator
» For tasks, accumulators are write-only
» Use to count errors seen in RDD across executors

Broadcast Variables

Keep ***read-only*** variable cached on executors

» Ship to each worker only once instead of with each task

Example: efficiently give every executor a large dataset

Usually distributed using efficient broadcast algorithms

```
At the driver:
>>> broadcastVar = sc.broadcast([1, 2, 3])

At an executor (in code passed via a closure)
>>> broadcastVar.value
[1, 2, 3]
```

Broadcast Variables Example

Country code lookup for HAM radio call signs

```python
# Lookup the locations of the call signs on the
# RDD contactCounts. We load a list of call sign
# prefixes to country code to support this lookup
signPrefixes = loadCallSignTable()

def processSignCount(sign_count, signPrefixes):
    country = lookupCountry(sign_count[0], signPrefixes)
    count = sign_count[1]
    return (country, count)

countryContactCounts = (contactCounts
                        .map(processSignCount)
                        .reduceByKey((lambda x, y: x+ y)))
```

Expensive to send large table
(Re-)sent for every processed file

From: http://shop.oreilly.com/product/0636920028512.do

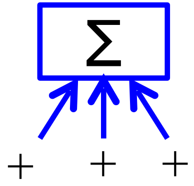
Broadcast Variables Example

Country code lookup for HAM radio call signs

```python
# Lookup the locations of the call signs on the
# RDD contactCounts. We load a list of call sign
# prefixes to country code to support this lookup
signPrefixes = sc.broadcast(loadCallSignTable())

def processSignCount(sign_count, signPrefixes):
    country = lookupCountry(sign_count[0], signPrefixes.value)
    count = sign_count[1]
    return (country, count)

countryContactCounts = (contactCounts
                        .map(processSignCount)
                        .reduceByKey((lambda x, y: x+ y)))
```

Efficiently sent once to executors

From: http://shop.oreilly.com/product/0636920028512.do



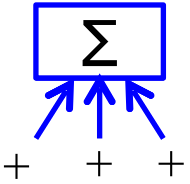
Accumulators

Variables that can only be "added" to by associative op

Used to efficiently implement parallel counters and sums

Only driver can read an accumulator's value, not tasks

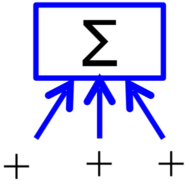
```
>>> accum = sc.accumulator(0)
>>> rdd = sc.parallelize([1, 2, 3, 4])
>>> def f(x):
>>>   global accum
>>>   accum += x

>>> rdd.foreach(f)
>>> accum.value
Value: 10
```



Accumulators Example

Counting empty lines

```
file = sc.textFile(inputFile)
# Create Accumulator[Int] initialized to 0
blankLines = sc.accumulator(0)

def extractCallSigns(line):
    global blankLines # Make the global variable accessible
    if (line == ""):
        blankLines += 1
    return line.split(" ")

callSigns = file.flatMap(extractCallSigns)
print "Blank lines: %d" % blankLines.value
```



Accumulators

Tasks at executors cannot access accumulator's values

Tasks see accumulators as write-only variables

Accumulators can be used in actions or transformations:

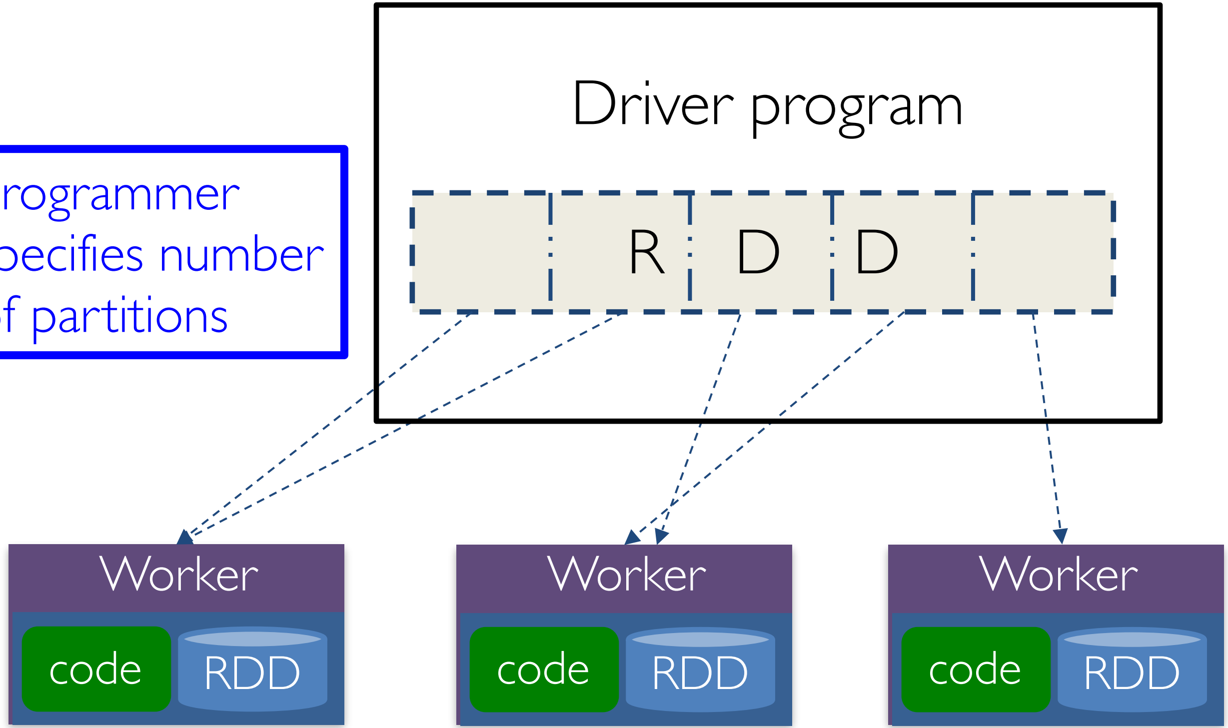
- » Actions: each task's update to accumulator is ***applied only once***
- » Transformations: ***no guarantees*** (use only for debugging)

Types: integers, double, long, float

- » See lab for example of custom type

Summary

Programmer specifies number of partitions



Spark automatically pushes closures to Spark executors at workers

Master parameter specifies number of executors

Spark Online Documentation

https://spark.apache.org/docs/latest/

Spark 1.6.1 Overview Programming Guides API Docs Deploying More

Spark Overview

Apache Spark is a fast and general-purpose cluster computing system. It provides high-level APIs in Java, Scala, Python and R, and an optimized engine that supports general execution graphs. It also supports a rich set of higher-level tools including Spark SQL for SQL and structured data processing, MLlib for machine learning, GraphX for graph processing, and Spark Streaming.

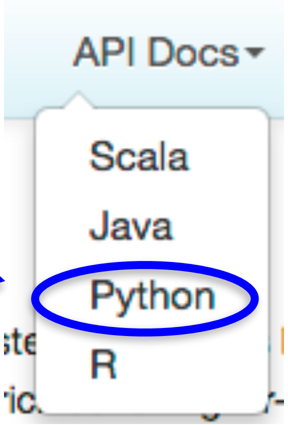
Downloading

Get Spark from the downloads page of the project website. This documentation is for Spark version 1.6.1. Spark uses Hadoop's client libraries for HDFS and YARN. Downloads are pre-packaged for a handful of popular Hadoop versions. Users can also download a "Hadoop free" binary and run Spark with any Hadoop version by augmenting Spark's classpath.

If you'd like to build Spark from source, visit Building Spark.

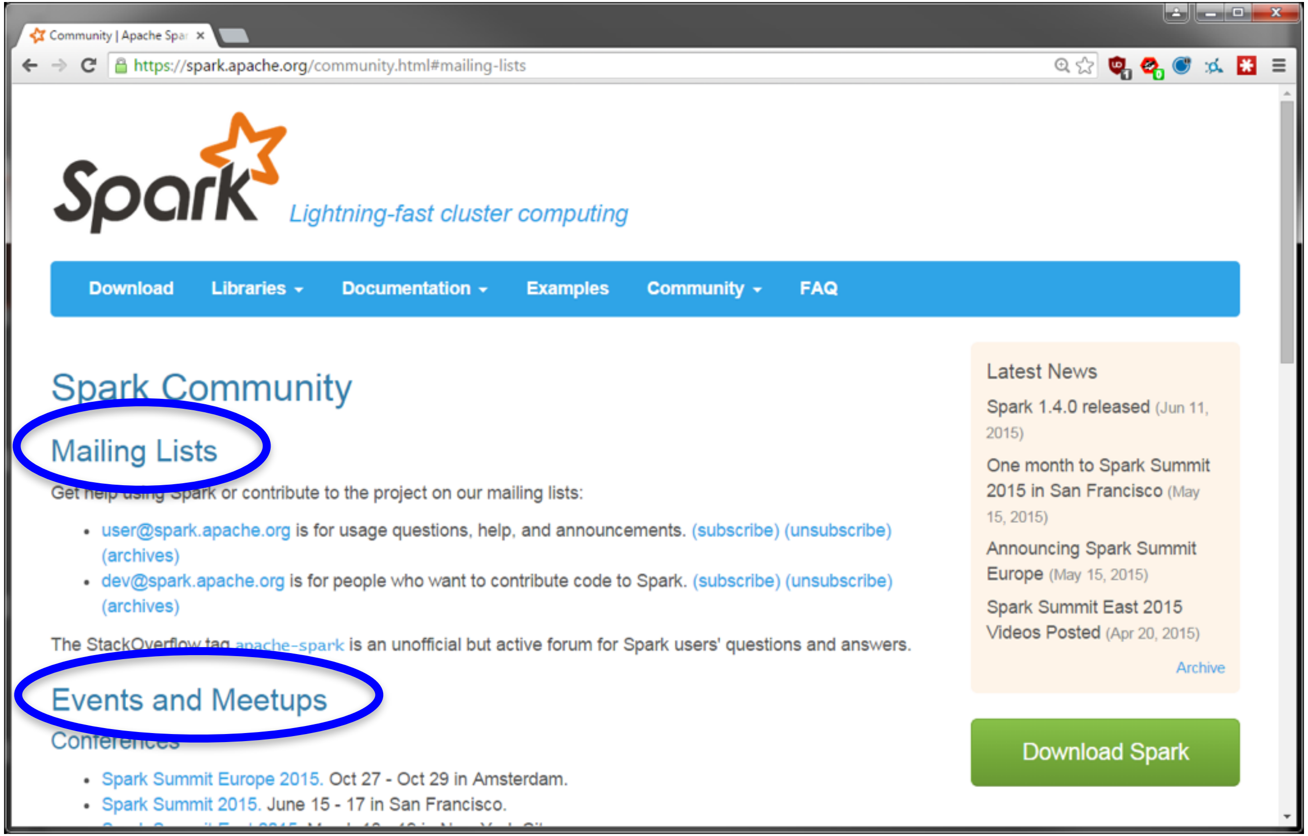
Spark runs on both Windows and UNIX-like systems (e.g. Linux, Mac OS). It's easy to run locally on one machine — all you need is to have `java` installed on your system `PATH`, or the `JAVA_HOME` environment variable pointing to a Java installation.

Spark runs on Java 7+, Python 2.6+ and R 3.1+. For the Scala API, Spark 1.6.1 uses Scala 2.10. You will need to use a compatible Scala version (2.10.x).



Community

http://spark.apache.org/community.html



Spark Meetups

Apache Spark

Find out what's happening in Apache Spark Meetup groups around the world and start meeting up with the ones near you.

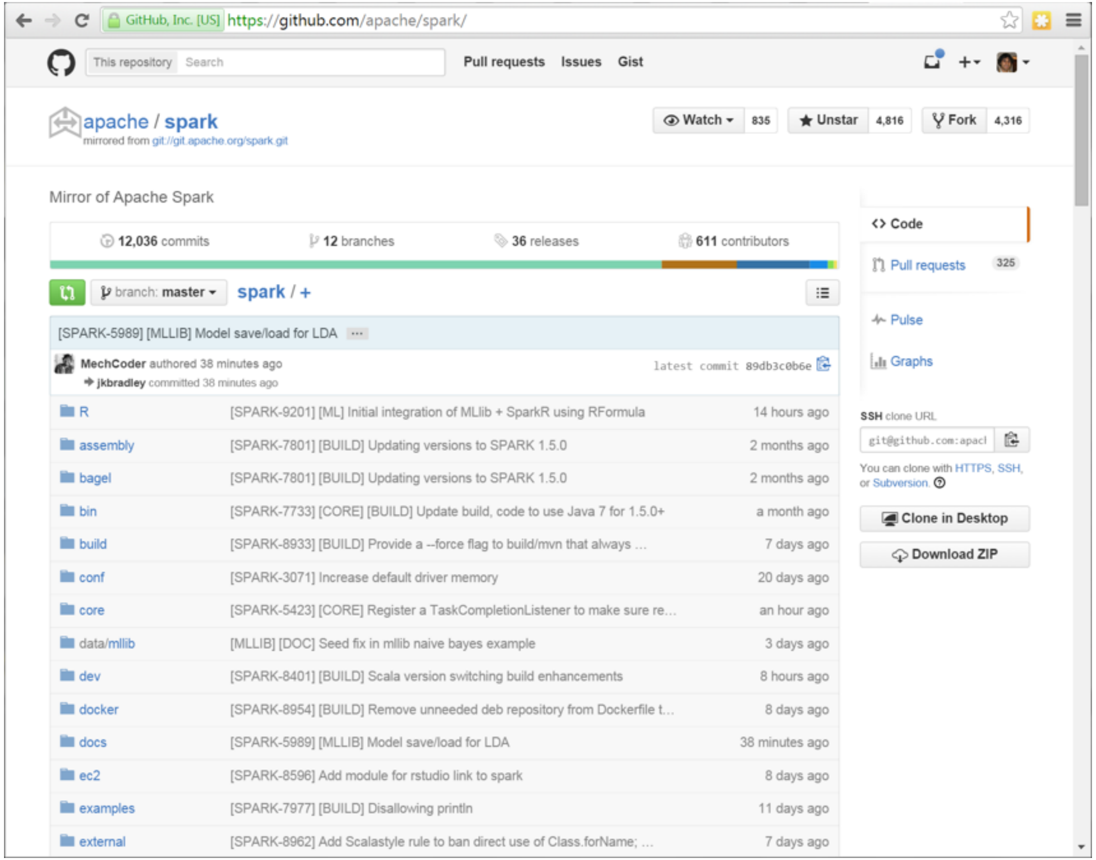
186,279 members | 421 Meetups

Join Apache Spark Meetups

http://spark.meetup.com/

Related topics: Big Data · Hadoop · Machine Learning · Data Analytics · Big Data Analytics · Data Science · Apache Kafka · MapReduce · Data Mining · Scala

Spark Source Code



https://github.com/apache/spark/

Hint: For detailed explanations, check out comments in code

Spark Research Papers

Spark: Cluster Computing with Working Sets

Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, Ion Stoica
University of California, Berkeley

Abstract

MapReduce and its variants have been highly successful in implementing large-scale data-intensive applications on commodity clusters. However, most of these systems are built around an acyclic data flow model that is not suitable for other popular applications. This paper focuses on one such class of applications: those that reuse a working set of data across multiple parallel operations. This includes many iterative machine learning algorithms, as well as interactive data analysis tools. We propose a new framework called Spark that supports these applications while retaining the scalability and fault tolerance of MapReduce. To achieve these goals, Spark introduces an abstraction called resilient distributed datasets (RDDs). An RDD is a read-only collection of objects partitioned across a set of machines that can be rebuilt if a partition is lost. Spark can outperform Hadoop by 10x in iterative machine learning jobs, and can be used to interactively query a 39 GB dataset with sub-second response time.

1 Introduction

A new model of cluster computing has become widely popular, in which data-parallel computations are executed on clusters of unreliable machines by systems that automatically provide locality-aware scheduling, fault tolerance, and load balancing. MapReduce [11] pioneered this model, while systems like Dryad [17] and Map-Reduce-Merge [24] generalized the types of data flows supported. These systems achieve their scalability and fault tolerance by providing a programming model where the user creates acyclic data flow graphs to pass input data through a set of operators. This allows the underlying system to manage scheduling and to react to faults without user intervention.

While this data flow programming model is useful for a large class of applications, there are applications that cannot be expressed efficiently as acyclic data flows. In this paper, we focus on one such class of applications: those that reuse a *working set* of data across multiple parallel operations. This includes two use cases where we have seen Hadoop users report that MapReduce is deficient:

- **Iterative jobs**: Many common machine learning algorithms apply a function repeatedly to the same dataset to optimize a parameter (e.g., through gradient descent). While each iteration can be expressed as a MapReduce/Dryad job, each job must reload the data from disk, incurring a significant performance penalty.
- **Interactive analytics**: Hadoop is often used to run ad-hoc exploratory queries on large datasets, through SQL interfaces such as Pig [21] and Hive [1]. Ideally, a user would be able to load a dataset of interest into memory across a number of machines and query it repeatedly. However, with Hadoop, each query incurs significant latency (tens of seconds) because it runs as a separate MapReduce job and reads data from disk.

This paper presents a new cluster computing framework called Spark, which supports applications with working sets while providing similar scalability and fault tolerance properties to MapReduce.

The main abstraction in Spark is that of a *resilient distributed dataset* (RDD), which represents a read-only collection of objects partitioned across a set of machines that can be rebuilt if a partition is lost. Users can explicitly cache an RDD in memory across machines and reuse it in multiple MapReduce-like *parallel operations*. RDDs achieve fault tolerance through a notion of *lineage*: if a partition of an RDD is lost, the RDD has enough information about how it was derived from other RDDs to be able to rebuild just that partition. Although RDDs are not a general shared memory abstraction, they represent a sweet-spot between expressivity on the one hand and scalability and reliability on the other hand, and we have found them well-suited for a variety of applications.

Spark is implemented in Scala [5], a statically typed high-level programming language for the Java VM, and exposes a functional programming interface similar to DryadLINQ [25]. In addition, Spark can be used interactively from a modified version of the Scala interpreter, which allows the user to define RDDs, functions, variables and classes and use them in parallel operations on a cluster. We believe that Spark is the first system to allow an efficient, general-purpose programming language to be used interactively to process large datasets on a cluster.

Although our implementation of Spark is still a prototype, early experience with the system is encouraging. We show that Spark can outperform Hadoop by 10x in iterative machine learning workloads and can be used interactively to scan a 39 GB dataset with sub-second latency.

This paper is organized as follows. Section 2 describes

Spark: Cluster Computing with Working Sets

http://www.cs.berkeley.edu/~matei/papers/2010/hotcloud_spark.pdf

June 2010

Research Papers

Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing

http://www.cs.berkeley.edu/~matei/papers/2012/nsdi_spark.pdf

April 2012

Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing

Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma,
Murphy McCauley, Michael J. Franklin, Scott Shenker, Ion Stoica
University of California, Berkeley

Abstract

We present Resilient Distributed Datasets (RDDs), a distributed memory abstraction that lets programmers perform in-memory computations on large clusters in a fault-tolerant manner. RDDs are motivated by two types of applications that current computing frameworks handle inefficiently: iterative algorithms and interactive data mining tools. In both cases, keeping data in memory can improve performance by an order of magnitude. To achieve fault tolerance efficiently, RDDs provide a restricted form of shared memory, based on coarse-grained transformations rather than fine-grained updates to shared state. However, we show that RDDs are expressive enough to capture a wide class of computations, including recent specialized programming models for iterative jobs, such as Pregel, and new applications that these models do not capture. We have implemented RDDs in a system called Spark, which we evaluate through a variety of user applications and benchmarks.

1 Introduction

Cluster computing frameworks like MapReduce [10] and Dryad [19] have been widely adopted for large-scale data analytics. These systems let users write parallel computations using a set of high-level operators, without having to worry about work distribution and fault tolerance.

Although current frameworks provide numerous abstractions for accessing a cluster's computational resources, they lack abstractions for leveraging distributed memory. This makes them inefficient for an important class of emerging applications: those that *reuse* intermediate results across multiple computations. Data reuse is common in many *iterative* machine learning and graph algorithms, including PageRank, K-means clustering, and logistic regression. Another compelling use case is *interactive* data mining, where a user runs multiple ad-hoc queries on the same subset of the data. Unfortunately, in most current frameworks, the only way to reuse data between computations (*e.g.*, between two MapReduce jobs) is to write it to an external stable storage system, *e.g.*, a distributed file system. This incurs substantial overheads due to data replication, disk I/O, and serialization, which can dominate application execution times.

Recognizing this problem, researchers have developed specialized frameworks for some applications that require data reuse. For example, Pregel [22] is a system for iterative graph computations that keeps intermediate data in memory, while HaLoop [7] offers an iterative MapReduce interface. However, these frameworks only support specific computation patterns (*e.g.*, looping a series of MapReduce steps), and perform data sharing implicitly for these patterns. They do not provide abstractions for more general reuse, *e.g.*, to let a user load several datasets into memory and run ad-hoc queries across them.

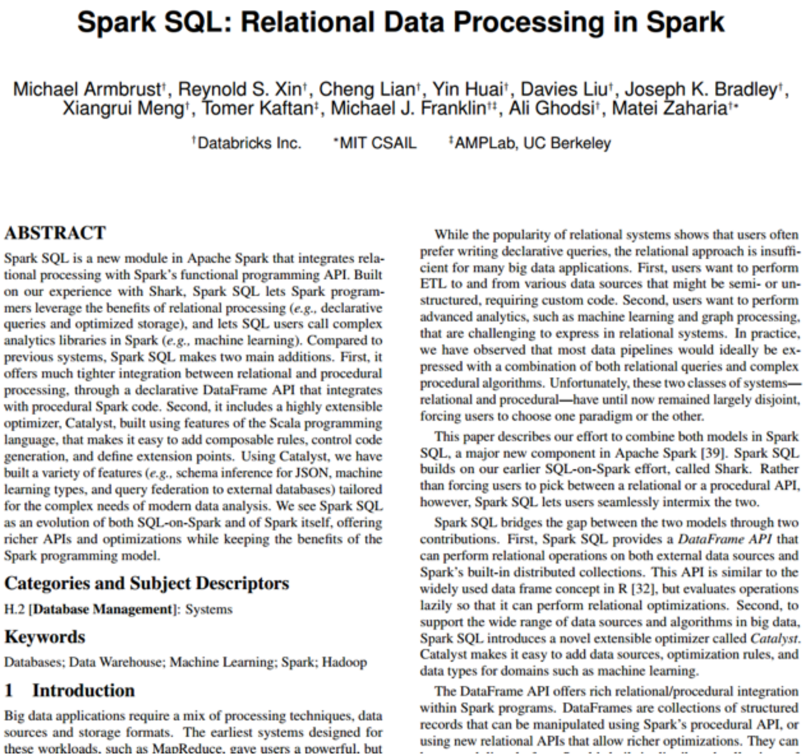
In this paper, we propose a new abstraction called *resilient distributed datasets (RDDs)* that enables efficient data reuse in a broad range of applications. RDDs are fault-tolerant, parallel data structures that let users explicitly persist intermediate results in memory, control their partitioning to optimize data placement, and manipulate them using a rich set of operators.

The main challenge in designing RDDs is defining a programming interface that can provide fault tolerance *efficiently*. Existing abstractions for in-memory storage on clusters, such as distributed shared memory [24], key-value stores [25], databases, and Piccolo [27], offer an interface based on fine-grained updates to mutable state (*e.g.*, cells in a table). With this interface, the only ways to provide fault tolerance are to replicate the data across machines or to log updates across machines. Both approaches are expensive for data-intensive workloads, as they require copying large amounts of data over the cluster network, whose bandwidth is far lower than that of RAM, and they incur substantial storage overhead.

In contrast to these systems, RDDs provide an interface based on *coarse-grained* transformations (*e.g.*, map, filter and join) that apply the same operation to many data items. This allows them to efficiently provide fault tolerance by logging the transformations used to build a dataset (its *lineage*) rather than the actual data.[1] If a partition of an RDD is lost, the RDD has enough information about how it was derived from other RDDs to recompute

[1] Checkpointing the data in some RDDs may be useful when a lineage chain grows large, however, and we discuss how to do it in §5.4.

SQL



Spark SQL: Relational Data Processing in Spark

Michael Armbrust[†], Reynold S. Xin[†], Cheng Lian[†], Yin Huai[†], Davies Liu[†], Joseph K. Bradley[†], Xiangrui Meng[†], Tomer Kaftan[‡], Michael J. Franklin[†‡], Ali Ghodsi[†], Matei Zaharia[†*]

[†]Databricks Inc. [*]MIT CSAIL [‡]AMPLab, UC Berkeley

ABSTRACT

Spark SQL is a new module in Apache Spark that integrates relational processing with Spark's functional programming API. Built on our experience with Shark, Spark SQL lets Spark programmers leverage the benefits of relational processing (*e.g.*, declarative queries and optimized storage), and lets SQL users call complex analytics libraries in Spark (*e.g.*, machine learning). Compared to previous systems, Spark SQL makes two main additions. First, it offers much tighter integration between relational and procedural processing, through a declarative DataFrame API that integrates with procedural Spark code. Second, it includes a highly extensible optimizer, Catalyst, built using features of the Scala programming language, that makes it easy to add composable rules, control code generation, and define extension points. Using Catalyst, we have built a variety of features (*e.g.*, schema inference for JSON, machine learning types, and query federation to external databases) tailored for the complex needs of modern data analysis. We see Spark SQL as an evolution of both SQL-on-Spark and of Spark itself, offering richer APIs and optimizations while keeping the benefits of the Spark programming model.

Categories and Subject Descriptors

H.2 [**Database Management**]: Systems

Keywords

Databases; Data Warehouse; Machine Learning; Spark; Hadoop

1 Introduction

Big data applications require a mix of processing techniques, data sources and storage formats. The earliest systems designed for these workloads, such as MapReduce, gave users a powerful, but

While the popularity of relational systems shows that users often prefer writing declarative queries, the relational approach is insufficient for many big data applications. First, users want to perform ETL to and from various data sources that might be semi- or unstructured, requiring custom code. Second, users want to perform advanced analytics, such as machine learning and graph processing, that are challenging to express in relational systems. In practice, we have observed that most data pipelines would ideally be expressed with a combination of both relational queries and complex procedural algorithms. Unfortunately, these two classes of systems—relational and procedural—have until now remained largely disjoint, forcing users to choose one paradigm or the other.

This paper describes our effort to combine both models in Spark SQL, a major new component in Apache Spark [39]. Spark SQL builds on our earlier SQL-on-Spark effort, called Shark. Rather than forcing users to pick between a relational or a procedural API, however, Spark SQL lets users seamlessly intermix the two.

Spark SQL bridges the gap between the two models through two contributions. First, Spark SQL provides a *DataFrame API* that can perform relational operations on both external data sources and Spark's built-in distributed collections. This API is similar to the widely used data frame concept in R [32], but evaluates operations lazily so that it can perform relational optimizations. Second, to support the wide range of data sources and algorithms in big data, Spark SQL introduces a novel extensible optimizer called *Catalyst*. Catalyst makes it easy to add data sources, optimization rules, and data types for domains such as machine learning.

The DataFrame API offers rich relational/procedural integration within Spark programs. DataFrames are collections of structured records that can be manipulated using Spark's procedural API, or using new relational APIs that allow richer optimizations. They can

Spark SQL: Relational Data Processing in Spark

Seemlessly mix SQL queries with Spark programs

June 2015

https://amplab.cs.berkeley.edu/wp-content/uploads/2015/03/SparkSQLSigmod2015.pdf

Historical References

- circa 1979 – Stanford, MIT, CMU, etc.: set/list operations in LISP, Prolog, etc., for parallel processing
  http://www-formal.stanford.edu/jmc/history/lisp/lisp.html
- circa 2004 – Google: *MapReduce: Simplified Data Processing on Large Clusters*
  Jeffrey Dean and Sanjay Ghemawat
  http://research.google.com/archive/mapreduce.html
- circa 2006 – Apache *Hadoop*, originating from the Yahoo!'s Nutch Project
  Doug Cutting
  http://nutch.apache.org/
- circa 2008 – Yahoo!: web scale search indexing
  Hadoop Summit, HUG, etc.
  http://hadoop.apache.org/
- circa 2009 – Amazon AWS: Elastic MapReduce
  Hadoop modified for EC2/S3, plus support for Hive, Pig, Cascading, etc.
  http://aws.amazon.com/elasticmapreduce/

Spark Research Papers

- *Spark: Cluster Computing with Working Sets*
  Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, Ion Stoica
  USENIX HotCloud (2010)
  people.csail.mit.edu/matei/papers/2010/hotcloud_spark.pdf

- *Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing*
  Matei Zaharia, Mosharaf Chowdhury, Tathagata Das,
  Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin,
  Scott Shenker, Ion Stoica
  NSDI (2012)
  usenix.org/system/files/conference/nsdi12/nsdi12-final138.pdf