pySpark Core Concepts

Learning Objectives

* **Understand Spark Cluster Architecture**
* Understand RDDs
* Understand Spark Application Execution

Spark - Cluster Architecture

Spark - Cluster Architecture

* Compute engine components
  * Driver
    * Launch spark applications
    * Can be run outside or inside the cluster
  * Executors
    * Containers on the worker nodes
    * Run units of work called tasks
    * Allocated on a per-application basis
  * Cluster Manager
    * Allocate compute resources (executors)

Spark - Cluster Architecture

* Cores
  * Each executor runs a configurable number of threads called Cores
  * Essentially, this is the number of slots available for running tasks
  * The cluster, then, is composed of a number of cores, say N
  * Each application can ask for a number of cores M <= N
    * Application will run with available cores if the requested number of cores are not available
* SparkContext (old) and SparkSession (New)
  * Handle to the execution environment

Spark - Cluster Architecture

* Cluster Manager
  * Local
    * Driver, Executor, etc. all run as threads in a single JVM
  * Standalone
    * Spark's own cluster manager
    * Spark Master
      * Driver works with the Master to allocate Executors on Worker nodes
    * Spark Worker
      * Manages Executor lifecycle
  * YARN
  * Mesos

Spark Cluster Architecture - Worker

Dataset Partitions

Spark Tasks

One task is launched
for each partition
Running task
Unused core
Memory
Disk
Worker
Worker
Worker

Tasks and Partitions

`map(f)`: Each task makes a new partition by calling `f(e)` on each entry `e` in the original partition

Spark Collect action

Spark Count action

`count()`: Each task counts the entries in one partition

Spark Filter

`filter()`: Each task makes a new partition with the entries in the original partition that have a value less than 10

Learning Objectives

* Understand Spark Cluster Architecture
* **Understand RDDs**
* Understand Spark Application Execution

RDD - Definition

* Spark's abstraction for distributed memory
  * Read-only collection of objects distributed across a cluster
  * Data is divided into chunks called *partitions*
* Resilient
  * Partitions, if lost due to node failure, can be recreated
* Distributed
  * Partitions distributed across the cluster
  * So we can parallelize operations on large datasets
* Dataset
  * Handle for working with large datasets in-memory
* Ideal for iterative and interactive applications

RDD - Properties

* Two types of operations
  * Transformations
    * Specify operators to manipulate datasets
    * Create RDDs from data in stable storage
    * Create RDDs from other RDDs
  * Actions
    * Return a value to the application
    * Write data to stable storage
* Immutable
  * Cannot be changed once created
* Encapsulate metadata required to transform one RDD into another

RDD - Lineage

* Lineage
  * Each transformation encodes instructions needed to create a child RDD from parent RDD(s)
  * A series of such transformations construct a lineage
    * More formally, the series of transformations form a Directed Acyclic Graph (DAG)
  * Allows for portions of RDD to be reconstructed when they are lost due to node failure

RDD - Operations

* Transformations
  * Are evaluated lazily
  * Do not actually materialize data
  * They just construct the DAG
* Actions
  * Force the DAG to be evaluated
  * Walk up the graph and materialize all ancestor RDDs required to produce the result

RDD - Construction

RDD - Evaluation

RDD - Evaluation

RDD data garbage collected once the Action is complete

RDD - Workflow

* Typical minimal workflow for manipulating data
  * Read
    * Construct “Base” RDD by reading data from stable storage
  * Transform
    * Apply a series of transformations constructing the Lineage DAG
  * Write (Call an Action)
    * Evaluate the DAG materializing the RDDs
    * Write the result back to the application or to stable storage
* Extend this flow by applying more transformations and actions

RDD – Lazy Evaluation

* Implications
  * Pros
    * Affords opportunities for applying optimizations
    * Uses cluster resources more efficiently
  * Cons
    * Runtime errors in transformations will not show up until an action is run

Spark Cluster

Spark Execution-By Example

sc.textFile("hdfs://..")

Spark Execution-By Example

sc.textFile("hdfs://..").flatMap(lambda line: line.split(" "))

Spark Execution-By Example

sc.textFile("hdfs://..") .flatMap(lambda line: line.split(" ")).map(lambda word: (word,1))

Spark Execution-By Example

```
sc.textFile("hdfs://").flatMap(lambda line: line.split(" ")).map(lambda word: (word,1))
.reduceByKey(lambda a,b : a + b)
```

Spark Execution-By Example

```
sc.textFile("hdfs://").flatMap(lambda line: line.split(" ")).map(lambda word: (word,1))
.reduceByKey(lambda a,b : a + b)
```

Spark Execution-By Example

```
sc.textFile("hdfs://").flatMap(lambda line: line.split(" ")).map(lambda word: (word,1))
.reduceByKey(lambda a,b : a + b)
```

RDD – Narrow Dependencies

* Narrow
  * Each parent partition contributes data to a single child partition
  * A sequence of operations involving narrow dependencies can be *pipelined*

RDD – Wide Dependencies

* Wide
  * Each parent partition contributes data to multiple child partitions
  * Requires a *shuffle*
    * An expensive operation in a distributed system
    * Limits performance of the overall *Job*

RDD - Caching

* RDDs in the lineage are garbage collected once the action is complete
* Not all RDDs are created equal
  * Some of them are expensive to create
    * RDDs created as a result of an expensive shuffle
    * RDDs created after an expensive read and extensive filtering process
  * Some, with a high out-degree, are reused multiple times
* For all such RDDs, it may be prudent to cache them
  * Spark can then terminate the DAG execution at the cached RDD

RDD - Caching

* Caching involves a time-space tradeoff
  * Cache too many and you will run out of space
  * Cache too little and you will spend too much time recreating them
* Caches managed using an LRU algorithm
* Cache can be in heap/off-heap memory and disk
  * In serialized and deserialized forms
* We will talk about memory management in much more detail in a separate module

RDD – Programmatic Creation

* Spark Context/SparkSession
  * Handle to the execution environment/cluster
  * Created by passing in a SparkConf object which contains
    * Cluster configuration information
      * Resource Manager location
    * Application execution parameters
      * Resources to be reserved for this application
      * Application package (executable jar, python script, etc.)
      * Application dependencies
  * Standalone applications must create this object
  * Spark REPL and Databricks Cloud create this object automatically and make it available as the variable ***sc***

RDD – Programmatic Creation

* Base RDDs
  * Created from SparkContext
    * textFile
    * newAPIHadoopFile
    * objectFile
    * sequenceFile
    * wholeTextFiles
    * range
    * parallelize
* Derived RDDs
  * Created by applying transformations on existing RDDs

Transformations and Actions

A Visual Guide of the API

RDD

RDD elements

Legend

Randomized operation

Set Theory / Relational operation

Numeric calculation

Spark Operations =

TRANSFORMATIONS

+

ACTIONS

Essential Core & Intermediate Spark Operations

TRANSFORMATIONS

General

- map
- filter
- flatMap
- mapPartitions
- mapPartitionsWith Index
- groupBy
- sortBy

Math / Statistical

- sample
- randomSplit

Set Theory / Relational

- union
- intersection
- subtract
- distinct
- cartesian
- zip

Data Structure / I/O

- keyBy
- zipWithIndex
- zipWithUniqueID
- zipPartitions
- coalesce
- repartition
- repartitionAndSortWithinPartitions
- pipe

ACTIONS

- reduce
- collect
- aggregate
- fold
- first
- take
- forEach
- top
- treeAggregate
- treeReduce
- forEachPartition
- collectAsMap

- count
- takeSample
- max
- min
- sum
- histogram
- mean
- variance
- stdev
- sampleVariance
- countApprox
- countApproxDist inct

- takeOrdered

- saveAsTextFile
- saveAsSequenceFile
- saveAsObjectFile
- saveAsHadoopDataset
- saveAsHadoopFile
- saveAsNewAPIHadoopDataset
- saveAsNewAPIHadoopFile

Essential Core & Intermediate PairRDD Operations

TRANSFORMATIONS

General	Math / Statistical	Set Theory / Relational	Data Structure
• flatMapValues • groupByKey • reduceByKey • reduceByKeyLocally • foldByKey • aggregateByKey • sortByKey • combineByKey • keys • values	• sampleByKey	• cogroup (=groupWith) • join • subtractByKey • fullOuterJoin • leftOuterJoin • rightOuterJoin	• partitionBy

ACTIONS

- countByKey
- countByValue
- countByValueApprox
- countApproxDistinctByKey
- countApproxDistinctByKey
- countByKeyApprox
- sampleByKeyExact

Narrow Vs Wide Dependency

narrow

vs

wide

each partition of the parent RDD is used by at most one partition of the child RDD

multiple child RDD partitions may depend on a single parent RDD partition

LINEAGE

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 and exploratory 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. Moreover, 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 some frameworks provide interfaces abstractions for more efficient in-memory computation, most lack abstractions for leveraging distributed memory. This makes them inefficient for an important class of emerging applications 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 emerging use case, interactive data mining, where a user runs ad-hoc queries on the same subset of the data. Unfortunately, in these special purpose frameworks, the only way to reuse data between computations (for example, between two MapReduce jobs) is to write it to an external stable storage system, e.g., a distributed file system. This incurs substantial overhead 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 iterate over a set of data, such as Pregel [22] for iterative graph computations that keep intermediate data in memory, which Haloop [7] offers an iterative MapReduce interface, and Twister [12] provides in-memory MapReduce, extended to support iterative jobs, e.g., looping a series of MapReduce steps[1], 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 then run ad-hoc queries on 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 (cells, records, or objects). These models force the system to provide fault tolerance by replicating the data across machines or to log updates across machines. Both approaches incur substantial overhead: replication requires copying large amounts of data over the cluster, which is slow relative to the cluster's aggregate computing capacity, and logging updates uses 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.[2] If a partition of an RDD is lost, the RDD has enough information about how it was derived from other RDDs to recompute

[1] Iterating over the same data is also why the world's fastest sorting benchmark uses Hadoop running on top of a 1400-node cluster.

[2] Checkpointing the data in some RDDs may be useful when a lineage chain grows large, however, we do not discuss how to do it in this paper.

"One of the challenges in providing RDDs as an abstraction is choosing a representation for them that can track lineage across a wide range of transformations."

"The most interesting question in designing this interface is how to represent dependencies between RDDs."

"We found it both sufficient and useful to classify dependencies into two types:

- narrow dependencies, where each partition of the parent RDD is used by at most one partition of the child RDD
- wide dependencies, where multiple child partitions may depend on it."

narrow

each partition of the parent RDD is used by at most one partition of the child RDD

map, filter

union

join w/ inputs co-partitioned

wide

multiple child RDD partitions may depend on a single parent RDD partition

groupBy Key

join w/ inputs not co-partitioned

Transformations

Core Operations

*filter

RDD: **x**

3 items in RDD

(partitions not shown)

*filter

RDD: x
RDD: y
Apply user function: keep item if function returns true
True
emits

*filter

RDD: x

RDD: y

*filter

RDD: x

RDD: y

filter

After `filter()` has been applied…

RDD: **x**

before

RDD: **y**

after

filter

filter(*f*)

Return a new RDD containing only the elements that satisfy a predicate

```
x = sc.parallelize([1,2,3])
y = x.filter(lambda x: x%2 == 1)
#keep odd values
print(x.collect())
print(y.collect())
```

---

x: [1, 2, 3]

y: [1, 3]

map

RDD: **x**

map

*map

RDD: x

RDD: y

map

RDD: x

RDD: y

map

After `map()` has been applied…

RDD: **x**

RDD: **y**

before

after

map

RDD: x

RDD: y

Return a new RDD by applying a function to each element of this RDD.

Return a new RDD by applying a function to each element of this RDD

```
x = sc.parallelize(["b", "a", "c"])
y = x.map(lambda z: (z, 1))
print(x.collect())
print(y.collect())
```

```
x: ['b', 'a', 'c']

y: [('b', 1), ('a', 1), ('c', 1)]
```

*flatMap

RDD: **x**

3 items in RDD

(partitions not shown)

*flatmap

RDD: x

RDD: y

*flatmap

RDD: x

RDD: y

*flatmap

RDD: x

RDD: y

flatmap

After `flatmap()` has been applied…

RDD: **x**

before

RDD: **y**

after

*flatmap

RDD: x

RDD: y

Return a new RDD by first applying a function to all elements of this RDD, and then flattening the results

flatmap

Return a new RDD by first applying a function to all elements of this RDD, and then flattening the results

```
x = sc.parallelize( [1,2,3])
y = x.flatMap(lambda n: [n, n*100, 42])
print(x.collect())
print(y.collect())
```

```
x: [1, 2, 3]

y: [1, 100, 42, 2, 200, 42, 3,
   300, 42]
```

RDD - Transformations

* map vs. flatMap
  * map is a 1:1 transformation
  * flatMap is a 1:N transformation but then the N elements are flattened out

groupBy

RDD: **x**

4 items in RDD

groupBy

RDD: x

RDD: y

groupBy

groupBy

groupBy

RDD: x
RDD: y
James
Anna
Fred
John
'J'
emits
A
["Anna"]
F
["Fred"]
J
["John", "James"]

groupBy

groupBy(*f*, *numPartitions=None*)

Group the data in the original RDD. Create pairs where the key is the output of a user function, and the value is all items for which the function yields this key.

```
x = sc.parallelize(     ["John", "Fred", "Anna", "James”])
y = x.groupBy(lambda w:  w.charAt(0))
sorted([(x, sorted(y)) for (x, y) in y])
```

```
x:  ['John', 'Fred', 'Anna', 'James']

y:  [('A',['Anna']),('J',['John','James']),('F',[
    'Fred'])]
```

groupByKey

Pair
RDD: **x**

5 items in RDD

groupByKey

Pair
RDD: **x**

RDD: **y**

groupByKey

Pair RDD:

x

RDD: y

groupByKey

```
groupByKey(numPartitions=None)
```

Group the values for each key in the original RDD. Create a new pair where the original key corresponds to this collected group of values.

```
>>> rdd = sc.parallelize([("a", 1), ("b", 1), ("a", 1)])
>>> sorted(rdd.groupByKey().mapValues(len).collect())
>>> sorted(rdd.groupByKey().mapValues(list).collect())
```

```
[('a', 2), ('b', 1)]
[('a', [1, 1]), ('b', [1])]
```

ReduceByKey

GroupByKey

(a, 1)
(b, 1)
(a, 1)
(a, 1)
(b, 1)
(b, 1)
(a, 1)
(a, 1)
(a, 1)
(b,1)
(b, 1)
a
b
(a, 1)
(a, 1)
(a, 1)
(a, 1)
(a, 1)
(a, 1)
(a,6)
(b, 1)
(b, 1)
(b, 1)
(b, 1)
(b, 1)
(b,5)

ReduceByKey vs GroupBykey

```
words = ["one", "two", "two", "three", "three", "three"]
wordPairsRDD = spark.sparkContext.parallelize(words).map(lambda
word: (word, 1))

wordCountsWithReduce = wordPairsRDD.reduceByKey(lambda a,b:a+b)
 print(wordCountsWithReduce.collect())

wordCountsWithGroup = wordPairsRDD.groupByKey()
wordCounts=wordCountsWithGroup.map(lambda t:(t[0], len(t[1])))
print(wordCounts.collect())
```

RDD - Transformations

* reduce and reduceByKey are different
  * reduce is an action
    * Aggregation over all elements of a dataset
  * reduceByKey is a transformation
    * Aggregation over all values associated with a key

*union

union

union(*otherRDD*)

Return a new RDD containing all items from two original RDDs. Duplicates are *not* culled.

```
x = sc.parallelize([1,2,3], 2)
y = sc.parallelize([3,4], 1)
z = x.union(y)
print(z.glom().collect())
```

```
x: [1, 2, 3]

y: [3, 4]

z: [[1], [2, 3], [3, 4]]
```

*join

RDD: x

RDD: y

*join

*join

*join

Return a new RDD containing all pairs of elements having the same key in the original RDDs

```
x = sc.parallelize([("a", 1), ("b", 2)])
y = sc.parallelize([("a", 3), ("a", 4), ("b", 5)])
z = x.join(y)
print(z.collect())
```

```
x: [("a", 1), ("b", 2)]

y: [("a", 3), ("a", 4), ("b", 5)]

z: [('a', (1, 3)), ('a', (1, 4)),
   ('b', (2, 5))]
```

*distinct

RDD: **x**

*distinct

RDD: x

RDD: y

*distinct

RDD: x

RDD: y

distinct

distinct(*numPartitions=None*)

Return a new RDD containing distinct items from the original RDD (omitting all duplicates)

```
x = sc.parallelize([1,2,3,3,4])
y = x.distinct()

print(y.collect())
```

```
x [1, 2, 3, 3, 4]
:
y [1, 2, 3, 4]
:
```

Actions

Core Operations

vs

distributed

occurs across the cluster

driver

result must fit in driver JVM

getNumPartitions

getNumPartitions

getNumPartitions()

Return the number of partitions in RDD

```
x = sc.parallelize([1,2,3], 2)
y = x.getNumPartitions()

print(x.glom().collect())
print(y)
```

```
x: [[1], [2, 3]]

y: 2
```

collect

collect

collect()

Return all items in the RDD to the driver in a single list

```
x = sc.parallelize([1,2,3], 2)
y = x.collect()

print(x.glom().collect())
print(y)
```

```
x: [[1], [2, 3]]

y: [1, 2, 3]
```

*reduce

*reduce

*reduce

reduce

reduce(*f*)

Aggregate all the elements of the RDD by applying a user function pairwise to elements and partial results, and returns a result to the driver

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

print(x.collect())
print(y.collect())
```

```
x: [1, 2, 3, 4]

y: 10
```

countByKey

{'A': 1, 'J': 2, 'F': 1}

countByKey

countByKey()

Return a map of keys and counts of their occurrences in the RDD

```
x =spark.sparkContext.parallelize(
[('J',"James"),('F',"Fred"),('A',"Anna"),('J',"John")])

y = x.countByKey()
println(y.collect())
```

```
x:[('J', 'James'), ('F','Fred'),
    ('A','Anna'), ('J','John')]

y:{'A': 1, 'J': 2, 'F': 1}
```

saveAsTextFile

saveAsTextFile

saveAsTextFile(*path, compressionCodecClass=None*)

Save the RDD to the filesystem indicated in the path

```
x = sc.parallelize([2,4,1])
x.saveAsTextFile("/temp/demo")

y = sc.textFile("/temp/demo")
print(y.collect())
```

```
x: [2, 4, 1]

y: ['2', '4', '1']
```

Creating RDD(With SparkSession)

```
import org.apache.spark.sql.SparkSession
object SparkSessionExample {
 def main(args:Array[String]){
  val   spark =  SparkSession.builder
                                .appName("SparkExample")
                                .master("local")
                                .getOrCreate()

  val dataRDD = spark.read.csv("input.txt").rdd
     val dataMap = dataRDD.map { x => (x,1) }
  dataMap.foreach { println  }
  }
}
```

Creating RDD(Parallelize)

```
from pyspark.sql import SparkSession
 spark =  SparkSession.builder
                                        .appName("SparkExample")
                                        .master("local")
                                        .getOrCreate()
data=spark.sparkContext.parallelize([("sun",1),("mon",2),("tue",3),
("wed",4),("thus",5)])
val sorted = data.sortByKey(False)
sorted.foreach(print)
   }
}
```

Lab Exercise 01

RDD – PairRDD Transformations

* Following apply only when datasets are organized as key-value pairs
  * groupByKey
  * reduceByKey
  * aggregateByKey
  * foldByKey
  * combineByKey
  * sortByKey
  * join
  * cogroup

RDD – PairRDD Transformations

* reduceByKey
  * Apply an aggregation function on values of each key
  * Runs aggregation locally as well – "combiner"
    * Aggregation function must be commutative and associative
  * Results in a shuffle using hash partitioner
    * If the parent is not already hash partitioned

RDD – PairRDD Transformations

* aggregateByKey – RDD[K,V] -> RDD[K,U]
  * Generic aggregator that allows you to specify
    * A “zero” value
    * A “combiner” – (U, V) -> U
      * Combine values within a partition
    * A “reducer” – (U, U) -> U
      * Reduce values across partitions
    * Useful when you need to return a value type (U) different than the value type (V) of the source RDD

RDD – PairRDD Transformations

* groupByKey
  * Groups all values corresponding to a key
  * No ordering of values – maybe different each run
  * Key and corresponding list of values must fit in memory on a single node
  * Results in a shuffle
  * Could be expensive if all you need to do is run an aggregation function on grouped values
    * aggregateByKey and reduceByKey
    * No "combine" functionality
  * Can be used in cases where the aggregation function is not commutative and associative

RDD – PairRDD Transformations

* sortByKey
  * Actually part of OrderedRDD
  * Global sort using range partitioning
  * Results in a shuffle
* join, leftOuterJoin, rightOuterJoin, fullOuterJoin
  * Join one dataset with another when they have a common key
* cogroup
  * Group values from each RDD separately and associate with key
  * Joins internally implemented using cogroup
* repartitionAndSortWithinPartitions
  * Pushes sorting down to shuffle
  * More efficient than repartition and then sort
* partitionBy

RDD - Actions

* reduce
  * Simple aggregation
  * Runs “Combiner”
    * Aggregation function must be commutative and associative
* collect
  * Return all elements to the driver as a sequence
* count
  * Returns number of elements in RDD
* countByKey on PairRDDs
  * Returns a count of values for each key

RDD - Actions

* saveAsTextFile
  * Call toString on each element of the RDD and store each partition to a part-n file
* saveAsSequenceFile
  * For PairRDDs
  * Write as Hadoop SequenceFiles
* saveAsObjectFile
  * Write as Java Serialized objects

RDD - Actions

* take
  * Return n smallest elements of the RDD to the driver
* takeOrdered
  * Take n smallest elements from each partition after ordering each partition
  * Take n smallest elements from the result set after reordering it
* takeSample
  * Return n random samples from the RDD
  * Sampling can be done with or without replacement
* top
  * Return n largest elements of the RDD to the driver
* Custom ordering can be specified for each of these operations

RDD – Types of RDDs

- HadoopRDD
- FilteredRDD
- MappedRDD
- PairRDD
- ShuffledRDD
- UnionRDD
- PythonRDD

- DoubleRDD
- JdbcRDD
- JsonRDD
- VertexRDD
- EdgeRDD

- CassandraRDD *(DataStax)*
- GeoRDD *(ESRI)*
- EsSpark *(ElasticSearch)*

* Each data source that wants to control how data is partitioned typically requires a new type of RDD to be implemented

Learning Objectives

* Understand Spark Cluster Architecture
* Understand RDDs
* **Understand Spark Application Execution**

Spark – Application Execution

* Transformations build the DAG
* Actions result in execution of the DAG constructed so far
* But first we need to figure out an optimal execution plan
  * Narrow dependencies can be pipelined together
  * Wide dependencies mean that all partitions of the parent RDD(s) need to be computed before the execution can proceed

Spark – DAG Scheduling

```
rdd1 = sc.textFile("file1.txt").filter(…).map(…).groupByKey(…)
rdd2 = sc.textFile("file2.txt").map(…).reduceByKey(…)
rdd3 = rdd1.map(…)
rdd4 = rdd2.filter(…)
rdd5 = rdd3.join(rdd4)
rdd5.saveAsTextFile("staging.txt")
rdd6 = rdd5.map(…).reduceByKey(…)
rdd7 = rdd6.filter(…).map(…).reduceByKey(…)
rdd8 = rdd7.map(…).combineByKey(…)
rdd8.saveAsTextFile("report.txt")
```

Spark – Task Scheduling

- Each stage is executed as a set of tasks
  - Number of tasks = number of partitions
- Stages operating on independent RDDs can be executed in parallel
- Task scheduling happens on the driver
- Each Task is bound to a partition
- Tasks are executed on the executors allocated to this application
- Executors are selected by the driver based on partition location awareness (data locality)
- An executor can run multiple tasks as multiple threads in the JVM each working on a different partition

Number of Tasks >>> Number of Cores

* Big Data = lots of partitions
* Lots of partitions = lots of tasks
* Tasks are launched in waves
* Number of tasks in each wave = number of cores available to the application

Spark – Location Awareness

* For Base RDDs, the driver gets location information from data source
  * This makes sense only when Spark is co-located with the distributed data store
* For derived RDDs, the driver keeps track of which executors have which partitions
* Spark will wait for a configurable amount of time for data-local scheduling
  * If the executor that has the partition is busy, then it allocates another executor
  * Network traffic is incurred

RDD - Partitioning

* RDD lineage information includes partitioning information as well
  * Partitioner
  * Number of Partitions
* Spark configuration parameter *spark.default.parallelism* controls the number of partitions in some cases
  * Can be set by user. If not, defaults to number of cores allocated to this application

RDD - Partitioning

* Base RDDs
  * RDDs created using *parallelize* are controlled by *spark.default.parallelism*
  * Partitioning depends on Data Source
    * HDFS files – number of partitions = number of HDFS blocks
    * Parquet – number of partitions = number of part files
    * Text files – partitioned based on block size defined by hadoop configuration, typically means 64/128MB blocks
  * Default minimum partitions is 2

RDD - Partitioning

* Derived RDDs
  * For narrow dependencies and wide dependencies generated by unary operators
    * Number of partitions and the partitioner carry forward
  * For wide dependencies generated by n-ary operators
    * Most transformations that cause a shuffle take an argument called *numPartitions/numTasks* so programmers can control this
    * If not set, then if all parents are copartitioned, use that partition count
    * Otherwise, use sum(parent partition count)

RDD - Partitioning

* Available partitioners
  * Range
  * Hash
  * Custom partitioners can be specified
* Transformations *coalesce* and *repartition* let programmers resize RDDs

RDD – Level of Parallelism

* Are you using the cluster resources efficiently?
  * Level of parallelism
    * How many parallel threads of execution (cores) do I have in the entire cluster?
    * How many of them are available to my application?
    * What size partitions make sure that each task spends a reasonable amount of time processing?
      * Make sure it takes significantly more time to process a partition than it takes to schedule a task
        * Task scheduling takes ~10-20ms
        * Task processing time should be >50ms, ideally ~200ms
      * Aggregation typically reduces that amount of data in each partition
        * May need to *repartition* to make sure each task has enough data to process

Fault Tolerance

* Executor failure
  * Driver asks the Cluster Manager to allocate new Executor
  * Driver uses the DAG to recreate lost partitions on the new Executor
* Driver failure
  * Driver failure can be catastrophic
  * Run Driver using --cluster and --supervise
* Failure of Cluster Manager components is handled by each Cluster Manager differently
  * Worker failures are handled by masters
  * Master failures are handled by having standby masters running and electing a new one using ZooKeeper

Summary

* Spark is a distributed compute engine for data-parallel applications
* Spark divides the compute environment into three components – the driver, the cluster manager and the executor
* RDD is Spark's abstraction for distributed memory
* RDDs support two types of operations – transformations and actions
* Applying a series of transformations results in an RDD DAG
* Actions result in this DAG being evaluated and the RDDs in the DAG being materialized
* Narrow dependency – each parent partition feeds data to one child partition
* Wide dependency – each parent partition feeds data to multiple child partitions
* DAG construction and scheduling happens on the driver
* Task scheduling also happens on the driver
* Task execution is farmed out to the executors

End of Spark core concepts