Architecture of Enterprise Applications 26 Spark

Haopeng Chen

***RE**liable, **IN**telligent and **S**calable Systems Group (**REINS**)*
Shanghai Jiao Tong University
Shanghai, China
http://reins.se.sjtu.edu.cn/~chenhp
e-mail: chen-hp@sjtu.edu.cn

Contents and Objectives

- Spark
  - Basic Concepts
  - Spark Deploying
  - Quick Start
  - RDD
  - Spark SQL
  - Other Issues

- Objectives
  - 能够针对高性能计算需求，设计并实现基于Spark和Spark SQL的的内存并行计算方案

- **Apache Spark** http://spark.apache.org/
  - Unified engine for large-scale data analytics
- **What is Apache Spark™?**
  - Apache Spark™ is a multi-language engine for executing data engineering, data science, and machine learning on single-node machines or clusters.

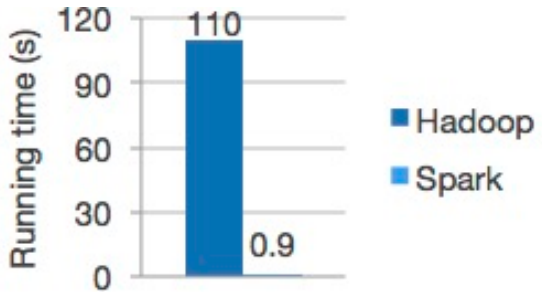
- **Simple. Fast. Scalable. Unified.**
- **The most widely-used engine for scalable computing**
  - Thousands of companies, including 80% of the Fortune 500, use Apache Spark™.
    Over 2,000 contributors to the open source project from industry and academia.

- **Batch/streaming data**
  - Unify the processing of your data in batches and real-time streaming, using your preferred language: Python, SQL, Scala, Java or R.
- **SQL analytics**
  - Execute fast, distributed ANSI SQL queries for dashboarding and ad-hoc reporting. Runs faster than most data warehouses.
- **Data science at scale**
  - Perform Exploratory Data Analysis (EDA) on petabyte-scale data without having to resort to downsampling
- **Machine learning**
  - Train machine learning algorithms on a laptop and use the same code to scale to fault-tolerant clusters of thousands of machines.

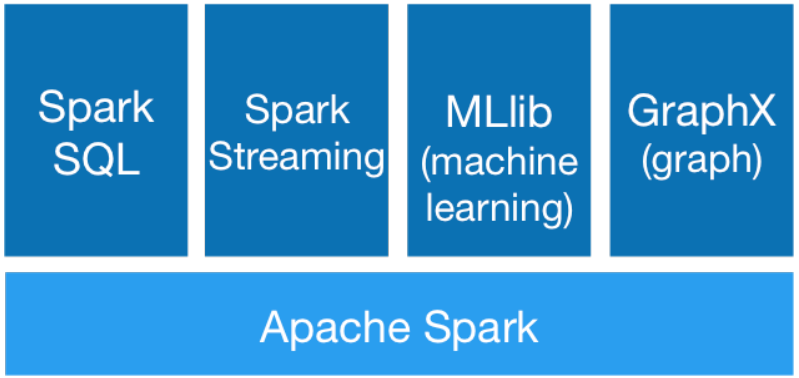
Spark简介

- Outstanding performance
  - Run workloads 100x faster
  - Memory computing

- Components for various usage
  - Spark Core
  - Spark SQL
  - Spark Streaming
  - MLlib
  - GraphX



Logistic regression in Hadoop and Spark



Spark & Hadoop

- Hadoop
  - HDFS + MapReduce

- MapReduce
  - MapReduce
  - Low utilization of memory and much disk I/O

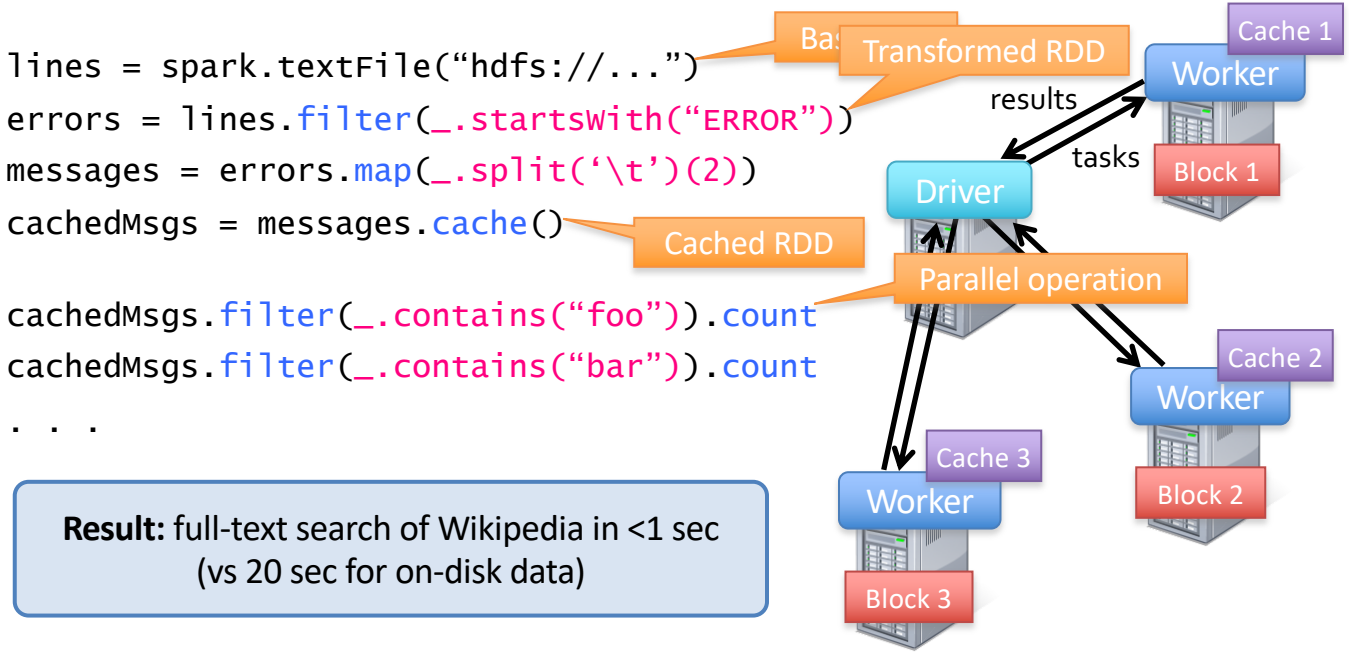
- Spark
  - Higher performance（100x faster）
  - Memory computing
  - More APIs
  - Capable of integrating with Hadoop (HDFS)
  - …

Example: Log Mining

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

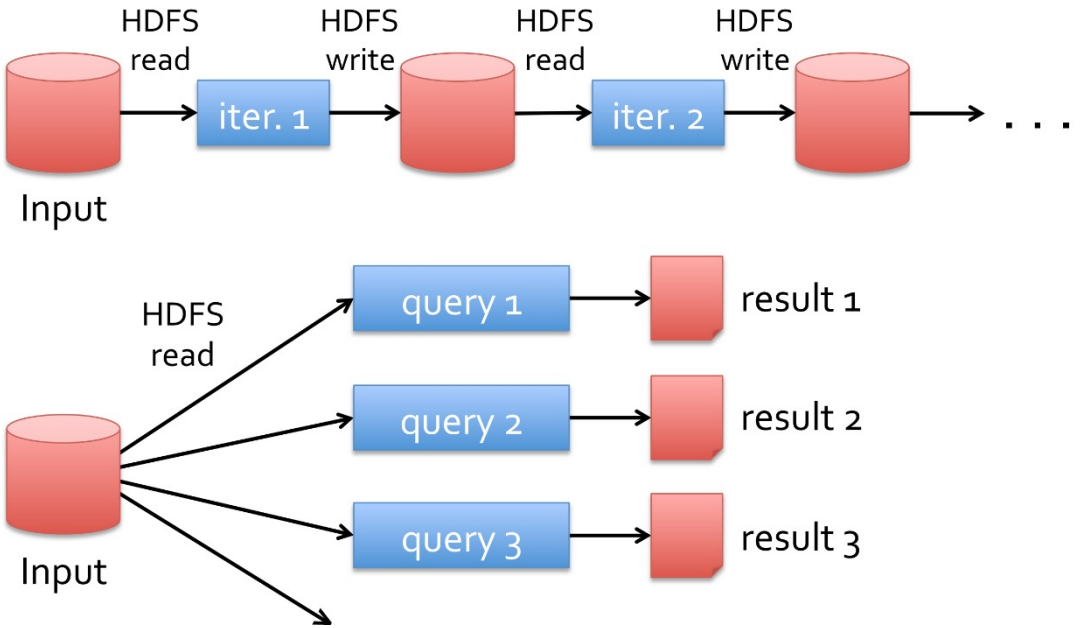
```
lines = spark.textFile(“hdfs://...”)
errors = lines.filter(_.startsWith(“ERROR”))
messages = errors.map(_.split(‘\t’)(2))
cachedMsgs = messages.cache()

cachedMsgs.filter(_.contains(“foo”)).count
cachedMsgs.filter(_.contains(“bar”)).count
. . .
```



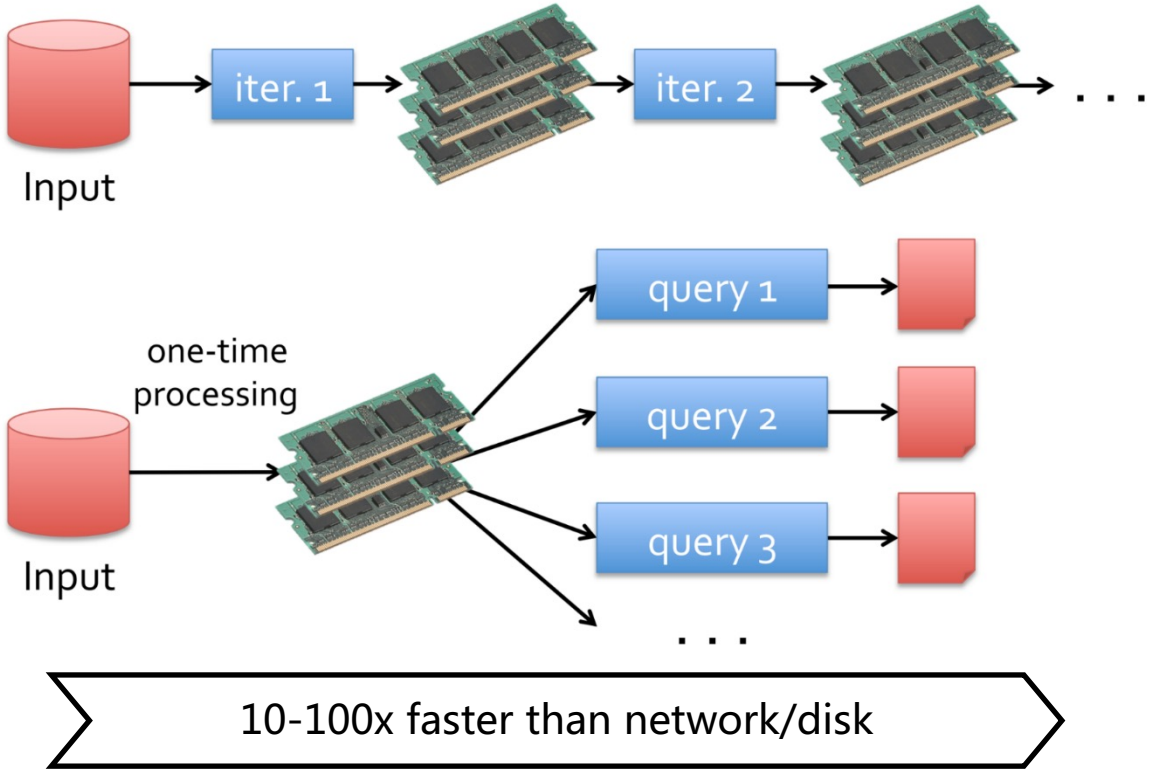
Result: full-text search of Wikipedia in <1 sec (vs 20 sec for on-disk data)

HDFS & Spark



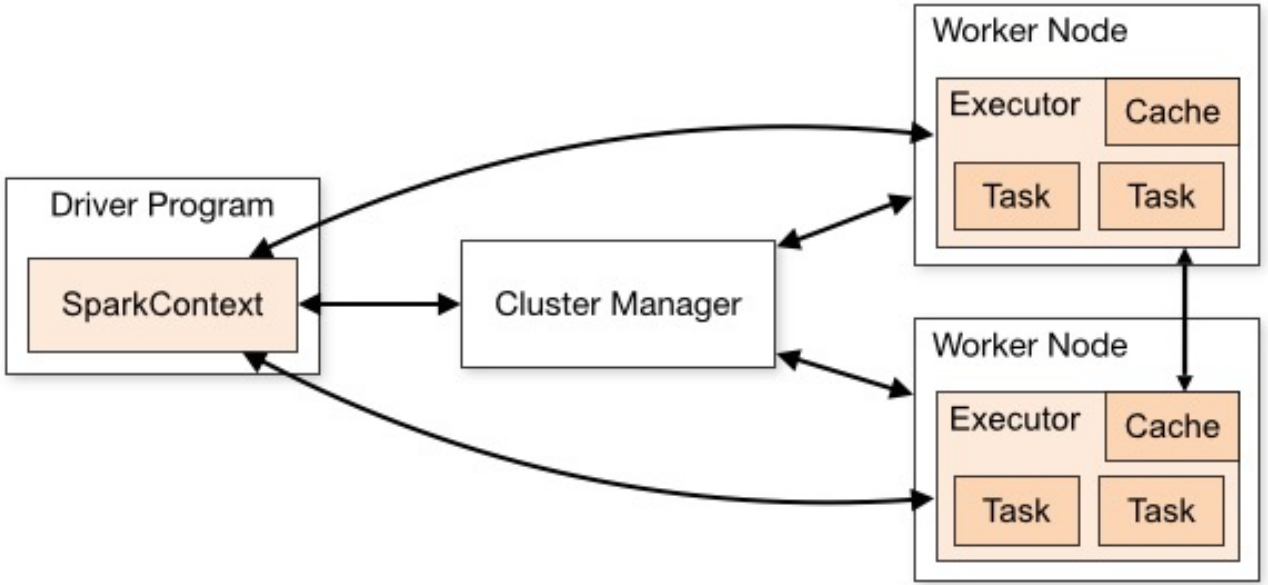
Slow due to replication and disk I/O, but necessary for fault tolerance

HDFS & Spark


iter. 1
iter. 2
. . .
Input
one-time
processing
query 1
query 2
query 3
. . .
Input
10-100x faster than network/disk

Spark Components

- Spark applications run as independent sets of processes on a cluster, coordinated by the SparkContext object in your main program (called the *driver program*).



Spark Components

- Specifically,
  - to run on a cluster, the SparkContext can connect to several types of *cluster managers* (either Spark's own standalone cluster manager, Mesos, YARN or Kubernetes), which allocate resources across applications.
  - Once connected, Spark acquires *executors* on nodes in the cluster, which are processes that run computations and store data for your application.
  - Next, it sends your application code (defined by JAR or Python files passed to SparkContext) to the executors.
  - Finally, SparkContext sends *tasks* to the executors to run.

Spark Components

- There are several useful things to note about this architecture:
  - Each application gets its own executor processes, which stay up for the duration of the whole application and run tasks in multiple threads.
  - Spark is agnostic to the underlying cluster manager. As long as it can acquire executor processes, and these communicate with each other, it is relatively easy to run it even on a cluster manager that also supports other applications (e.g. Mesos/YARN/Kubernetes).
  - The driver program must listen for and accept incoming connections from its executors throughout its lifetime. As such, the driver program must be network addressable from the worker nodes.
  - Because the driver schedules tasks on the cluster, it should be run close to the worker nodes, preferably on the same local area network.

Cluster Manager Types

- The system currently supports several cluster managers:
  - Standalone – a simple cluster manager included with Spark that makes it easy to set up a cluster.
  - Apache Mesos – a general cluster manager that can also run Hadoop MapReduce and service applications. (Deprecated)
  - Hadoop YARN – the resource manager in Hadoop 2.
  - Kubernetes – an open-source system for automating deployment, scaling, and management of containerized applications.

How Spark Works

- **Submitting Applications**
  - Applications can be submitted to a cluster of any type using the spark-submit script.
- **Monitoring**
  - Each driver program has a web UI, typically on port 4040, that displays information about running tasks, executors, and storage usage. Simply go to http://<driver-node>:4040 in a web browser to access this UI.
- **Job Scheduling**
  - Spark gives control over resource allocation both *across* applications (at the level of the cluster manager) and *within* applications (if multiple computations are happening on the same SparkContext).

Glossary

Term	Meaning
Application	User program built on Spark. Consists of a *driver program* and *executors* on the cluster.
Application jar	A jar containing the user's Spark application. In some cases users will want to create an "uber jar" containing their application along with its dependencies. The user's jar should never include Hadoop or Spark libraries, however, these will be added at runtime.
Driver program	The process running the main() function of the application and creating the SparkContext
Cluster manager	An external service for acquiring resources on the cluster (e.g. standalone manager, Mesos, YARN, Kubernetes)
Deploy mode	Distinguishes where the driver process runs. In "cluster" mode, the framework launches the driver inside of the cluster. In "client" mode, the submitter launches the driver outside of the cluster.
Worker node	Any node that can run application code in the cluster
Executor	A process launched for an application on a worker node, that runs tasks and keeps data in memory or disk storage across them. Each application has its own executors.
Task	A unit of work that will be sent to one executor
Job	A parallel computation consisting of multiple tasks that gets spawned in response to a Spark action (e.g. save, collect); you'll see this term used in the driver's logs.
Stage	Each job gets divided into smaller sets of tasks called *stages* that depend on each other (similar to the map and reduce stages in MapReduce); you'll see this term used in the driver's logs.

Spark Standalone Mode

- **Standalone Mode**
  - You can launch a standalone cluster either manually, by starting a master and workers by hand, or use our provided launch scripts.

- **Security**
  - Security features like authentication are not enabled by default.
  - When deploying a cluster that is open to the internet or an untrusted network, it's important to secure access to the cluster to prevent unauthorized applications from running on the cluster.
- **Installing Spark Standalone to a Cluster**
  - To install Spark Standalone mode, you simply place a compiled version of Spark on each node on the cluster. You can obtain pre-built versions of Spark with each release or build it yourself.

Spark Standalone Mode

REliable, INtelligent & Scalable Systems

- **Starting a Cluster Manually**
  - You can start a standalone master server by executing:
  - `$ ./sbin/start-master.sh`
  - Once started, the master will print out a spark://HOST:PORT URL for itself, which you can use to connect workers to it, or pass as the "master" argument to SparkContext.
  - You can also find this URL on the master's web UI, which is http://localhost:8080 by default.

  - Similarly, you can start one or more workers and connect them to the master via:
  - `$ ./sbin/start-worker.sh <master-spark-URL>`
  - Once you have started a worker, look at the master's web UI (http://localhost:8080 by default). You should see the new node listed there, along with its number of CPUs and memory (minus one gigabyte left for the OS).

Spark Standalone Mode

REliable, INtelligent & Scalable Systems

Spark Master at spark://MacBook-Pro-7.local:7077

URL: spark://MacBook-Pro-7.local:7077
Alive Workers: 1
Cores in use: 16 Total, 0 Used
Memory in use: 15.0 GiB Total, 0.0 B Used
Resources in use:
Applications: 0 Running, 0 Completed
Drivers: 0 Running, 0 Completed
Status: ALIVE

Workers (1)

Worker Id	Address	State	Cores	Memory	Resources
worker-20211211095752-192.168.1.5-51412	192.168.1.5:51412	ALIVE	16 (0 Used)	15.0 GiB (0.0 B Used)	

Running Applications (0)

Application ID	Name	Cores	Memory per Executor	Resources Per Executor	Submitted Time	User	State	Duration

Completed Applications (0)

Application ID	Name	Cores	Memory per Executor	Resources Per Executor	Submitted Time	User	State	Duration

Spark Standalone Mode

- **Cluster Launch Scripts**
  - To launch a Spark standalone cluster with the launch scripts, you should create a file called conf/workers in your Spark directory, which must contain the hostnames of all the machines where you intend to start Spark workers, one per line.
  - If conf/workers does not exist, the launch scripts defaults to a single machine (localhost), which is useful for testing.

  - Once you've set up this file, you can launch or stop your cluster with the following shell scripts, based on Hadoop's deploy scripts, and available in SPARK_HOME/sbin:
    - sbin/start-master.sh - Starts a master instance on the machine the script is executed on.
    - sbin/start-workers.sh - Starts a worker instance on each machine specified in the conf/workers file.
    - sbin/start-worker.sh - Starts a worker instance on the machine the script is executed on.
    - sbin/start-all.sh - Starts both a master and a number of workers as described above.
    - sbin/stop-master.sh - Stops the master that was started via the sbin/start-master.sh script.
    - sbin/stop-worker.sh - Stops all worker instances on the machine the script is executed on.
    - sbin/stop-workers.sh - Stops all worker instances on the machines specified in the conf/workers file.
    - sbin/stop-all.sh - Stops both the master and the workers as described above.

- **Interactive Analysis with the Spark Shell**
  - Spark's shell provides a simple way to learn the API, as well as a powerful tool to analyze data interactively. It is available in either Scala (which runs on the Java VM and is thus a good way to use existing Java libraries) or Python.
  - Start it by running the following in the Spark directory:
  - **Scala:** `$ ./bin/spark-shell`
  - **Python:** `$ ./bin/pyspark`

```
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 3.2.0
      /_/

Using Scala version 2.12.15 (Java HotSpot(TM) 64-Bit Server VM, Java 13.0.2)
Type in expressions to have them evaluated.
Type :help for more information.
```

- **Interactive Analysis with the Spark Shell**
  - Spark's shell provides a simple way to learn the API, as well as a powerful tool to analyze data interactively. It is available in either Scala (which runs on the Java VM and is thus a good way to use existing Java libraries) or Python.
  - Start it by running the following in the Spark directory:
  - **Scala:** `$ ./bin/spark-shell`
  - **Python:** `$ ./bin/pyspark`

```
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 3.2.0
      /_/

Using Scala version 2.12.15 (Java HotSpot(TM) 64-Bit Server VM, Java 13.0.2)
Type in expressions to have them evaluated.
Type :help for more information.
```

Quick Start

- Spark's primary abstraction is a distributed collection of items called a Dataset.
  - Datasets can be created from Hadoop InputFormats (such as HDFS files) or by transforming other Datasets.
  - Let's make a new DataFrame from the text of the README file in the Spark source directory:

```
scala> val textFile = spark.read.textFile("README.md")
textFile: org.apache.spark.sql.Dataset[String] = [value: string]

scala> textFile.count()
res0: Long = 109

scala> textFile.first()
res1: String = # Apache Spark

scala> val linesWithSpark = textFile.filter(line => line.contains("Spark"))
linesWithSpark: org.apache.spark.sql.Dataset[String] = [value: string]

scala> textFile.filter(line => line.contains("Spark")).count() // How many lines contain
"Spark"?
res2: Long = 19

scala> textFile.map(line => line.split(" ").size).reduce((a, b) => if (a > b) a else b)
res3: Int = 16
```

Quick Start

- Spark's primary abstraction is a distributed collection of items called a Dataset.
  - `scala> import java.lang.Math`
  - `import java.lang.Math`

  - `scala> textFile.map(line => line.split(" ").size).reduce((a, b) => Math.max(a, b))`
  - `res4: Int = 16`

  - `scala> val wordCounts = textFile.flatMap(line => line.split(" ")).groupByKey(identity).count()`
  - `wordCounts: org.apache.spark.sql.Dataset[(String, Long)] = [key: string, count(1): bigint]`

  - `scala> wordCounts.collect()`
  - `res5: Array[(String, Long)] = Array(([![PySpark,1), (online,1), (graphs,1), (["Building,1), (documentation,3), (command,,2), (abbreviated,1), (overview,1), (rich,1), (set,2), (-DskipTests,1), (1,000,000,000:,2), (name,1), (["Specifying,1), (stream,1), (run:,1), (not,1), (programs,2), (tests,2), (./dev/run-tests,1), (will,1), ([run,1), (particular,2), (Alternatively,,1), (must,1), (using,3), (./build/mvn,1), (you,4), (MLlib,1), (DataFrames,,1), (variable,1), (Note,1), (core,1), (protocols,1), (Guide](https://spark.apache.org/docs/latest/configuration.html),1), (guidance,2), (shell:,2), (can,6), (site,,1), (*,4), (systems.,1), ([building,1), (configure,1), (for,12), (README,1), (Interactive,2), (how,3), ([Configuration,1), (Hive,2), (provides,1), (Hadoop-supporte...`

- **Caching**
  - Spark also supports pulling data sets into a cluster-wide in-memory cache.
  - This is very useful when data is accessed repeatedly, such as when querying a small “hot” dataset or when running an iterative algorithm like PageRank.
  - As a simple example, let’s mark our linesWithSpark dataset to be cached:

```
scala> linesWithSpark.cache()
res6: linesWithSpark.type = [value: string]

scala> linesWithSpark.count()
res7: Long = 19

scala> linesWithSpark.count()
res8: Long = 19
```

Quick Start

- We'll create a very simple Spark application in Scala–so simple, in fact, that it's named SimpleApp.scala:

```scala
/* SimpleApp.scala */
import org.apache.spark.sql.SparkSession
object SimpleApp {
  def main(args: Array[String]) {
    val logFile = "YOUR_SPARK_HOME/README.md" // Should be some file on your system
    val spark = SparkSession.builder.appName("Simple Application").getOrCreate()
    val logData = spark.read.textFile(logFile).cache()
    val numAs = logData.filter(line => line.contains("a")).count()
    val numBs = logData.filter(line => line.contains("b")).count()
    println(s"Lines with a: $numAs, Lines with b: $numBs")
    spark.stop()
  }
}
```

Quick Start

- Our application depends on the Spark API, so we'll also include an sbt configuration file, build.sbt,
  - which explains that Spark is a dependency.
  - This file also adds a repository that Spark depends on:

```
name := "SE3353_26_Spark"

version := "0.1"

scalaVersion := "2.13.7"

libraryDependencies += "org.apache.spark" %% "spark-sql" % "3.2.0"
```

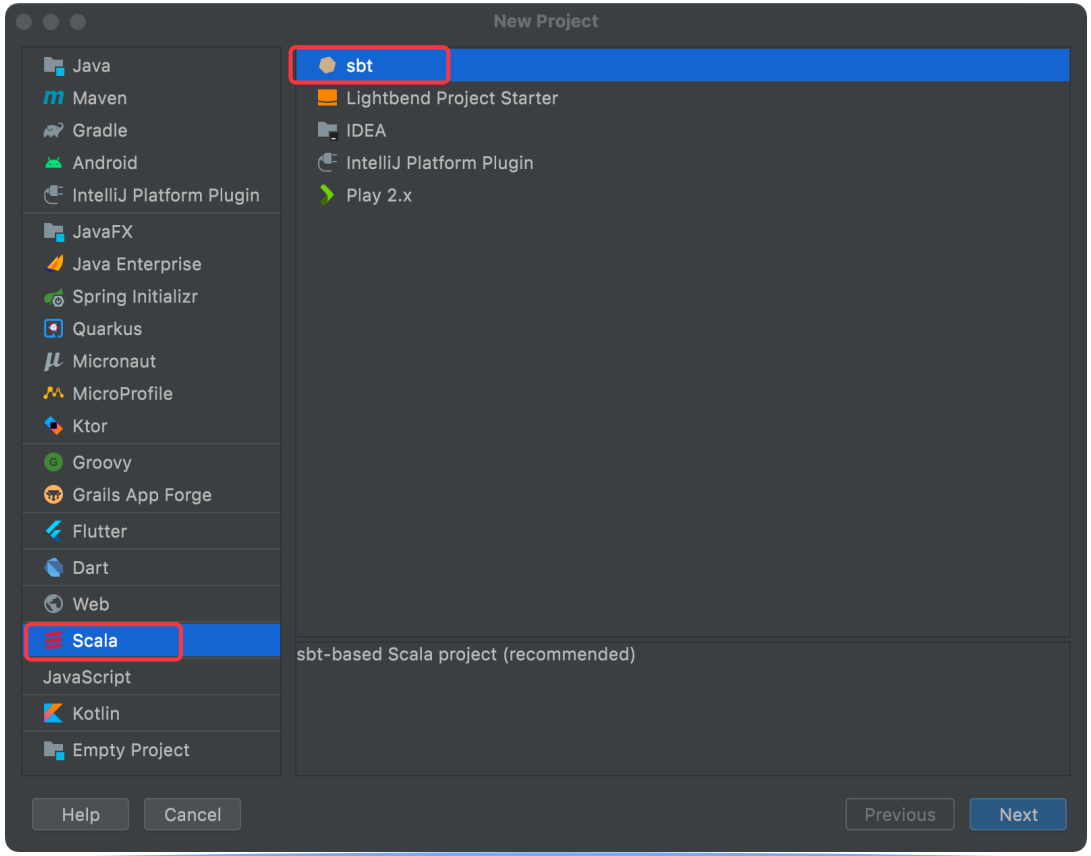
Quick Start

- For sbt to work correctly, we'll need to
  - layout SimpleApp.scala and build.sbt according to the typical directory structure. Once that is in place, we can create a JAR package containing the application's code, then use the spark-submit script to run our program.

```
# Your directory layout should look like this
$ find .
.
./build.sbt
./src
./src/main
./src/main/scala
./src/main/scala/SimpleApp.scala
# Package a jar containing your application
$ sbt package
...
[info] Packaging {..}/{..}/target/scala-2.12/simple-project_2.12-1.0.jar
```

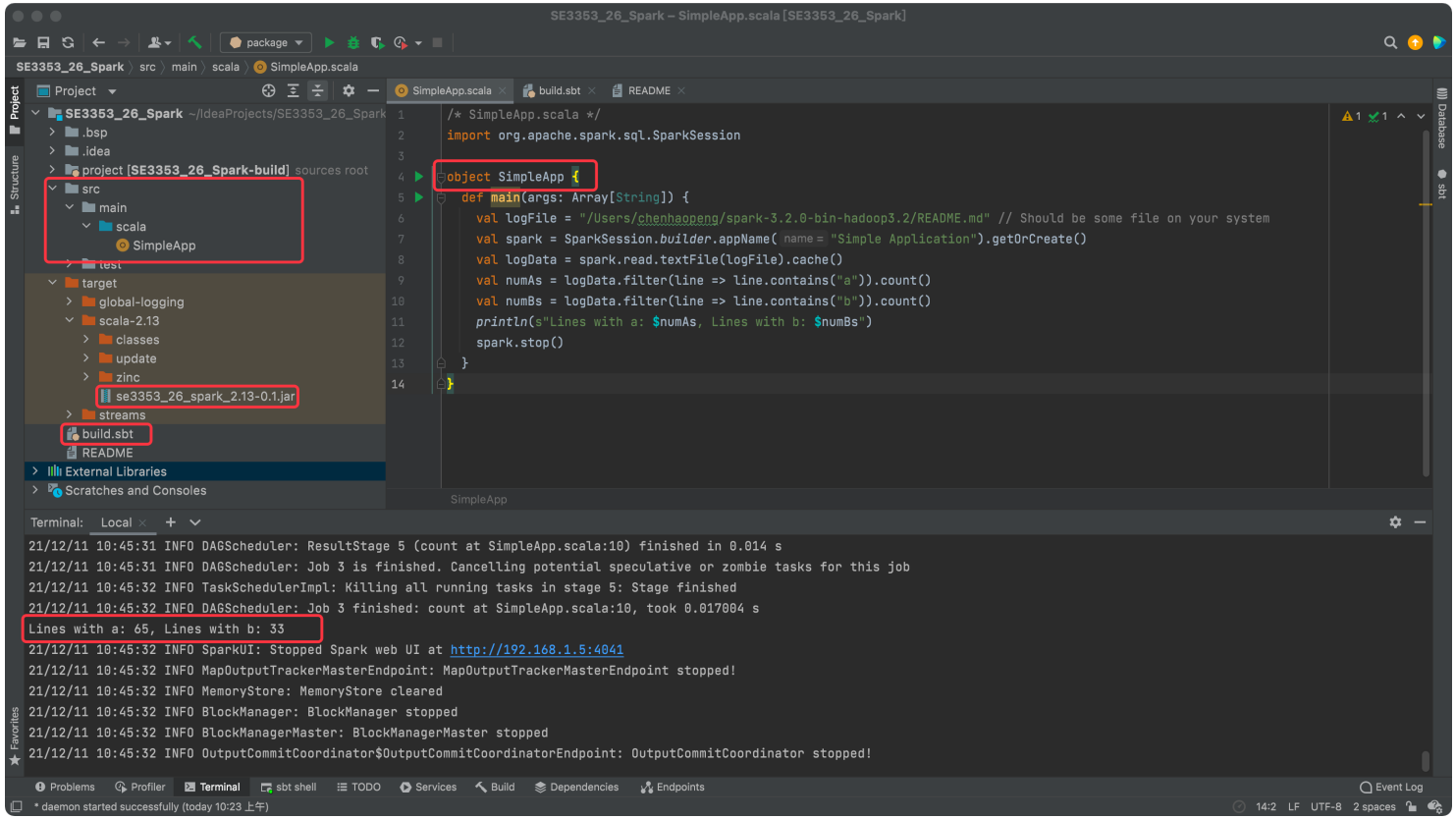
```
# Use spark-submit to run your application
$ YOUR_SPARK_HOME/bin/spark-submit \
  --class "SimpleApp" \
  --master local\
  target/scala-2.13/se3353_26_spark_2.13-0.1.jar
...
Lines with a: 46, Lines with b: 23
```

Quick Start

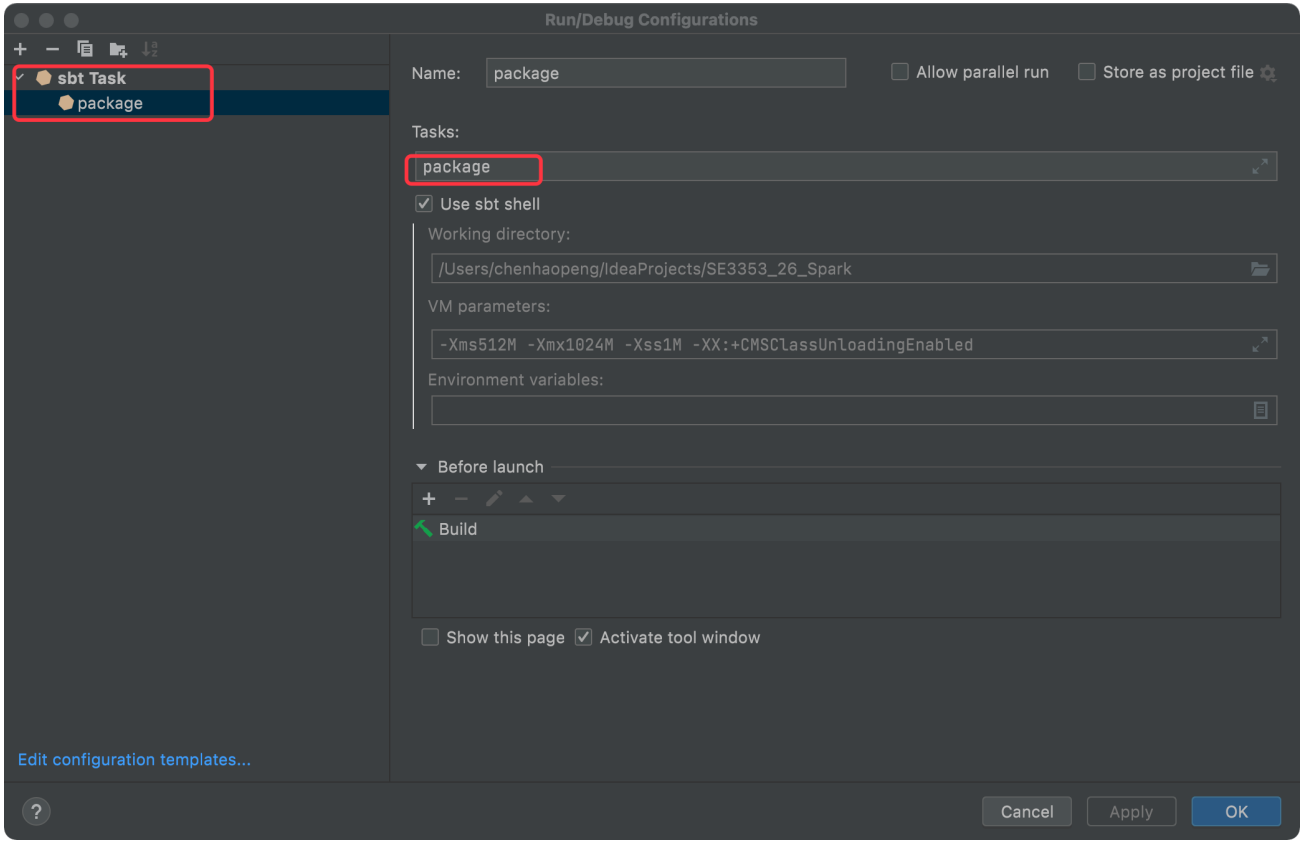

New Project
Java
Maven
Gradle
Android
IntelliJ Platform Plugin
JavaFX
Java Enterprise
Spring Initializr
Quarkus
Micronaut
MicroProfile
Ktor
Groovy
Grails App Forge
Flutter
Dart
Web
Scala
JavaScript
Kotlin
Empty Project
sbt
Lightbend Project Starter
IDEA
IntelliJ Platform Plugin
Play 2.x
sbt-based Scala project (recommended)
Help
Cancel
Previous
Next

Quick Start

REliable, INtelligent & Scalable Systems



Quick Start


Run/Debug Configurations
sbt Task
package
Name: package
Allow parallel run
Store as project file
Tasks:
package
Use sbt shell
Working directory:
/Users/chenhaopeng/IdeaProjects/SE3353_26_Spark
VM parameters:
-Xms512M -Xmx1024M -Xss1M -XX:+CMSClassUnloadingEnabled
Environment variables:
Before launch
Build
Show this page
Activate tool window
Edit configuration templates...
Cancel
Apply
OK

- Python Application

```
"""SimpleApp.py"""
from pyspark.sql import SparkSession

logFile = "/Users/chenhaopeng/spark-3.2.0-bin-hadoop3.2/README.md"  # Should be some file on your system
spark = SparkSession.builder.appName("SimpleApp").getOrCreate()
logData = spark.read.text(logFile).cache()

numAs = logData.filter(logData.value.contains('a')).count()
numBs = logData.filter(logData.value.contains('b')).count()

print("Lines with a: %i, lines with b: %i" % (numAs, numBs))

spark.stop()
```

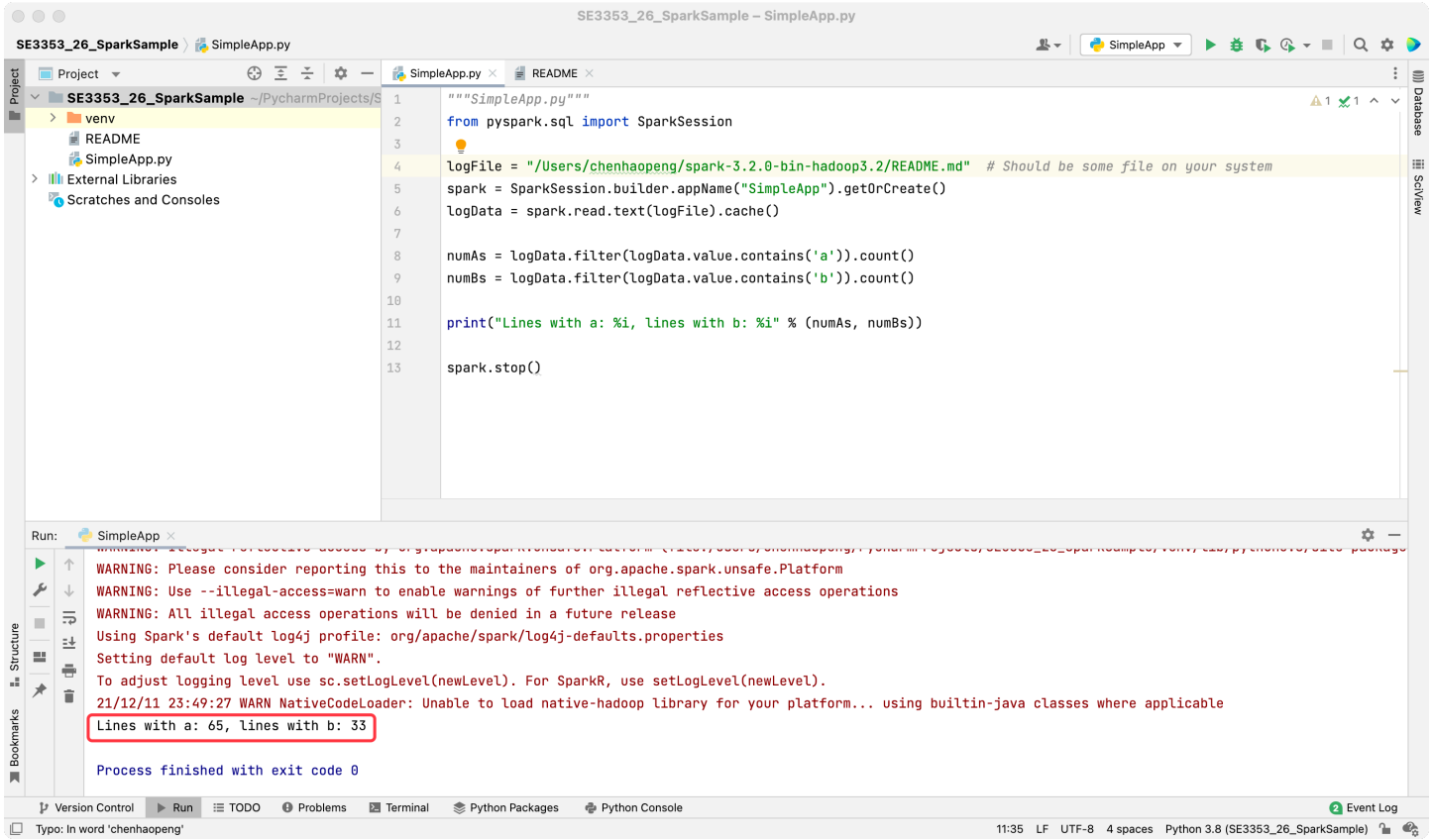
Quick Start

- Python Application

```
$ /Users/chenhaopeng/spark-3.2.0-bin-hadoop3.2/bin/spark-submit \
  --class "SimpleApp" \
  --master local \
  SimpleApp.py
```

Quick Start

REin
REliable, INtelligent & Scalable Systems



- **Resilient Distributed Datasets (RDDs)**
  - Spark revolves around the concept of a *resilient distributed dataset* (RDD), which 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 filesystem, HDFS, HBase, or any data source offering a Hadoop InputFormat.

- **Parallelized Collections**
  - Parallelized collections are created by calling SparkContext's parallelize method on an existing collection in your driver program
  - `val data = Array(1, 2, 3, 4, 5)`
  - `val distData = sc.parallelize(data)`

- **External Datasets**
  - Spark can create distributed datasets from any storage source supported by Hadoop, including your local file system, HDFS, Cassandra, HBase, Amazon S3, etc.
  - Spark supports text files, SequenceFiles, and any other Hadoop InputFormat.

  - `scala> val distFile = sc.textFile("data.txt")`
  - `distFile: org.apache.spark.rdd.RDD[String] = data.txt MapPartitionsRDD[10] at textFile at <console>:26`

- **Partition**
  - One important parameter for parallel collections is the number of *partitions* to cut the dataset into.
  - Spark will run one task for each partition of the cluster.

RDD Operations

- RDDs support two types of operations:
  - *transformations*, which create a new dataset from an existing one, and
  - *actions*, which return a value to the driver program after running a computation on the dataset.

  - For example, map is a transformation that passes each dataset element through a function and returns a new RDD representing the results.
  - On the other hand, reduce is an action that aggregates all the elements of the RDD using some function and returns the final result to the driver program.

RDD Operations

- All transformations in Spark are *lazy*,
  - in that they do not compute their results right away.
  - Instead, they just remember the transformations applied to some base dataset (e.g. a file). The transformations are only computed when an action requires a result to be returned to the driver program.
  - This design enables Spark to run more efficiently. For example, we can realize that a dataset created through map will be used in a reduce and return only the result of the reduce to the driver, rather than the larger mapped dataset.
  - 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 using the persist (or cache) method, 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.

RDD Transformations

- map

```
nums = sc.parallelize([1, 2, 3, 4])
squared = nums.map(lambda x: x * x).collect()
```

- flatMap

```
lines = sc.parallelize(["hello world", "hi"])
words = lines.flatMap(lambda line: line.split(" "))
```

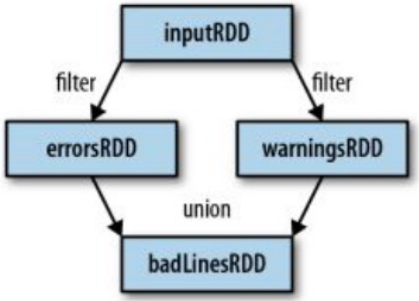
RDD Transformations

- filter

```
inputRDD = sc.textFile("log.txt")
errorsRDD = inputRDD.filter(lamda x: "error" in x)
warningsRDD = inputRDD.filter(lamda x: "warning" in x)
```

- union

```
badLinesRDD = errorsRDD.union(warningsRDD)
```



RDD Transformations

For RDD {1, 2, 3, 3}

Transformations	Meaning	Example	Result
map(*func*)	Return a new distributed dataset formed by passing each element of the source through a function *func*.	rdd.map(x => x + 1)	{2, 3, 4, 4}
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).	rdd.flatMap(x => x.to(3))	{1, 2, 3, 2, 3, 3, 3}
filter(*func*)	Return a new dataset formed by selecting those elements of the source on which *func* returns true.	rdd.filter(x => x != 1)	{2, 3, 3}
distinct()	Return a new dataset that contains the distinct elements of the source dataset.	rdd.distinct()	{1, 2, 3}

RDD Transformations

For RDDs {1, 2, 3} and {3, 4, 5}

Transformations	Meaning	Example	Result
union(*otherDataset*)	Return a new dataset that contains the union of the elements in the source dataset and the argument.	rdd.union(other)	{1, 2, 3, 3, 4, 5}
intersection (*otherDataset*)	Return a new RDD that contains the intersection of elements in the source dataset and the argument.	rdd.intersection(other)	{3}
cartesian (*otherDataset*)	When called on datasets of types T and U, returns a dataset of (T, U) pairs (all pairs of elements).	rdd.cartesian(other)	{(1, 3), (1, 4), ..., (3,5)}

RDD Actions

- reduce

```
rdd = sc.parallelize([1, 2, 3, 4])
sum = rdd.reduce(lambda x, y: x + y)
```

- count
  - 为每个输入元素产生多个输出元素

```
print “Input had” + badLinesRDD.count() + "concerning lines"
```

RDD Actions

For RDD {1, 2, 3, 3}

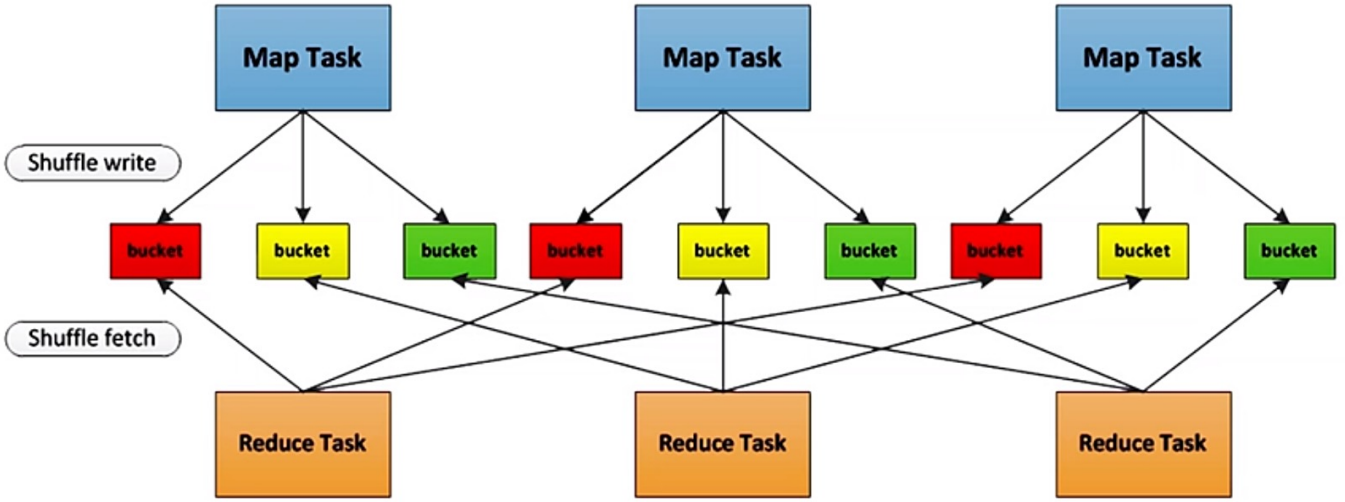
Actions	Meaning	Example	Result
collect()	Return all the elements of the dataset as an array at the driver program. This is usually useful after a filter or other operation that returns a sufficiently small subset of the data.	rdd.collect()	{1, 2, 3, 3}
count()	Return the number of elements in the dataset.	rdd.count()	4
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.	rdd.reduce((x,y) => x+y)	9

Spark 程序流程示例

- 基本流程示例
  - 从外部数据创建一些作为输入的RDD
  - 使用类似filter之类的变换(Transformations)来定义出新的RDD
  - 要求Spark对需要重用的任何中间RDD进行persist
  - 启用类似count之类的动作(Actions)进行并行计算

Shuffle Operation

- Certain operations within Spark trigger an event known as the shuffle.
  - The shuffle is Spark's mechanism for re-distributing data so that it's grouped differently across partitions. This typically involves copying data across executors and machines, making the shuffle a complex and costly operation.
  - The Shuffle is an expensive operation since it involves disk I/O, data serialization, and network I/O.

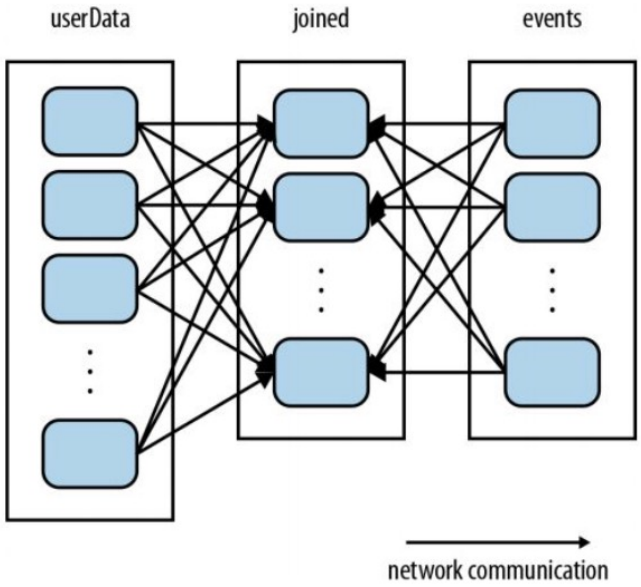


Partitions

- An Easy Example
  - 假设程序在内存中持有一个非常大的用户信息表 (UserID，UserInfo)，其中UserInfo 包含用户订阅的主题列表
  - 有一个很小的表，记录过去5分钟里在网页上点击过链接的事件，键值对为(UserID, LinkInfo)
  - 程序周期性地将这两个表合并、查询

Partitions

- No Partitions
  - join操作不知道数据集中的主键如何分区
  - userData需要周期性地进行Hash并Shuffle，即使其没发生任何变化

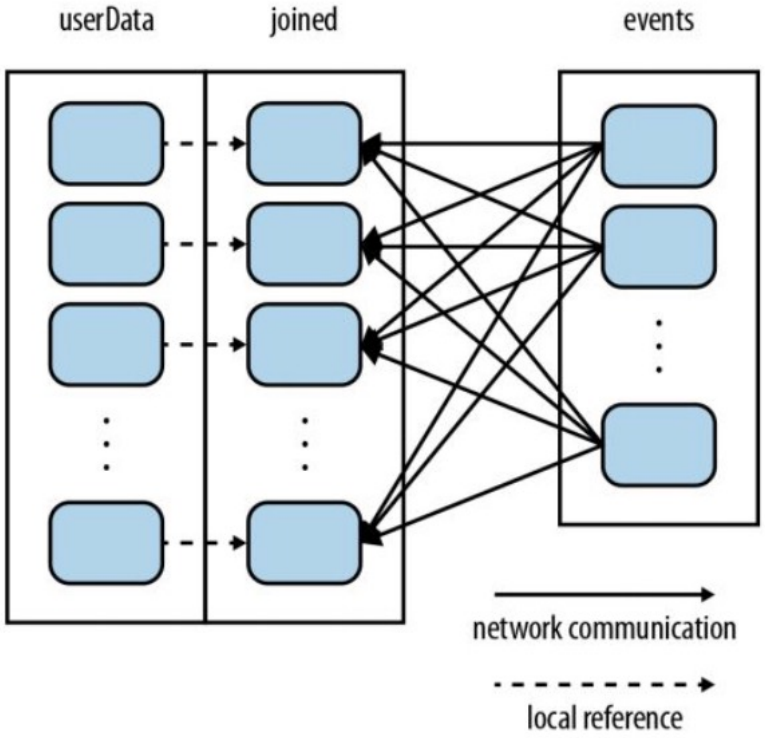


Partitions

- partitionBy - 指定分区
  - 指定分区
  - Spark 知道其为哈希分区，在执行join时会利用这一信息

```
val sc = new SparkContext(...)
val userData = sc.sequenceFile[UserID, UserInfo]("hdfs://...")
 .partitionBy(new HashPartitioner(100)) // Create 100 partitions
 .persist()
```

Partitions

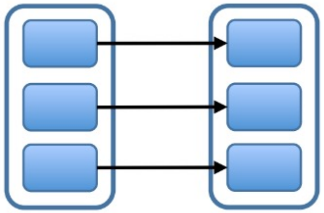

userData
joined
events
network communication
local reference

Dependencies

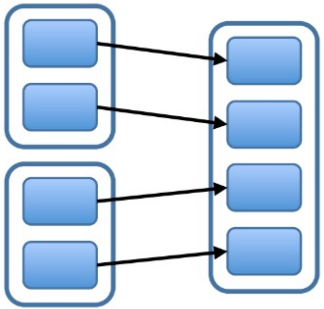
- Narrow Dependencies
  - 窄依赖
  - 父RDD的每个分区只被子RDD的一个分区所使用
  - map, union, …
- Wide Dependencies
  - 宽依赖
  - 父RDD的每个分区都可能被多个子RDD分区所使用
  - Shuffle
  - Join with inputs not co-partitioned, …

Dependencies

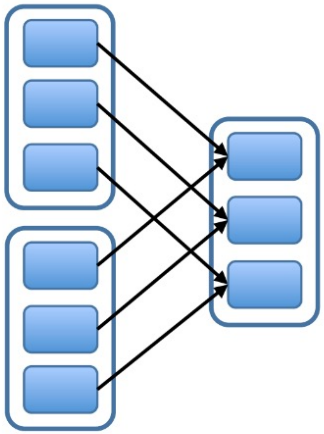
Narrow Dependencies:



map, filter

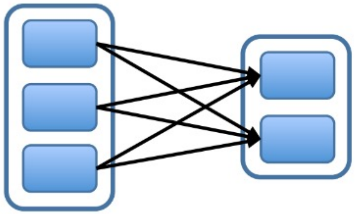


union

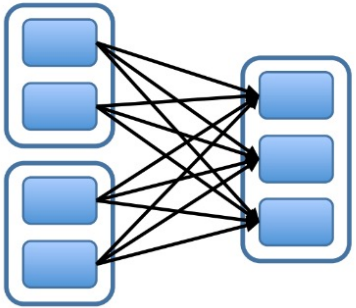


join with inputs co-partitioned

Wide Dependencies:



groupByKey

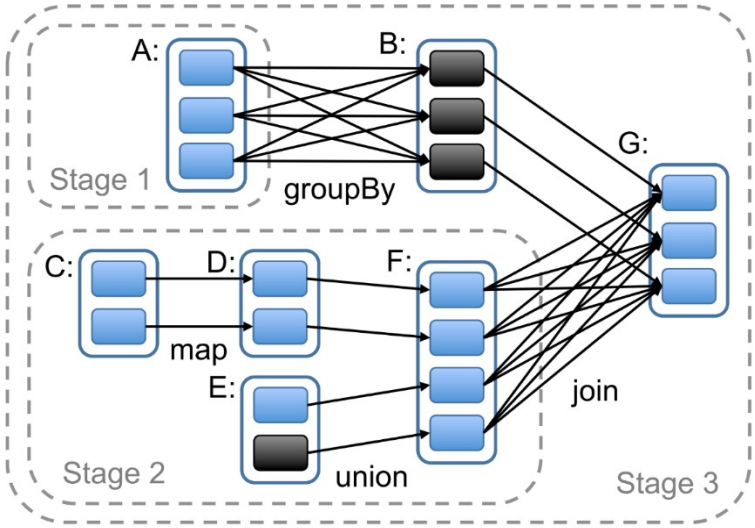


join with inputs not co-partitioned

Dependencies

- Narrow Dependencies VS. Wide Dependencies
  - 宽依赖往往意味着Shuffle操作，可能涉及多个节点的数据传输
  - 当RDD分区丢失时，Spark会对数据进行重算
  - 窄依赖只需计算丢失RDD的父分区，不同节点间可以并行计算，能更有效地进行节点的恢复
  - 宽依赖中，重算的子RDD分区往往来源自多个父RDD分区，其中只有一部分数据用于恢复，造成了不必要的冗余，甚至需要整体重新计算

- Stage
  - 每个阶段stage内部尽可能多地包含一组具有窄依赖关系的transformations操作，以便将它们流水线并行化（pipeline）
  - 边界有两种情况：一是宽依赖上的Shuffle操作；二是已缓存分区



RDD Persistence

- One of the most important capabilities in Spark is *persisting* (or *caching*) a dataset in memory across operations.
  - 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).

- You can mark an 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.
  - Spark's cache is fault-tolerant – if any partition of an RDD is lost, it will automatically be recomputed using the transformations that originally created it.

RDD Persistence

- In addition, each persisted RDD can be stored using a different *storage level.*
- The full set of storage levels is:

Storage Level	Meaning
MEMORY_ONLY	Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, some partitions will not be cached and will be recomputed on the fly each time they're needed. This is the default level.
MEMORY_AND_DISK	Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, store the partitions that don't fit on disk, and read them from there when they're needed.
MEMORY_ONLY_SER (Java and Scala)	Store RDD as *serialized* Java objects (one byte array per partition). This is generally more space-efficient than deserialized objects, especially when using a fast serializer, but more CPU-intensive to read.
MEMORY_AND_DISK_SER (Java and Scala)	Similar to MEMORY_ONLY_SER, but spill partitions that don't fit in memory to disk instead of recomputing them on the fly each time they're needed.
DISK_ONLY	Store the RDD partitions only on disk.
MEMORY_ONLY_2, MEMORY_AND_DISK_2, etc.	Same as the levels above, but replicate each partition on two cluster nodes.
OFF_HEAP (experimental)	Similar to MEMORY_ONLY_SER, but store the data in off-heap memory. This requires off-heap memory to be enabled.

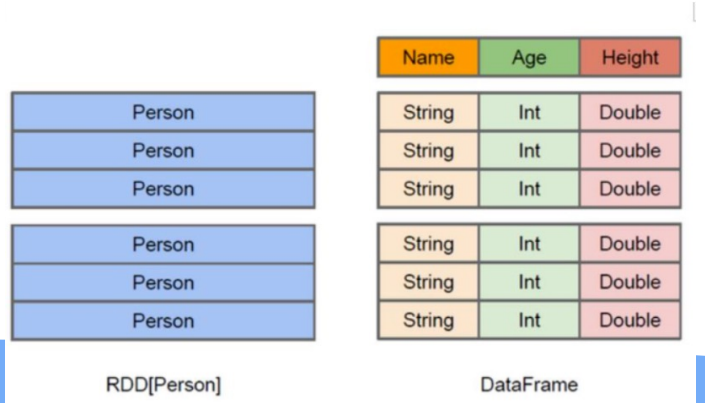
- **Removing Data**
  - Spark automatically monitors cache usage on each node and drops out old data partitions in a least-recently-used (LRU) fashion.
  - If you would like to manually remove an RDD instead of waiting for it to fall out of the cache, use the RDD.unpersist() method.
  - Note that this method does not block by default.
  - To block until resources are freed, specify blocking=true when calling this method.

- Spark SQL
  - Spark SQL is a Spark module for structured data processing.
  - Unlike the basic Spark RDD API, the interfaces provided by Spark SQL provide Spark with more information about the structure of both the data and the computation being performed.
  - Internally, Spark SQL uses this extra information to perform extra optimizations.
  - There are several ways to interact with Spark SQL including SQL and the Dataset API.
  - When computing a result, the same execution engine is used, independent of which API/language you are using to express the computation.
  - This unification means that developers can easily switch back and forth between different APIs based on which provides the most natural way to express a given transformation.

Datasets and DataFrames

- A Dataset is a distributed collection of data.
  - A Dataset can be constructed from JVM objects and then manipulated using functional transformations (map, flatMap, filter, etc.).

- A DataFrame is a *Dataset* organized into named columns.
  - It is conceptually equivalent to a table in a relational database or a data frame in R/Python, but with richer optimizations under the hood.
  - DataFrames can be constructed from a wide array of sources such as: structured data files, tables in Hive, external databases, or existing RDDs.



Creating DataFrames

- With a SparkSession, applications can create DataFrames from an existing RDD, from a Hive table, or from Spark data sources.
- As an example, the following creates a DataFrame based on the content of a JSON file:

```
val df = spark.read.json("examples/src/main/resources/people.json")
// Displays the content of the DataFrame to stdout
df.show()
// +----+-------+
// | age|   name|
// +----+-------+
// |null|Michael|
// |  30|   Andy|
// |  19| Justin|
// +----+-------+
```

Untyped Dataset Operations


```scala
// This import is needed to use the $-notation
import spark.implicits._

// Print the schema in a tree format
df.printSchema()
// root
// |-- age: long (nullable = true)
// |-- name: string (nullable = true)

// Select only the "name" column
df.select("name").show()
// +-------+
// |   name|
// +-------+
// |Michael|
// |   Andy|
// | Justin|
// +-------+

// Select everybody, but increment the age by 1
df.select($"name", $"age" + 1).show()
// +-------+---------+
// |   name|(age + 1)|
// +-------+---------+
// |Michael|     null|
// |   Andy|       31|
// | Justin|       20|
// +-------+---------+

// Select people older than 21
df.filter($"age" > 21).show()
// +---+----+
// |age|name|
// +---+----+
// | 30|Andy|
// +---+----+

// Count people by age
df.groupBy("age").count().show()
// +----+-----+
// | age|count|
// +----+-----+
// |  19|    1|
// |null|    1|
// |  30|    1|
// +----+-----+
// $example off:untyped_ops$
```

Running SQL Queries Programmatically

- The sql function on a SparkSession enables applications to run SQL queries programmatically and returns the result as a DataFrame.

```
// $example on:run_sql$
// Register the DataFrame as a SQL temporary view
df.createOrReplaceTempView("people")

val sqlDF = spark.sql("SELECT * FROM people")
sqlDF.show()
// +----+-------+
// | age|   name|
// +----+-------+
// |null|Michael|
// |  30|   Andy|
// |  19| Justin|
// +----+-------+
// $example off:run_sql$
```

Global Temporary View

- Temporary views in Spark SQL are session-scoped and will disappear if the session that creates it terminates.
  - If you want to have a temporary view that is shared among all sessions and keep alive until the Spark application terminates, you can create a global temporary view.

  - ```
    // $example on:global_temp_view$
    // Register the DataFrame as a global temporary view
    df.createGlobalTempView("people")
    ```
    ```

Untyped Dataset Operations


```
// Global temporary view is tied to a system preserved database `global_temp`
spark.sql("SELECT * FROM global_temp.people").show()
// +----+-------+
// | age|   name|
// +----+-------+
// |null|Michael|
// |  30|   Andy|
// |  19| Justin|
// +----+-------+

// Global temporary view is cross-session
spark.newSession().sql("SELECT * FROM global_temp.people").show()
// +----+-------+
// | age|   name|
// +----+-------+
// |null|Michael|
// |  30|   Andy|
// |  19| Justin|
// +----+-------+
```

Creating Datasets

- Datasets are similar to RDDs,
  - however, instead of using Java serialization or Kryo they use a specialized Encoder to serialize the objects for processing or transmitting over the network.

```
// $example on:create_ds$
case class Person(name: String, age: Long)
// $example off:create_ds$


// Encoders are created for case classes
val caseClassDS = Seq(Person("Andy", 32)).toDS()
caseClassDS.show()
// +----+---+
// |name|age|
// +----+---+
// |Andy| 32|
// +----+---+
```

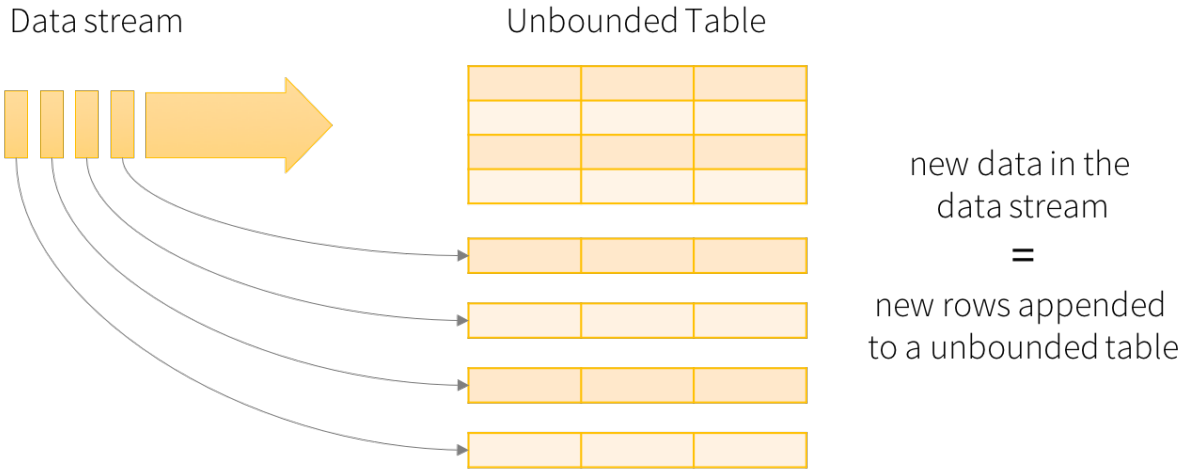
Creating Datasets


```
// Encoders for most common types are automatically provided by importing
spark.implicits._
val primitiveDS = Seq(1, 2, 3).toDS()
primitiveDS.map(_ + 1).collect() // Returns: Array(2, 3, 4)

// DataFrames can be converted to a Dataset by providing a class. Mapping
will be done by name
val path = "examples/src/main/resources/people.json"
val peopleDS = spark.read.json(path).as[Person]
peopleDS.show()
// +----+-------+
// | age|   name|
// +----+-------+
// |null|Michael|
// |  30|   Andy|
// |  19| Justin|
// +----+-------+
// $example off:create_ds$
```

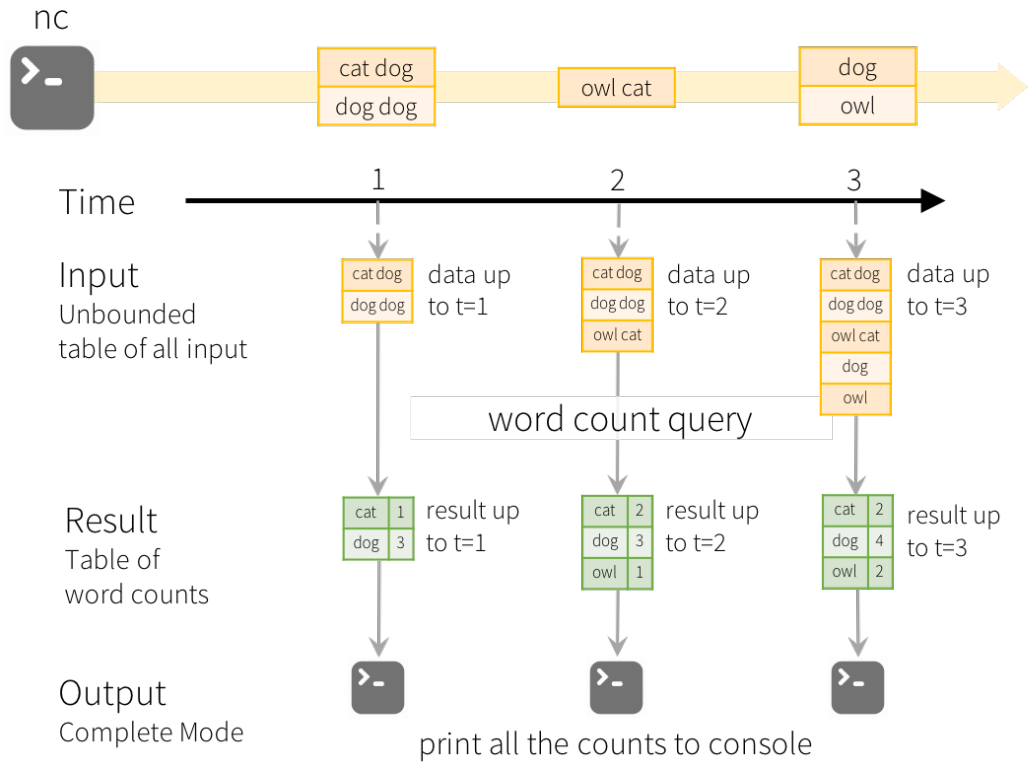
Other Issues

- Structured Streaming
  - a scalable and fault-tolerant stream processing engine built on the Spark SQL engine.



Data stream as an unbounded table

Other Issues

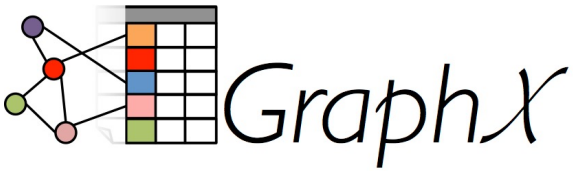


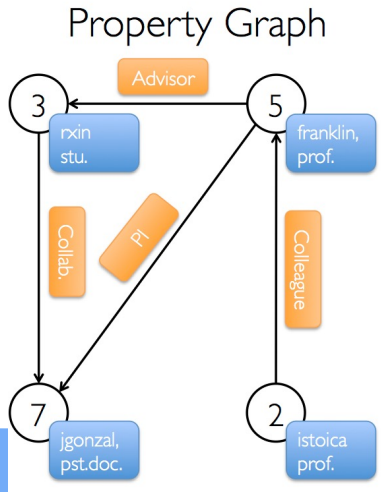
Model of the Quick Example

- Spark Streaming
  - an extension of the core Spark API that enables scalable, high-throughput, fault-tolerant stream processing of live data streams.

Other Issues

- **Machine Learning Library (MLlib) Guide**
  - MLlib is Spark's machine learning (ML) library. Its goal is to make practical machine learning scalable and easy. At a high level, it provides tools such as:
- **GraphX**
  - a new component in Spark for graphs and graph-parallel computation.
  - At a high level, GraphX extends the Spark RDD by introducing a new Graph abstraction: a directed multigraph with properties attached to each vertex and edge.





Vertex Table

Id	Property (V)
3	(rxin, student)
7	(jgonzal, postdoc)
5	(franklin, professor)
2	(istoica, professor)

Edge Table

SrcId	DstId	Property (E)
3	7	Collaborator
5	3	Advisor
2	5	Colleague
5	7	PI

References

- Spark Standalone Mode
  - https://spark.apache.org/docs/latest/spark-standalone.html
- Cluster Mode Overview
  - https://spark.apache.org/docs/latest/cluster-overview.html
- Quick Start
  - https://spark.apache.org/docs/latest/quick-start.html
- RDD Programming Guide
  - https://spark.apache.org/docs/latest/rdd-programming-guide.html
- Spark SQL, DataFrames and Datasets Guide
  - https://spark.apache.org/docs/latest/sql-programming-guide.html
- Python
  - http://spark.apache.org/docs/latest/api/python/index.html
- Java
  - http://spark.apache.org/docs/latest/api/java/index.html
- Scala
  - http://spark.apache.org/docs/latest/api/scala/index.html

References

- Structured Streaming Programming Guide
  - https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html
- Spark Streaming Programming Guide
  - https://spark.apache.org/docs/latest/streaming-programming-guide.html
- Machine Learning Library (MLlib) Guide
  - https://spark.apache.org/docs/latest/ml-guide.html
- GraphX Programming Guide
  - https://spark.apache.org/docs/latest/graphx-programming-guide.html

Thank You!