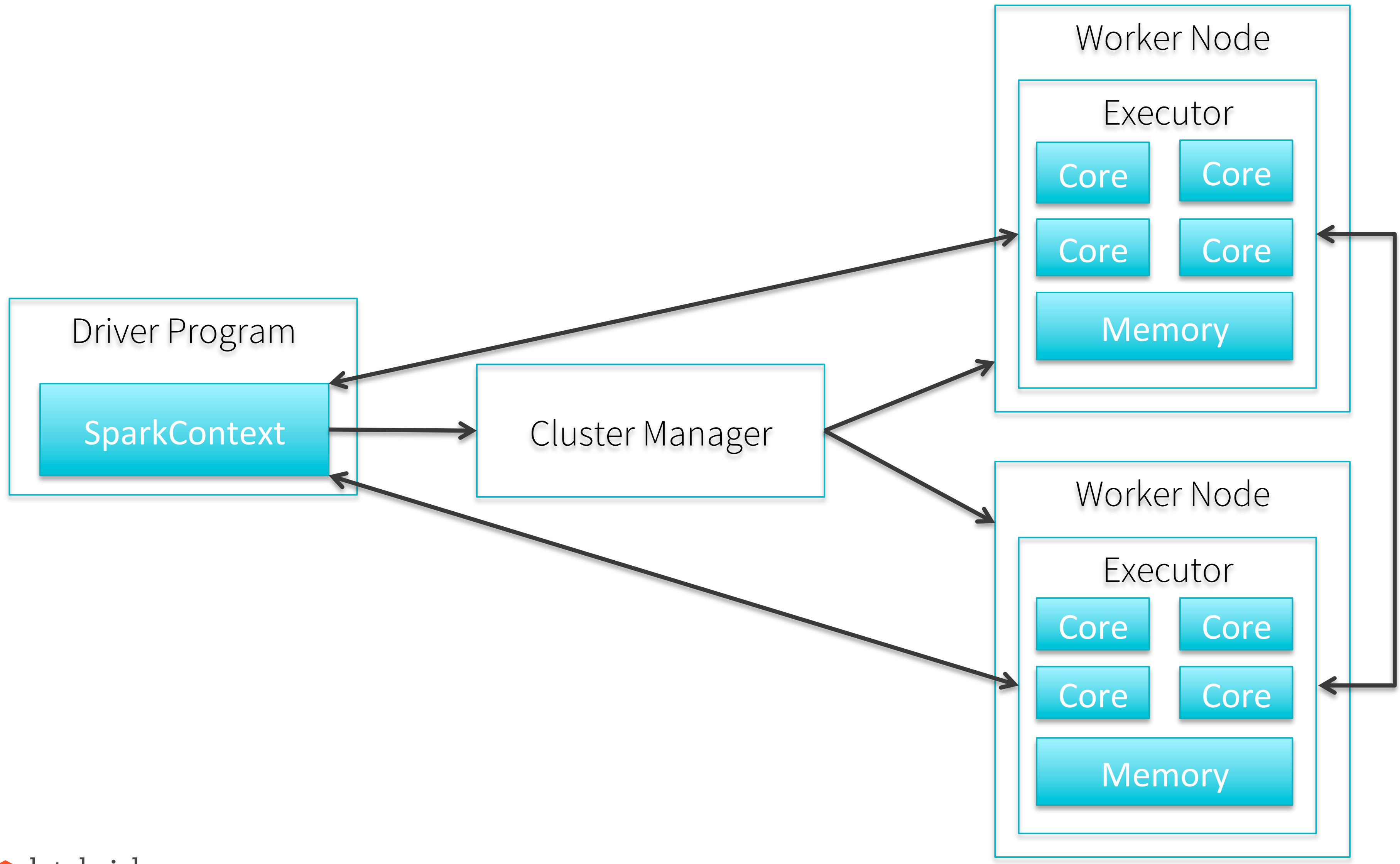
Spark Core Concepts

Learning Objectives

- **Understand Spark Cluster Architecture**
- Understand RDDs
- Understand Spark Application Execution
- Understand Data Partitioning

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
- Spark Context
  - 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 - REPL

- Spark comes with REPLs for Scala, Python, R and SQL

```
ubuntu@ip-10-0-53-24: ~
ubuntu@ip-10-0-53-24:~$ dse spark
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 0.9.1
      /_/

Using Scala version 2.10.3 (Java HotSpot(TM) 64-Bit Server VM, Java 1.7.0_51)
Type in expressions to have them evaluated.
Type :help for more information.
Creating SparkContext...
Created spark context..
Spark context available as sc.
Type in expressions to have them evaluated.
Type :help for more information.

scala> val myRDD = sc.cassandraTable("tinykeyspace", "keyvaluetable")
myRDD: com.datastax.bdp.spark.CassandraRDD[com.datastax.bdp.spark.CassandraRow] = Cassan
draRDD[0] at RDD at CassandraRDD.scala:32

scala> myRDD.count()
res2: Long = 5

scala>
```

Learning Objectives

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

RDD - Definition

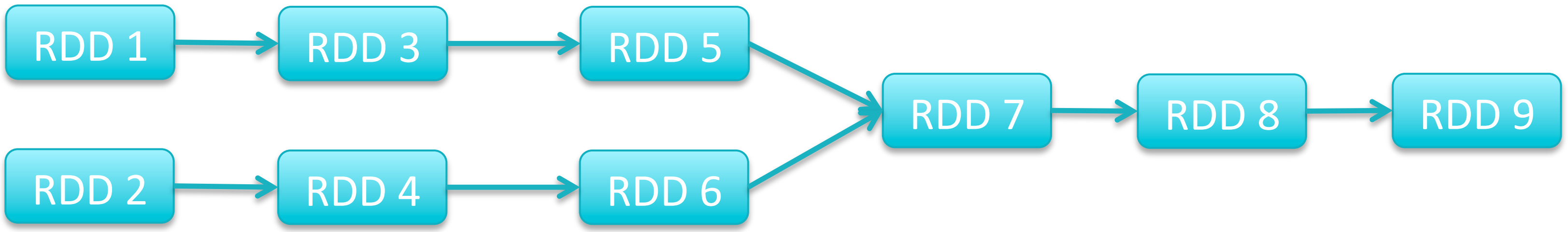
- Spark's abstraction for distributed memory
  - Read-only collection of objects distributed across a cluster
  - Data is divided into chunks called *partitions*
- Resilient
  - Partitions, if lost due to node failure, can be recreated
- Distributed
  - Partitions distributed across the cluster
  - So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

RDD - Properties

- Immutable
  - Cannot be changed once created
- 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
- Encapsulate metadata required to transform one RDD into another

RDD - Lineage

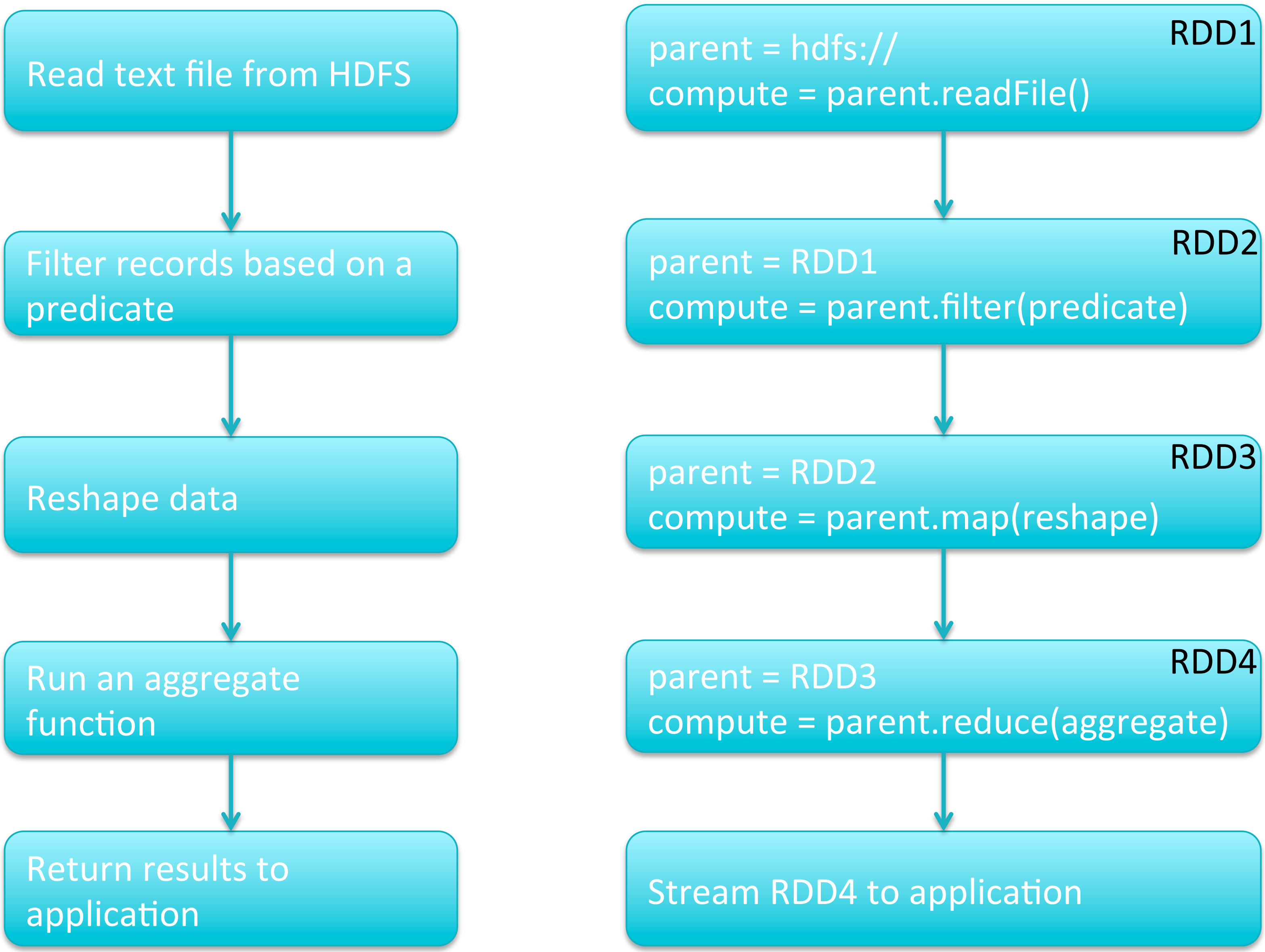
- Lineage
  - Each transformation encodes instructions needed to create a child RDD from parent RDD(s)
  - A series of such transformations construct a lineage
    - More formally, the series of transformations form a Directed Acyclic Graph (DAG)
  - Allows for portions of RDD to be reconstructed when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



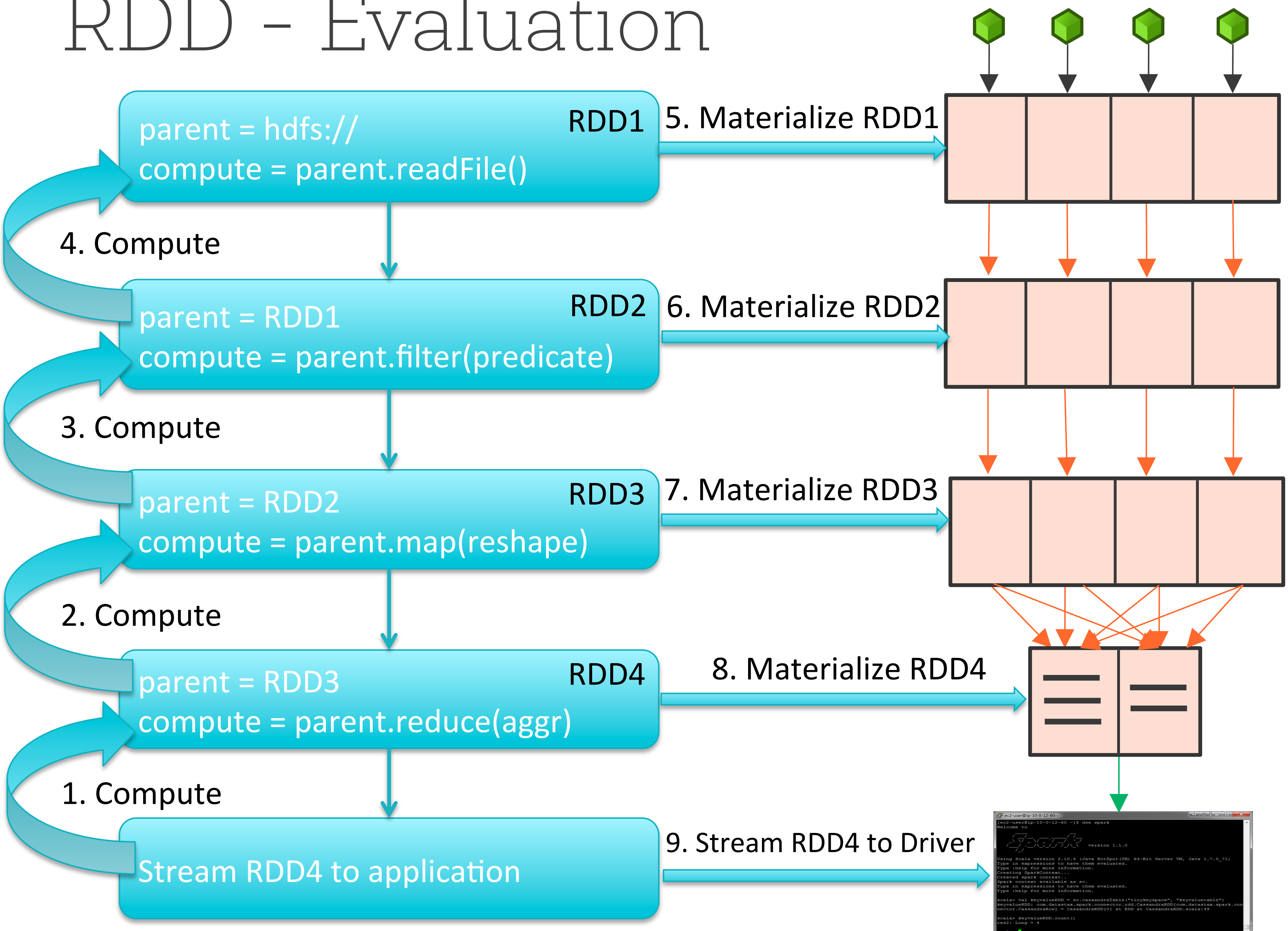
RDD - Operations

- 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

RDD1
parent = hdfs://
compute = parent.readFile()

↓

RDD2
parent = RDD1
compute = parent.filter(predicate)

↓

RDD3
parent = RDD2
compute = parent.map(reshape)

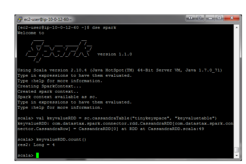
↓

RDD4
parent = RDD3
compute = parent.reduce(aggregate)

↓

Stream RDD4 to application

RDD data garbage collected
once the Action is complete



Driver

databricks

RDD - Workflow

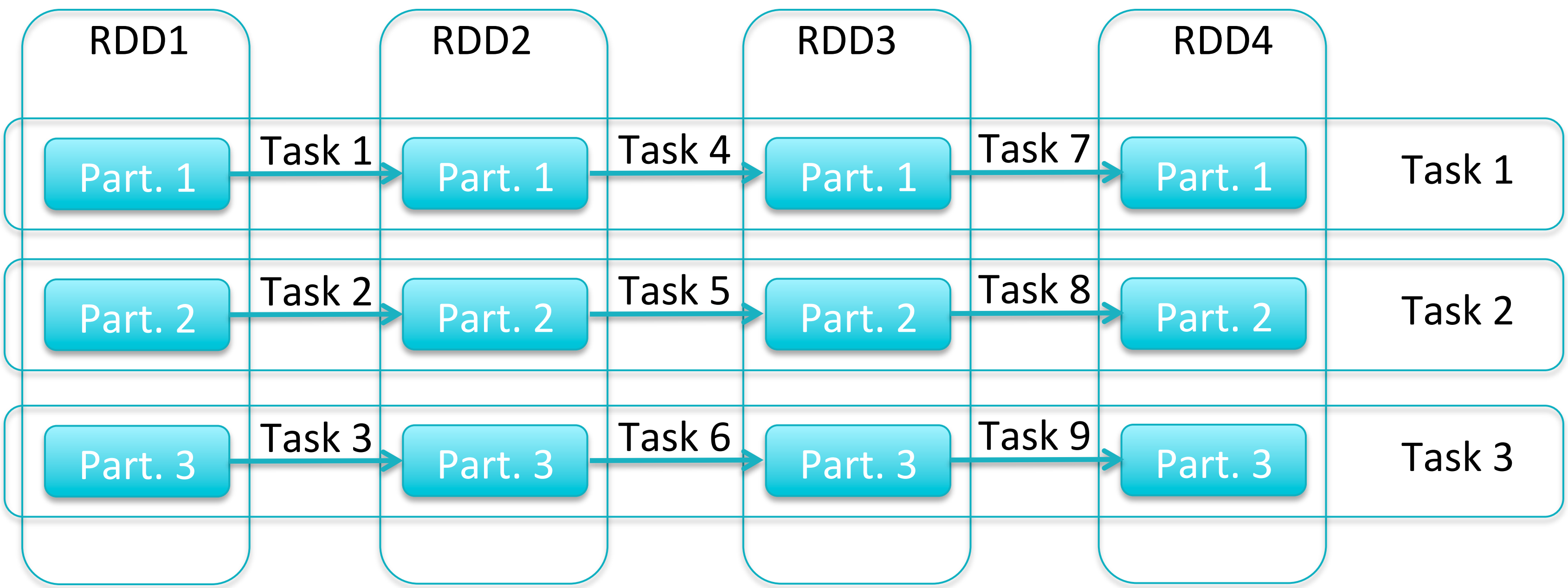
- Typical minimal workflow for manipulating data
  - Read
    - Construct "Base" RDD by reading data from stable storage
  - Transform
    - Apply a series of transformations constructing the Lineage DAG
  - Write (Call an Action)
    - Evaluate the DAG materializing the RDDs
    - Write the result back 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

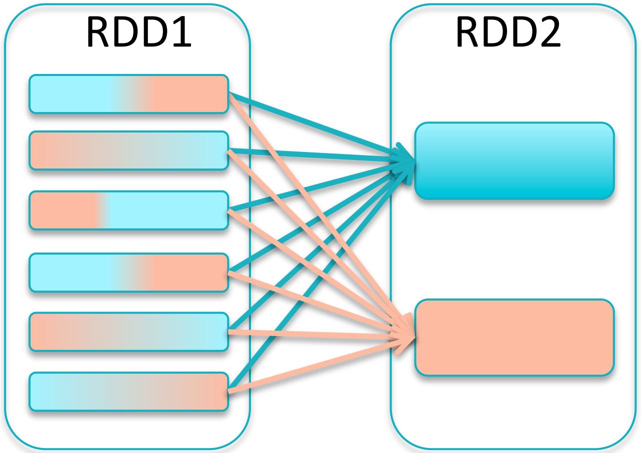
RDD – Dependencies

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



RDD – Dependencies

- Wide
  - Each parent partition contributes data to multiple child partitions
  - Example: Aggregation operators like group by
  - Requires a *shuffle*
    - An expensive operation in a distributed system
    - Limits performance of the overall application



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
  - 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

RDD API Visual Guide and Lab

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
- 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

RDD – Partition Control Transformations

- coalesce
  - Used to handle data skew across partitions
  - Can increase or decrease number of partitions
  - Shuffle=false (default) causes no shuffle
    - Used to merge smaller partitions into bigger ones
  - Shuffle=true causes shuffle
    - Used to breakup large partitions into smaller ones using hash partitioner
- repartition
  - Implemented as *coalesce(shuffle=true)*

RDD - Transformations

- filter
  - Filter RDDs using a selector function
- filterByRange
  - Filter for elements within a range
  - Efficient if RDD already partitioned by RangePartitioner, otherwise same as filter
- glom
  - Merge all elements within each partition into an array
- keyBy
  - Create key-value pair RDD from a regular RDD
- zip
  - Create key-value pair RDD with key from first RDD and value from second RDD
  - Variations allow for
    - Running a function on the key-value pair
    - Associating the global element index with each element
    - Associating a generated unique ID with each element

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

- Aggregation function dimensions
  - [K,V] -> [K,V] vs. [K,V] -> [K,W]
    - Change the type of value during aggregation
  - Whether to do a map-side combine or not
  - Whether the combiner function is the same as the reducer function or not
  - Whether an identity value is provided or not
  - Whether to just group things together or actually run an aggregation function

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
  - Could be expensive if all you need to do is run an aggregation function on grouped values

RDD – PairRDD Transformations

- combineByKey
  - Generic aggregator all other functions are implemented as
- sortByKey
  - Global sort using range partitioning

RDD – PairRDD Transformations

- 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
- partitionBy
  - Use a custom partitioner

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

RDD Fundamentals Lab

Learning Objectives

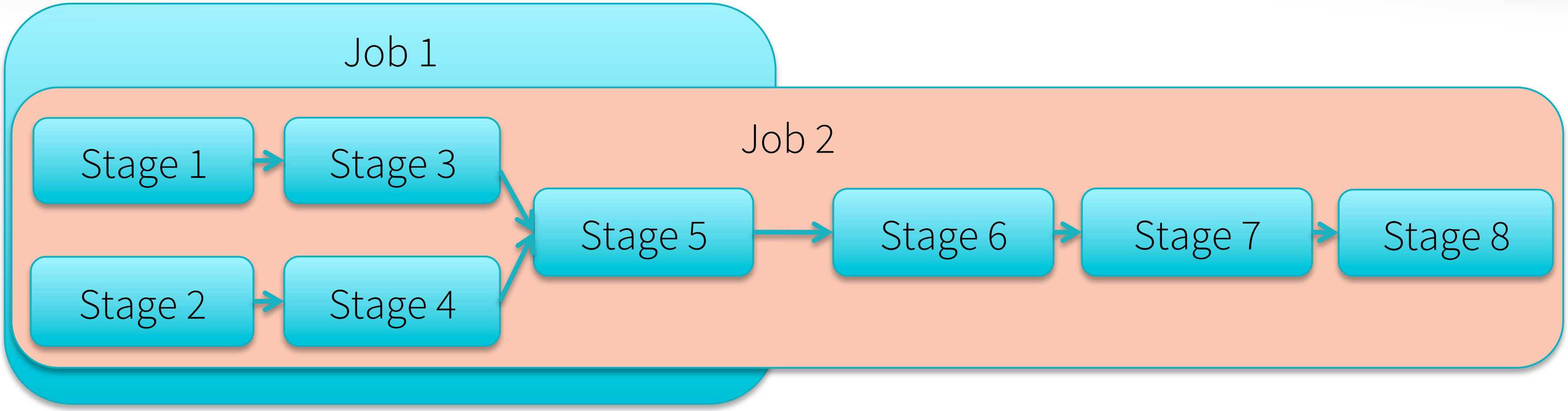
- Understand Spark Cluster Architecture
- Understand RDDs
- **Understand Spark Application Execution**
- Understand Data Partitioning

Spark – Application Execution

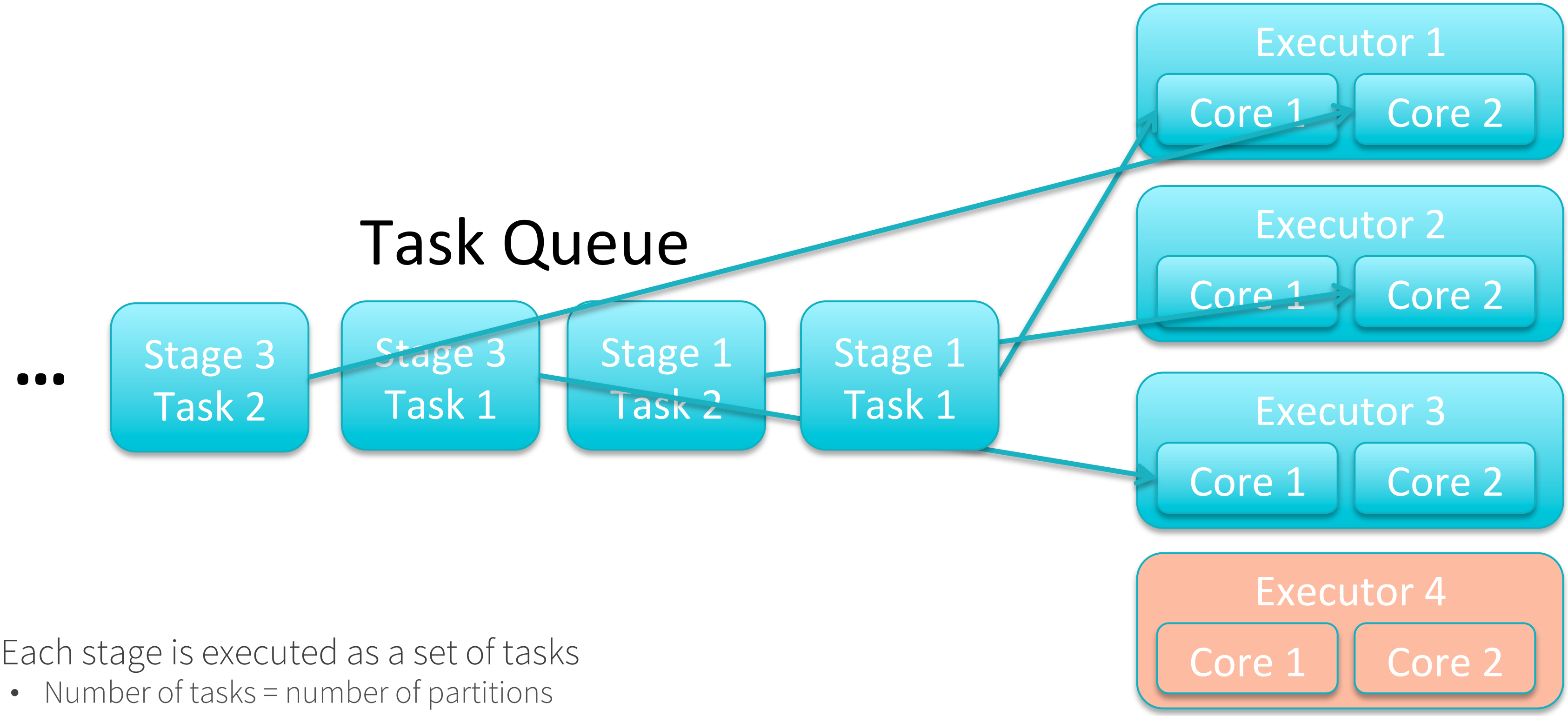
- Transformations build the DAG
- Actions result in execution of the DAG constructed so far
- But first we need to figure out an optimal execution plan
  - Narrow dependencies can be pipelined together
  - Wide dependencies mean that all partitions of the parent RDD(s) need to be computed before the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

Learning Objectives

- Understand Spark Cluster Architecture
- Understand RDDs
- Understand Spark Application Execution
- **Understand Data Partitioning**

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

RDD – Custom Partitioners

- Custom partitioners can be implemented
- If a data store has a custom partitioning mechanism, a custom partitioner can be used to implement data-local save
- For Example, Cassandra has a specific way of partitioning data
  - If you want to save an RDD to Cassandra, repartitioning using Cassandra token ranges ensure data-local save

Partitioning Lab/Demo

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

RDD – Shuffle managers

- hash
  - Default Spark < 1.2.0
  - Each mapper creates a number of files equal to the number of reducers
  - Pros
    - Fast – no sorting or hash table maintenance
    - No memory overhead for sorting
    - Positive I/O characteristics
      - Map output doesn't have to fit memory
      - Each file written/read exactly once
  - Cons
    - Potential for creating too many files thereby approaching file system limits
    - May result in random I/O which can be inefficient

RDD – Shuffle managers

- sort
  - Default Spark >= 1.2.0
  - Assigns each reducer an ID, assigns reducer ID to each record and sorts data based on reducer ID
  - Stores all data in a single file indexed by reducer ID
  - Falls back on hash if the number of reducers < a threshold
  - Pros
    - More efficient I/O because it is sequential
      - When map output fits memory
    - Does not create too many files like hash shuffle manager
  - Cons
    - Sort penalty
    - I/O performance degradation when map outputs do not fit memory
      - Spills need to be managed

RDD – Shuffle managers

- tungsten-sort
  - Available in Spark >=1.4.0
  - Operate directly on sun.misc.Unsafe memory
  - NUMA-aware data structures and algorithms
  - May use off-heap storage

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

Review Questions

- What is Spark?
- What is an RDD?
- What are the properties of an RDD?
- What kinds of operations are supported on an RDD?
- What is Lazy Evaluation?
- What are narrow and wide dependencies?
- What are the components of the Spark Compute Engine?
- What are the responsibilities of the Driver?
- What is a Job, Stage, Task?
- What is the SparkContext?

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

Questions?

databricks™