CLOUDERA
Educational Services

Spark Application Performance Tuning with CDP

Version 20201026

202007

CLOUDERA
Educational Services

Introduction

Trademark Information

- The names and logos of Apache products mentioned in Cloudera training courses, including those listed below, are trademarks of the Apache Software Foundation

Apache Accumulo	Apache Hive	Apache Pig
Apache Avro	Apache Impala	Apache Ranger
Apache Ambari	Apache Kafka	Apache Sentry
Apache Atlas	Apache Knox	Apache Solr
Apache Bigtop	Apache Kudu	Apache Spark
Apache Crunch	Apache Lucene	Apache Sqoop
Apache Druid	Apache Mahout	Apache Storm
Apache Flume	Apache NiFi	Apache Tez
Apache Hadoop	Apache Oozie	Apache Tika
Apache HBase	Apache Parquet	Apache Zeppelin
Apache Catalog	Apache Phoenix	Apache Zookeeper

- All other product names, logos, and brands cited herein are the property of their respective owners

CLOUDERA
Educational Services

About This Course

About this Course

- This is not a beginning course in Spark
- We assume you can write Spark code in Python
  - Basic API knowledge
  - Basic concepts
- The focus is to improve Spark application performance
- Application design best practices and pitfalls

Agenda

Day 1	Day 2	Day 3	Optional (*)
Introduction	Dealing with Skewed Data	Pyspark Overhead and UDFs	Partition Processing
Spark Architecture	Catalyst and Tungsten Overview	Caching Data for Reuse	Broadcasting
Data Sources and Formats	Mitigating Spark Shuffles	WXM Introduction	Scheduling
Inferring Schemas	Partitioned and Bucketed Tables	What's New in Spark 3.0?	
	Improving Joins Performance		

(*) if time allows

CLOUDERA
Educational Services

Introductions

Introductions

- About your instructor
- About you
  - Currently, what do you do at your workplace?
  - What is your experience with database technologies, programming, and query languages?
  - What is your experience with Spark and Big Data?
  - What do you expect to gain from this course? What would you like to be able to do at the end that you cannot do now?

CLOUDERA
Educational Services

About Cloudera

About Cloudera

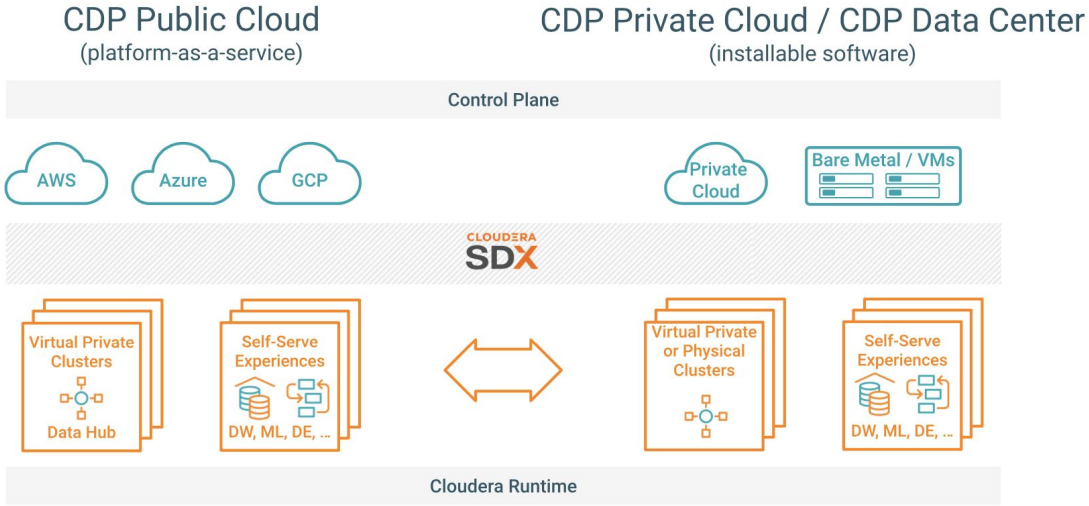
- Cloudera (founded 2008) and Hortonworks (founded 2011) merged in 2019
- The new Cloudera improves on the best of companies
  - Introduced the world's first Enterprise Data Cloud
  - Delivers an comprehensive platform for any data from the Edge to AI
  - Leads in training, certification, support, and consulting for data professionals
  - Remains committed to open source and open standards

CLOUDERA

THE ENTERPRISE DATA CLOUD COMPANY

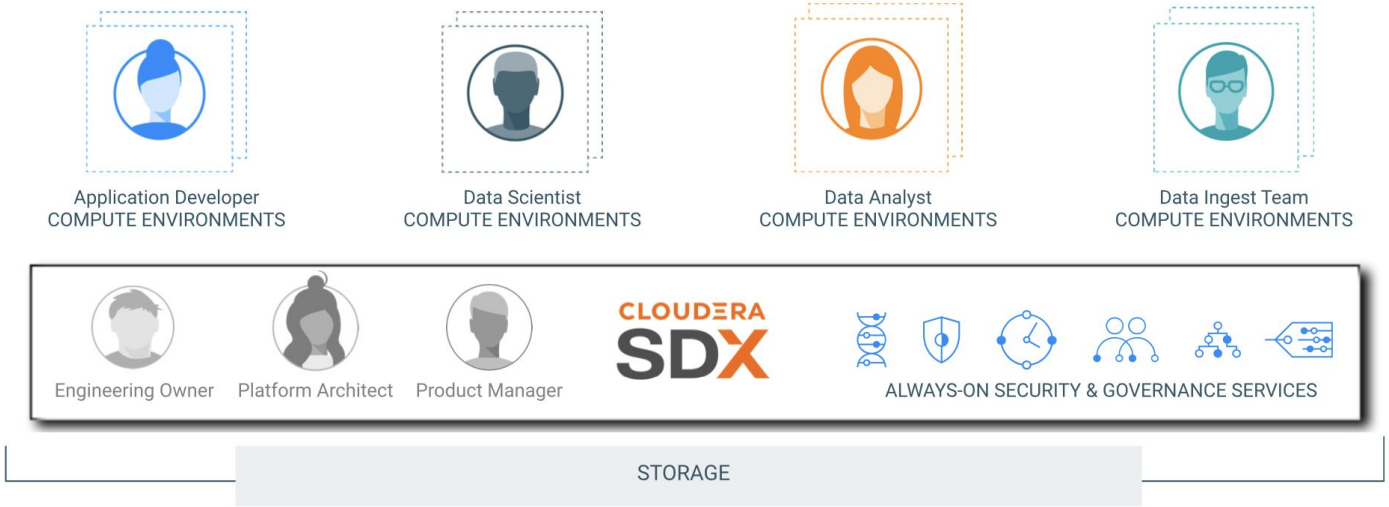
Cloudera Data Platform

- A suite of products to collect, curate, report, serve, and predict
  - Cloud native or bare metal deployment
  - Analytics from the Edge to AI
  - Powered by open source
  - Unified data control plane Shared Data Experience (SDX)



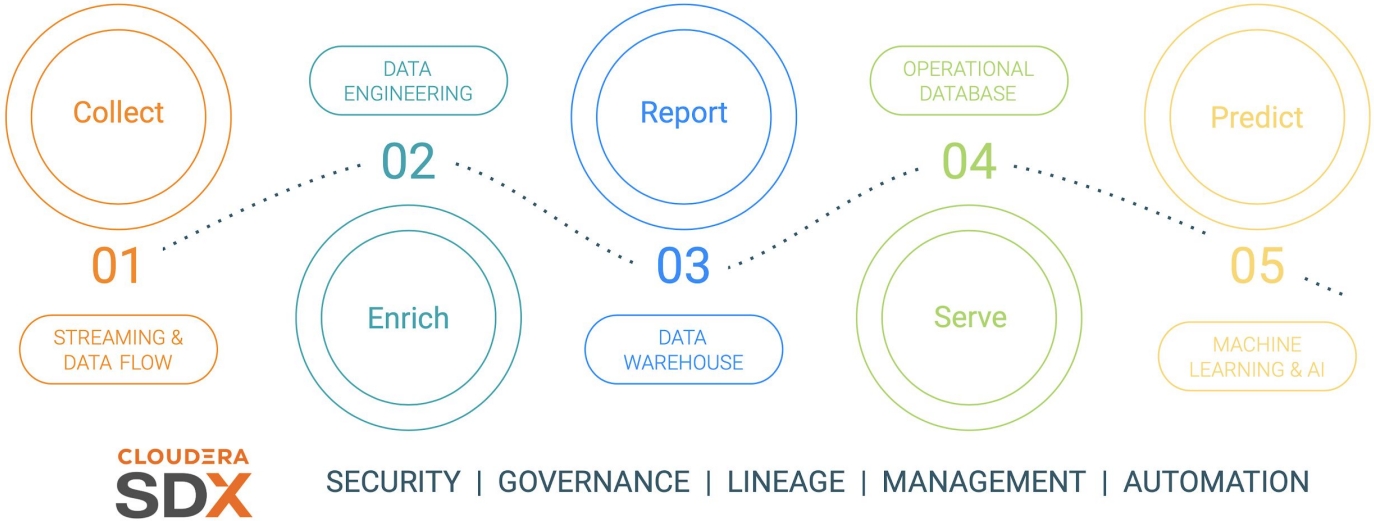
Cloudera Shared Data Experience (SDX)

- **Full data lifecycle:** Manages your data from ingestion to actionable insights
- **Unified security:** Protects sensitive data with consistent controls
- **Consistent governance:** Enables safe self-service access



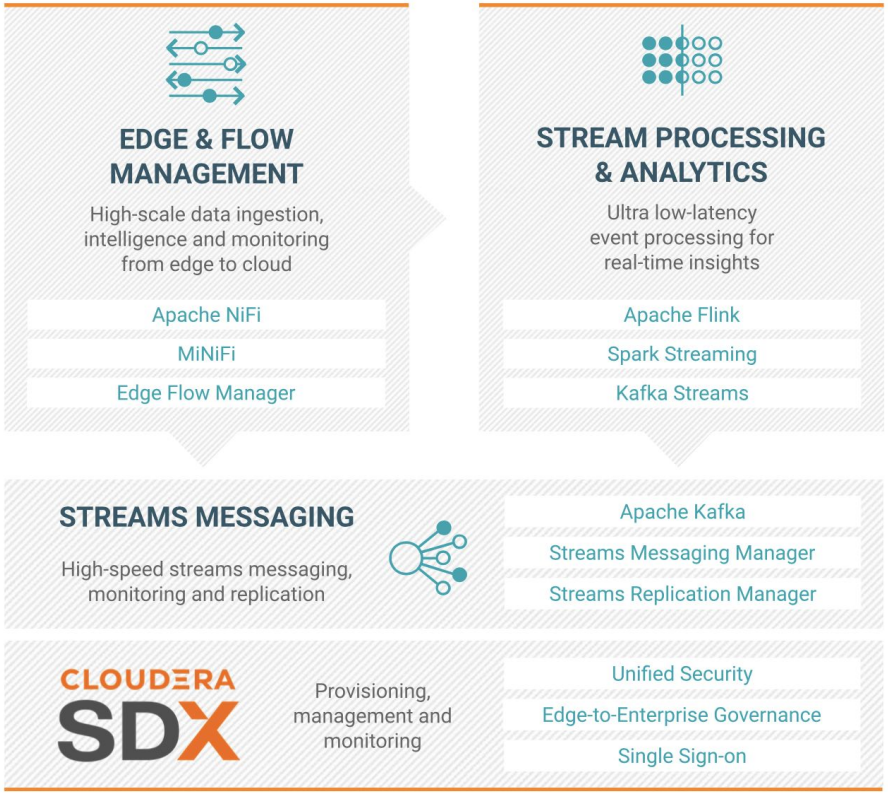
Self Serve Experiences

- Services customized for specific steps in the data lifecycle
- Emphasize productivity and ease of use
- Auto-scale compute resources to match changing demands
- Isolate compute resources to maintain workload performance



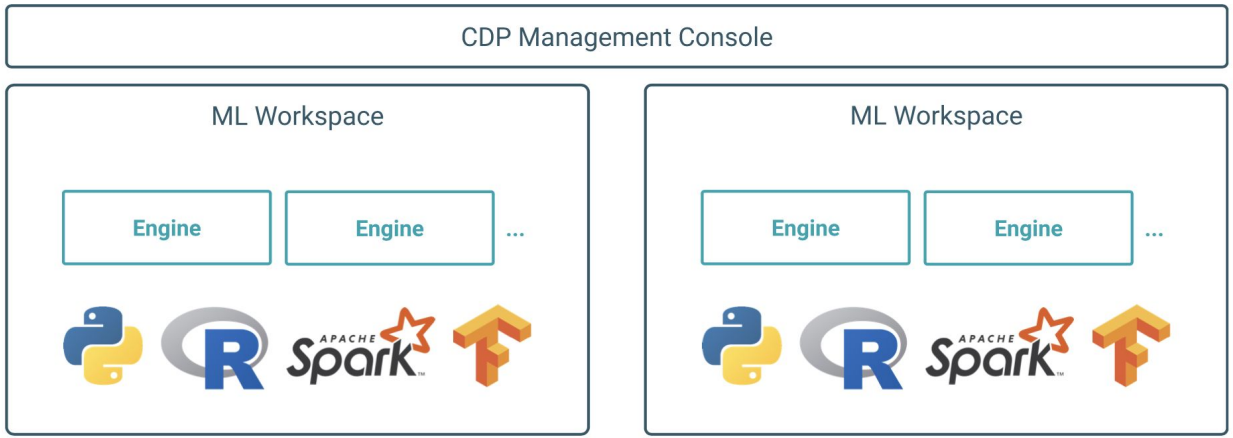
Cloudera DataFlow

- Data-in-motion Platform
- Reduces data integration development time
- Manages and secures your data from edge to enterprise



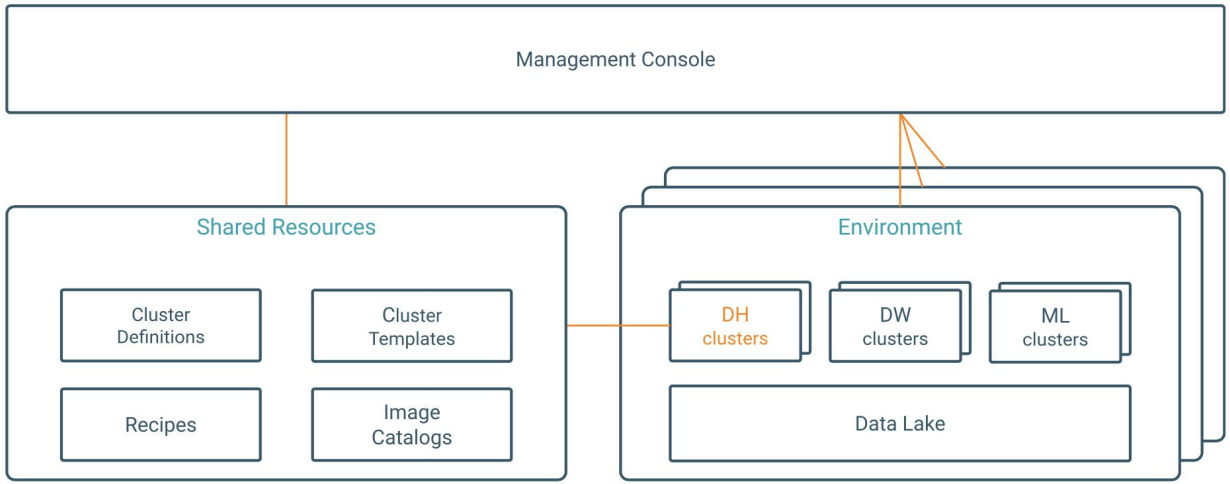
Cloudera Machine Learning

- **Cloud-native enterprise machine learning**
  - Fast, easy, and secure self-service data science in enterprise environments
  - Direct access to a secure cluster running Spark and other tools
  - Isolated environments for running Python, R, and Scala code
  - Teams, version control, collaboration, and project sharing



Cloudera Data Hub

- **Customize your own experience**
  - Integrated suite of analytic engines
  - Cloudera SDX applies consistent security and governance
  - Fueled by open source innovation



CLOUDERA
Educational Services

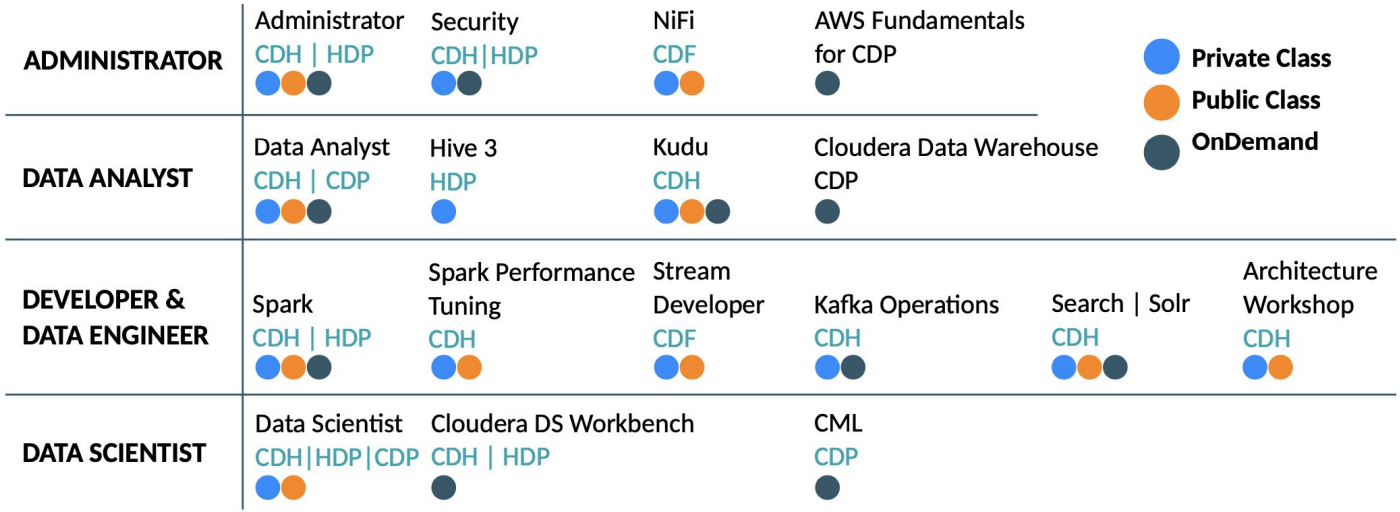
About Cloudera Educational Services

Cloudera Educational Services

- We offer a variety of ways to take our courses
  - Instructor-led, both in physical and virtual classrooms
    - Private and customized courses also available
  - Self-paced, through Cloudera OnDemand
- Courses for all kinds of data professionals
  - Executives and managers
  - Data scientists and machine learning specialists Data analysts
  - Developers and data engineers
  - System administrators
  - Security professionals

Cloudera Education Catalog

- A broad portfolio across multiple platforms
  - Not all courses shown here
  - See our website for the complete catalog



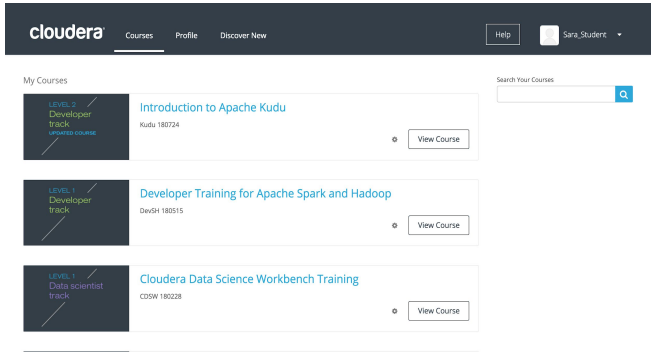
Cloudera On Demand

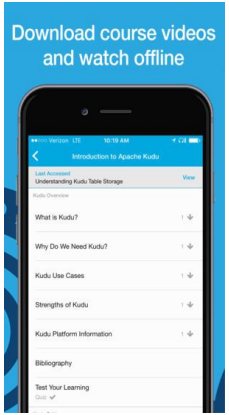
- Our OnDemand catalog includes
  - Exclusive OnDemand-only courses on security, Cloudera Data Science Workbench, and installing on Microsoft Azure
  - Courses for developers, data analysts, administrators, and data scientists, updated regularly
  - Free courses such as *Essentials* and *Cloudera Director* available to all with or without an OnDemand account
- Features include
  - Video lectures and demonstrations with searchable transcripts
  - Hands-on exercises through a browser-based virtual environment identical to the ones used in our instructor led classes
  - Discussion forums monitored by Cloudera course instructors
  - Searchable content within and across courses
- Purchase access to a library of courses or individual courses
- See the Cloudera OnDemand information page for more details or to make a purchase, or go directly to the OnDemand Course Catalog

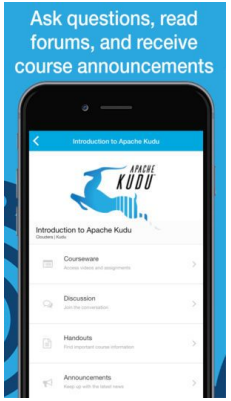
Accessing Cloudera On Demand

- Cloudera OnDemand subscribers can access their courses online through a web browser

- Cloudera OnDemand is also available through an IOS app
  - Search for “Cloudera OnDemand” in the IOS App Store







Cloudera Certification

- The leader in Apache Hadoop-based certification
- All Cloudera professional certifications favor hands-on, performance-based problems that require execution of a set of real-world tasks against a live, working cluster
- We offer two levels of certifications
  - Cloudera Certified Associate (CCA)
    To achieve CCA certification, you complete a set of core tasks on a working CDH cluster instead of guessing at multiple-choice questions
    - CCA Spark and Hadoop Developer
    - CCA Data Analyst
    - CCA Administrator and HDP Administrator Exam
  - Cloudera Certified Professional (CCP)
    - CCP Data Engineer
- For more information check our Preparing for Cloudera Certification free online course

CLOUDERA
Educational Services

Course Logistics

Logistics

Your instructor will give you details on how to access the course materials for the class

- Class start and finish time
- Lunch
- Breaks
- Restrooms
- Wi-Fi access
- Virtual machines

CLOUDERA
Educational Services

Methodology

Course Methodology

- We are going to spend most of the time writing and executing Pyspark code
  - Lots of demonstrations
  - Follow on exercises
  - Some slides, mostly diagrams
- We have deployed a CDP Private Cloud Base Edition cluster for each of you
  - Light – a single node
  - Sampled, smallish, data sets

Course Methodology

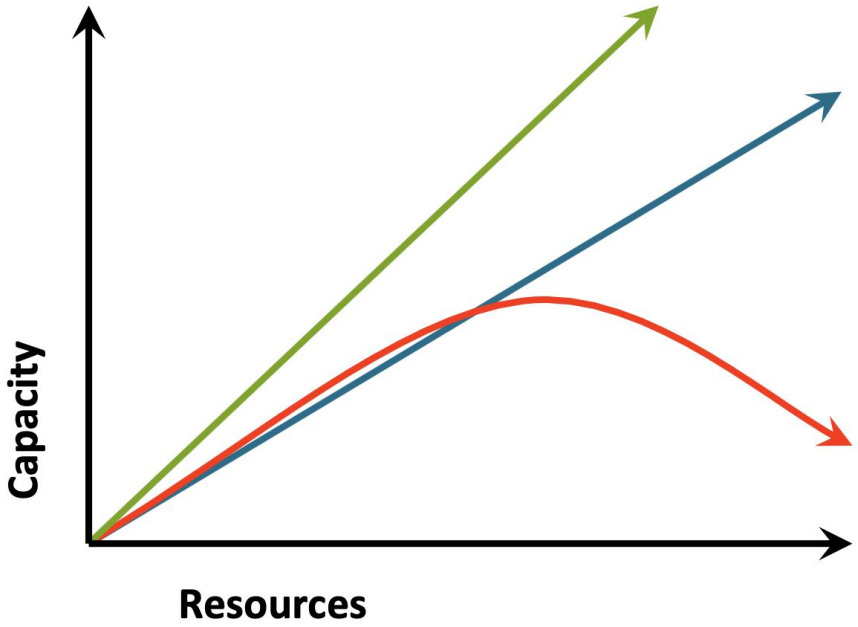
- Goal is to find and fix performance issues in a scaled down environment before they happen on Big Data in a hefty cluster
- For example, an ***inefficient*** join algorithm applied to
  - small sampled data sets takes a *second* running on a small cluster
  - terabytes of data takes *hours* using a large hefty cluster
- An ***efficient*** join algorithm applied to
  - small sampled data sets takes a *second* running on a small cluster
  - terabytes of data takes a *second* in a large hefty cluster by taking advantage of parallelism
- In short, we want to find solutions that *scale*

Scalability

- Scalability is the capability to increase hardware resources to yield an ideally linear increase in service capacity
- Additional load is accommodated by
  - adding additional hardware resources
  - not extensive modification of the application
- Hardware resources may be “scaled up” or “scaled out”

Scalability

- If we can add resources and have a linear increase in capacity without bounds, then we say the architecture *scales*



Metrics

- In the demonstrations and exercises we will use the metrics available in the Spark UI to compare approaches and focus on relative amounts of data and work, for example
  - was the data shuffled
  - if so, the amount of shuffled data
  - balanced partitions
  - cached vs. streamed and recomputed

CLOUDERA
Educational Services

Data Sets

Data Sets

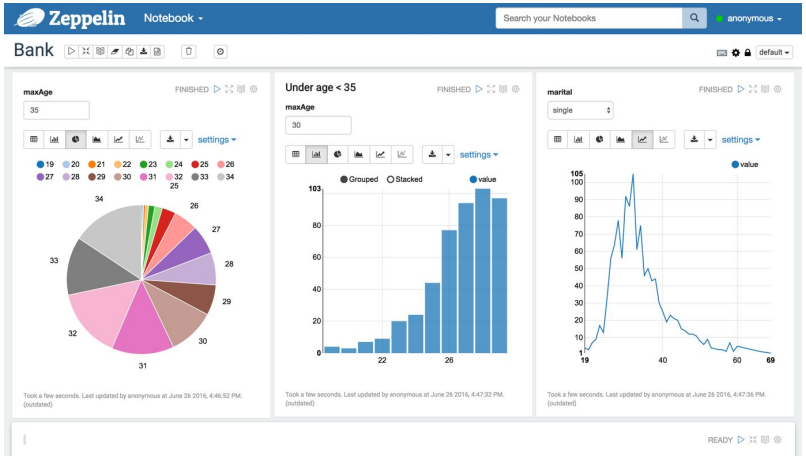
- The demonstrations and exercises use three data sets in various formats:
  - Ride sharing data
    - *rides, riders, drivers* – Ecommerce data
- – *weblogs, accounts* – Retail data
- – *customers, sales, stores*
- One exercise uses the complete works of William Shakespeare!

CLOUDERA
Educational Services

Tools

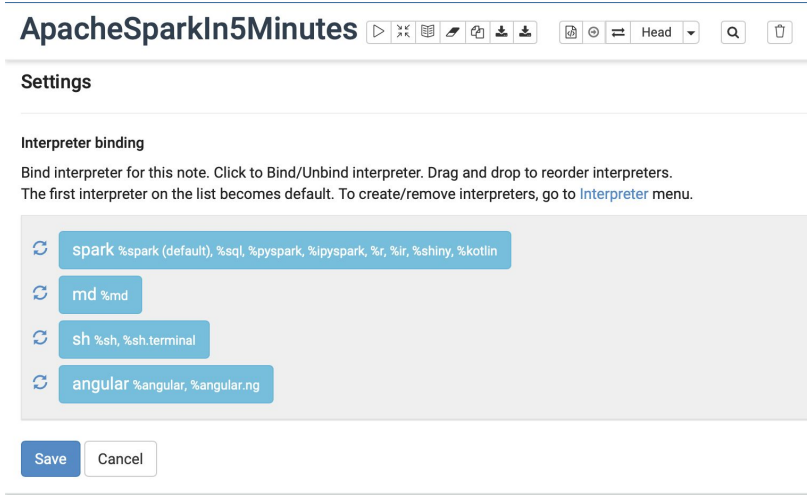
Apache Zeppelin

- An open source project from the Apache Software Foundation
  - 2013, NFLabs started the Zeppelin project
  - 2014-12-23, the Zeppelin project became incubation project in Apache Software Foundation
  - 2016-06-18, the Zeppelin project graduated incubation and became a Top Level Project in Apache Software Foundation
- A multi-purpose notebook system for
  - Data ingestion
  - Data discovery
  - Data analytics
  - Data visualization and collaboration



Anatomy of a Note

- A Zeppelin note is a sequence of paragraphs
- Each paragraph is bound to an interpreter
- Each note has a list of available interpreters which is a subset of all the interpreters installed on the Zeppelin server
- The first element of the list is the default interpreter for the note
- Each paragraph should start by specifying the interpreter to which it is bound unless it is the default

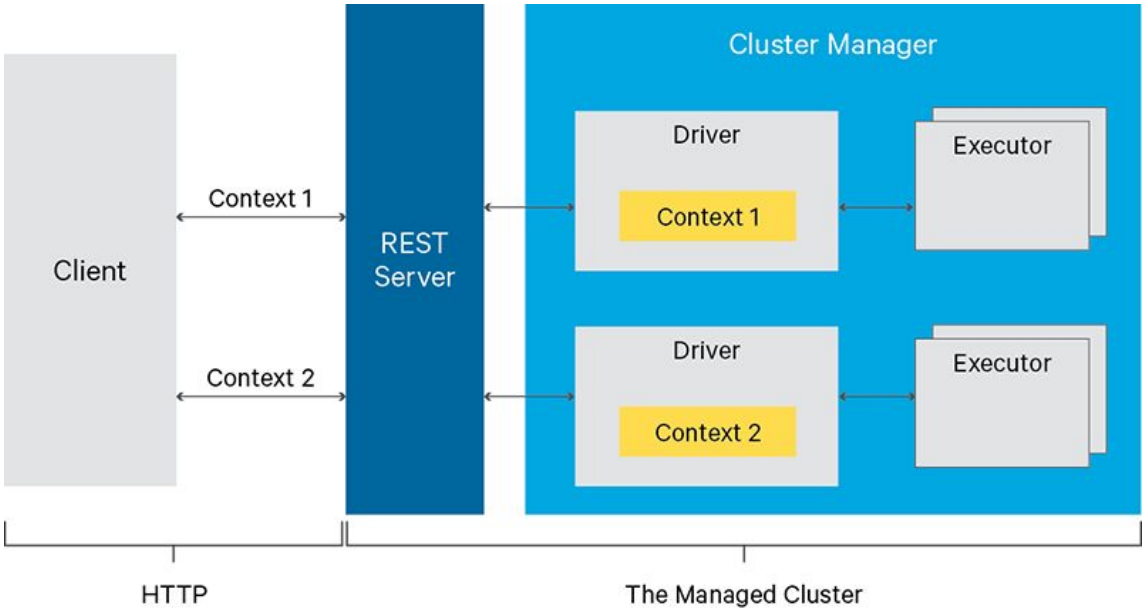


Available Interpreters

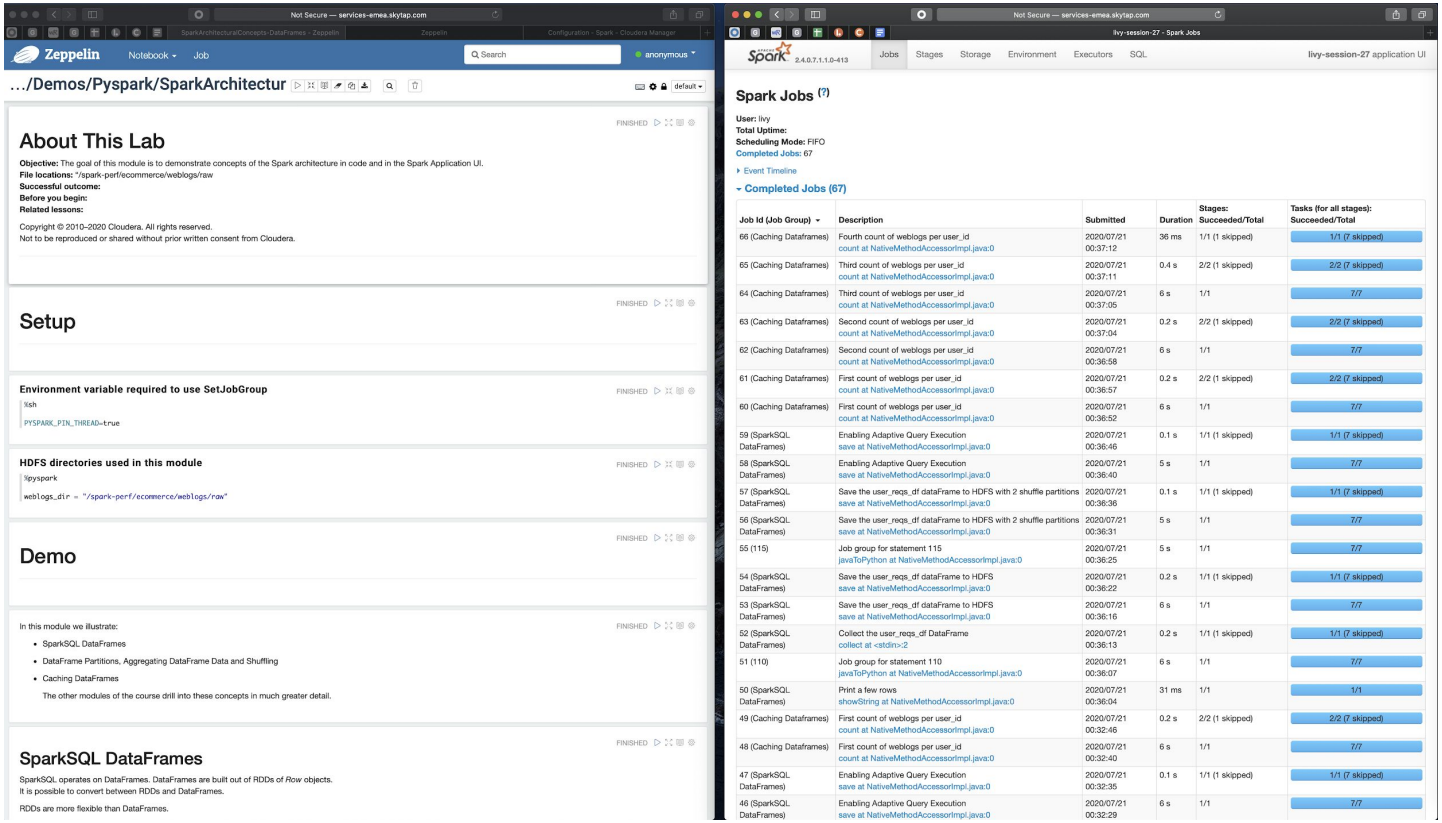
- The latest version of Zeppelin contains over 30 interpreters
- In CDP DC the following interpreters are available by default:
  - angular
  - livy
  - md
- The default one is livy
- This is a direct consequence of the secured design of CDP DC
- If you need an additional interpreter check with your favourite admin whether it can be installed securely

Apache Livy

Livy enables multitenant and secure communication with a Spark cluster over a REST interface



The Spark UI



CLOUDERA
Educational Services

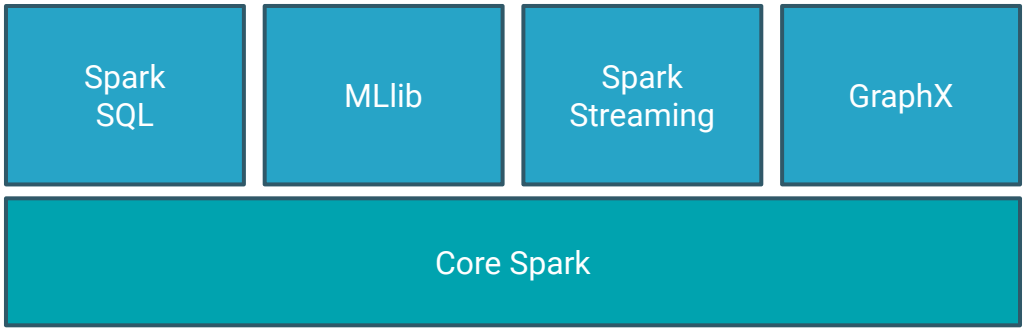
Demos/Labs:
Apache Spark in 5 Minutes

CLOUDERA
Educational Services

Apache Spark Architecture

The Spark Stack

- Spark provides a stack of libraries built on core Spark
  - Core Spark provides the fundamental Spark abstraction: Resilient Distributed Datasets (RDDs)
  - Spark SQL for working with structured data
  - MLlib for scalable machine learning
  - Spark Streaming for processing data in real time
  - GraphX for working with graphs and graph-parallel computation



Resilient Distributed Datasets (RDDs)

- Distributed collection of *serializable* objects
  - Any kind of data you can express as an object in Scala, Java or Python
  - Must be serializable: passed over the network, stored in temporary files
- Can be created from / saved to an extensible set of data sources
- Immutable
- Transformed using functional programming
  - Apply pure functions that developer writes
- Streamed by default – not in memory
  - Developer can cache when appropriate
- Scale: processed in parallel by multiple machines
- Fault tolerant

SparkSQL DataFrames

- Distributed collection of *Row* objects
  - Data must be specified in terms of Row objects
- Can be created from / saved to an extensible set of data sources
- Immutable
- Transformed using SQL transformations
- Streamed by default – not in memory
  - Developer can cache when appropriate
- Scale: processed in parallel by multiple machines
- Fault tolerant
- Always have a *schema*
- Optimized by the Catalyst and Tungsten optimizers

RDDs vs. DataFrames

- RDDs and DataFrames have a lot in common
  - Convert between RDDs and DataFrames
    - Via RDD of Row objects and schema
  - Same execution model
- RDDs are more flexible
  - The power of functional programming
  - The flexibility of objects
- DataFrames are simpler
  - SQL transformations well understood
  - DataSources are easier to use and “smarter”
- DataFrames are faster
  - Catalyst optimizer applies typical database optimizations
  - Tungsten optimizer applies physical optimizations
  - Sequence of RDD transformations are not optimized

Lazy Execution (1)

Language: Python

```
> peopleDF =
  spark.read.json("people.json")
```

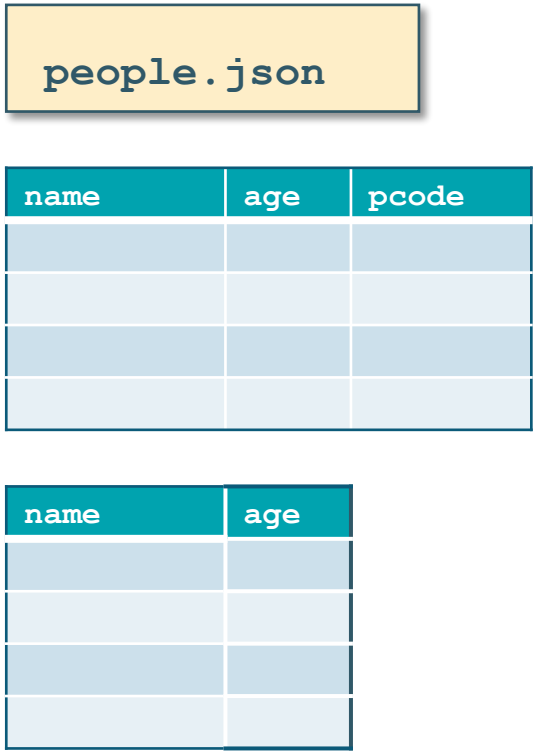
people.json

name	age	pcode

Lazy Execution (2)

Language: Python

```
> peopleDF =
  spark.read.json("people.json")
> nameAgeDF =
  peopleDF.select("name","age")
```



Lazy Execution (3)

Language: Python

```
> peopleDF =
  spark.read.json("people.json")
> nameAgeDF =
  peopleDF.select("name","age")
> nameAgeDF.show()
```

people.json

name	age	pcode
Alice		94304
Brayden	30	94304
Carla	19	10036
…	…	…

name	age
Alice	
Brayden	30
Carla	19
…	

Lazy Execution (4)

Language: Python

```
> peopleDF =
  spark.read.json("people.json")
> nameAgeDF =
  peopleDF.select("name","age")
> nameAgeDF.show()
+-------+----+
|   name| age|
+-------+----+
|  Alice|null|
|Brayden|  30|
|  Carla|  19|
…
```

people.json

name	age	pcode
Alice		94304
Brayden	30	94304
Carla	19	10036
…	…	…

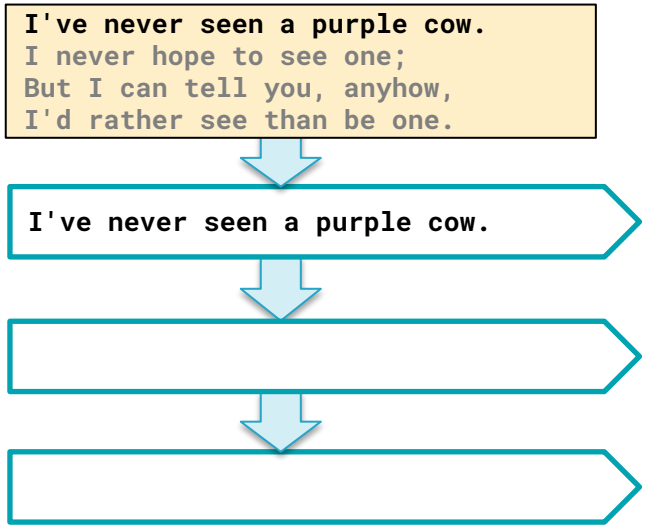
name	age
Alice	
Brayden	30
Carla	19
…	

Pipelining (1)

- When possible, Spark will perform sequences of transformations by element so no data is stored

Language: Scala

```
val myFilteredRDD =
  sc.textFile("purplecow.txt").
  map(line => line.toUpperCase).
  filter(line => line.startsWith("I"))
myFilteredRDD.take(2)
```

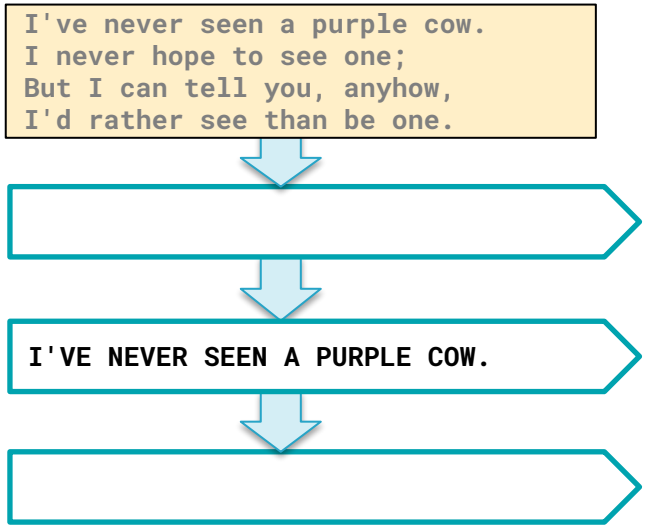


Pipelining (2)

- When possible, Spark will perform sequences of transformations by element so no data is stored

Language: Scala

```
val myFilteredRDD =
  sc.textFile("purplecow.txt").
  map(line => line.toUpperCase).
  filter(line => line.startsWith("I"))
myFilteredRDD.take(2)
```

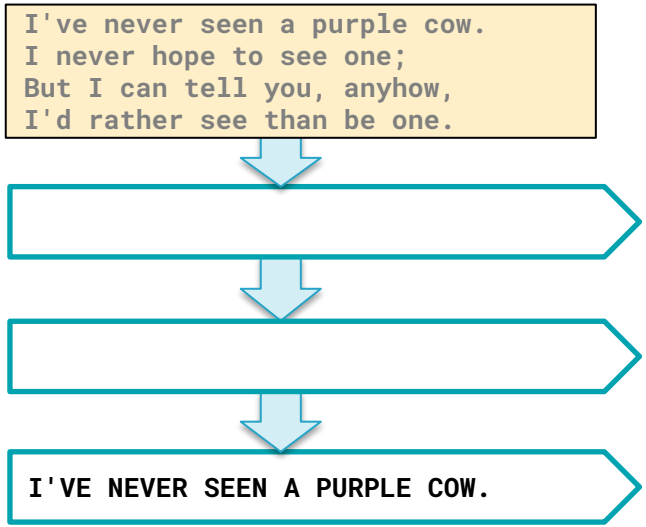


Pipelining (3)

- When possible, Spark will perform sequences of transformations by element so no data is stored

Language: Scala

```
val myFilteredRDD =
  sc.textFile("purplecow.txt").
  map(line => line.toUpperCase).
  filter(line => line.startsWith("I"))
myFilteredRDD.take(2)
```

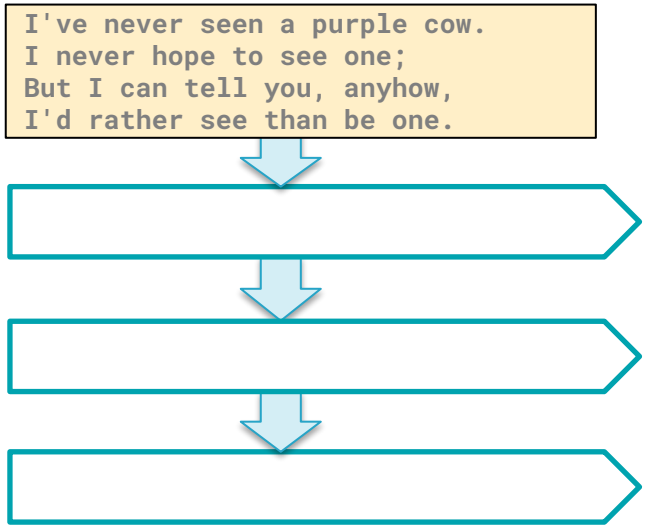


Pipelining (4)

- When possible, Spark will perform sequences of transformations by element so no data is stored

Language: Scala

```
val myFilteredRDD =
  sc.textFile("purplecow.txt").
  map(line => line.toUpperCase).
  filter(line => line.startsWith("I"))
myFilteredRDD.take(2)
I'VE NEVER SEEN A PURPLE COW.
```

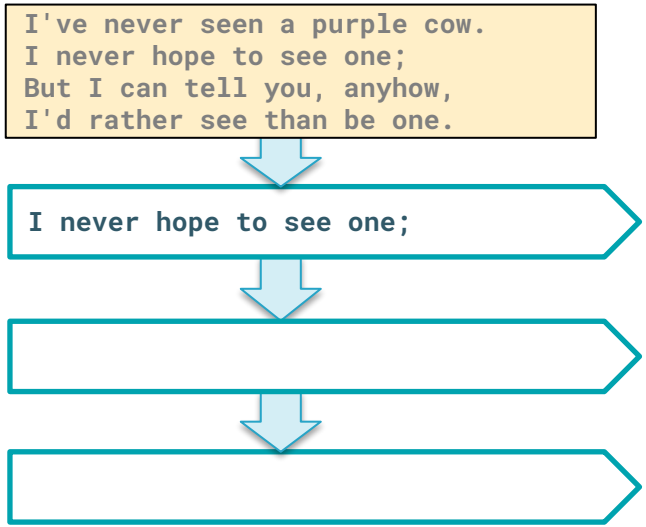


Pipelining (5)

- When possible, Spark will perform sequences of transformations by element so no data is stored

Language: Scala

```
val myFilteredRDD =
  sc.textFile("purplecow.txt").
  map(line => line.toUpperCase).
  filter(line => line.startsWith("I"))
myFilteredRDD.take(2)
I'VE NEVER SEEN A PURPLE COW.
```

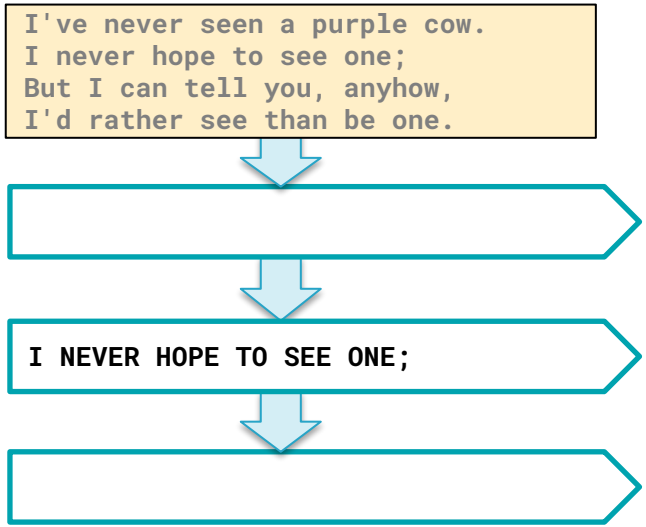


Pipelining (6)

- When possible, Spark will perform sequences of transformations by element so no data is stored

Language: Scala

```
val myFilteredRDD =
  sc.textFile("purplecow.txt").
  map(line => line.toUpperCase).
  filter(line => line.startsWith("I"))
myFilteredRDD.take(2)
I'VE NEVER SEEN A PURPLE COW.
```

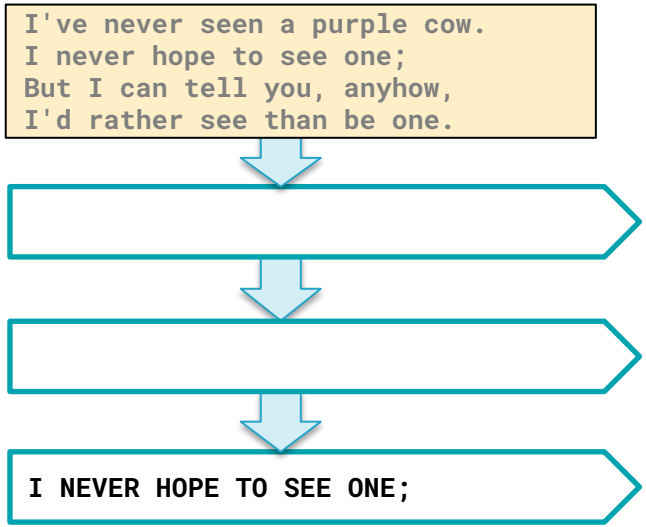


Pipelining (7)

- When possible, Spark will perform sequences of transformations by element so no data is stored

Language: Scala

```
val myFilteredRDD =
  sc.textFile("purplecow.txt").
  map(line => line.toUpperCase).
  filter(line => line.startsWith("I"))
myFilteredRDD.take(2)
I'VE NEVER SEEN A PURPLE COW.
```

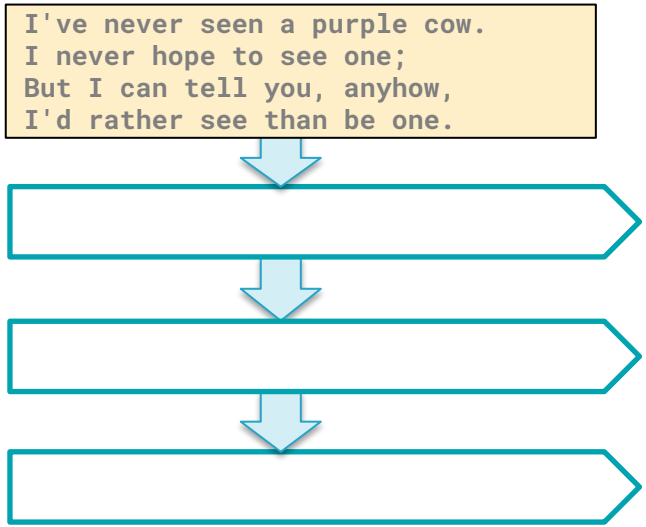


Pipelining (8)

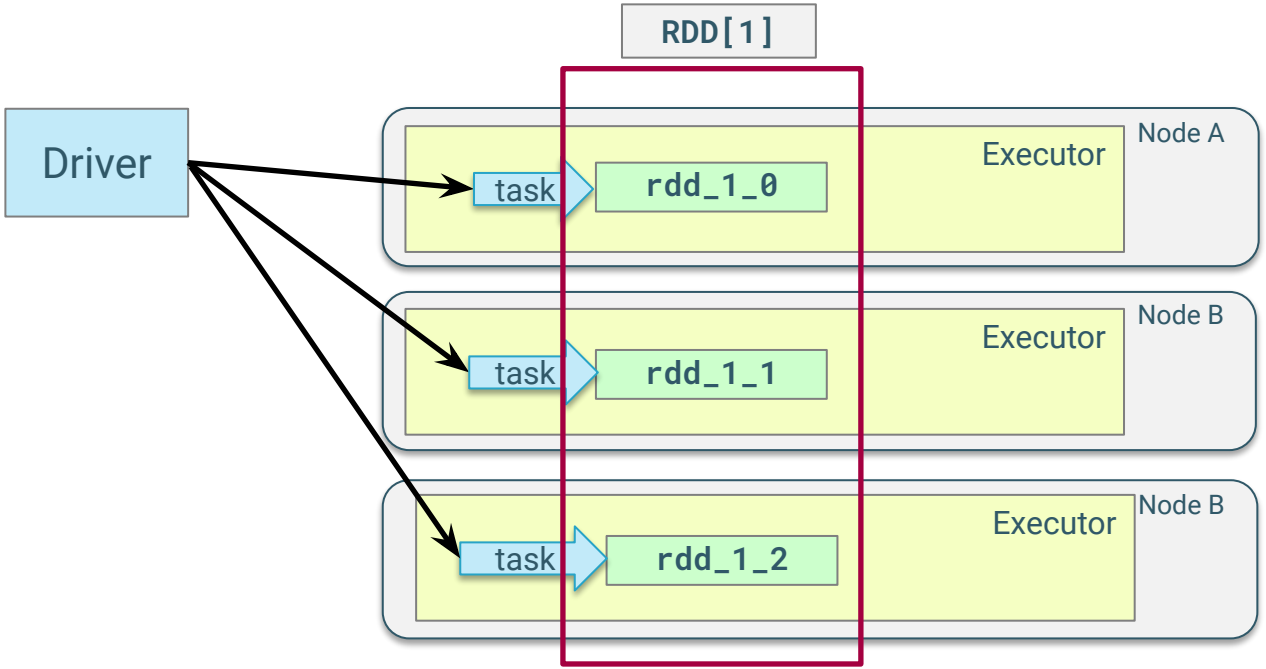
- When possible, Spark will perform sequences of transformations by element so no data is stored

Language: Scala

```
val myFilteredRDD =
  sc.textFile("purplecow.txt").
  map(line => line.toUpperCase).
  filter(line => line.startsWith("I"))
myFilteredRDD.take(2)
I'VE NEVER SEEN A PURPLE COW.
I NEVER HOPE TO SEE ONE;
```



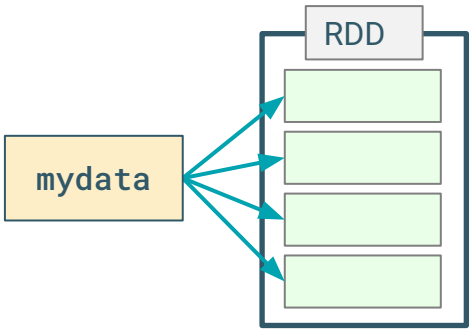
RDDs and DataFrames are Partitioned



Example: Average Word Length by Letter (1)

Language: Python

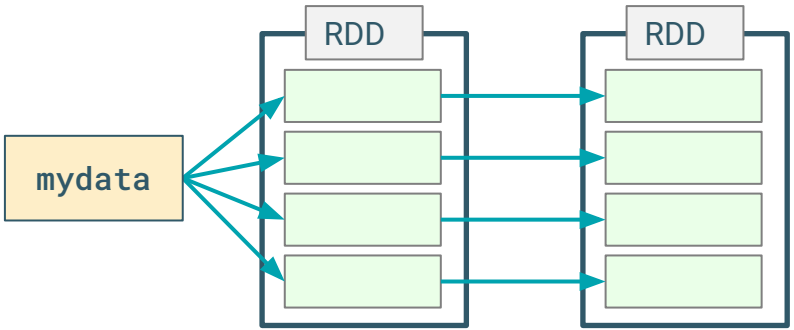
```
> avglens = sc.textFile(file)
```



Example: Average Word Length by Letter (2)

Language: Python

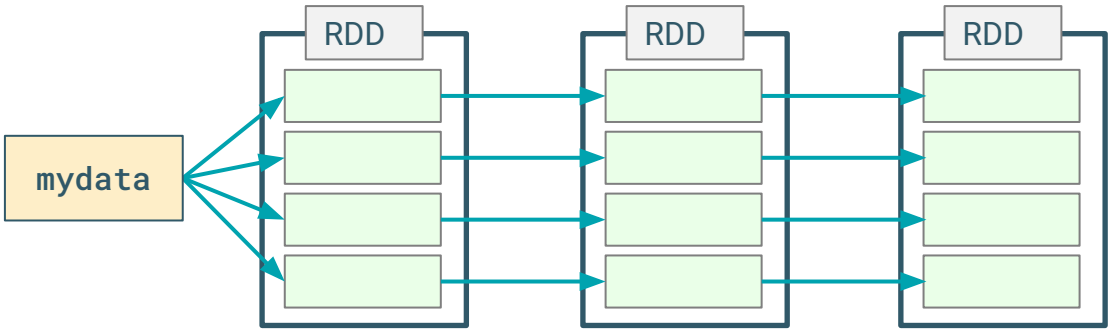
```
> avglens = sc.textFile(file) \
  .flatMap(lambda line: line.split(' '))
```



Example: Average Word Length by Letter (3)

Language: Python

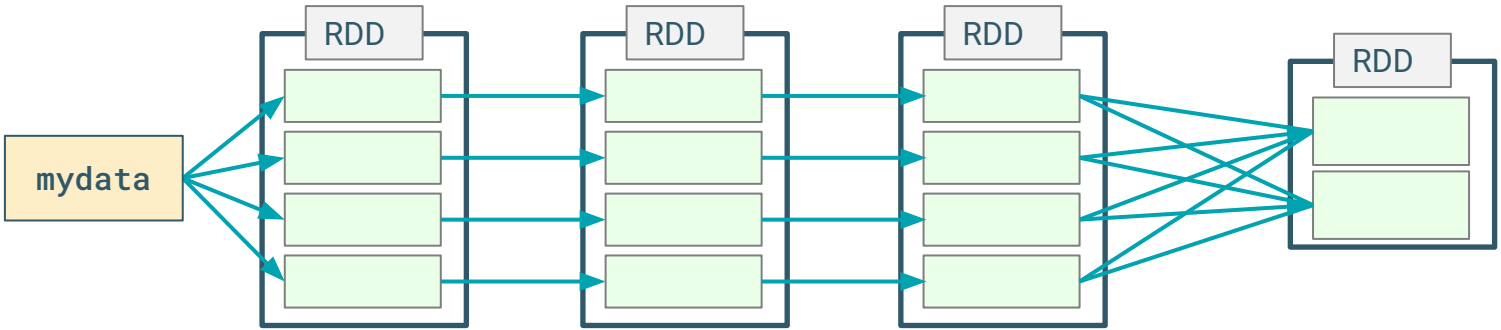
```
> avglens = sc.textFile(file) \
  .flatMap(lambda line: line.split(' ')) \
  .map(lambda word: (word[0],len(word[0])))
```



Example: Average Word Length by Letter (4)

Language: Python

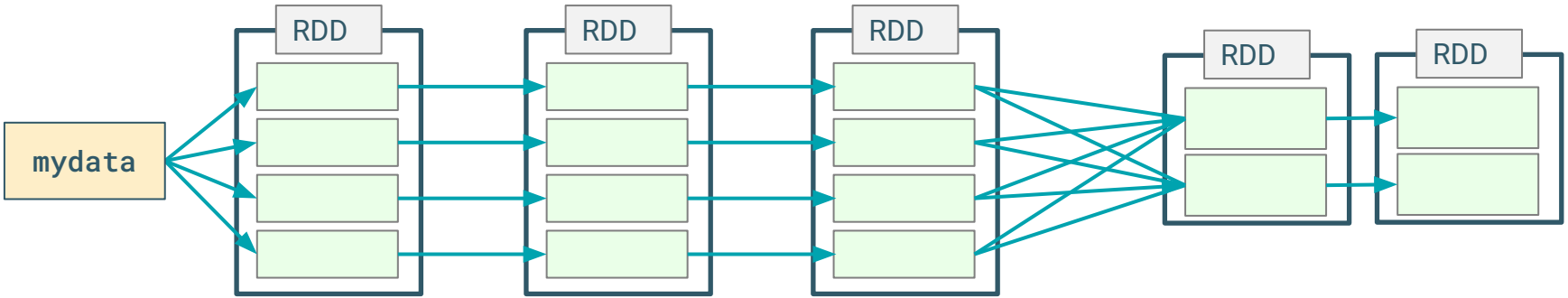
```
> avglens = sc.textFile(file) \
  .flatMap(lambda line: line.split(' ')) \
  .map(lambda word: (word[0],len(word[0]))) \
  .groupByKey(2)
```



Example: Average Word Length by Letter (5)

Language: Python

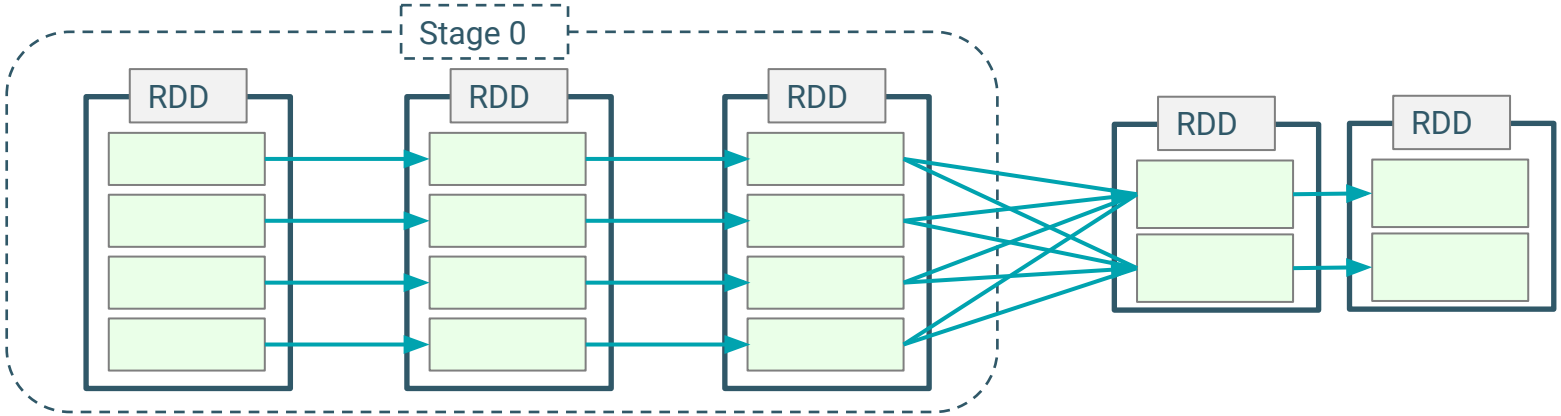
```
> avglens = sc.textFile(file) \
  .flatMap(lambda line: line.split(' ')) \
  .map(lambda word: (word[0],len(word[0]))) \
  .groupByKey(2) \
  .map(lambda (k, values): \
     (k, sum(values)/len(values)))
```



Spark Execution: Stages (1)

Language: Scala

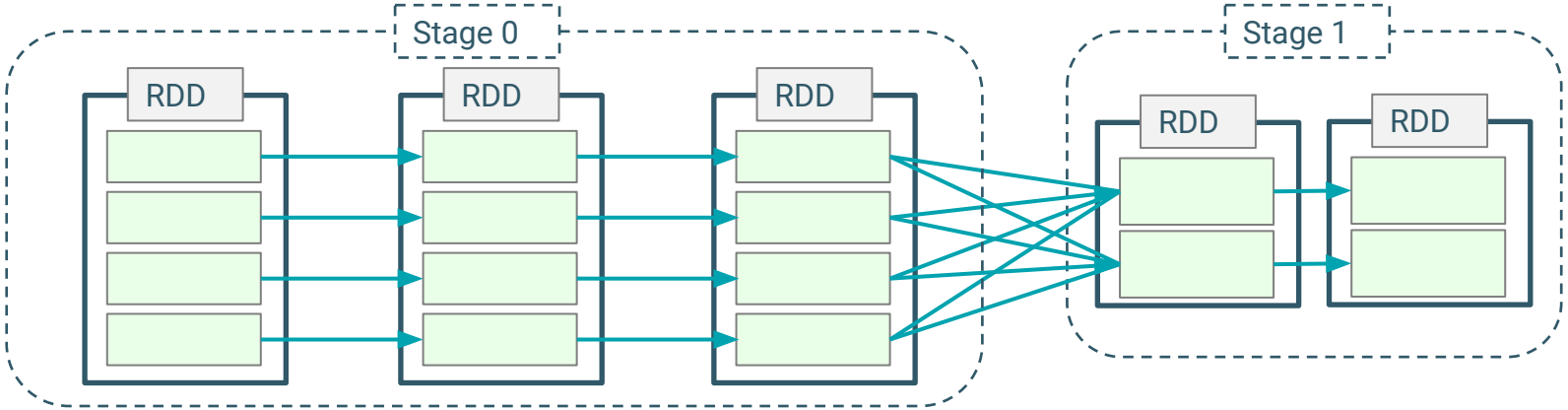
```
> val avglens = sc.textFile(myfile).
    flatMap(line => line.split(' ')).
    map(word => (word(0),word.length)).
    groupByKey(2).
    map(pair => (pair._1, pair._2.sum/pair._2.size.toDouble))

> avglens.saveAsTextFile("avglen-output")
```



Spark Execution: Stages (2)

Language: Scala

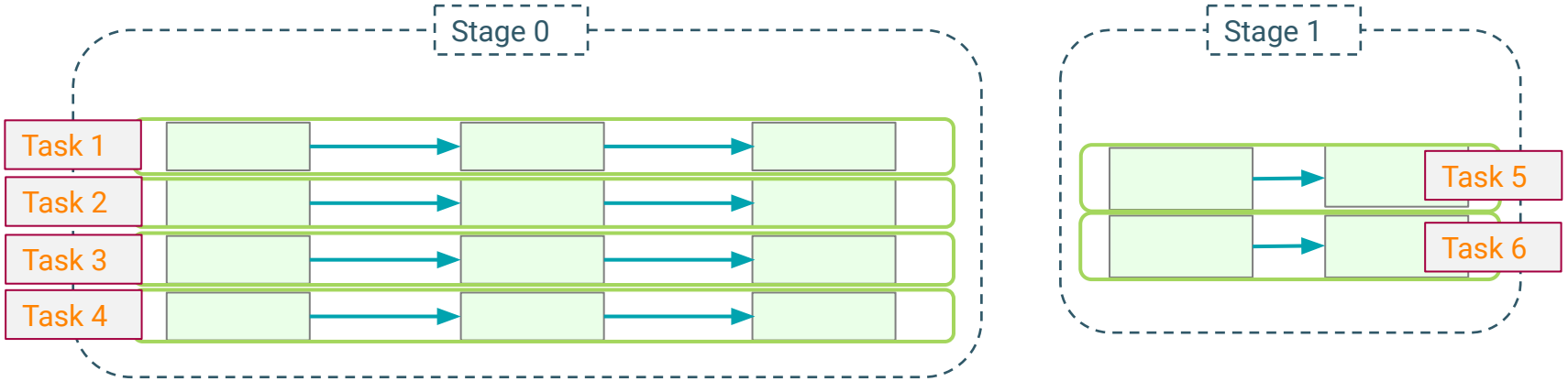
```
> val avglens = sc.textFile(myfile).
    flatMap(line => line.split(' ')).
    map(word => (word(0),word.length)).
    groupByKey(2).
    map(pair => (pair._1, pair._2.sum/pair._2.size.toDouble))

> avglens.saveAsTextFile("avglen-output")
```



Spark Execution: Stages (3)

Language: Scala

```
> val avglens = sc.textFile(myfile).
    flatMap(line => line.split(' ')).
    map(word => (word(0),word.length)).
    groupByKey(2).
    map(pair => (pair._1, pair._2.sum/pair._2.size.toDouble))

> avglens.saveAsTextFile("avglen-output")
```



Spark Execution: Stages (4)

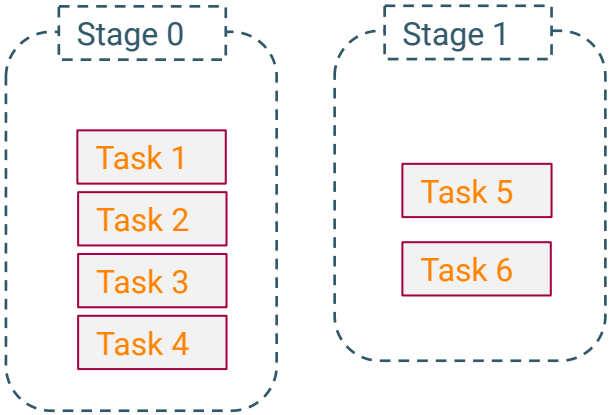
Language: Scala

```
> val avglens = sc.textFile(myfile).
    flatMap(line => line.split(' ')).
    map(word => (word(0),word.length)).
    groupByKey(2).
    map(pair => (pair._1, pair._2.sum/pair._2.size.toDouble))

> avglens.saveAsTextFile("avglen-output")
```

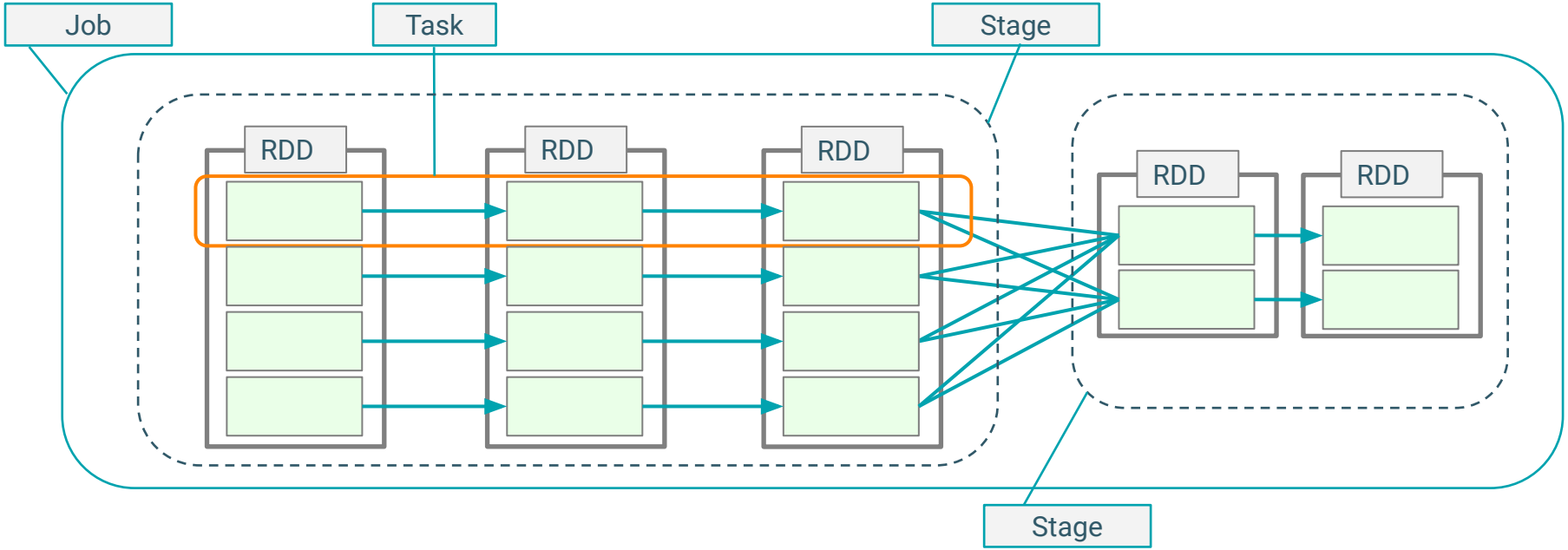
Spark Execution: Stages (5)

Language: Scala

```
> val avglens = sc.textFile(myfile).
    flatMap(line => line.split(' ')).
    map(word => (word(0),word.length)).
    groupByKey(2).
    map(pair => (pair._1, pair._2.sum/pair._2.size.toDouble))

> avglens.saveAsTextFile("avglen-output")
```



Summary of Spark Terminology



CLOUDERA
Educational Services

Demos/Labs:

Spark Architectural Concepts - RDDs
Spark Architectural Concepts - DataFrames

CLOUDERA
Educational Services

Data Sources and Formats

Extensible Set of Data Sources and Formats

Data Sources

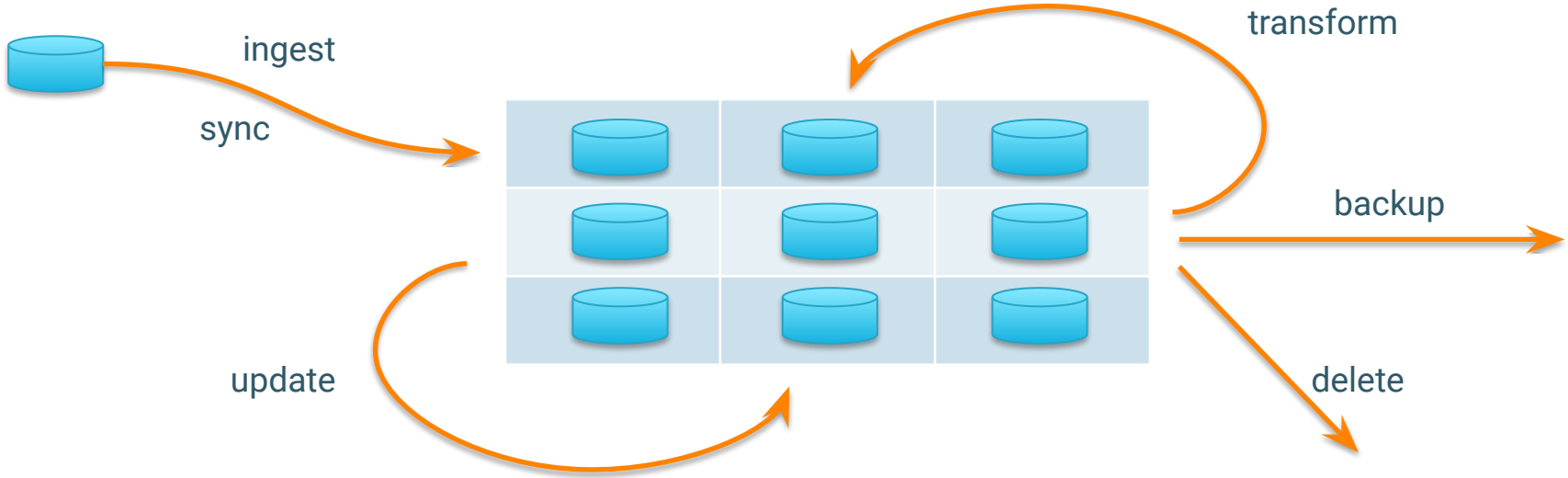
- JDBC
- HDFS
- local files
- HBase
- Kudu
- Object stores (e.g. S3)
- Hive Metastore
- In memory data structures
- Extensible!

Data Formats

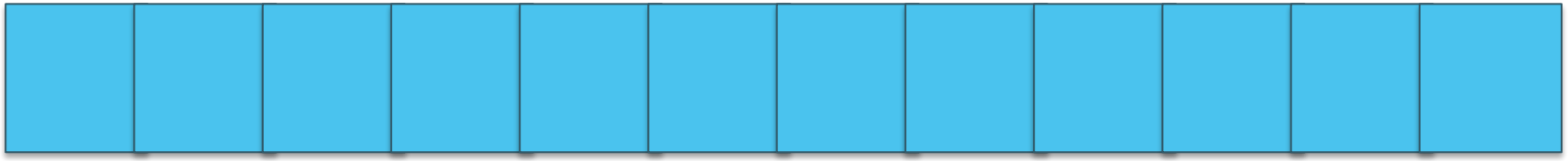
- Text
- CSV
- JSON
- XML
- Sequence files
- Avro
- Parquet
- ORC
- Extensible!

Data Lifecycle

- No requirement to use the same format for all phases
  - Straightforward to change formats



Splittable Formats



- Splittable file formats can be processed
  - Separately, in pieces called “input splits”
  - In parallel
  - Without having to read the whole file to identify the pieces
- Subsets of records are recognized from each piece
- Some file formats are splittable, others are not
- Some compression algorithms are splittable, others are not

The *Small Files* Problem

- Spark is a *coarse grain parallel* system
  - Overhead creating tasks
  - Parallel tasks are meant to do a significant amount of work
- Creating RDDs from lots of small files can result in lots of parallel tasks
  - Each task has little work to do
  - RDDs suffer from this because Spark usually creates at least one partition per file
  - DataFrame data sources are usually smarter about creating partitions from multiple files

Non-Splittable, Small Text Files

- Lots and lots of small XML or JSON file
  - For RDDs use:
    - `wholeTextFiles()`
  - For DataFrames use
    - `DataFrameReader.text(wholetext=True)`
- `wholeTextFiles`
  - Maps entire contents of each file in a directory to a single RDD element
  - Works only for small files (element must fit in memory)

user1.json

```
{
"firstName":"Fred",
"lastName":"Flintstone",
"userid":"123"
}
```

user2.json

```
{
"firstName":"Barney",
"lastName":"Rubble",
"userid":"234"
}
```

Example of `wholeTextFiles()`

```
userRDD = spark.sparkContext. \
  wholeTextFiles("userFiles")
```

Language: *Python*

userRDD

("user1.json",{"firstName":"Fred", "lastName":"Flintstone","userid":"123"})
("user2.json",{"firstName":"Barney", "lastName":"Rubble","userid":"234"})
("user3.json",…)
("user4.json",…)

Text

- Text files are the most basic file type
  - Can be read or written from virtually any programming language
  - Comma and tab-delimited files are compatible with many applications
- Text files are human-readable
  - All values are represented as strings
  - Useful when debugging
- At scale, this format is inefficient
  - Representing numeric values as strings wastes storage space
  - Difficult to represent binary data such as images
    - Often resort to techniques such as Base64 encoding
  - Parsing text degrades performance
- Recognizing complex structures in text can be accomplished with regular expressions

XML / JSON

- XML are text files with hierarchical structure
  - Nested tagged elements and attributes
  - Usually validated against schema
    - Syntactic structure and data constraints
  - Parsers available in most programming languages
  - Not splittable in general
    - Need to parse the entire document in order to recognize records
    - Or resort to schema dependent parsing code
  - Effective as big data when represented as a huge number of reasonably sized XML files
    - Spark's `wholeTextFiles()` designed for exactly that use case
- JSON is very similar to XML, usually without a schema
  - JSON schemas are only recently defined as an IETF standard
- Both XML and JSON are good data interoperability formats
  - Usually a good idea for performance to transform data into another format

Apache Avro Data Files

- Efficient storage due to optimized binary encoding
- Widely supported throughout the Hadoop ecosystem
  - Can also be used outside of Hadoop
- Ideal for long-term storage of important data
  - Many languages can read and write Avro files
  - Embeds schema in the file, so will always be readable
    - In JSON format and not Java-specific
  - Schema evolution can accommodate changes

Columnar Formats

- *Columnar* formats organize data storage by column, rather than by row
  - Very efficient when selecting only a subset of a table's columns

id	name	city	occupation
1	Alice	Palo Alto	Accountant
2	Bob	Sunnyvale	Accountant
3	Bob	Palo Alto	Dentist
4	Bob	Palo Alto	Manager
5	Carol	Palo Alto	Manager
6	David	Sunnyvale	Mechanic

Organization of data in traditional row-based formats

id	name	city	occupation
1	Alice	Palo Alto	Accountant
2	Bob	Sunnyvale	Accountant
3	Bob	Palo Alto	Dentist
4	Bob	Palo Alto	Manager
5	Carol	Palo Alto	Manager
6	David	Sunnyvale	Mechanic

Organization of data in columnar formats

Apache Parquet Files

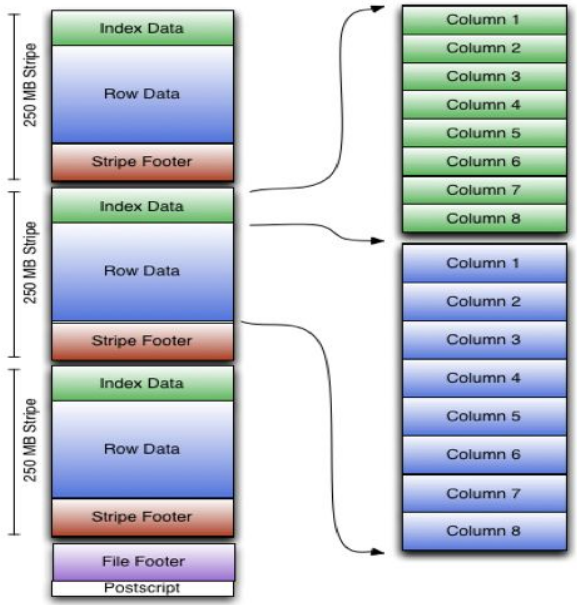
- Parquet is a columnar format developed by Cloudera and Twitter
  - Supported in Spark, MapReduce, Hive, Pig, Impala, and others
  - Schema metadata is embedded in the file (like Avro)
- Uses advanced optimizations described in Google's Dremel paper
  - Reduces storage space
  - Increases performance
- Most efficient when adding many records at once
  - Some optimizations rely on identifying repeated patterns

Apache ORC Files

- The **Optimized Row Columnar (ORC)** file format provides highly efficient way to store data
- Light-weight indexes stored within the **ORC** file format
- Block-mode compression based on data type
- Hive default storage format
  - Hive type support including datetime,decimal, and complex types(struct, list, map and union)

ORC File - Columnar Format

- **High Performance:** Columnar storage File.
- **Efficient Reads:** Break into large “stripes” of data efficient read.
- **Fast Filtering:** Built in index, min/max, metadata for fast filtering blocks.
- **Efficient Compression:** Decompose complex row types into primitives: massive compression and efficient comparisons for filtering.
- **Precomputation:** Built in aggregates per block (min, max, count, sum, etc.)
- **Proven at 300 PB scale:** Facebook uses ORC for their 300 PB Hive Warehouse.



CLOUDERA
Educational Services

Demos/Labs:
File Formats

CLOUDERA
Educational Services

Inferring Spark SQL Schemas

CLOUDERA
Educational Services

Demos/Labs:
Inferring Schemas

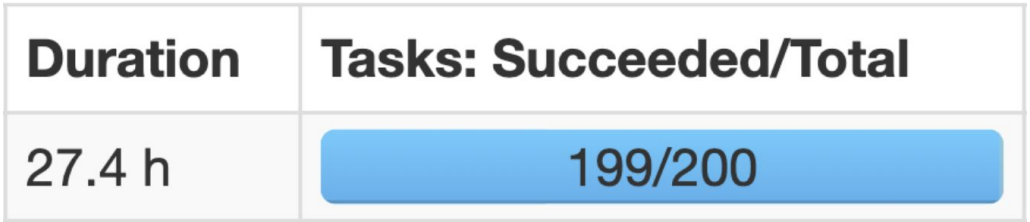
CLOUDERA
Educational Services

Dealing with Skewed Data

Dealing with Skewed Data

This issue cannot be fixed with parameter tuning. The code needs to be changed.

- Common strategies
  - Find a better partition key
  - Use a map-side (broadcast) join
  - Use a salted key
  - AQE skew join (sub partitions)



CLOUDERA
Educational Services

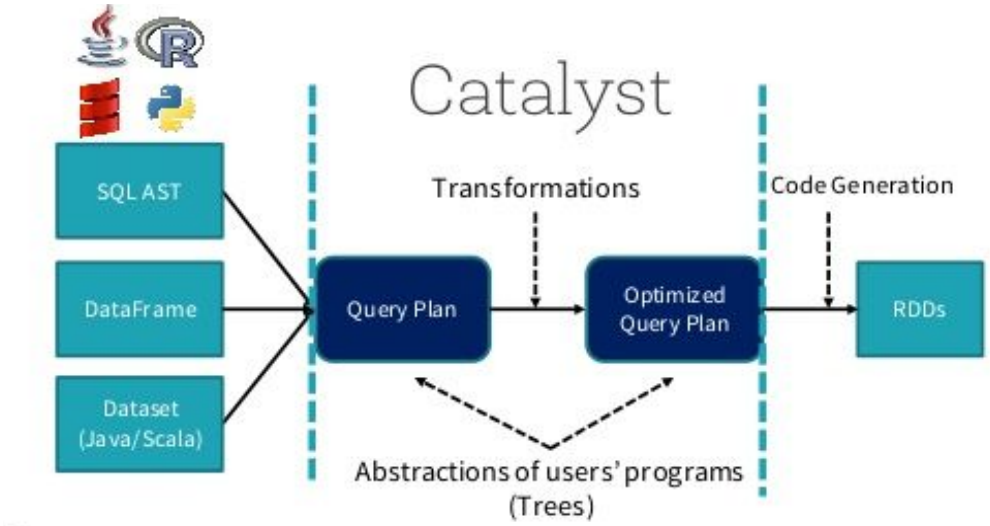
Demos/Labs:
Skewed Data

CLOUDERA
Educational Services

Catalyst Overview

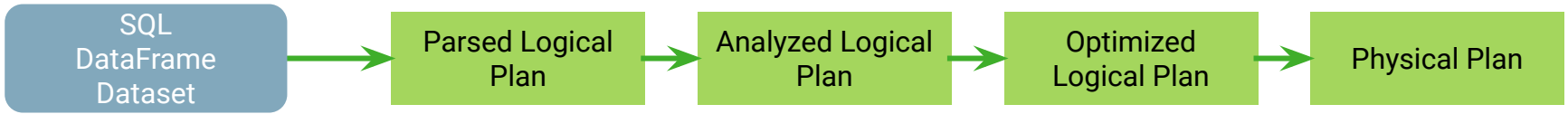
Catalyst Overview

- Catalyst **automatically optimizes** your Spark code
  - When using SQL, DataFrames, or Datasets
  - You focus on the WHAT
  - Catalyst automatically figures out the best HOW



How Catalyst Works

- Optimizes your total transformation
  - Creates an abstract representation (a tree)
    - The **Parsed Logical Plan**
  - Analyzes and optimizes the abstract representation
    - Using many optimization rules
    - Creating the **Analyzed Logical Plan** and **Optimized Logical Plan**
  - Converts the abstract representation to actual transformations
    - Creates multiple physical plan, applies cost model, chooses best one
    - Creating the **Physical Plan**



What Optimizations does Catalyst Do?

- Likely, more than you can think of, including:
  - **Predicate Pushdown**: Push filtering as early as possible
    - Eliminate data rows not satisfying preconditions early
  - **Projection Pushdown**: Project as early as possible
    - Eliminating data columns not needed early
  - **Reduce shuffling**: e.g. by reducing before grouping
    - Or joining in the most efficient manner
  - Constant Folding: Fold constant calculations into literals
  - Other Optimizations: e.g. convert Decimal ops to long ops
  - And more

- Let's look at examples to see it in action
  - Enough for a clear understanding
  - Don't need all the low level details to benefit from it

Example: Word Count with the DSL

- Uses the DSL for all transformations
  - Will need to shuffle data (uses groupBy)
  - Filters out results at two different times
    - Once **before** aggregation (counting) is done, and once **after**

```
// Prepare the data
> val linesDF = sc.parallelize(Seq("Twinkle twinkle little star", "How I wonder what you
are", "Twinkle twinkle little star")).toDF("line")

// Split into words
> val splitWordsDF = linesDF.select(explode (split('line, "\\s+")).as("word")
).select(lower('word).as("word"))

// Filter then count
> val filterThenCountDF = splitWordsDF  // Filter before count
      .filter('word =!= "twinkle").groupBy('word).count (1)

// Count then filter
> val countThenFilterDF = splitWordsDF.groupBy('word)
      .count.filter('word =!= "twinkle")  // Filter after count
```

"Explaining" Catalyst Optimizations

- Call explain() on a DataFrame/Dataset to view Catalyst's plans
  explain(): Unit — Display the Physical Plan
  explain(extended: Boolean): Unit — Display the Logical and Physical plans

- We'll look at filtering before counting, and after counting
  - Both will result in the same plan due to Catalyst

```
filterThenCountDF.explain
== Physical Plan ==
*HashAggregate(keys=[word#1168], functions=[count(1)])
+- Exchange hashpartitioning(word#1168, 200)
   +- *HashAggregate(keys=[word#1168],functions=[partial_count(1)])
// Remaining detail omitted
```

Overview of Explain Output

- In the output below
  - 1-3 prepares the data
  - 4 explodes/splits the data
  - 5 filters the data (**note where this is** — more on this soon)
  - 7 does a partial_count (**this is important** — more on this soon)
  - 8 does the shuffling (called an exchange)
  - 9 does the final counting (an aggregation on word which counts (1))

```
countThenFilterDF.explain  // Filter after counting
== Physical Plan ==
9 *HashAggregate(keys=[word#1168], functions=[count(1)])
8 +- Exchange hashpartitioning(word#1168, 200)
7  +- *HashAggregate(keys=[word#1168], functions=[partial_count(1)])
6    +- *Project [lower(word#1165) AS word#1168]
5      +- *Filter NOT (lower(word#1165) = twinkle)
4        +- Generate explode(split(line#852, \s+)), false, false,[word#1165]
3          +- *Project [value#850 AS line#852]
2            +- *SerializeFromObject [staticinvoke(…) AS value#850]
1              +- Scan ExternalRDDScan[obj#849]
```

Optimization: Predicate Pushdown

- The plan below is for filtering before counting
  - The previous slide showed filtering after counting
- Both transformations **have exactly the same plan**
  - Even though our transformations are written differently
  - Catalyst **pushes the filter down** as far as it can

```
filterThenCountDF.explain  // Filter before counting
== Physical Plan ==
9 *HashAggregate(keys=[word#1168], functions=[count(1)])
8 +- Exchange hashpartitioning(word#1168, 200)
7  +- *HashAggregate(keys=[word#1168], functions=[partial_count(1)])
6    +- *Project [lower(word#1165) AS word#1168]
5     +- *Filter NOT (lower(word#1165) = twinkle)
4        +- Generate explode(split(line#852, \s+)), false, false,[word#1165]
3          +- *Project [value#850 AS line#852]
2            +- *SerializeFromObject [staticinvoke(…) AS value#850]
1              +- Scan ExternalRDDScan[obj#849]
```

Optimization: Reduce Shuffling

- The plan **minimizes shuffling**
  - It does the count in two stages — at 7 (locally) and 9 (after shuffle)
  - This is possible because counting is commutative
  - Catalyst has chosen the equivalent of our RDD reduceByKey (1)
  - And not the more expensive groupByKey

```
filterThenCountDF.explain  // Filter before counting
== Physical Plan ==
9 *HashAggregate(keys=[word#1168], functions=[count(1)])
8 +- Exchange hashpartitioning(word#1168, 200)
7    +- *HashAggregate(keys=[word#1168], functions=[partial_count(1)])
6       +- *Project [lower(word#1165) AS word#1168]
5          +- *Filter NOT (lower(word#1165) = twinkle)
4             +- Generate explode(split(line#852, \s+)), false, false,[word#1165]
3                +- *Project [value#850 AS line#852]
2                   +- *SerializeFromObject [staticinvoke(…) AS value#850]
1                      +- Scan ExternalRDDScan[obj#849]
```

More Plan Detail

- Below, we show how the filter is pushed down
  - From the Parsed to the Optimized Logical Plan
  - It then generates physical plans, and picks the best one

```
countThenFilterDF.explain(true)
== Parsed Logical Plan ==
'Filter NOT ('word = twinkle)
+- Aggregate [word#1168], [word#1168, count(1) AS count#1292L]
   +- Project [lower(word#1165) AS word#1168]
   // …

== Analyzed Logical Plan ==
word: string, count: bigint
Filter NOT (word#1168 = twinkle)
// …

== Optimized Logical Plan ==
Aggregate [word#1168], [word#1168, count(1) AS count#1292L]
+- Project [lower(word#1165) AS word#1168]
   +- Filter NOT (lower(word#1165) = twinkle)
   // …
```

Lambdas Impede Catalyst Optimization

- Below, we filter with a lambda, after counting
  - Where does filtering happens in the physical plan?
  - After the aggregation
  - Catalyst can't optimize with a lambda (or UDF) — it's opaque (1)
  - Result — less efficient transformation

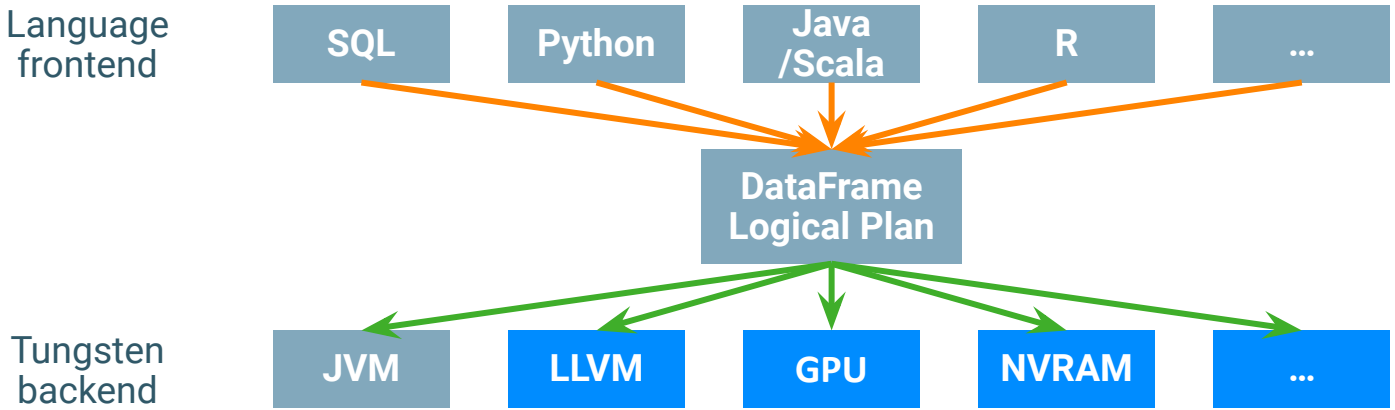
```
// Count then filter with lambda.  splitWordsDF uses split/explode
> val countThenFilterDF = splitWordsDF.groupBy('word)
      .count.filter(w => w.getString(0)!="twinkle").explain
== Physical Plan ==
*Filter <function1>.apply
8 *HashAggregate(keys=[word#1168], functions=[count(1)])
7 +- Exchange hashpartitioning(word#1168, 200)
6  +- *HashAggregate(keys=[word#1168], functions=[partial_count(1)])
5    +- *Project [lower(word#1165) AS word#1168]
4       +- Generate explode(split(line#852, \s+)), false, false,
                                                                    [w
ord#1165]
3          +- *Project [value#850 AS line#852]
2            +- *SerializeFromObject [staticinvoke(…) AS value#850]
1              +- Scan ExternalRDDScan[obj#849]
```

CLOUDERA
Educational Services

Tungsten Overview

Tungsten Overview

- Improve Spark execution by optimizing CPU / memory usage
  - Understands and optimizes for hardware architectures
  - Tunes optimizations for Spark's characteristics



Tungsten's Binary Format

- Binary representation of Java objects (Tungsten row format)
  - Different from Java serialization and Kryo
- Advantages include:
  - Much smaller size than native Java serialization
  - Supports off-heap allocation, **default is on-heap**
  - Structure supports Spark operations **without deserialization**
    - e.g. You can sort data while it remains in the binary format
  - Avoids GC overhead if using off-heap
- Result:
  - Much faster, less memory, less CPU
  - Can process much larger datasets

Whole-Stage Code Generation

- Optimization to improve execution performance
  - Collapses a query expression into a single optimized function

- For example suppose we were filtering on the following

```
filter('age>25 && 'age<50)
```

  - Code generation dynamically generates bytecode for this
    - All code contained in one function
  - Instead of classic interpretation
    - With boxing of primitives, polymorphic function calls …

- Result:
  - Eliminates virtual function calls
  - Leverages CPU registers for intermediate data
  - Can meet/exceed performance of hand-tuned function for a task

Cache-Aware Computation

- Modern hardware has many types of memory
  - Main memory
  - Multiple levels of cache (L1, L2, …)
  - Registers
  - These all have different characteristics
- Tungsten is aware of the different types of memory and different hardware architectures
  - It generates code that is optimized to utilize modern hardware
- Result: Much faster performance
  - For instance, it will use a cache-aware sorting algorithm
  - Giving 3x improvement over the non-cache aware version

How Do You Use Tungsten?

- Generally, don't think about it much
  - Just enjoy the benefits (but you are using DF schema and supported data types?)
- Ahhh — Except sometimes for lambdas!
  - These require Java objects, and can't be executed with data in the Tungsten format
  - This adds a layer of complexity when they interact with Tungsten
- Let's look at it in action
  - We can see it at work in the Physical Plan
- We will look at a DataFrame and Dataset Physical plan
  - Illustrating what Tungsten is doing
  - Exploring some issues with lambdas

DataFrame Physical Plan

- **SerializeFromObject** converts data to Tungsten binary format
  - First thing that's done after reading data in
  - All operations after that are done on this highly efficient format
  - Operations with asterisk (*) use Whole-Stage Code Gen

```
// Prepare the data
> val linesDF = sc.parallelize(Seq("Twinkle twinkle little star", "How I wonder what you
are", "Twinkle twinkle little star")).toDF()
// View physical plan with DataFrames
> linesDF.select(explode (split('value, "\\s+")).as("word")  )
       .select(lower('word).as("word")).groupBy('word).count.explain
== Physical Plan ==
*HashAggregate(keys=[word#563], functions=[count(1)])
+- Exchange hashpartitioning(word#563, 200)
 +- *HashAggregate(keys=[word#563], functions=[partial_count(1)])
  +- *Project [lower(word#560) AS word#563]
   +- Generate explode(split(value#555, \s+)), false, false, [word#560]
    +- *SerializeFromObject staticinvoke(class org.apache.spark.unsafe.types.UTF8String,
StringType, fromString, input[0, java.lang.String, true], true) AS value#555]
      +- Scan ExternalRDDScan[obj#554]
```

Tungsten Improves Efficiency

- The previous example demonstrates several things
- Operations are performed on data in Tungsten Binary Format
  - Data is serialized into this format (SerializeFromObject) at the very start of the pipeline
  - All succeeding operations work on data in this format
    - Highly memory efficient.
- Most operations use Whole-Stage Code Generation
  - Indicated by the asterisk (*) before the operation name
  - Highly CPU efficient
- Result: Significantly faster execution and reduced memory usage
  - Huge win for big data workflows

Dataset Physical Plan

- **AppendColumnsWithObject** converts to Tungsten format
  - After MapPartitions, which must work on Java objects
  - Note that **AppendColumnsWithObject** does **NOT** use code gen
  - Less efficient than the previous plan

```
// Convert linesDF to Dataset, set up word count, and look at plan
linesDF.as[String].flatMap(_.toLowerCase().split("\\s")).groupByKey(s =>
s).count.explain
== Physical Plan ==
*HashAggregate(keys=[value#496], functions=[count(1)])
+- Exchange hashpartitioning(value#496, 200)
 +- *HashAggregate(keys=[value#496], functions=[partial_count(1)])
  +- *Project [value#496]
   +- AppendColumnsWithObject <function1>, […]
    +- MapPartitions <function1>, obj#492: java.lang.String
     +- Scan ExternalRDDScan[obj#483]
```

Extra Serialize/Deserialize

- We've made a trivial change when loading the data
  - Naming our input row "line" instead of the default "value"
  - Note how this adds extra work in our plan
  - There is an extra serialize (to Tungsten format) then deserialize (to Java format) step — the flatMap lambda needs a Java object
  - This is unexpected, and adds extra overhead (1)

```
// Trivial change – name column "line" instead of default "value"
val linesDF = sc.parallelize(Seq("Twinkle twinkle" … )).toDF("line")
linesDF.as[String].flatMap(_.toLowerCase().split("\\s")).groupByKey(s => s).count.explain
== Physical Plan ==  // rest of plan as before
  +- *Project [value#520]
    +- AppendColumnsWithObject <function1>, … AS value#520]
      +- MapPartitions <function1>, obj#516: java.lang.String
        +- DeserializeToObject line#509.toString, obj#515: java.lang.String
          +- *Project [value#507 AS line#509]
            +- *SerializeFromObject [… AS value#507]
              +- Scan ExternalRDDScan[obj#506]
```

CLOUDERA
Educational Services

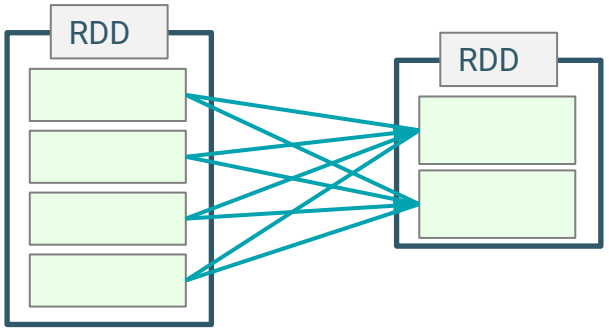
Demos/Labs:
Seeing Catalyst at Work

CLOUDERA
Educational Services

Mitigating the Shuffle

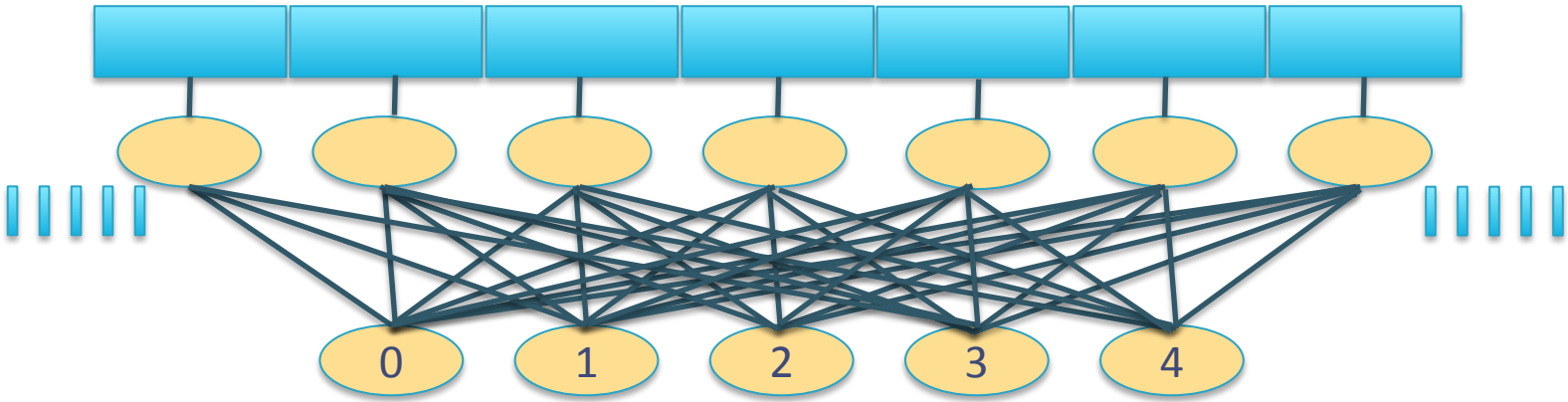
The Shuffle Is a Very Disruptive Event

- Let us consider a high level sequence of events that take place during that innocuous looking part of processing
  - The pipelining stops, intermediate data is materialized and held potentially putting a stress on **memory**
  - Intermediate data is serialized requiring **CPU** cycles
  - Intermediate data is spilled on local disks potentially putting a stress on **local storage**
  - Intermediate data is transferred over the network putting a stress on the **network**
  - Intermediate data is stored on local disks potentially putting a stress on **local storage**
  - Intermediate data is deserialized requiring **CPU** cycles
  - The Spark engine **waits** for all the intermediate data to be transferred and deserialized to resume processing which can result in a loss of efficiency when the shuffled data is **skewed**



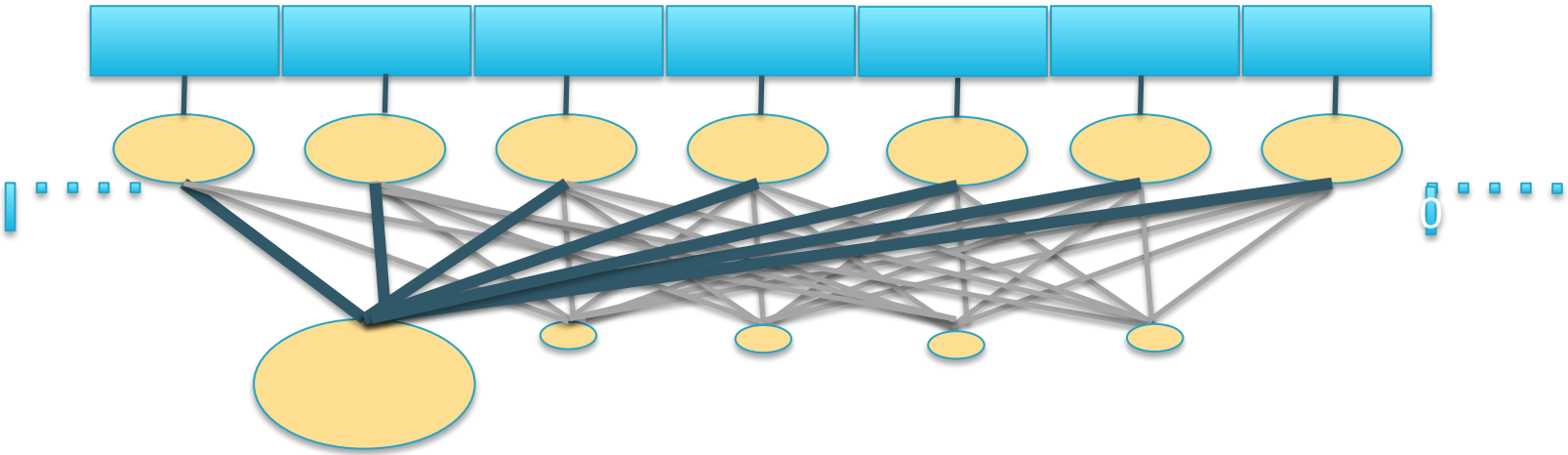
Balanced Shuffle

- Partition function computes target partition for each source record
- Goals:
  - Amount of shuffled data is small
  - Target partitions are similar in size
- Parallel processing on target partitions is balanced



Skewed Shuffle

- Most of the shuffled data ends up in the same partition
- Little parallelism processing the shuffled data



Consider the Shuffle As a Risk

As a distributed processing developer you might want to consider the shuffle as a risk and start leveraging various strategies to

- Avoid the risk if you can using,
  - Denormalization
  - Map side joins
- Limit the impact of risk if you cannot using.
  - Map side reduce
  - Sort Merge Bucket joins

Risk Avoiding Strategy #1: Denormalization

- Flat is beautiful!
  - Remove the joins with lookup tables
  - Add new columns to your datasets
  - Leave the foreign keys, you will need them for Machine Learning algorithms
- Whose job is it?
  - Yours
- This is the most powerful tool in your toolbox
  - Just because Spark 'speaks' SQL does not mean it is ready to tackle your pages long legacy queries
  - Without Adaptive SQL you might want to consider Spark SQL has a very convenient tool to query big flat tables

Flattening a Star Schema

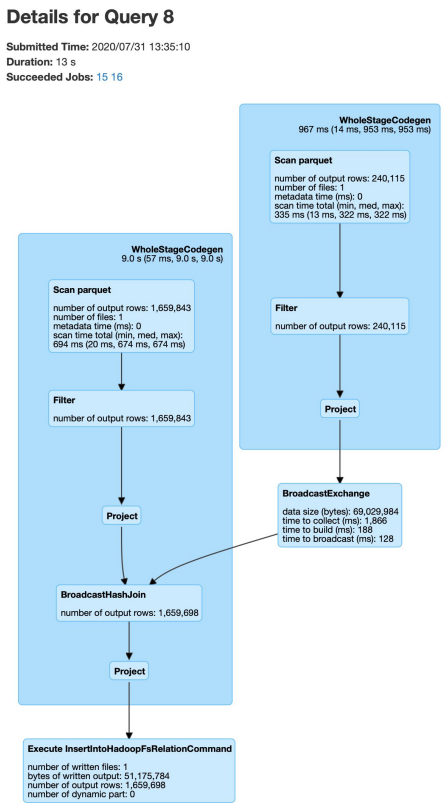
```
CREATE TABLE flatOrders STORED AS ORC AS
SELECT orders.SalesOrderID,
       orders.SalesOrderDetailID,
       orders.OrderDate,
       orders.DueDate,
       orders.ShipDate,
       orders.EmployeeID,
       orders.CustomerID,
       orders.SubTotal,
       orders.TaxAmt,
       orders.Freight,
       orders.TotalDue,
       orders.ProductID,
       orders.OrderQty,
       orders.UnitPrice,
       orders.UnitPriceDiscount,
       orders.LineTotal,
       employees.ManagerID,
       employees.FirstName AS EmployeeFirstName,
       employees.LastName  AS EmployeeLastName,
       employees.FullName  AS EmployeeFullName,
       employees.JobTitle,
       employees.OrganizationLevel,
       employees.MaritalStatus,
       employees.Gender,
       employees.Territory,
       employees.Country,
       employees.Group,
       customers.FirstName AS CustomerFirstName,
       customers.LastName  AS CustomerLastName,
       customers.FullName  AS CustomerFullName,
       products.ProductNumber,
       products.ProductName,
       products.ModelName,
       products.MakeFlag,
       products.StandardCost,
       products.ListPrice,
       productSubCategories.CategoryID,
       productCategories.CategoryName,
       products.SubCategoryID,
       productSubCategories.SubCategoryName
FROM orders
INNER JOIN employees ON orders.EmployeeID=employees.EmployeeID
INNER JOIN customers ON orders.CustomerID=customers.CustomerID
INNER JOIN products ON orders.ProductID=products.ProductID
INNER JOIN productSubCategories ON products.SubCategoryID=productSubCategories.SubCategoryID
INNER JOIN productCategories ON productSubCategories.CategoryID=productCategories.CategoryID
```

Risk Avoiding Strategy #2: Map Side Joins

- If one of the tables can fit in memory do the joins in memory

- Whose job is it?
  - Yours if you are using the distributed cache
  - Catalyst if you are using Spark Dataframes
  - The Hive Optimizer if you are using Hive

Map Side (Broadcast) Joins

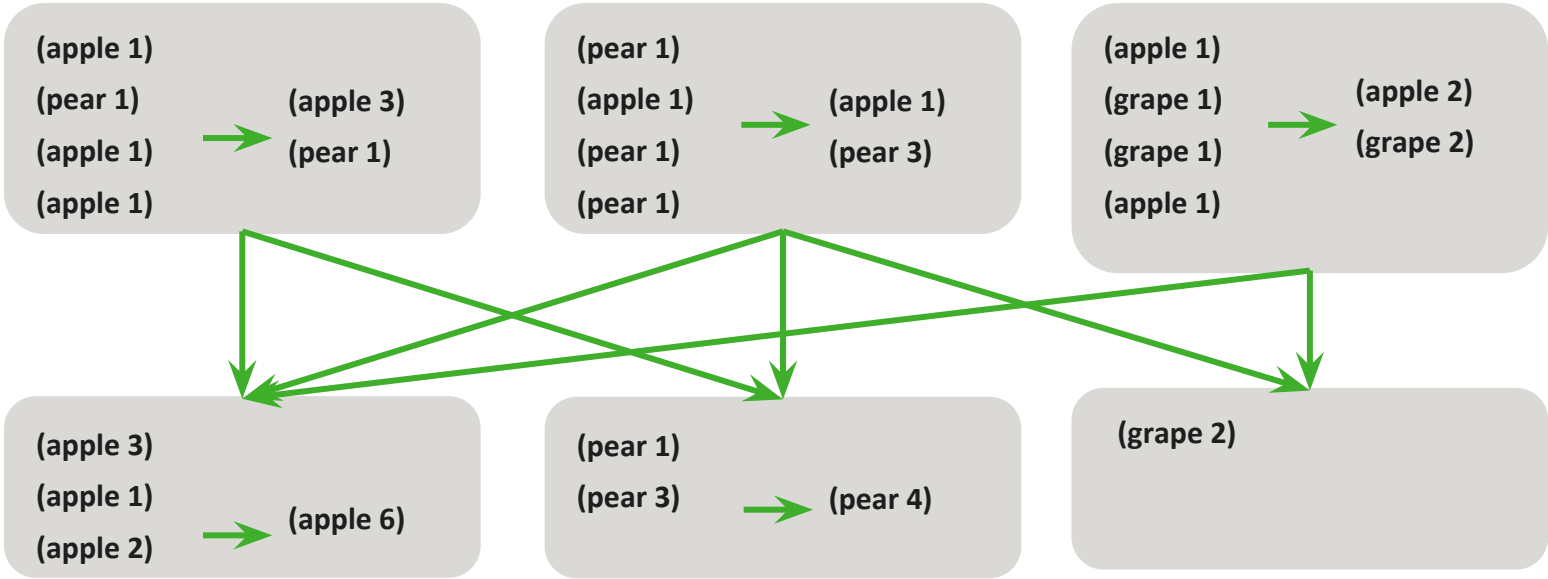


Impact Limiting Strategy #1: Map Side Reduce

- If some part of the process can be done on the map side thus limiting the amount of data sent over the network
- Whose job is it?
  - Yours if you are using Java, add a Combiner class to your job
  - The Spark Engine for instance when you are using ReduceByKey

Spark ReduceByKey

Note how pairs on the same partition are combined first which significantly reduces size before shuffling

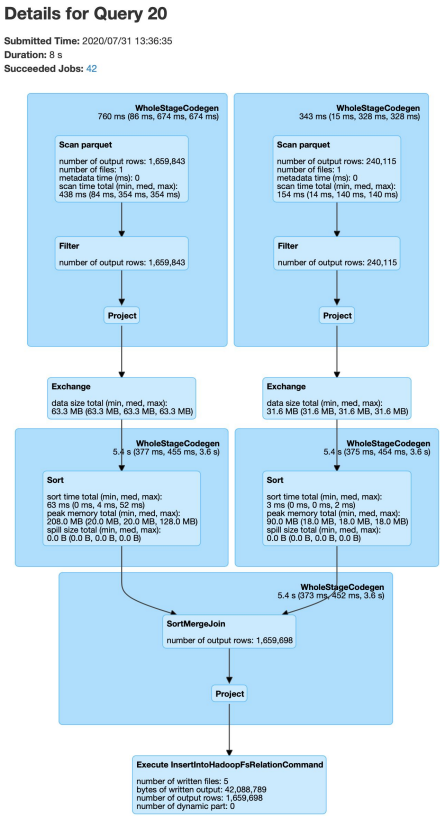


Impact Limiting Strategy #2: Sort Merge (Bucket) Joins

- If some part of the process can be done on the map side thus limiting the amount of data sent over the network using specific partitioning of the data

- Whose job is it?
  - Catalyst if you are using Spark Dataframes
  - The Hive Optimizer if you are using Hive

Sort Merge Joins



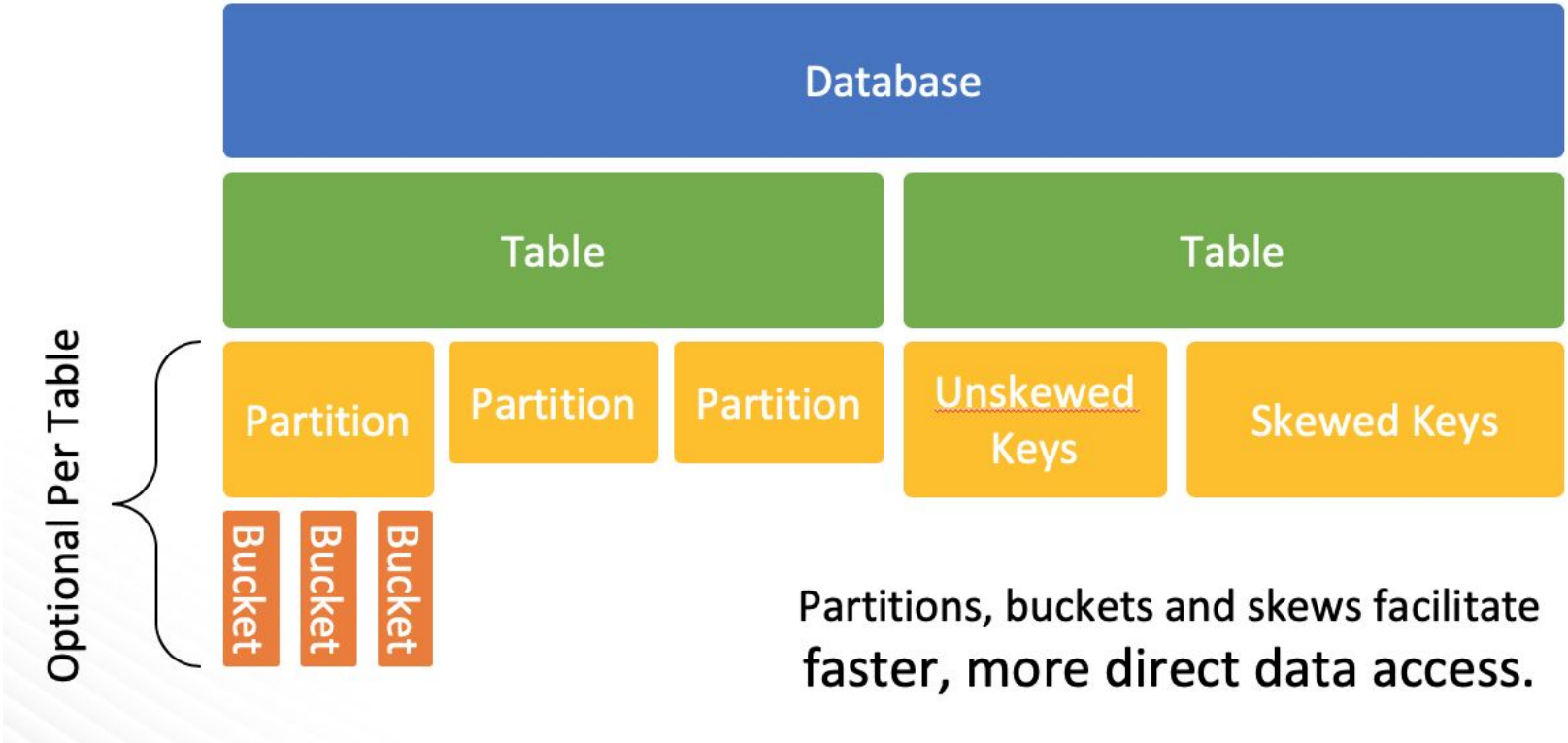
CLOUDERA
Educational Services

Demos/Labs:
Optimizing Shuffles

CLOUDERA
Educational Services

Partitioned and Bucketed Tables

Data Abstractions in Hive



Partitioned Tables

- Hive defines a directory structure for physically partitioning data into files
  - SparkSQL supports this directory structure
- By default, all data files for a table are stored in a single directory
  - All files in the directory are read during a query
- Partitioning subdivides the data
  - Data is physically divided during loading, based on values from one or more columns
- Speeds up queries that filter on partition columns
  - Only the files containing the selected data need to be read
- Does not prevent you from running queries that span multiple partitions

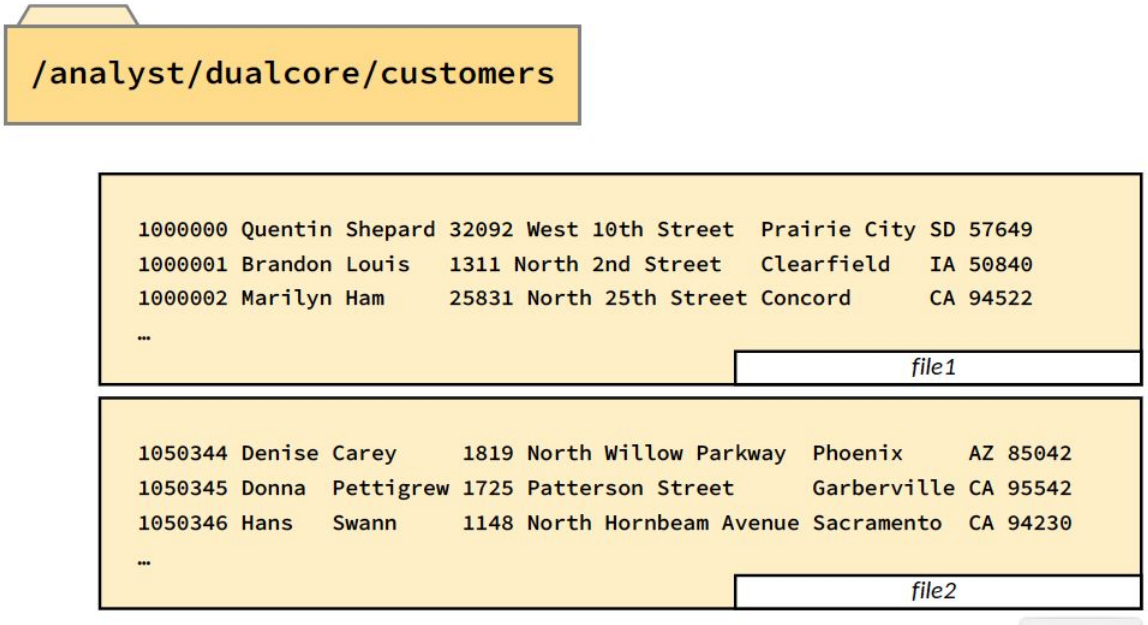
Example: Partitioning Customers by State (1)

- Example: `customers` in a non-partitioned table

```
CREATE EXTERNAL TABLE customers (
    cust_id INT,
    fname STRING,
    lname STRING,
    address STRING,
    city STRING,
    state STRING,
    zipcode STRING)
  ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t'
  LOCATION '/dualcore/customers';
```

Example: Partitioning Customers by State (2)

- Data files are stored in a single directory
- All files are scanned for every query



Partitioning File Structure

- Partitioned tables store data in subdirectories
  - Queries that filter on partitioned fields limit amount of data read

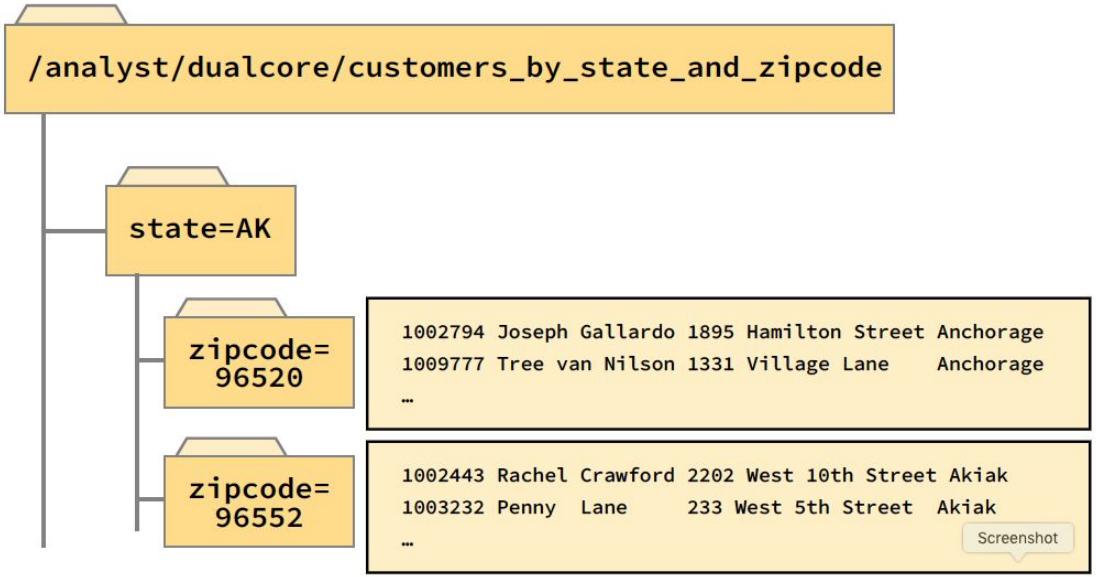
Creating a Partitioned Table

- Create a partitioned table using PARTITIONED BY

```
CREATE EXTERNAL TABLE customers_by_state (
    cust_id INT,
    fname STRING,
    lname STRING,
    address STRING,
    city STRING,
    zipcode STRING)
  PARTITIONED BY (state STRING)
  ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t'
  LOCATION '/dualcore/customers_by_state';
```

Nested Partitions

- You can also create nested partitions `PARTITIONED BY (state STRING, zipcode STRING)`



Bucketed Tables

- Bucketing data is another way of subdividing data
  - Unlike partitioned tables, bucketed tables have a flat directory structure
  - A *bucket* is a file
    - Computes a hash function on column values to assign records to a bucket
- The primary use for buckets is in optimizing joins
- A bucket is also a random sample of data

Creating a Bucketed Table

```
CREATE EXTERNAL TABLE sales_bucket (
  sale_id BIGINT,
  cust_id BIGINT,
  sale_date TIMESTAMP,
  store_id BIGINT
)
  CLUSTERED BY (store_id) INTO 4 BUCKETS
  STORED AS PARQUET
  LOCATION '/spark-perf/retail/clean/sales_bucket';
```

CLOUDERA
Educational Services

Demos/Labs:
Partitioned Tables

CLOUDERA
Educational Services

Improving Joins Performance

Example Data Sets

- Sales

  Bruce, shoes, 1997, $75, 500

  Susan, PC, 2012, $500, 250

  José, TV, 2019, $750, 500

  Bob, pants, 2018, $50, 300

  . . .

- **Stores**

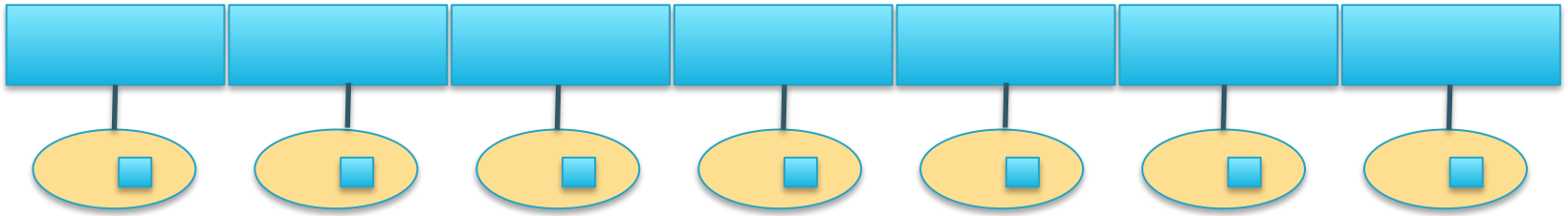
  500, SF, Market, Fred …

  250, NYC, Broadway, Rene …

  300, Cd. México, Reforma, María

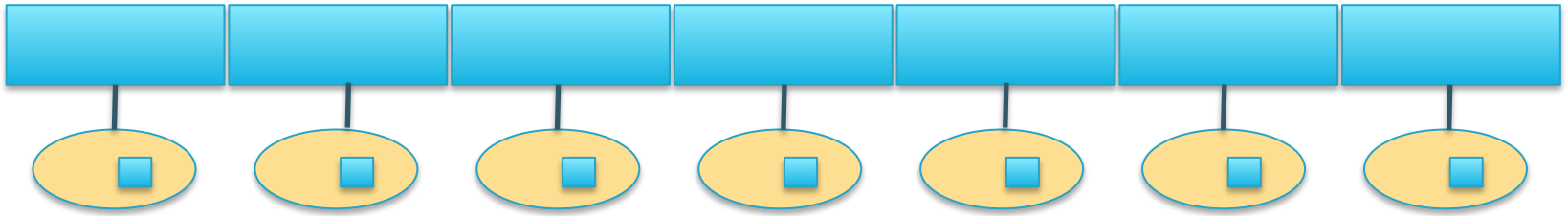
  . . .

Big Data Set (Sales) Joined with Small Data Set (Stores)



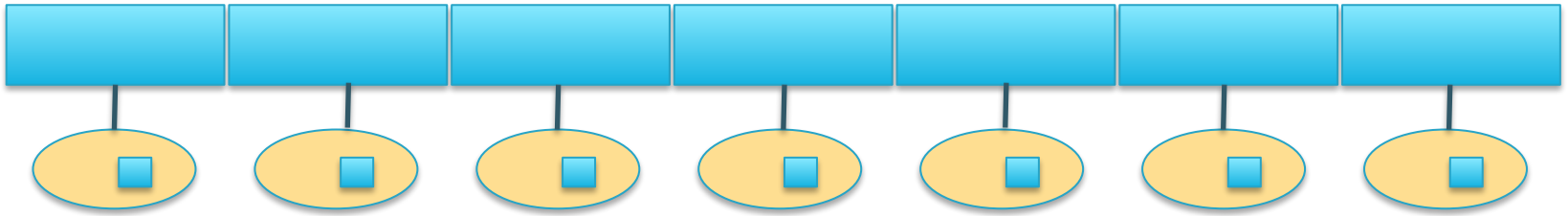
- Send copy of stores to all partitions of sales
- Keep store data in memory as hash table with join key as key
- Stream all partitions of sales in parallel, looking up key for store data
- Output joined records
- Fast
  - Just requires sending small data set to each partition of large data set
  - Executes the join in parallel

System Support



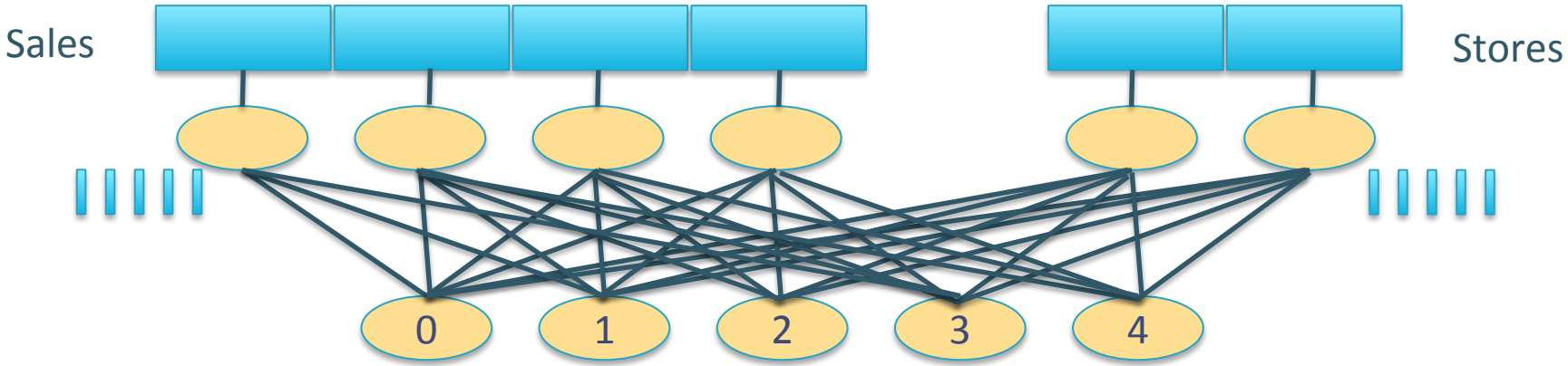
- Hive – map join
  - `SET hive.auto.convert.join=true`
  - `SET hive.mapjoin.smalltable.filesize=25000000`
- Impala – broadcast join
  - COMPUTE STATS (or)
  - [BROADCAST] and [SHUFFLE] hints

System Support



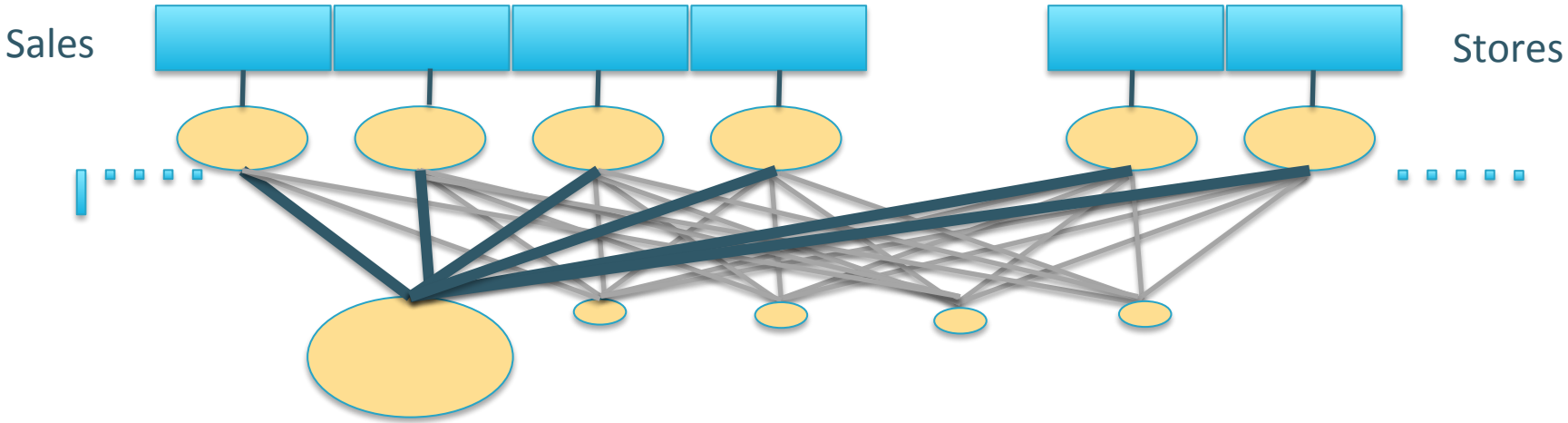
- Core Spark – don't use RDD join
  - Define small data set as hash table broadcast variable
  - Big data set is mapped RDD, looking up join key
- SparkSQL – broadcast join
  - `spark.sql.autoBroadcastJoinThreshold`
- Pig – replicated join

Big Data Set Joined with Big Data Set



- Partition and shuffle by join key
- Each reducer processes subset of records and outputs joined records
- A lot of data! Both data sets sent over network
- Hive documentation refers to as “common join”

Skewed Data Joins



- What if 80% of the sales were made at store 500?
  - The reducer processing store 500 records would do at least 80% of the work
  - The resulting joined file would have at least 80% of the records

Skewed Data Joins in Hive

- Hive supports “compile-time” and “run-time” skewed joins

- Compile-time support requires popular values in metadata:

```
create table Sales skewed by (storeID) on (500);
SET hive.optimize.skewjoin.compiletime=true;
```

- At compile time, the plan is broken into different joins:
  - One for the popular values -- map join
  - Other for the remaining values – common join

Skewed Data Joins in Hive

- Run-time support computes “popular” values for join key
  - Must be activated

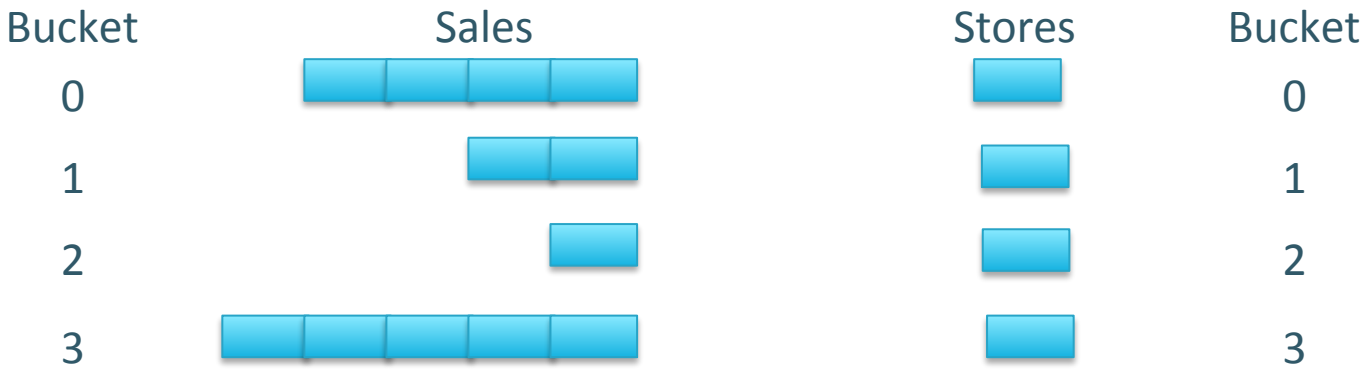
```
SET hive.optimize.skewjoin=true;
```

  - Set threshold for being popular

```
SET hive.skewjoin.key=100000;
```

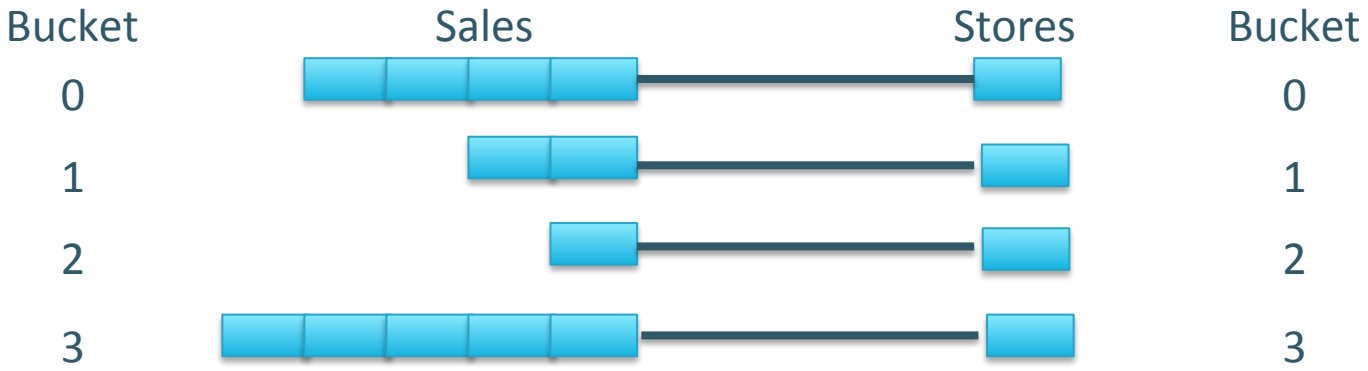
- Hive then performs
  - a map join for the popular values
  - a common join for the rest

Bucketed Joins in Hive



- Recall bucket tables are stored as multiple buckets defined by bucket column.
  - One file per bucket
- To improve join performance with buckets:
  - Both tables must be bucketed on join column
  - Number of buckets of each table must be multiples each other

Bucketed Joins in Hive



- Map join, bucket by bucket

```
SET hive.optimize.bucketmapjoin=true;
```

- Sort merge join if buckets are too big for map join

```
SET hive.optimize.bucketmapjoin.sortedmerge=true;
```

Incremental Inner Join, One Data Set Grows

- Additional sales, same stores
- At time T_0:
  - Compute inner join sales and stores
  - Store the result
- At time T_1:
  - New sales arrive (sales'), stores unchanged
  - Compute inner join of sales' and stores
  - Append the result
  - No need to re-join sales and stores

Incremental Inner Join, Both Data Sets Grow

- Additional sales, same stores
- At time T_0:
  - Compute inner join sales and stores
  - Store the result
- At time T_1:
  - New sales arrive (sales')
  - New stores arrive (stores')
  - Compute inner join of sales' and stores
  - Compute inner join of sales and stores'
  - Compute inner join of sales' and stores'
  - Append the results
  - No need to re-join sales and stores

Incremental Outer Joins

- Incremental outer joins are more complex:
  - Unmatched joined records could be replaced with joined records
  - May not be worth it
- Demo will explore a single example:
  - Left outer join sales and stores
    - Sales with unmatched stores are included
  - Later additional stores arrive

Demo and Exercises

- SparkSQL doesn’t support skewed join optimization directly
  - Demo will show how to achieve it in SparkSQL code
- SparkSQL doesn’t support bucketed join optimization directly
  - Demo will show how to achieve it in SparkSQL code

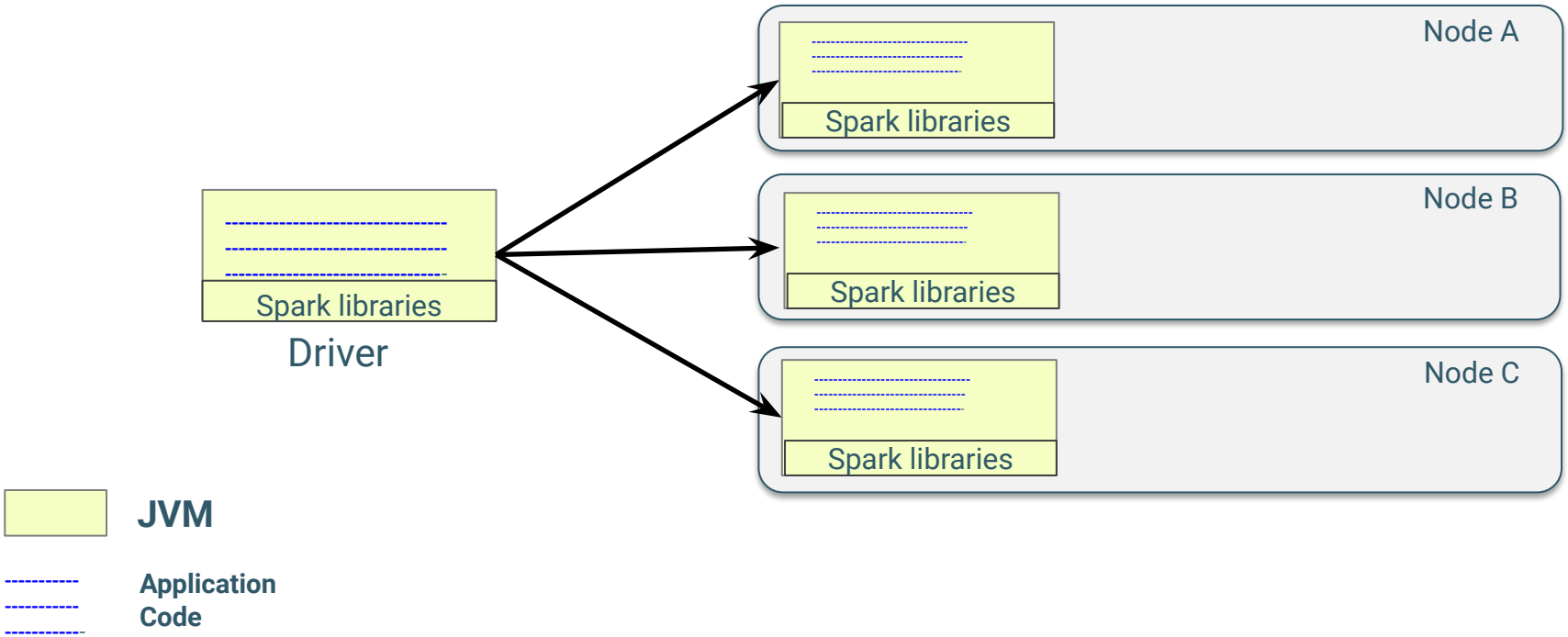
CLOUDERA
Educational Services

Demos/Labs:
Improving Joins Performance
Key Salting

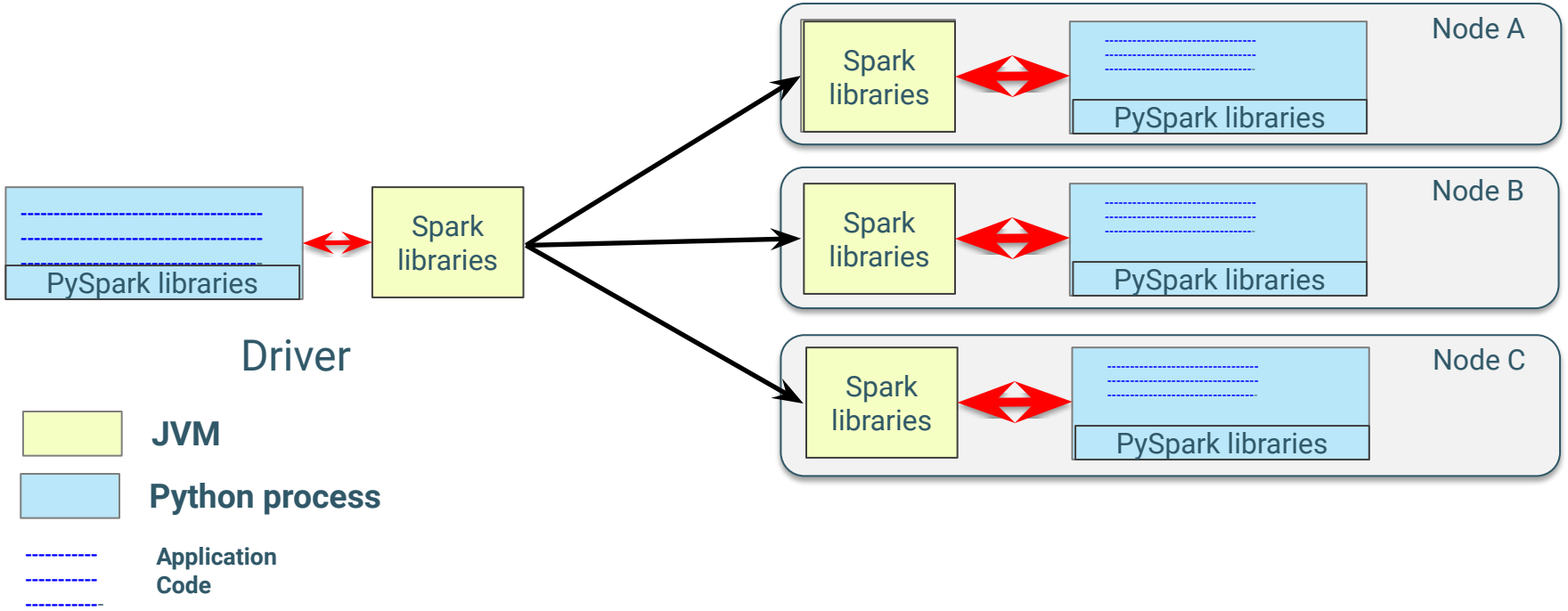
CLOUDERA
Educational Services

Pyspark Overhead

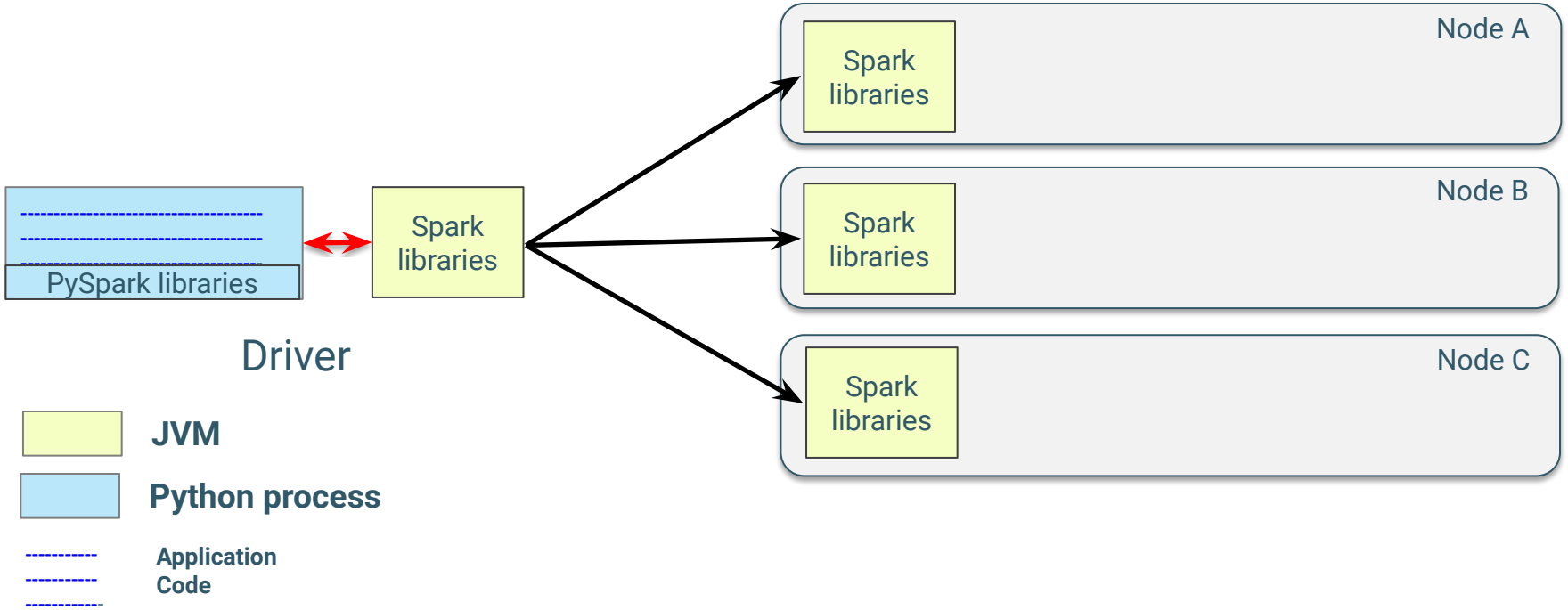
Scala / Java Spark Application: RDDs



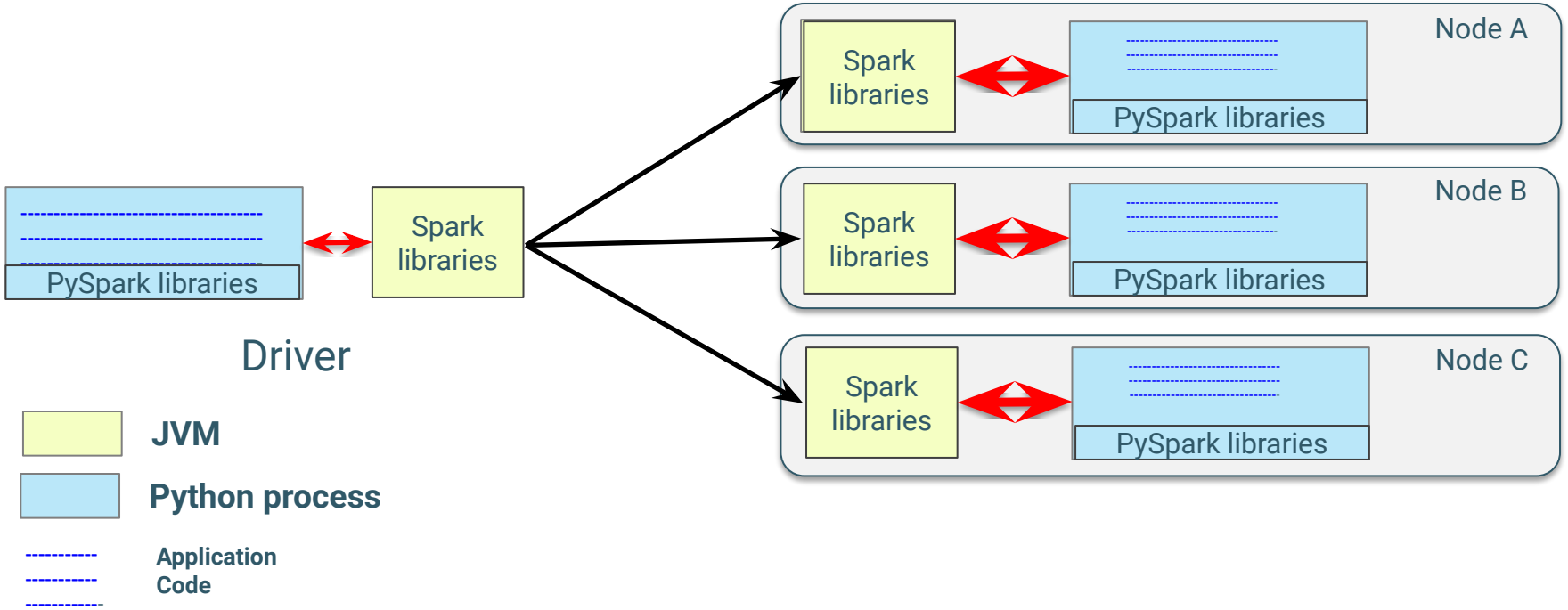
Python Spark Application: RDDs



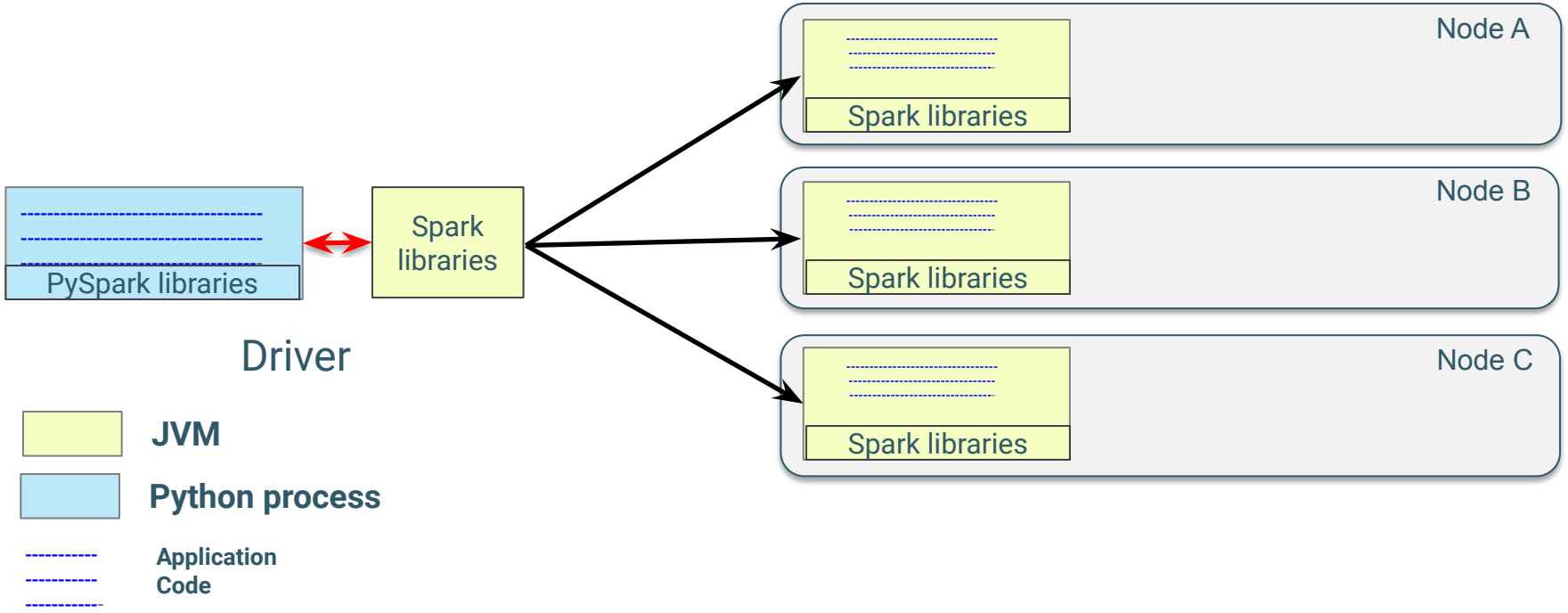
PySpark DataFrames without UDFs



PySpark DataFrames with Python UDFs



PySpark DataFrames with Scala/Java UDFs



CLOUDERA
Educational Services

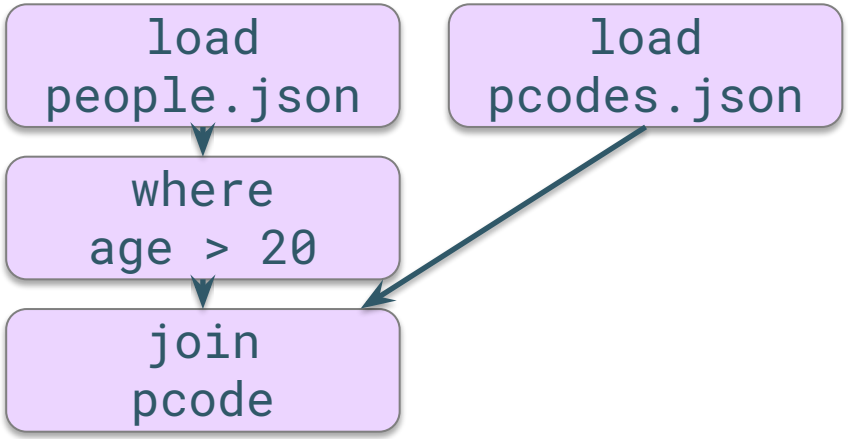
Demos/Labs:
Pyspark Overhead

CLOUDERA
Educational Services

Caching Data for Reuse

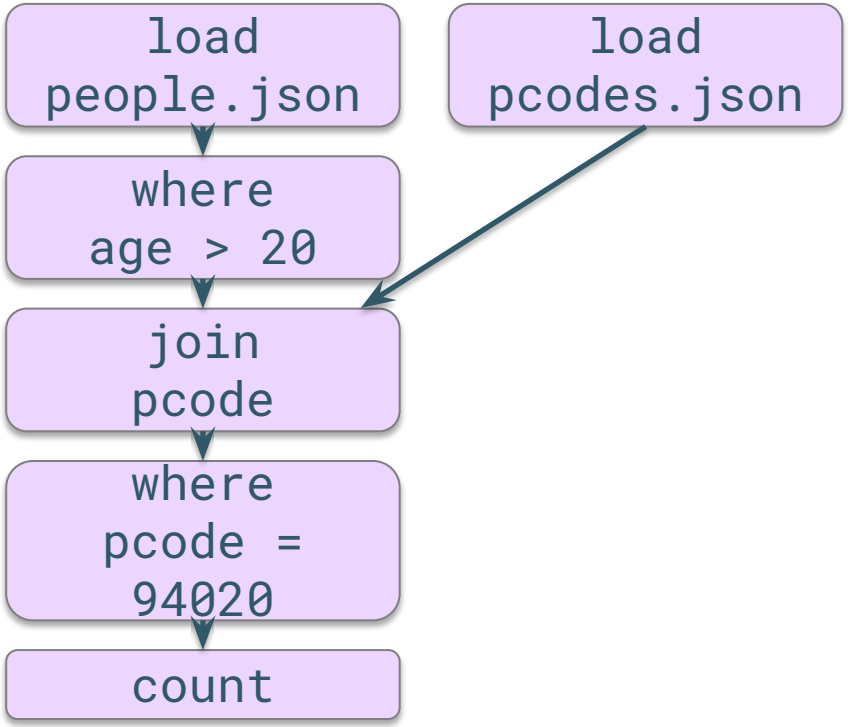
Example: DataFrame Caching (1)

```
over20DF = spark.read.        \
      json("people.json").    \
      where("age > 20")
pcodesDF = spark.read.        \
      json("pcodes.json").    \
      persist()
joinedDF = over20DF.          \
      where("pcode = 94020"). \
      count()
```



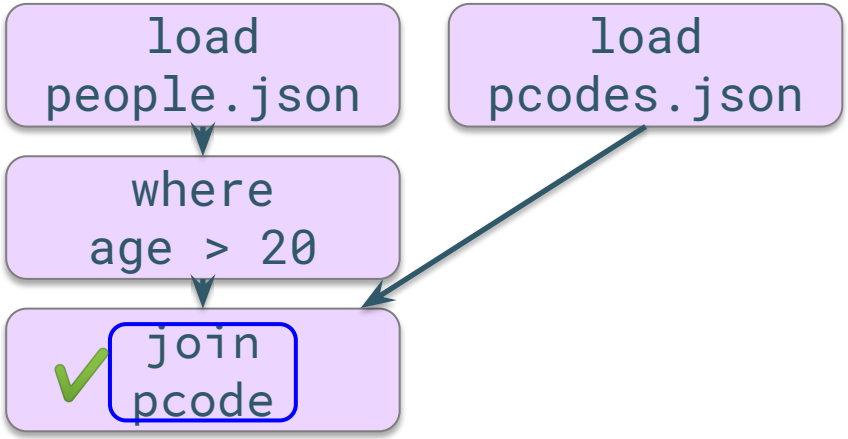
Example: DataFrame Caching (2)

```
over20DF = spark.read.        \
      json("people.json").    \
      where("age > 20")
pcodesDF = spark.read.        \
      json("pcodes.json")
joinedDF = over20DF.          \
      where("pcode = 94020"). \
      count()
```



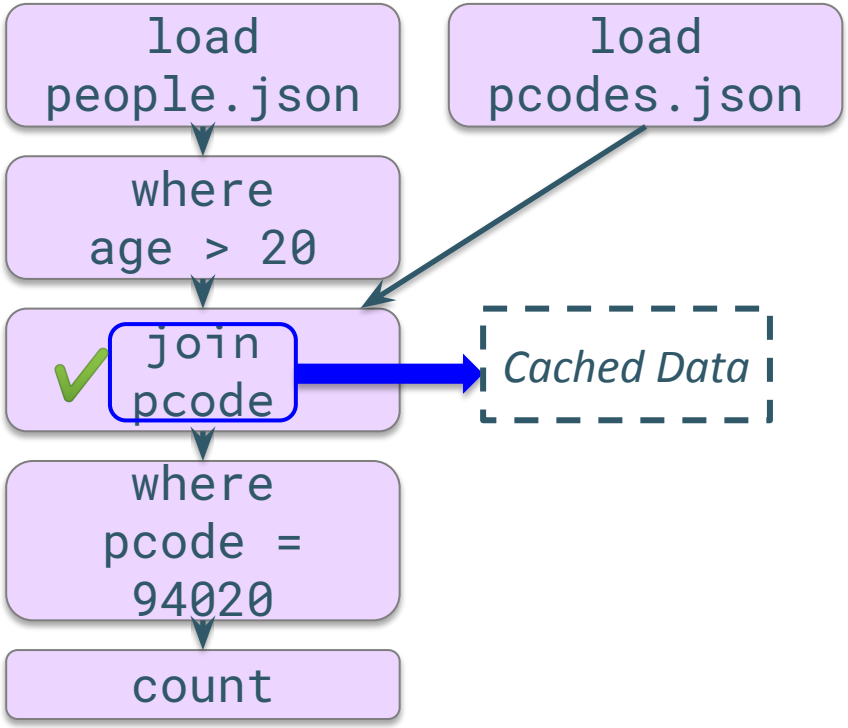
Example: DataFrame Caching (3)

```
over20DF = spark.read.        \
      json("people.json").    \
      where("age > 20")
pcodesDF = spark.read.        \
      json("pcodes.json").    \
      persist()
```



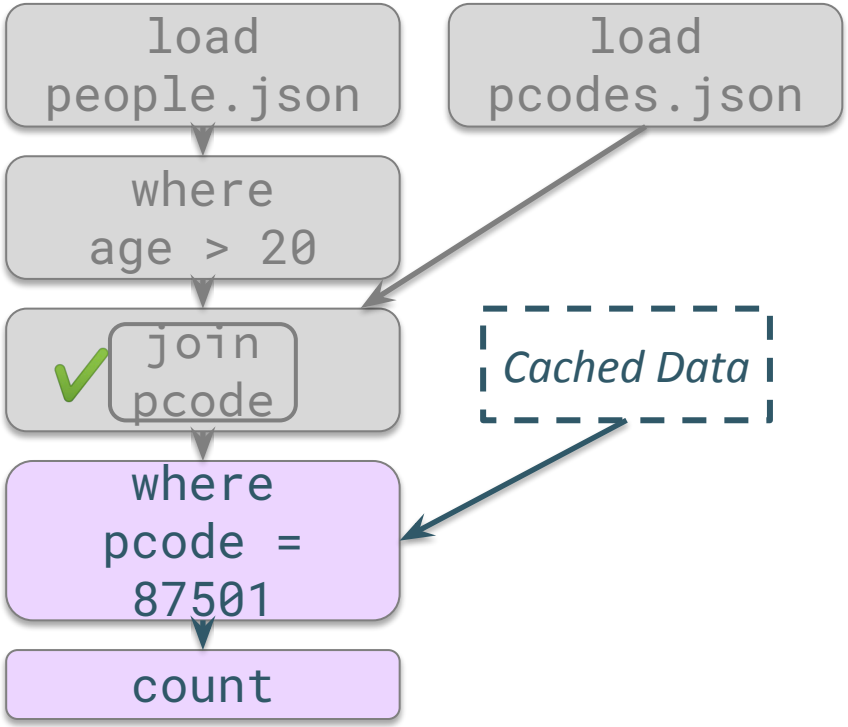
Example: DataFrame Caching (4)

```
over20DF = spark.read.          \
      json("people.json").      \
      where("age > 20")
pcodesDF = spark.read.          \
      json("pcodes.json").      \
      persist()
joinedDF = over20DF.            \
      where("pcode = 94020"). \
      count()
joinedDF.                       \
      where("pcode = 87501"). \
      count()
```

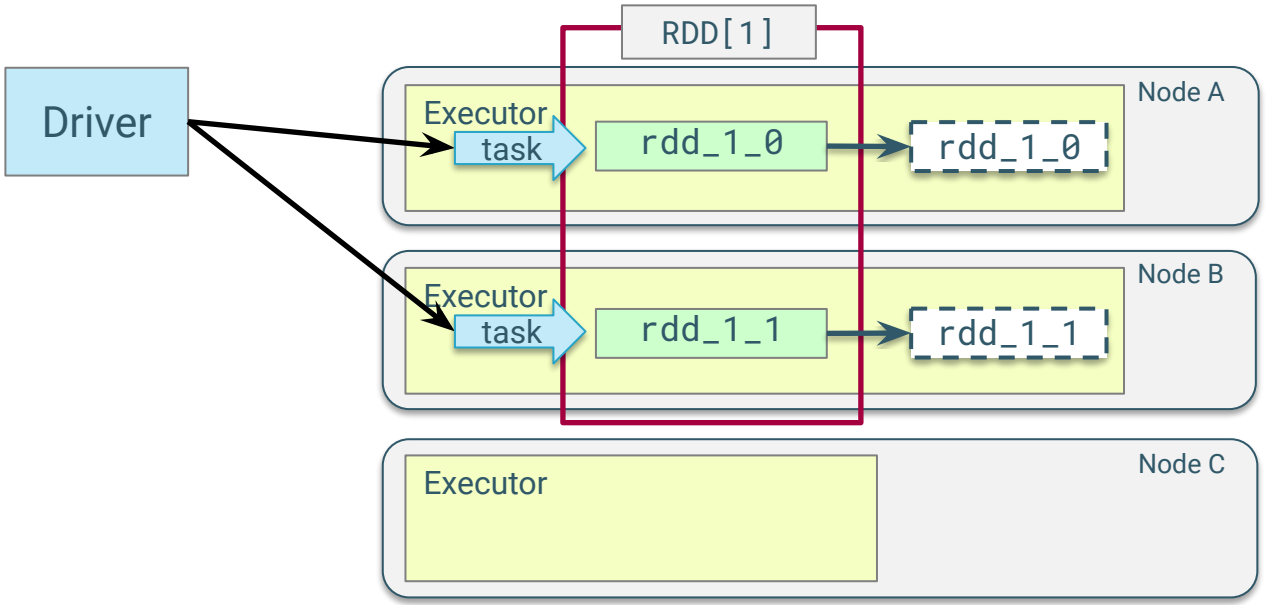


Example: DataFrame Caching (5)

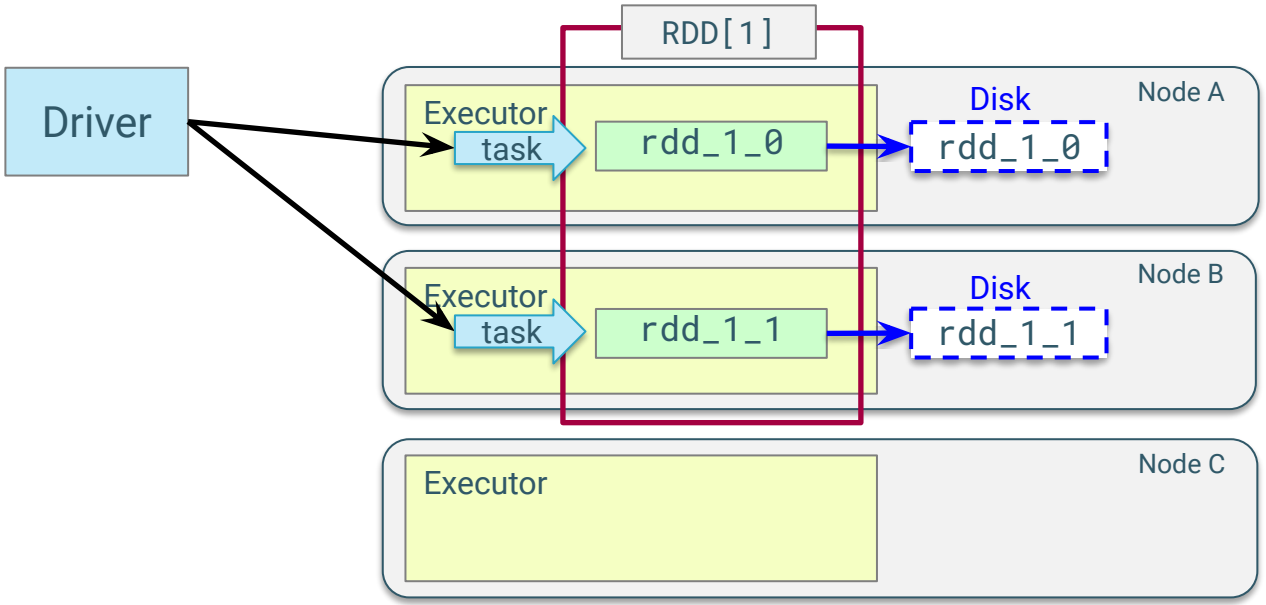
```
over20DF = spark.read.          \
      json("people.json").      \
      where("age > 20")
pcodesDF = spark.read.          \
      json("pcodes.json").      \
      persist()
joinedDF = over20DF.            \
      where("pcode = 94020"). \
      count()
joinedDF.                       \
      where("pcode = 87501"). \
      count()
```



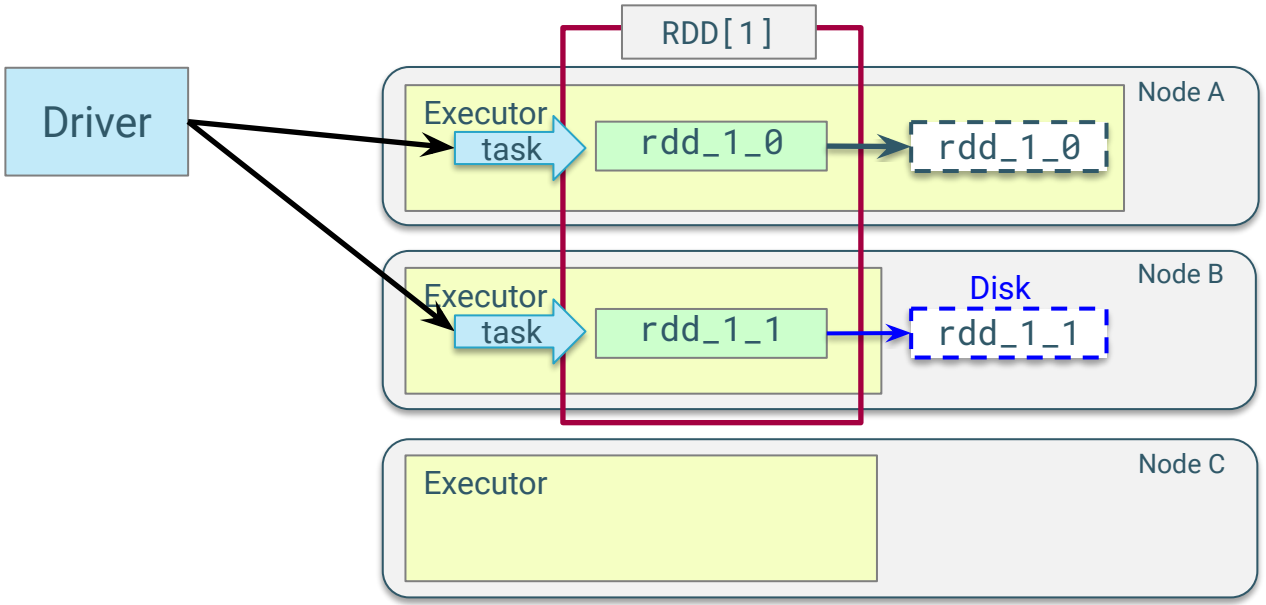
Memory Caching



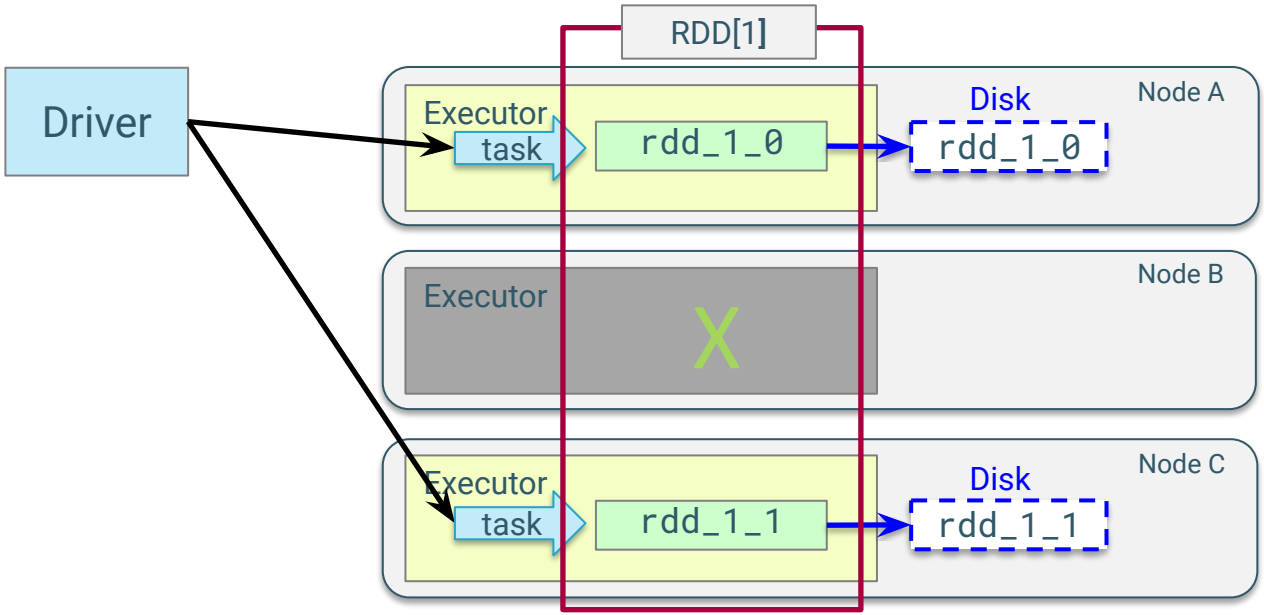
Disk Caching



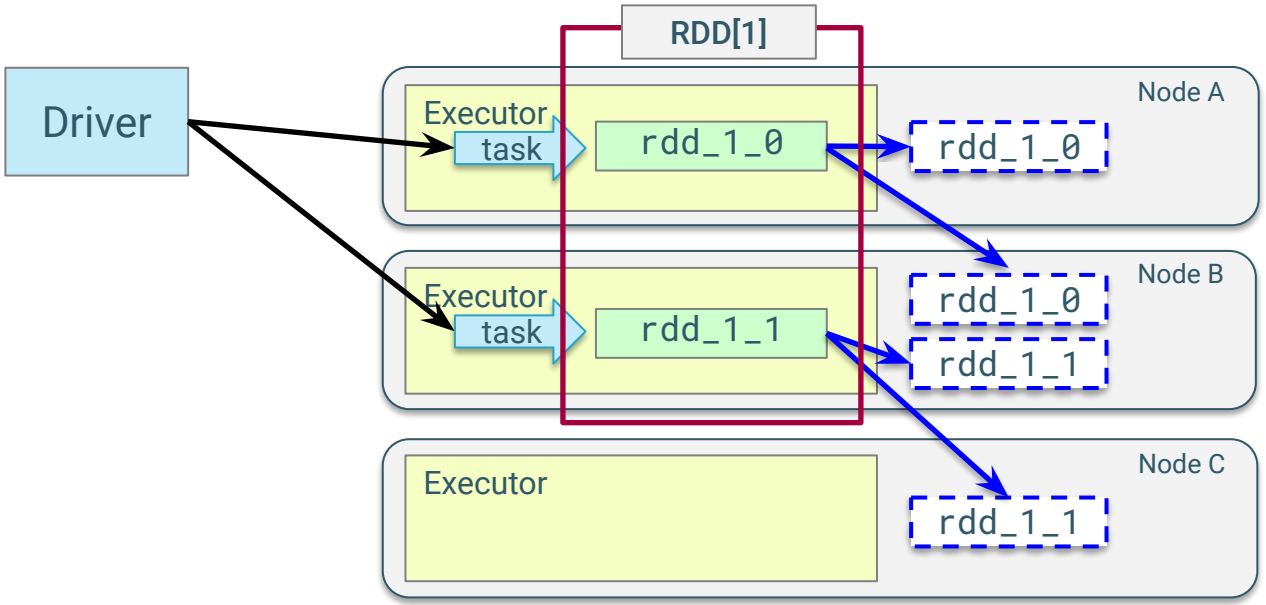
Memory and Disk Caching



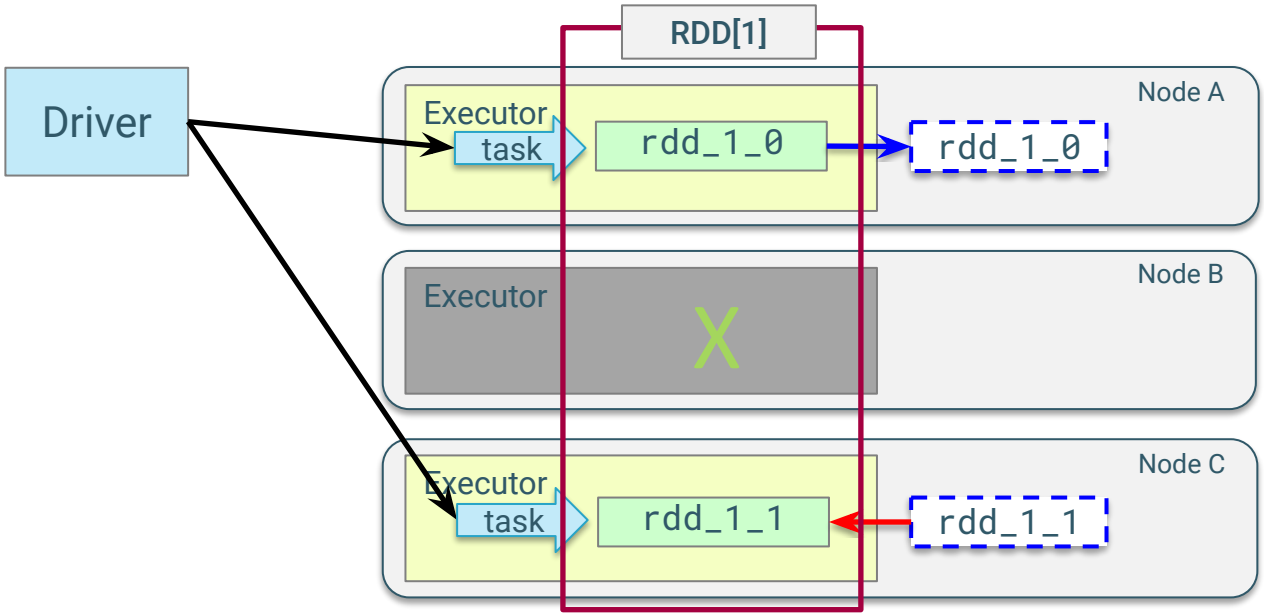
Failover Without Replication



Caching with Replicated Partitions



Failover with Replicated Partitions



CLOUDERA
Educational Services

Demos/Labs:
Persistence

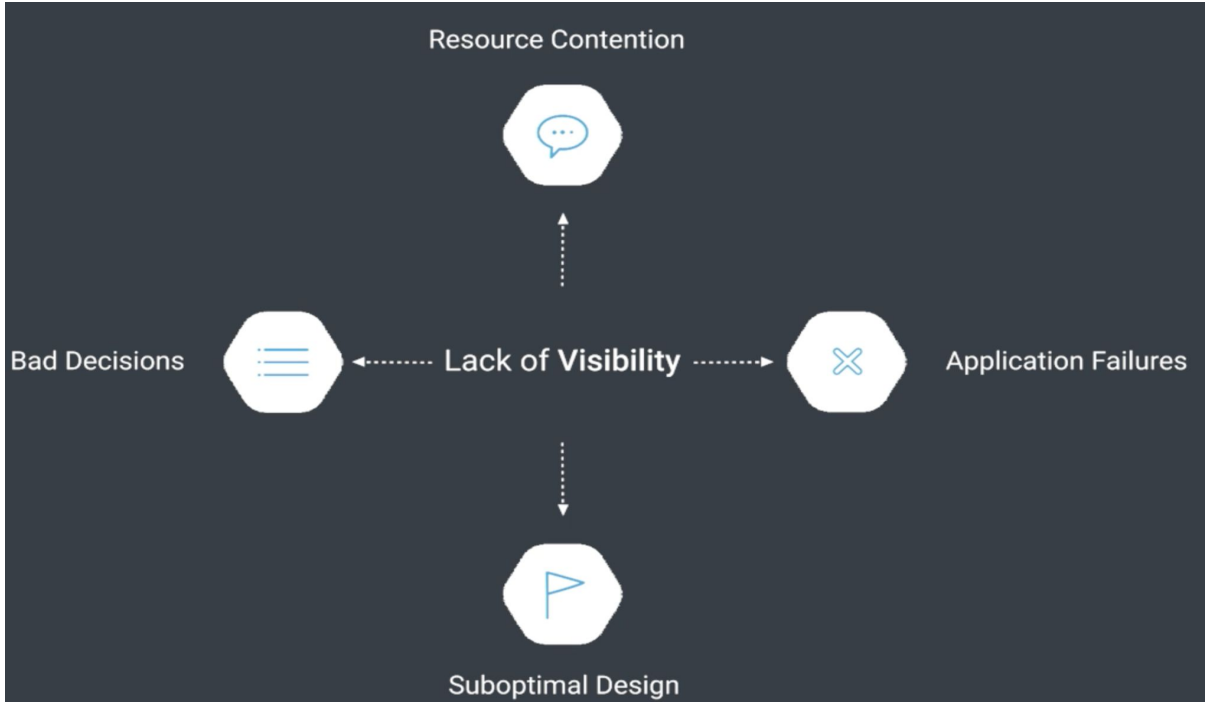
CLOUDERA
Educational Services

Workload Manager (WXM) Introduction

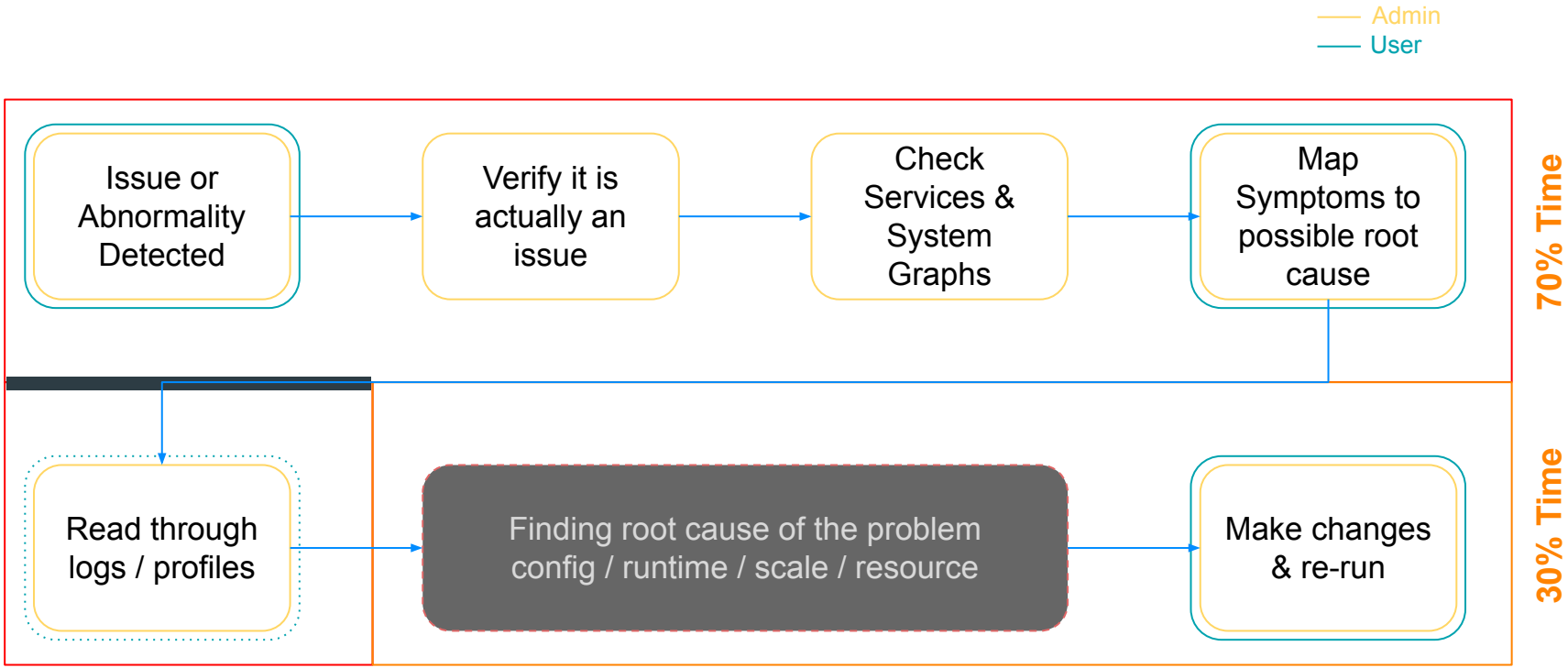
CLOUDERA
Educational Services

Why Should You Care?

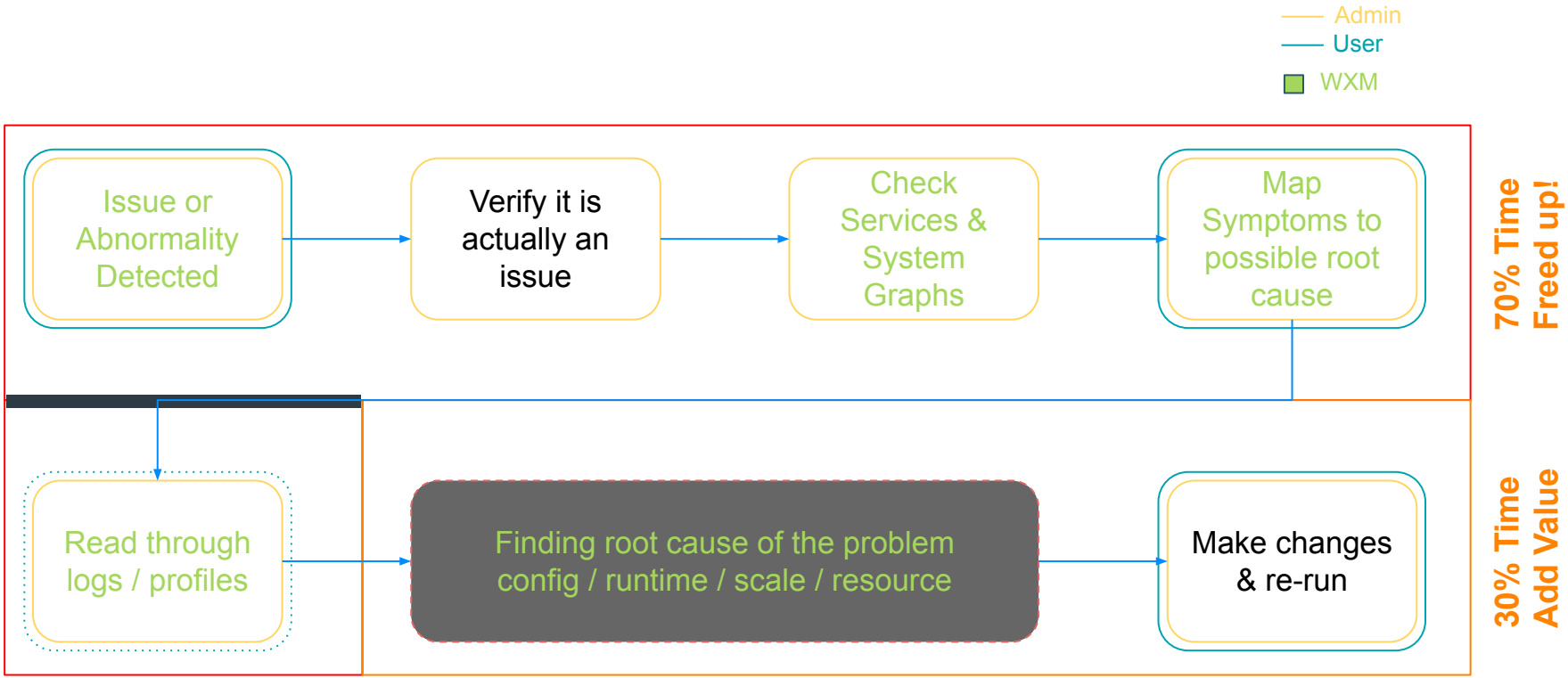
Workload Manager Increases Visibility



Typical Issue Lifecycle



Typical Issue Lifecycle with Workload Manager



CLOUDERA
Educational Services

What Is It?

What is Workload Manager?

- Workload Manager is a tool that provides
  - insights to help you gain in-depth understanding of the workloads you send to clusters managed by Cloudera Manager
  - information that can be used for troubleshooting failed jobs and optimizing slow jobs that run on those clusters
- It uses information about job executions to display metrics about the performance of a job and compares the current run of a job to previous runs of the same job by creating baselines.
- You can use the knowledge gained from this information to identify and address abnormal or degraded performance or potential performance improvements

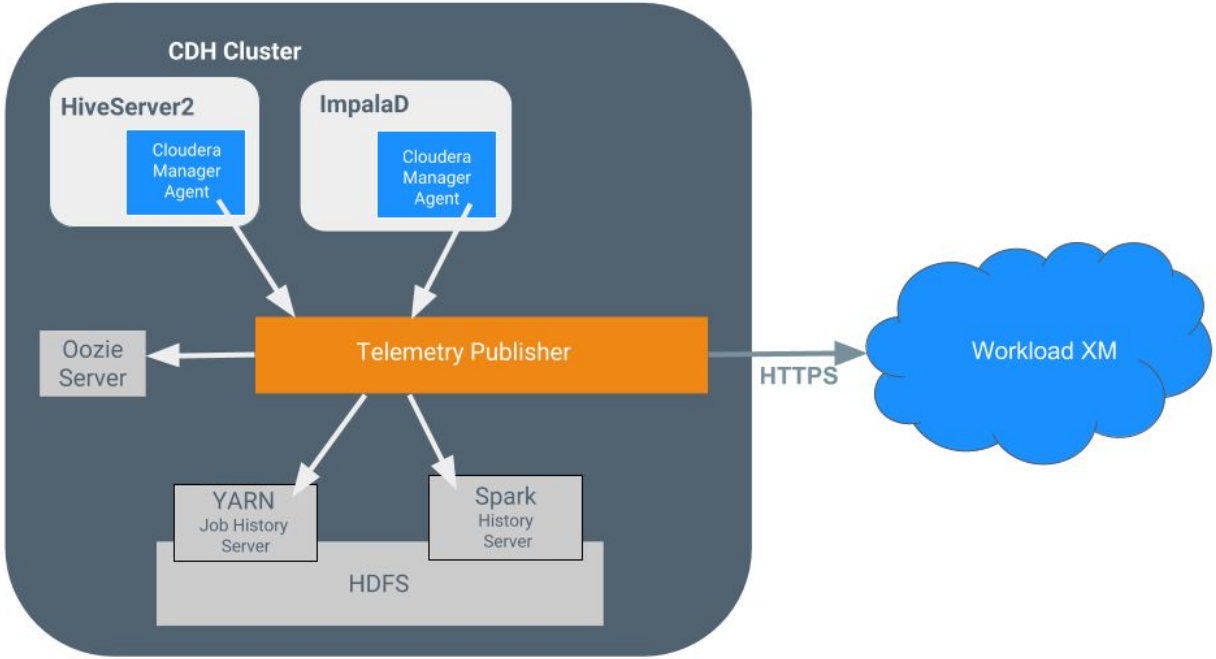
CLOUDERA
Educational Services

How Does It Work?

How Does it Work?

- When you enable Workload Manager, the Cloudera Management Service starts the Telemetry Publisher role
- Telemetry Publisher collects and transmits metrics as well as configuration and log files from Impala, Oozie, Hive, YARN, and Spark services for jobs running on CDH clusters to Workload Manager
- This data is collected in the following ways:
  - **Pull** — Telemetry Publisher pulls diagnostic data from these services periodically (once per minute, by default). That's for Oozie, YARN, and Spark
  - **Push** — A Cloudera Manager Agent pushes diagnostic data from these services to Telemetry Publisher within 5 seconds after a job finishes. That's for Hive and Impala
- After the diagnostic data reaches Telemetry Publisher, it is stored temporarily in its data directory and periodically (once per minute) exported to Workload Manager

The Telemetry Publisher Connections



CLOUDERA
Educational Services

Who Is It For?

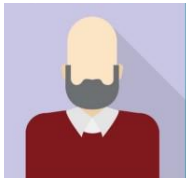
Who Is It For?

Hadoop / System / Database Admin

Data Architects

Data Engineering / BI Developer

System Integrator



Central Support Team

Sales Engg / PS

CLOUDERA
Educational Services

WXM For Developers

CLOUDERA
Educational Services

Diagnostic Data Collection Details

MapReduce Jobs

- Telemetry Publisher polls the YARN Job History Server for recently completed MapReduce jobs
- For each of these jobs, Telemetry Publisher collects the configuration and jhist file, which is the job history file that contains job and task counters, from HDFS
- Telemetry Publisher can be configured to collect MapReduce task logs from HDFS and send them to Workload Manager
  - By default, this log collection is turned off

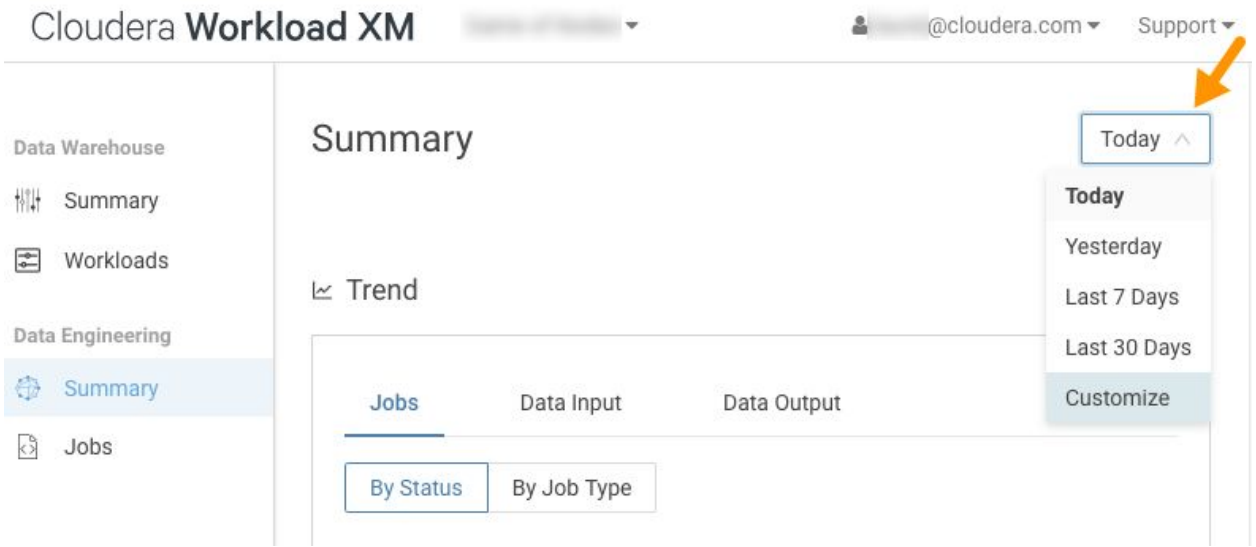
Spark Applications

- Telemetry Publisher polls the Spark History Server for recently completed Spark applications.
- For each of these applications, Telemetry Publisher collects their event log from HDFS.
- Telemetry Publisher only collects Spark application data from Spark version 2.2 and later.
- Telemetry Publisher can be configured to collect the executor logs of Spark applications from HDFS and send them to Workload Manager,
    - but this data collection is turned off by default.
- **Important**: CDH version 5.x is packaged with Spark 1.6 so you cannot configure Telemetry Publisher data collection for CDH 5.x clusters unless you are using CDS 2.2 Powered by Apache Spark or later versions with those clusters

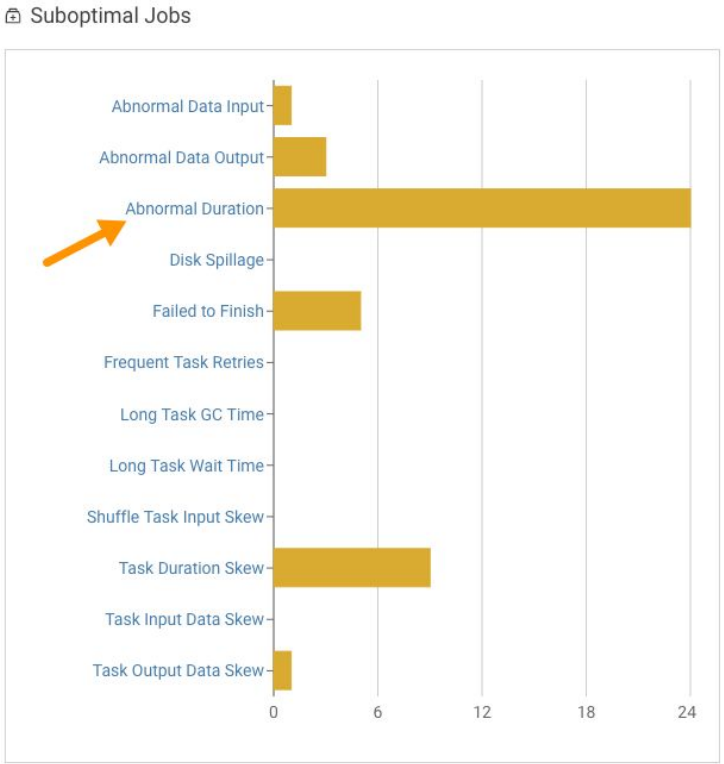
CLOUDERA
Educational Services

Use Case Example:
Troubleshooting Abnormal Job Durations

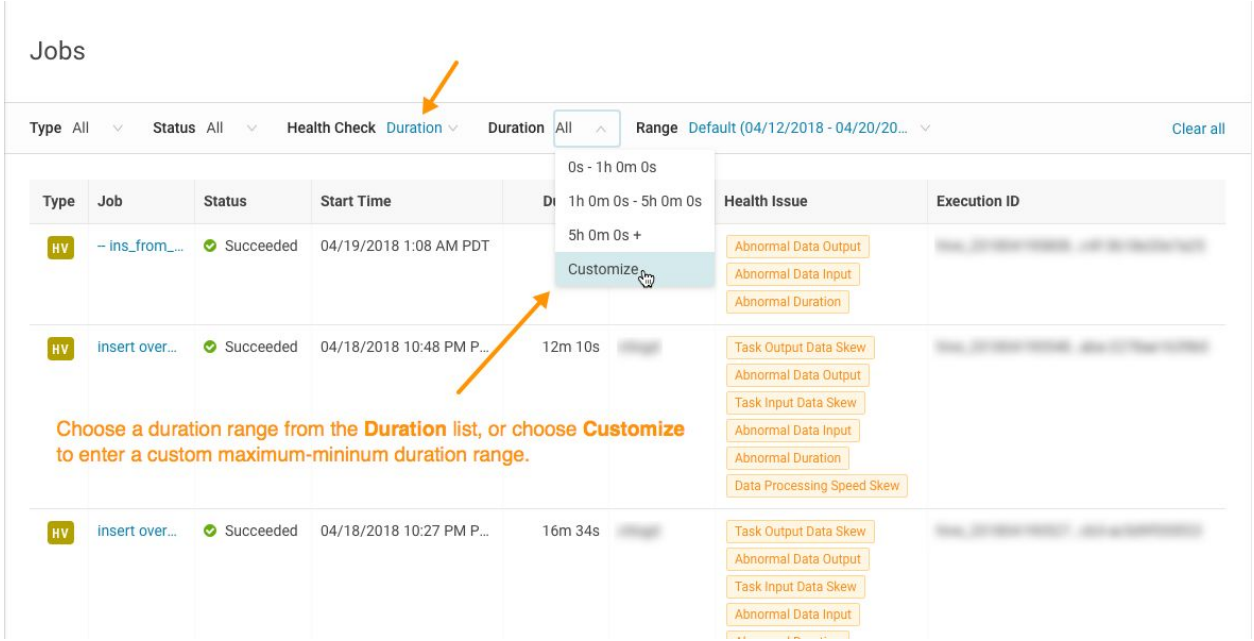
1 - Specify a Time Range



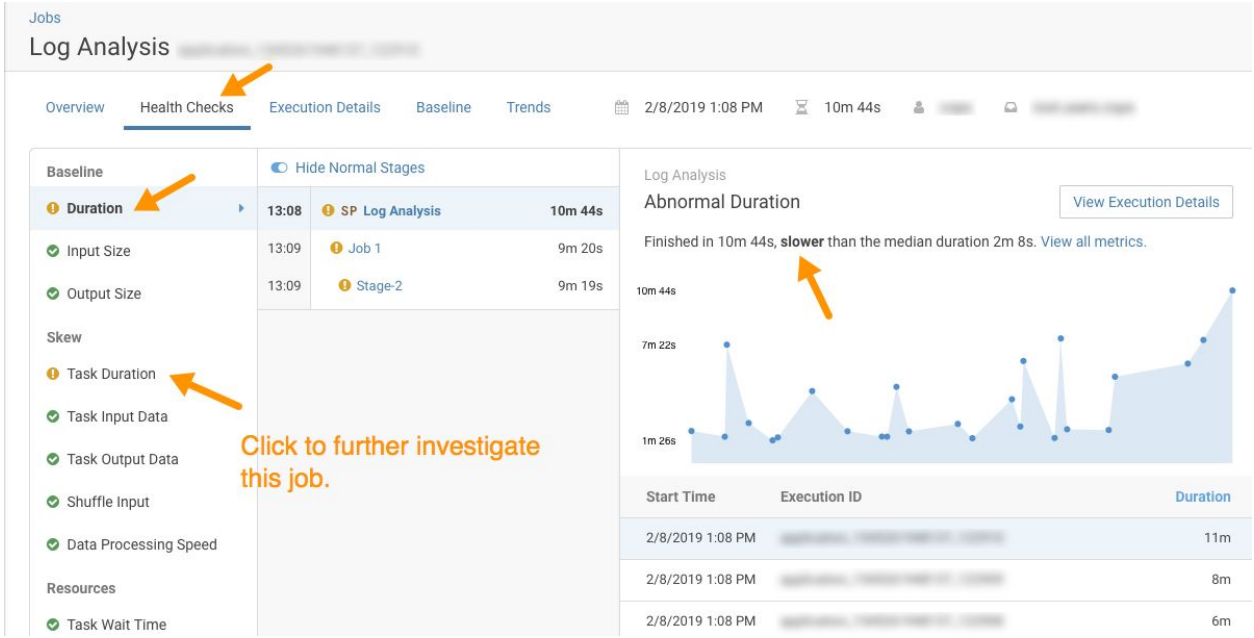
2 - Select Abnormal Duration



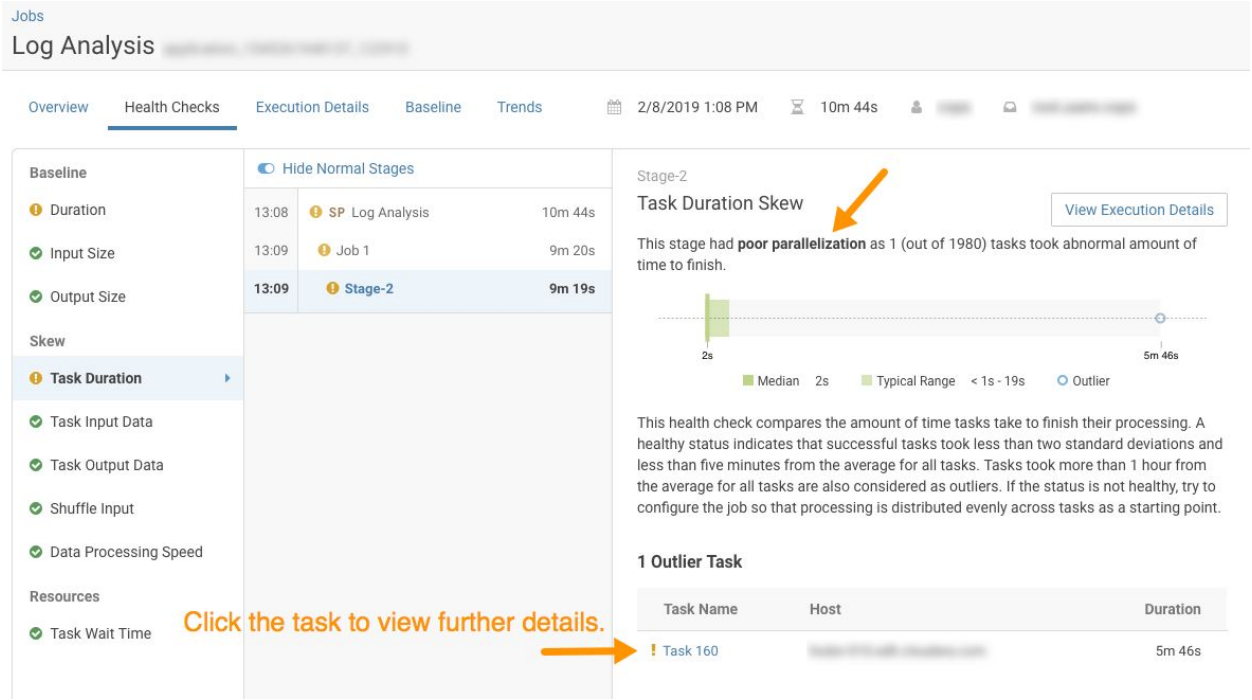
3 - Select a Duration Range



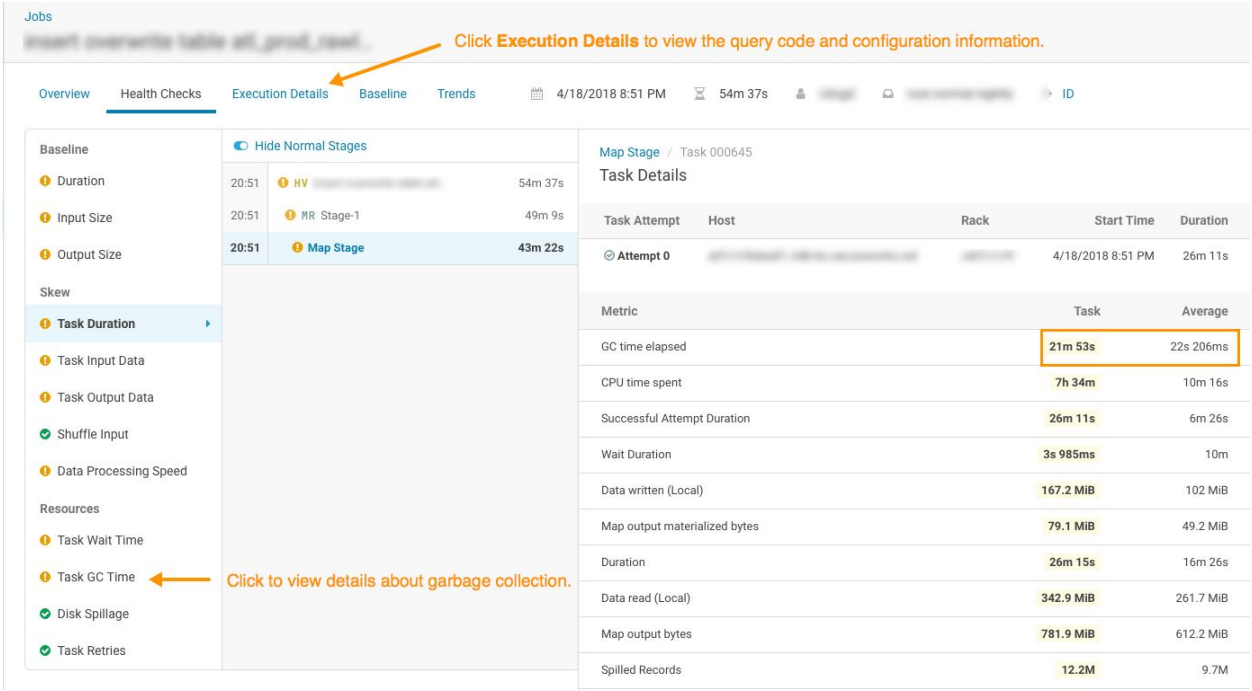
4 - Inspect the Job Details



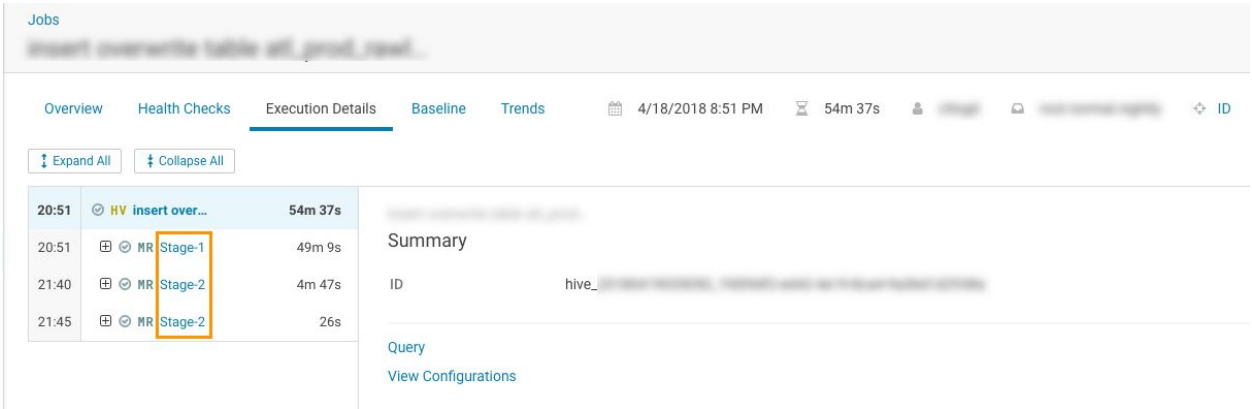
5 - Identify the Outlier Task



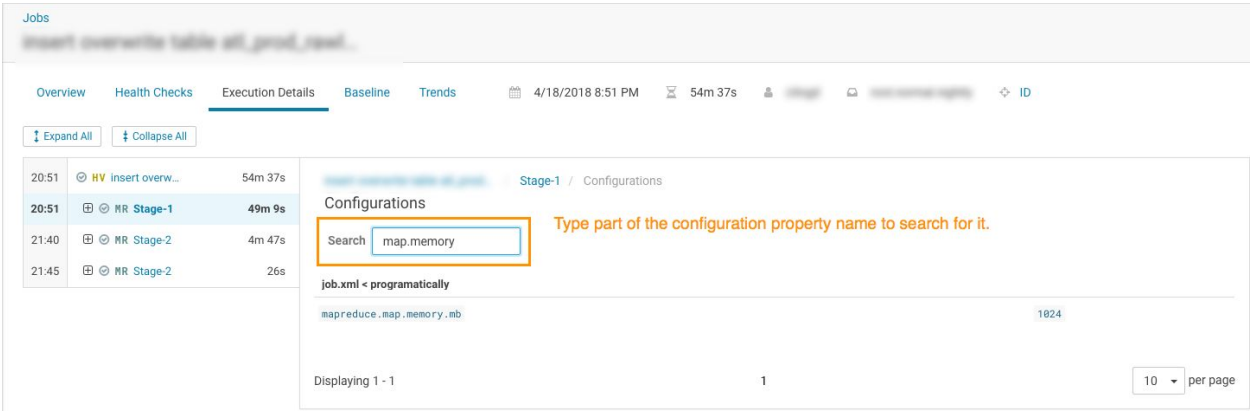
6 - Identify the Cause of the Abnormal Duration



7 - Drill Down on the Execution Details



8 - View the Related Configuration Settings



CLOUDERA
Educational Services

Demo:

WXM for Spark Deep Dive (Mar 2020)

CLOUDERA
Educational Services

What's New in Spark 3.0?

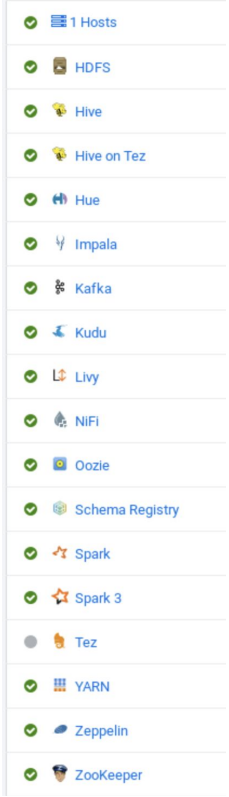
Spark 3.0 in CDP

Spark 3 service can coexist with Spark 2 service.

Spark History Server port - Spark 2 - 18088, Spark 3 - 18089

This latest release does not support the following features:

- Hive Warehouse Connector
- Kudu
- HBase Connector
- Oozie
- Zeppelin

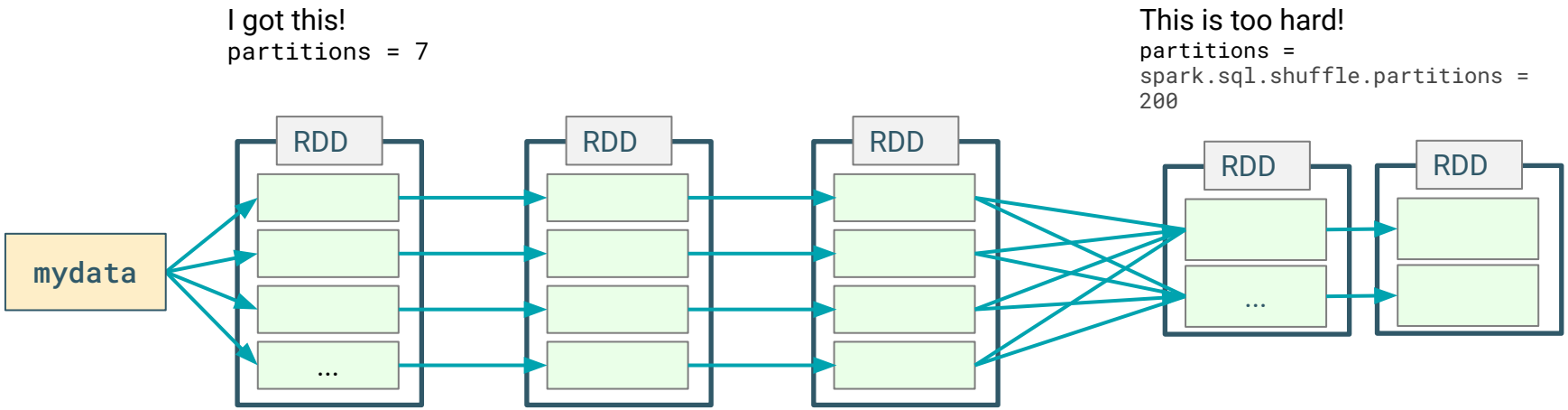


Supported Languages

- Spark 3 will move to Python 3
- Scala version is upgraded to version 2.12
- In addition it will fully support JDK 11

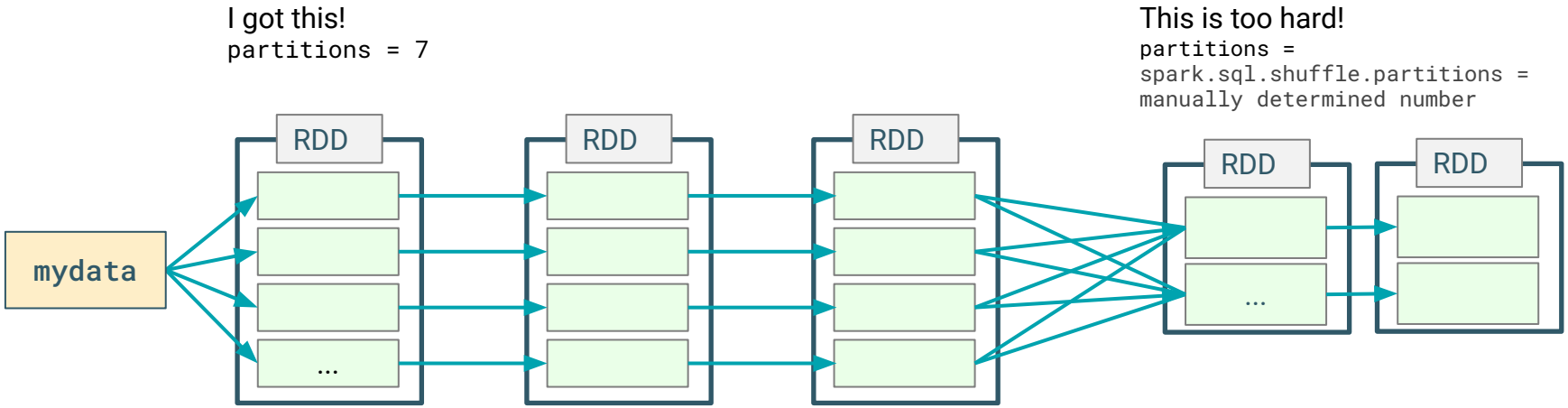
Catalyst Limitations (1)

- Catalyst knows how to optimize the number of partitions for the first stage
- But uses a magic number (200) for the number of partitions for the second stage



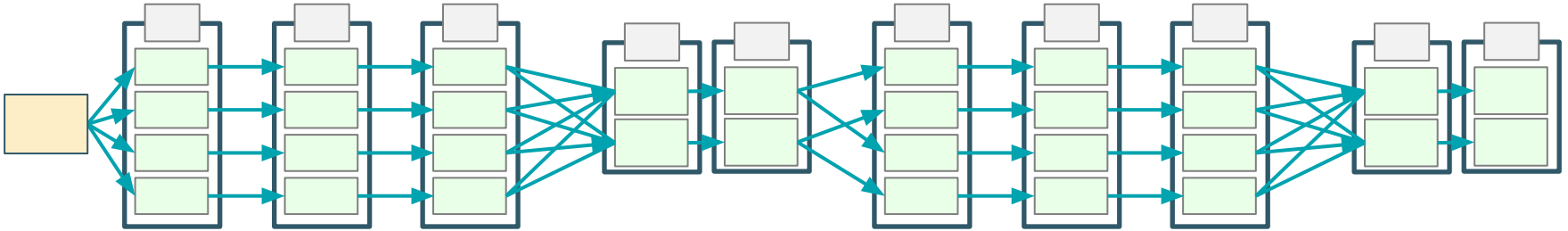
Catalyst Limitations (2)

- To work around this you can manually tune `spark.sql.shuffle.partitions` to a suitable number



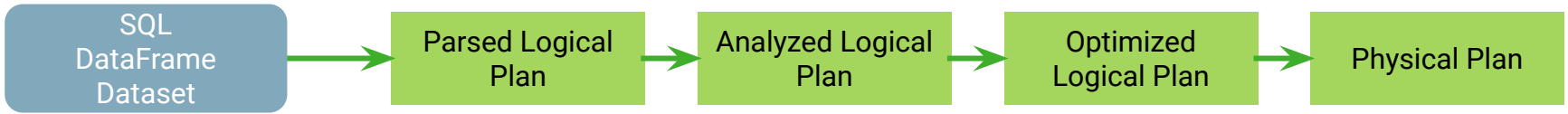
Catalyst Limitations (3)

- But what happens if your query involves more than one shuffle?
- You can only set `spark.sql.shuffle.partitions` once before the execution



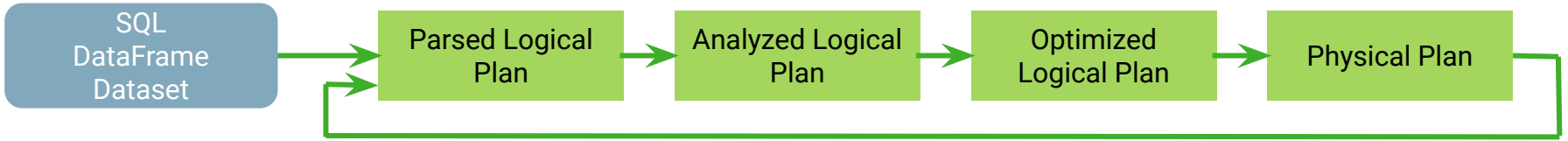
The Flaw in the Initial Catalyst Design

- The plans are completely finalized before any execution
- So the first stage is optimized because Catalyst has all the information required (size and distribution of the data to transform) to find out the ideal number of partitions
- But the later stages not so much because that information is missing and finding it out would equate to actually performing the transformations
- Hence the resort to a default magic number (200) for partitions and missed opportunities to use Broadcast Hash joins instead of Sort Merge Joins



Adaptive Query Execution Design Principle

- Break down the static monolithic plan into new ‘query stages’ abstractions delimited by stages
- Allow Catalyst to plan for one query stage
- Execute the query stage
- Plan the next query stage using the results of the previous query stage and apply all the following optimizations...



Adaptive Query Execution

Adaptive Query Execution, AQE, is a layer on top of the spark catalyst which will modify the spark plan on the fly

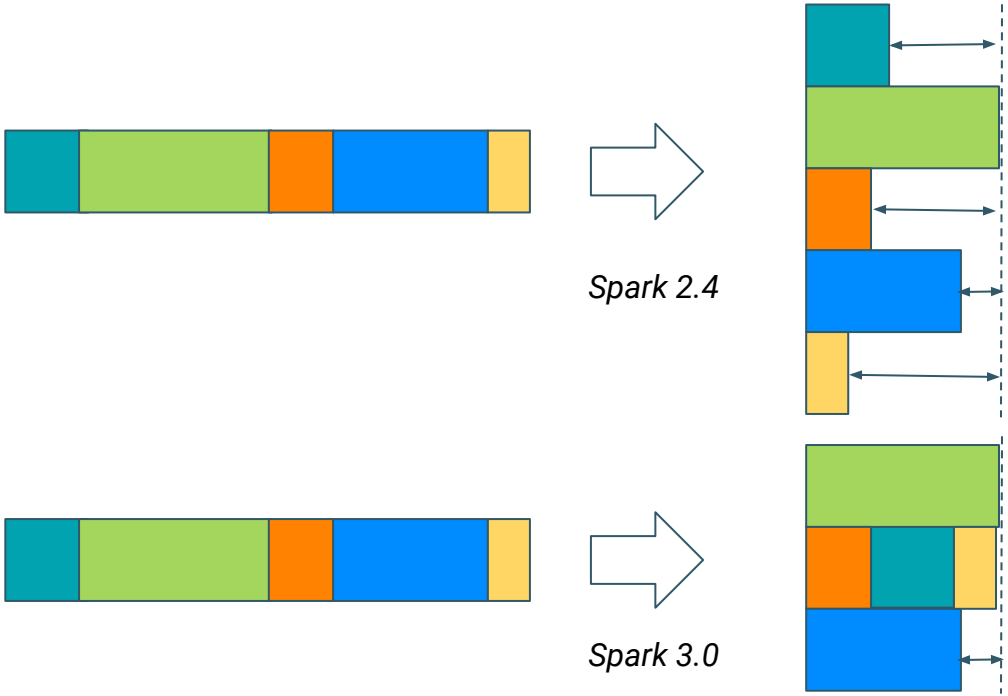
- **Adaptive Number of Shuffle Partitions**
  - When AQE is enabled, the number of shuffle partitions are automatically adjusted (no longer to the default 200 or manually set value).

- **Handling Skew Joins**
  - When adaptive execution is enabled, spark can recognise skew in the query and then automatically redistributes the data to go join faster. This will make skew join go faster than normal joins.

- **Converting Sort Merge Join to BroadCast Join**
  - AQE converts sort-merge join to broadcast hash join when the runtime statistics of any join side is smaller than the broadcast hash join threshold.

```
How to enable AQE - sparkConf.set("spark.sql.adaptive.enabled", "true")
```

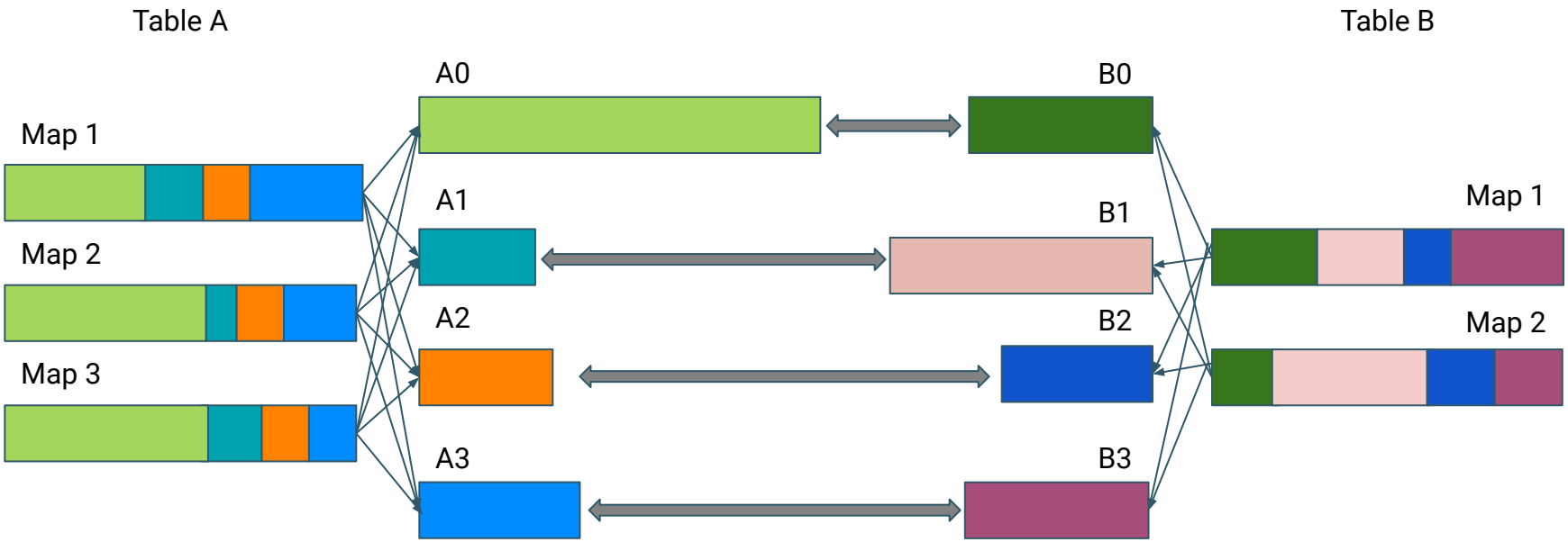
Dynamically Coalesce Shuffle Partitions



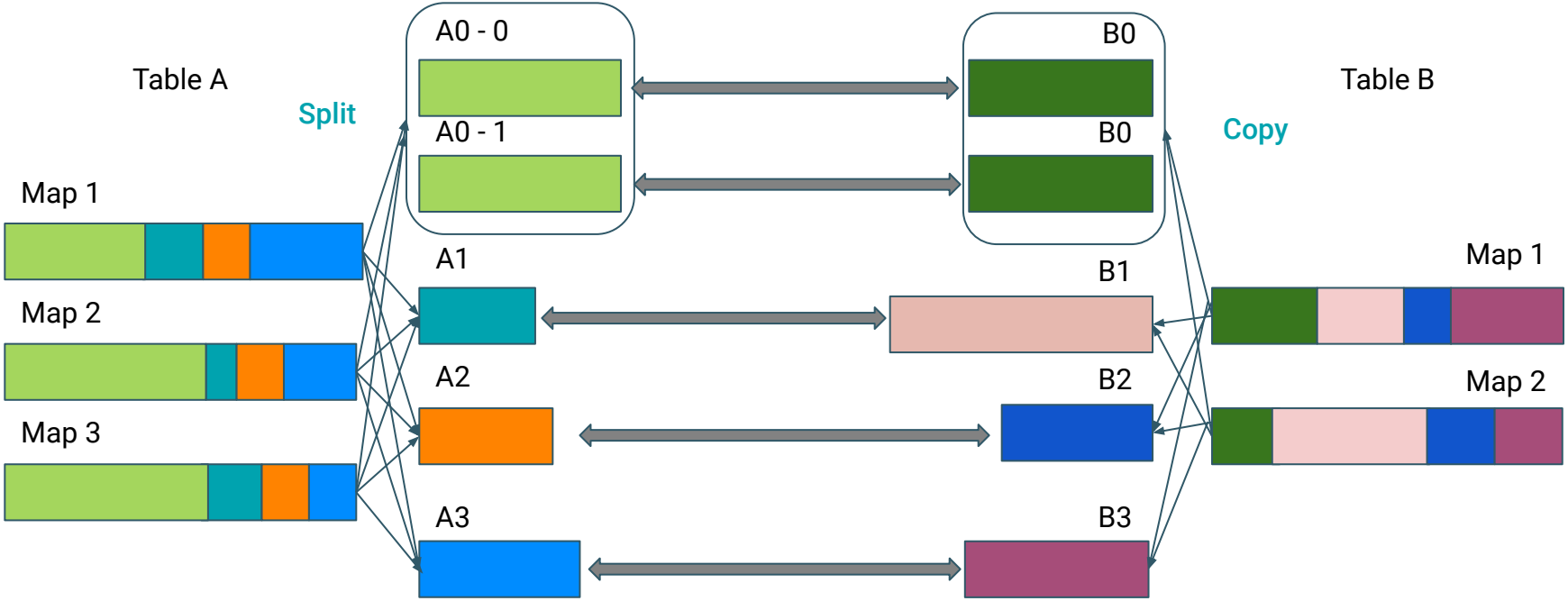
- If the number of shuffle partitions is > than the number of the group by keys then a lot of CPU cycles are lost due to the unbalanced distribution of the keys
- When both properties are set to true, Spark will coalesce contiguous shuffle partitions according to the target size (specified by `spark.sql.adaptive.advisoryPartitionSizeInBytes`), to avoid too many small tasks.

```
spark.sql.adaptive.enabled -> true (false in Spark 3.0)
spark.sql.adaptive.coalescePartitions.enabled -> true (false in Spark 3.0)
```

Skewed Join with no AQE



Skewed Join with AQE



The skew join optimization will thus split partition A0 into to subpartitions and join each of them to the corresponding partition B0 of table B.

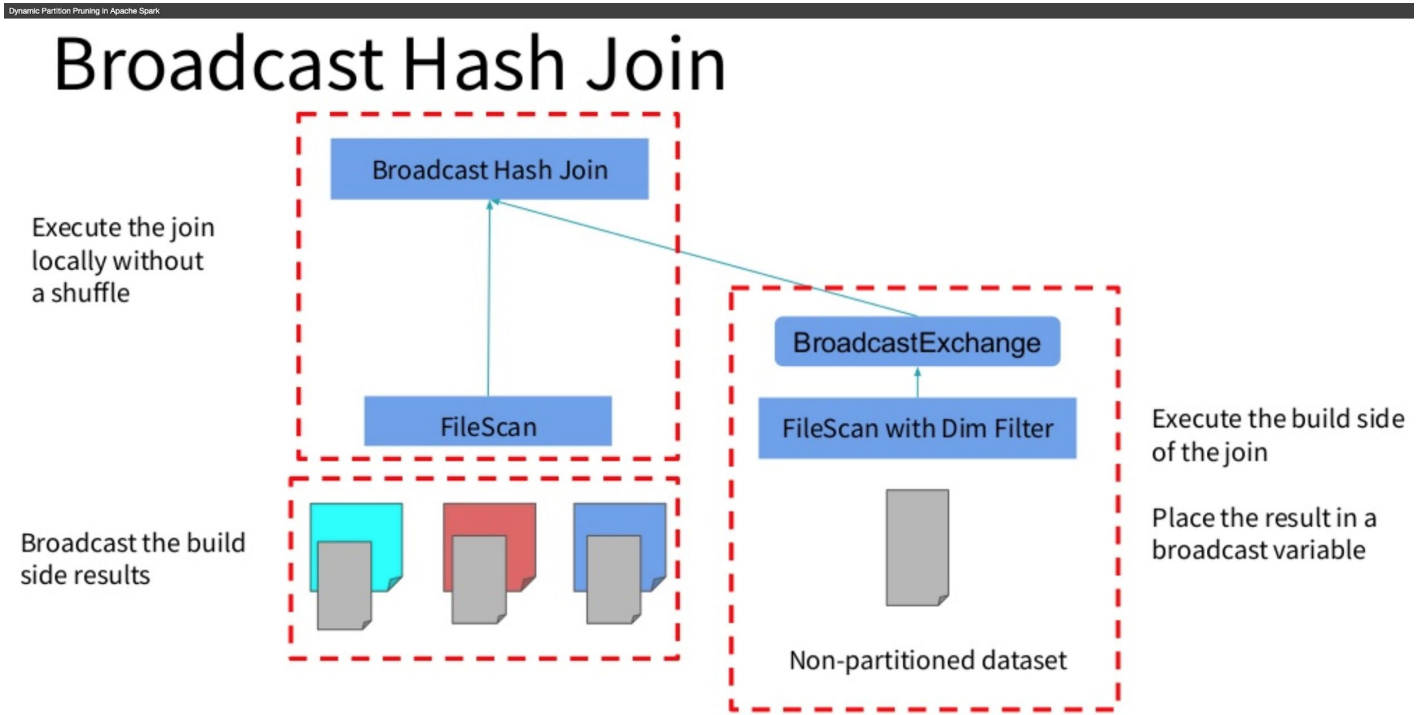
Adaptive Skew Joins Controlling Properties

Property Name	Default	Meaning
`spark.sql.adaptive.skewJoin.enabled`	True	When true and `spark.sql.adaptive.enabled` is true, Spark dynamically handles skew in sort-merge join by splitting (and replicating if needed) skewed partitions.
`spark.sql.adaptive.skewJoin.skewedPartitionFactor`	5	A partition is considered as skewed if its size is larger than this factor multiplying the median partition size and also larger than `spark.sql.adaptive.skewedPartitionThresholdInBytes`.
`spark.sql.adaptive.skewJoin.skewedPartitionThresholdInBytes`	256MB	A partition is considered as skewed if its size in bytes is larger than this threshold and also larger than `spark.sql.adaptive.skewJoin.skewedPartitionFactor` multiplying the median partition size. Ideally this config should be set larger than spark.sql.adaptive.advisoryPartitionSizeInBytes.

Dynamic Partition Pruning

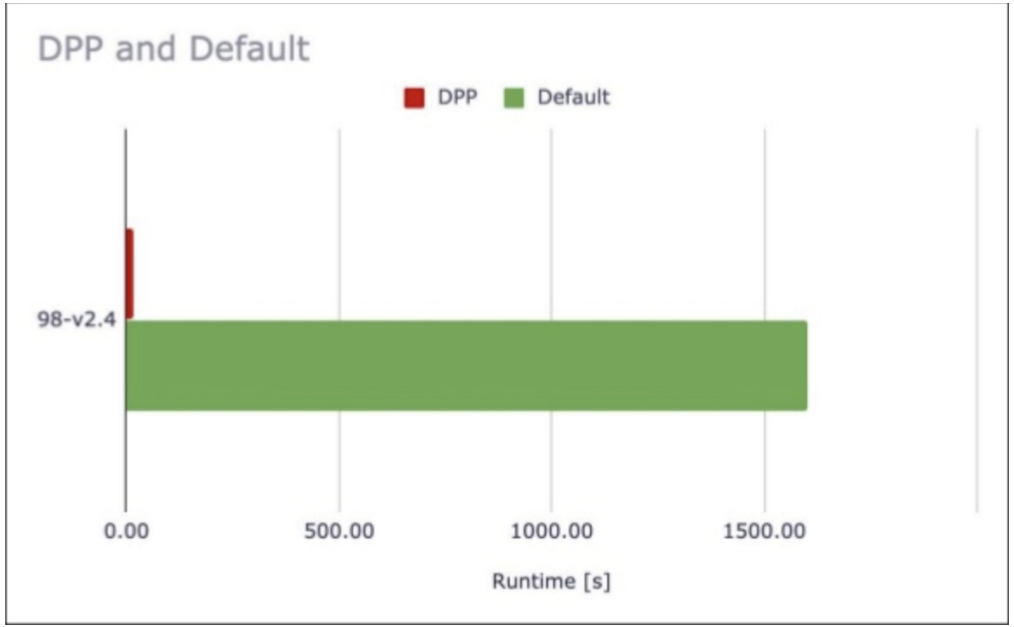
- Read only the data you need
- DPP's Optimisation is implemented both on the logical plan optimization and the physical planning
- Create Logical plan by - column pruning, constant folding, filter push down
- Logical planning level to find the **dimensional filter** and propagated across the join to the other side of the scan
- Physical level to wire it together in a way that this filter executes only once on the dimension side and then creates broadcast variable
- Then the results of the filter is directly reused for the table scan
  - With this two fold approach we can achieve significant speed ups in many queries in Spark.

Dynamic Partition Pruning - Broadcast Hash Join



TPCDS 10 TB

- Highly selective dimension filter that retains only one month out of 5 years of data



Other Notable Features

- Python 3, Scala 2.12 and JDK 11
- Spark 3 support GPU on Yarn.
  - spark 3.0 framework can now auto-discover GPU on a cluster or a system
  - It supports heterogeneous GPUs like AMD, Intel, and Nvidia
- You can use binary files as the data source of your spark dataframe, however, as of now write operations in binary are not allowed

```
val df = spark.read.format(BINARY_FILE).load(dir.getPath)
```

- Kafka Streaming: includeHeaders - Allow reading headers from kafka streaming

```
val df = spark.readStream.format("kafka")
                .option("kafka.bootstrap.servers", "host1:port1,host2:port2")
                .option("subscribe", "topic1")
                .option("includeHeaders", "true").load()
```

Other Notable Features

- Kubernetes support - k8s support began in v 2.3.x new features are
  - Pod Templates
  - GPU-aware scheduling
  - Sub path mounting
  - Improve behavior with dynamic allocation
  - The Kerberos authentication protocol is now supported in Kubernetes resource manager.
- Koalas is a pandas API - Koalas can now scale to the distributed environment
- Analyze Cached Data

```
withTempView("cachedQuery") {
 sql("""CACHE TABLE cachedQuery AS
         |  SELECT c0, avg(c1) AS v1, avg(c2) AS v2
         |  FROM (SELECT id % 3 AS c0, id % 5 AS c1, 2 AS c2 FROM range(1, 30))
         |  GROUP BY c0
      """.stripMargin)
// Analyzes one column in the cached logical plan
sql("ANALYZE TABLE cachedQuery COMPUTE STATISTICS FOR COLUMNS v1")
```

What's being deprecated or removed

- Deprecate MLLIB based on RDD
- Python 2.7 is heavily deprecated
  - Python 2.7 will still work, but not anymore tested in the release lifecycle of Spark

Essential Points

- Deprecated Python 2 and replaced with Python 3
- Deprecated MLLib, the one based on RDD and replaced with SparkML based on dataframes
- Spark 3.0 is said to perform 17x faster compared with current versions on the TPCDS benchmark
- Enhanced Support for Deep Learning
- Better Kubernetes Integration
- Spark 3.0 supports binary file data source
- Spark 3.0 introduces a whole new module named SparkGraph with major features for Graph processing
- ACID Transactions with Delta Lake

CLOUDERA
Educational Services

Demos/Labs:
Seeing Adaptive SQL at Work

CLOUDERA
Educational Services

Conclusion

Course Objectives

During this course, you have learned

- Which file formats to use for best performance
- How to avoid the small files problem
- How to deal with schemas
- How to manage skewed data
- How Catalyst and Tungsten work
- How to mitigate the shuffles
- What is the impact on performance of partitioned and bucketed tables
- How to improve joins performance
- How to work around the overhead of Pyspark UDFs
- How to cache data for reuse
- How WXM can help troubleshoot Spark applications issues
- What is new in Spark 3.0
- How Adaptive Query Execution works to improve performance

Which Course to Take Next

- For developers
  - Cloudera Search Training
  - Cloudera Training for Apache HBase
- For system administrators
  - Cloudera Administrator Training for Apache Hadoop
  - Cloudera Security Training
- For data analysts and data scientists
  - Cloudera Data Analyst Training
  - Cloudera Data Scientist Training

CLOUDERA
Educational Services

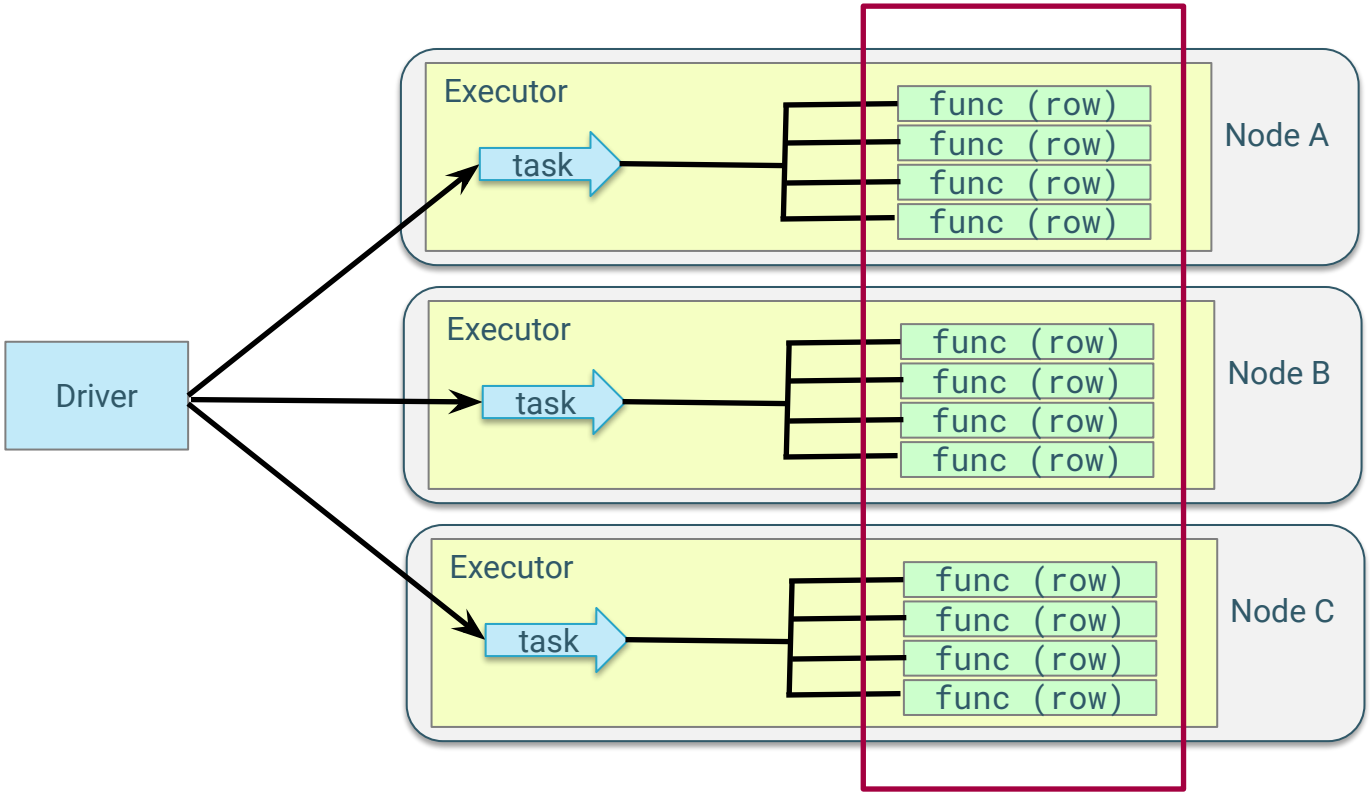
CLOUDERA
Educational Services

Appendix

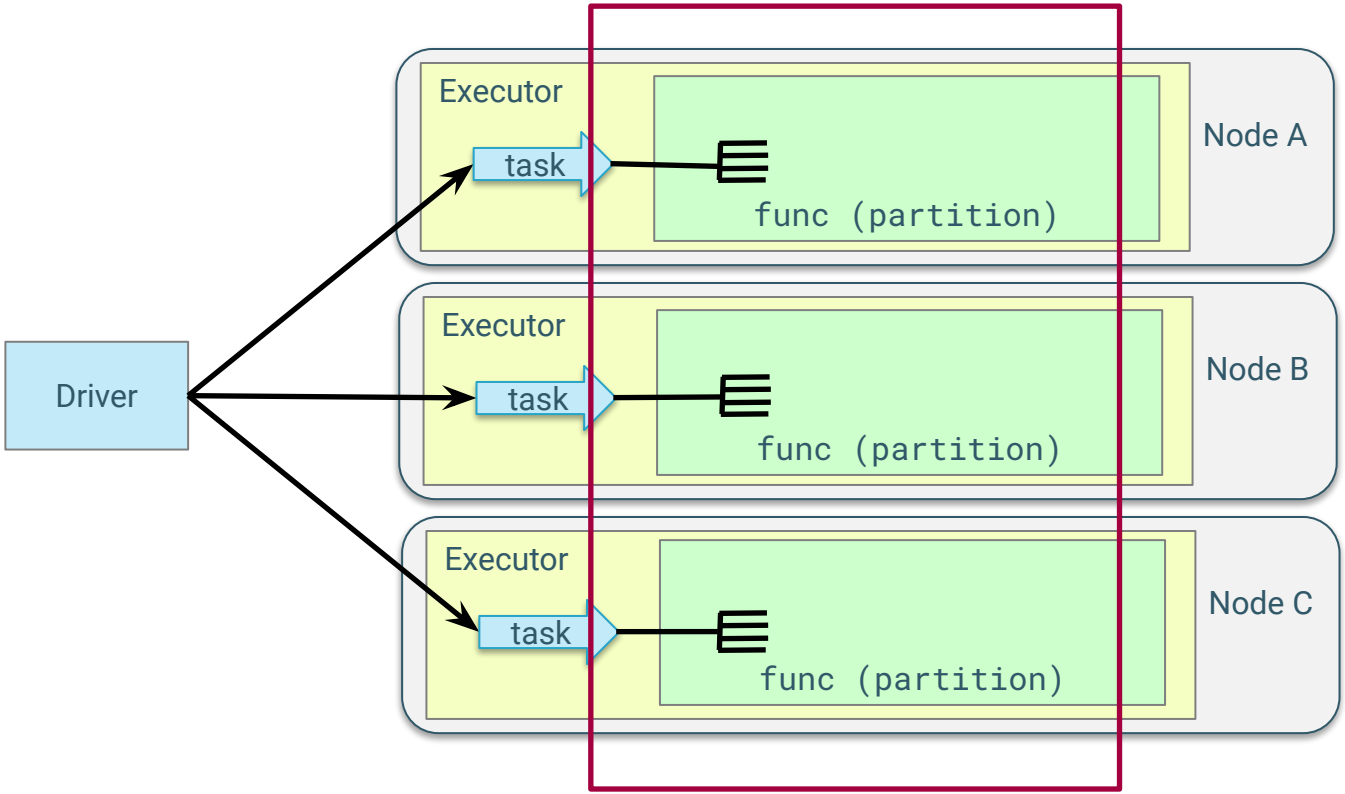
CLOUDERA
Educational Services

Partition Processing

Coarse-grain Parallel Processing



Coarse-grain Partition Parallel Processing



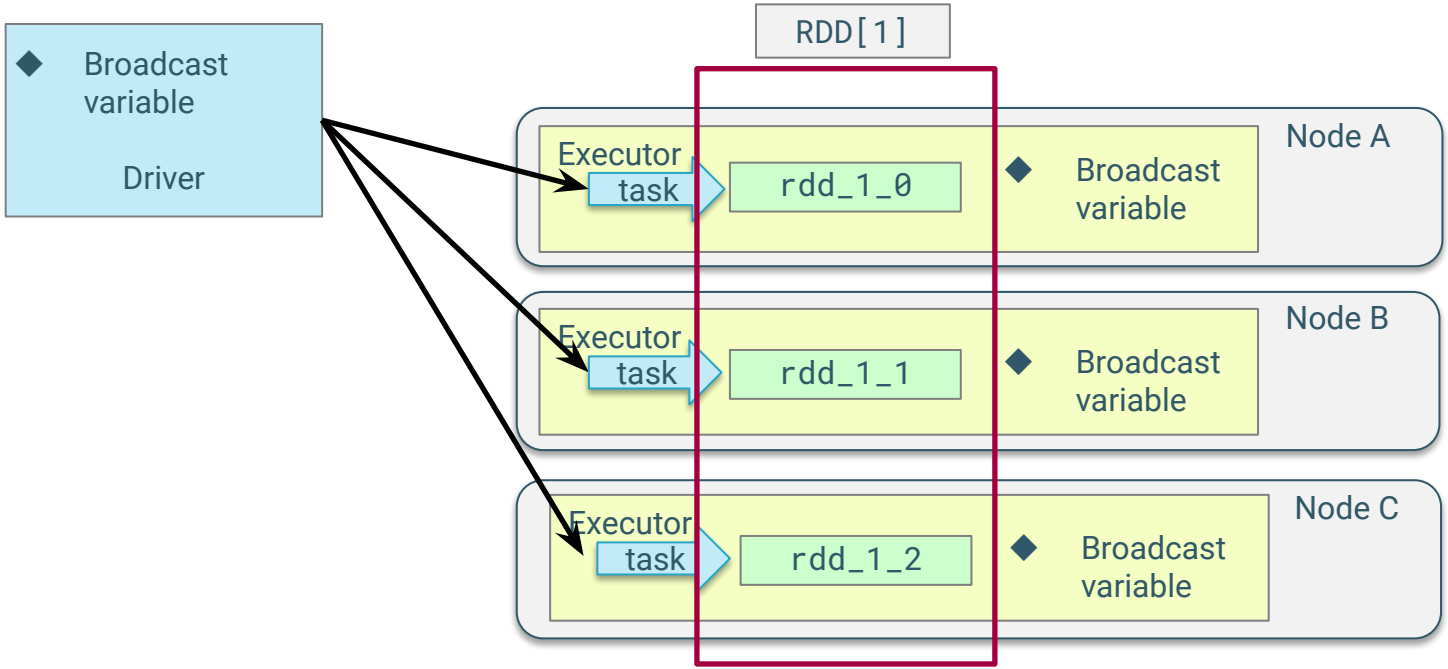
CLOUDERA
Educational Services

Demos/Labs:
Partition Processing

CLOUDERA
Educational Services

Broadcasting

Broadcast Variables



CLOUDERA
Educational Services

Demos/Labs:
Broadcasting

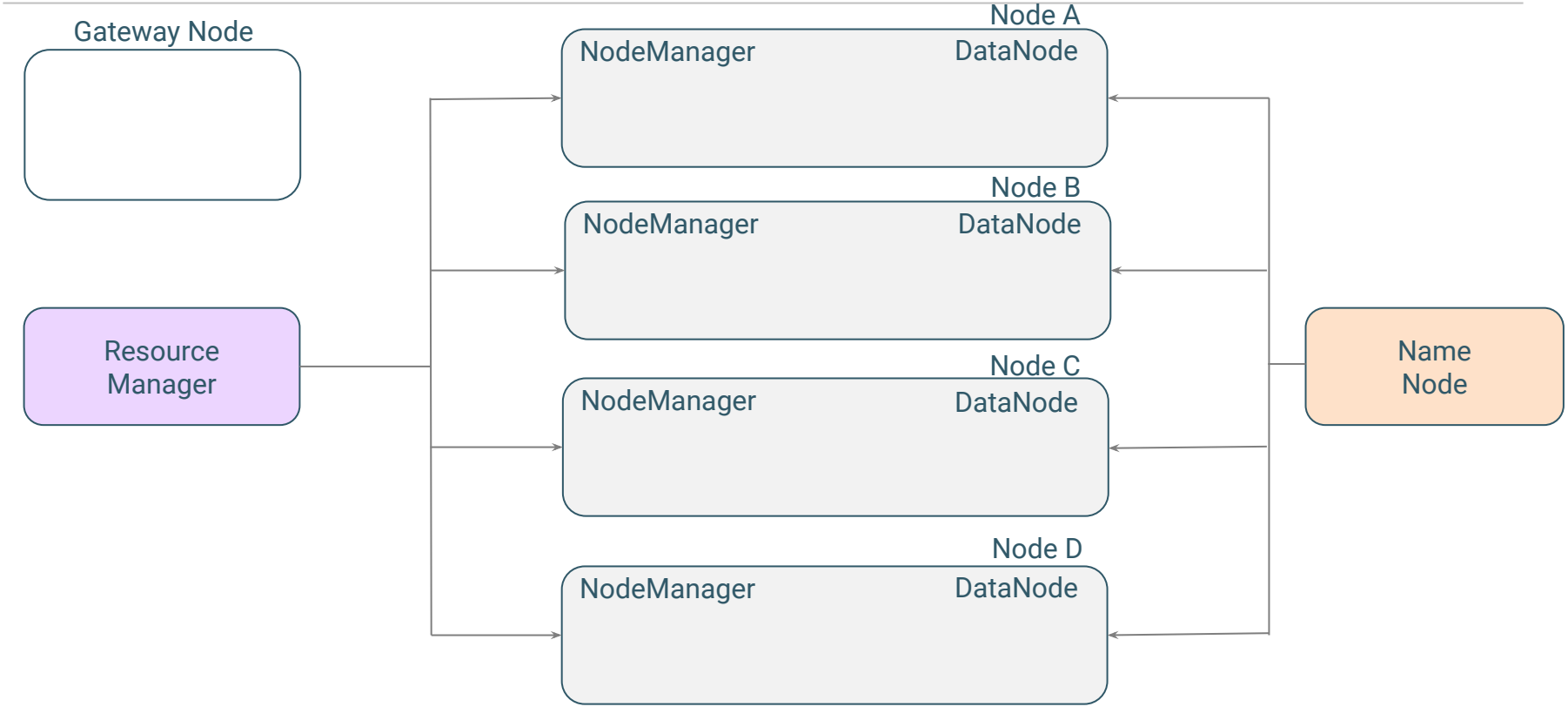
CLOUDERA
Educational Services

Scheduling

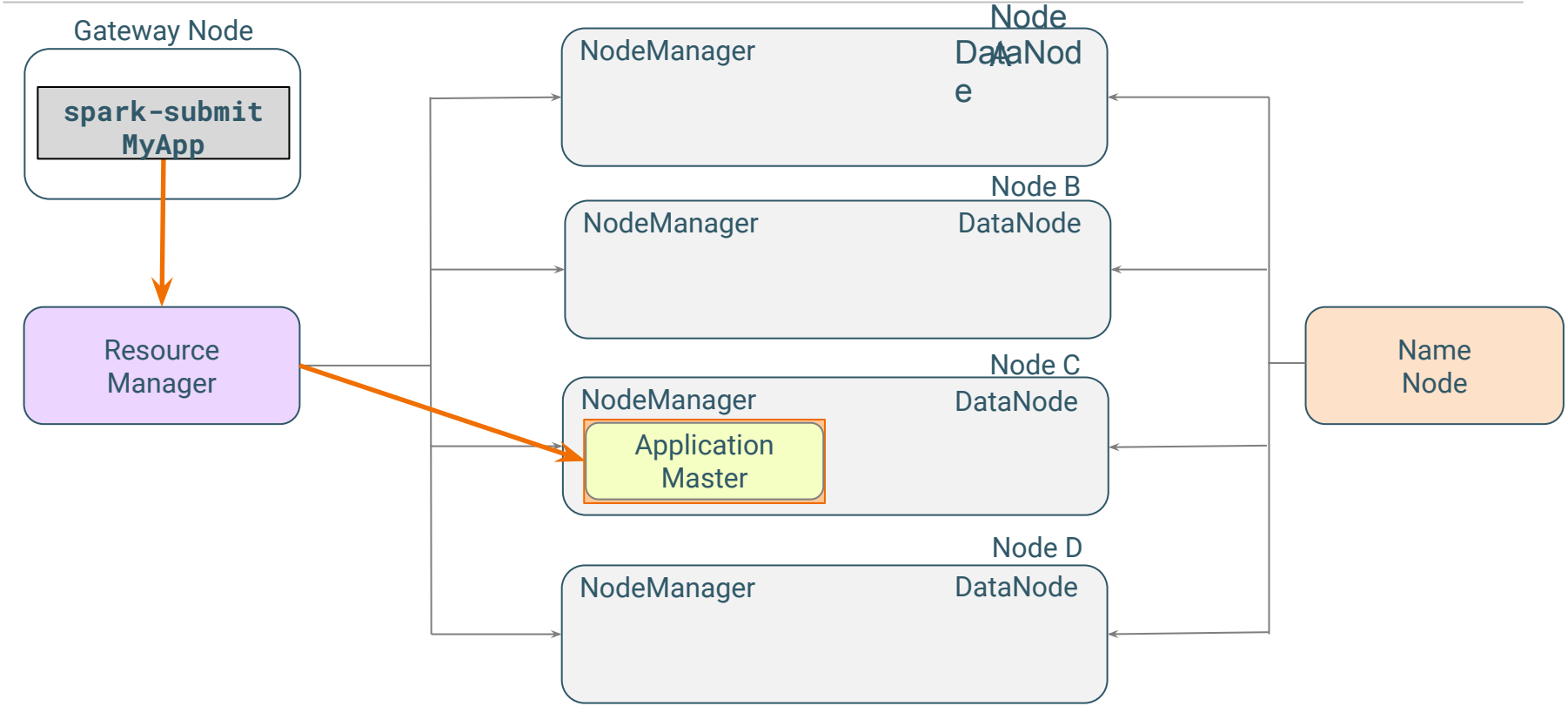
Spark Static Allocation of Resources in YARN

- Application is submitted to cluster for execution
- Resource Manager creates Application Master to represent application
- Application Master reads configured properties and requests resources from Resource Manager
  - The resource request is *static*
  - It is fixed in the configuration
- YARN schedules the request
- Once granted, the application holds the resources until it terminates

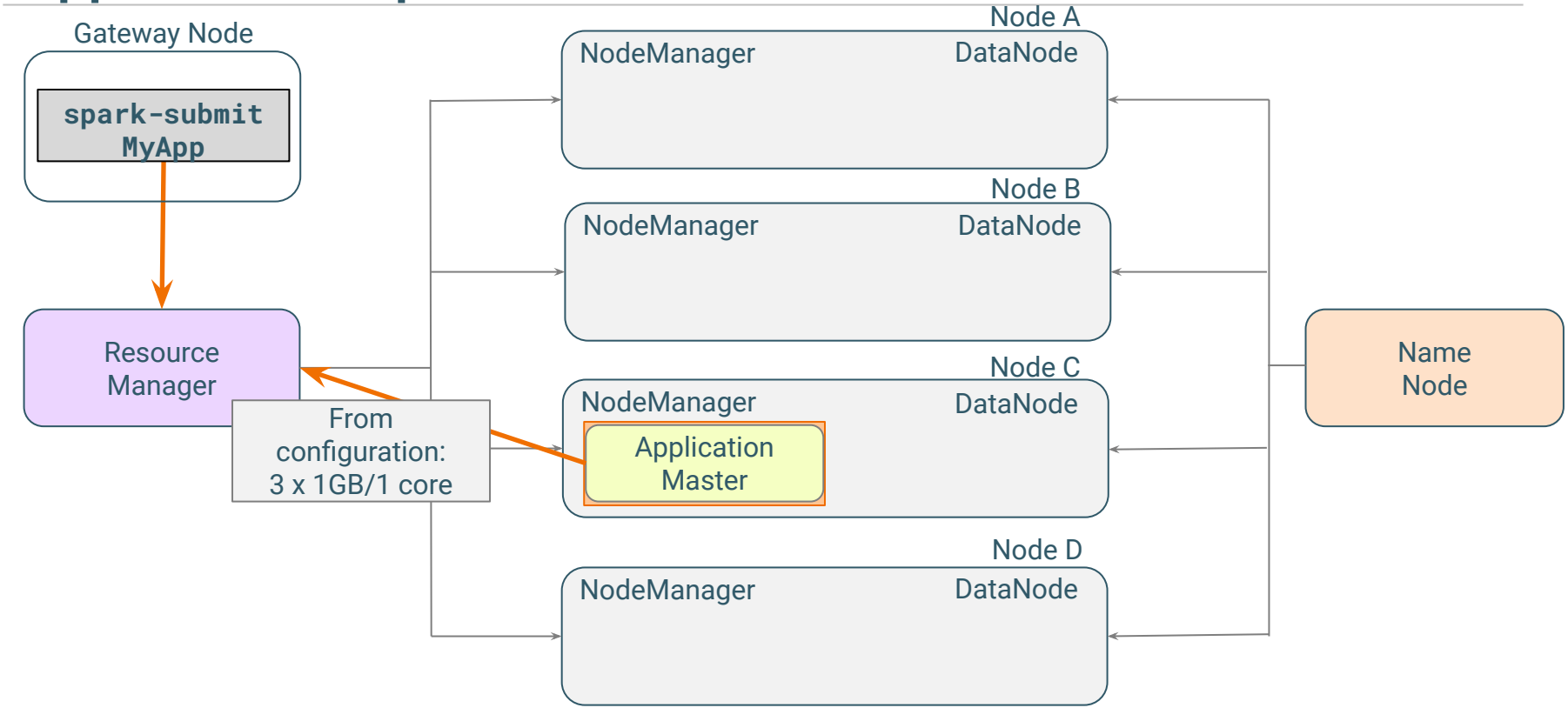
Static Resource Allocation (1): Before Application



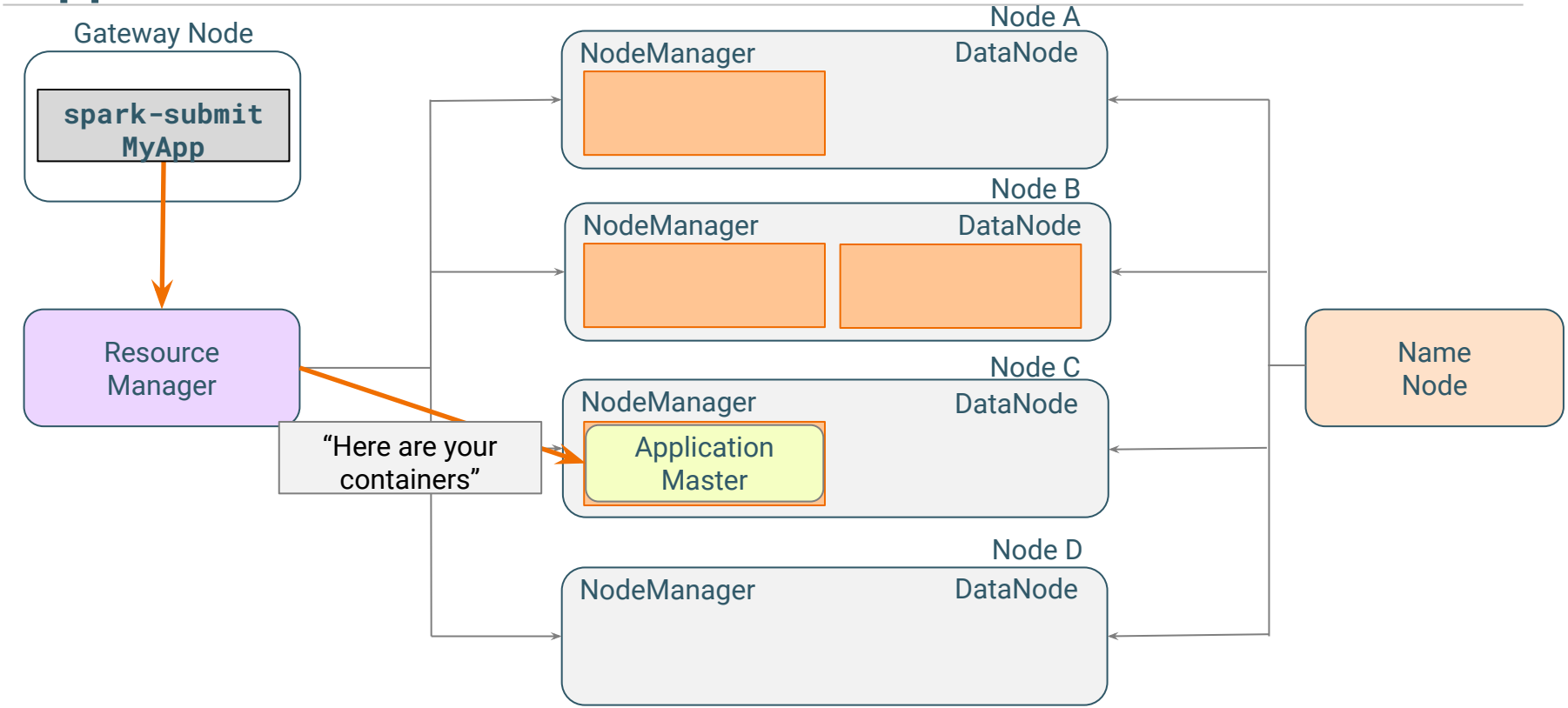
Static Resource Allocation (2): Applications Starts



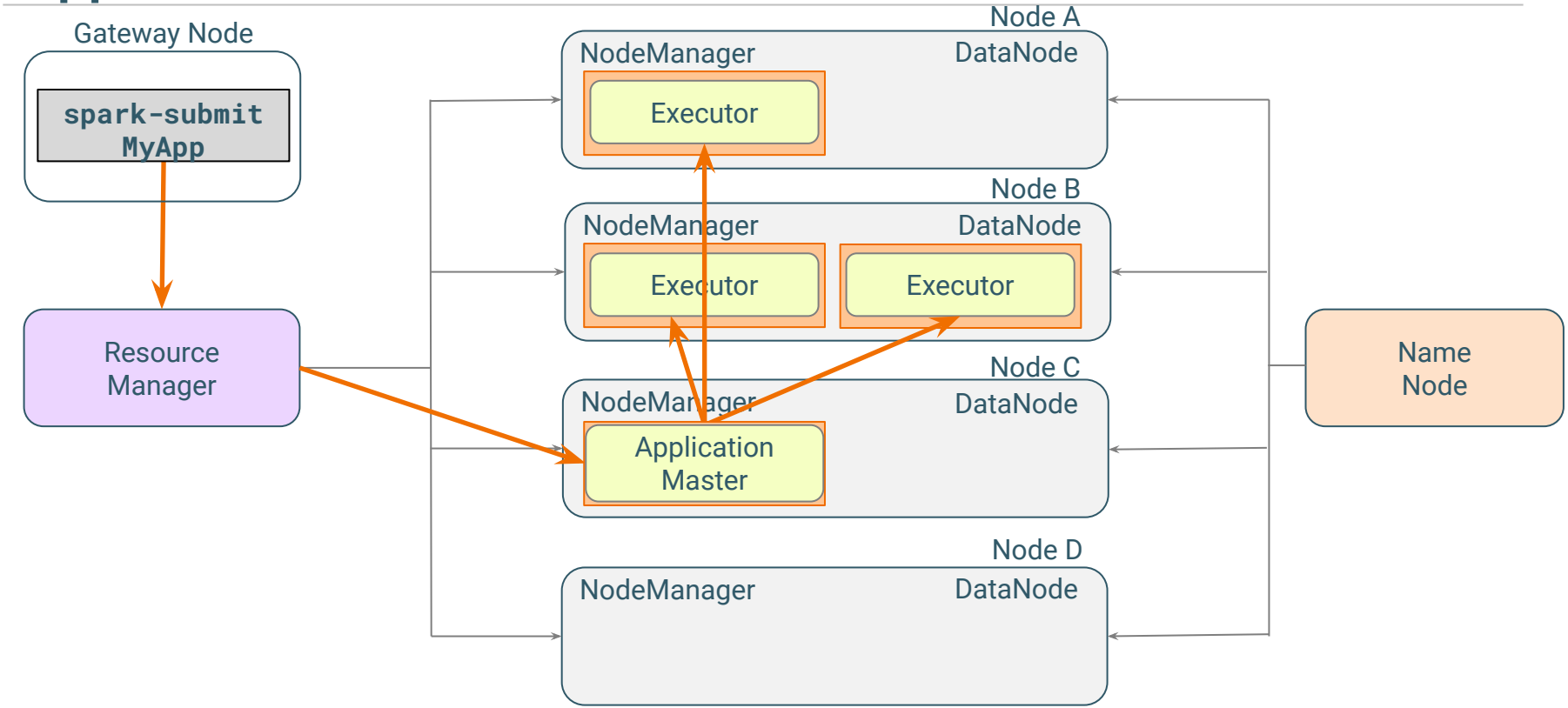
Static Resource Allocation (3): Application Requests Resources



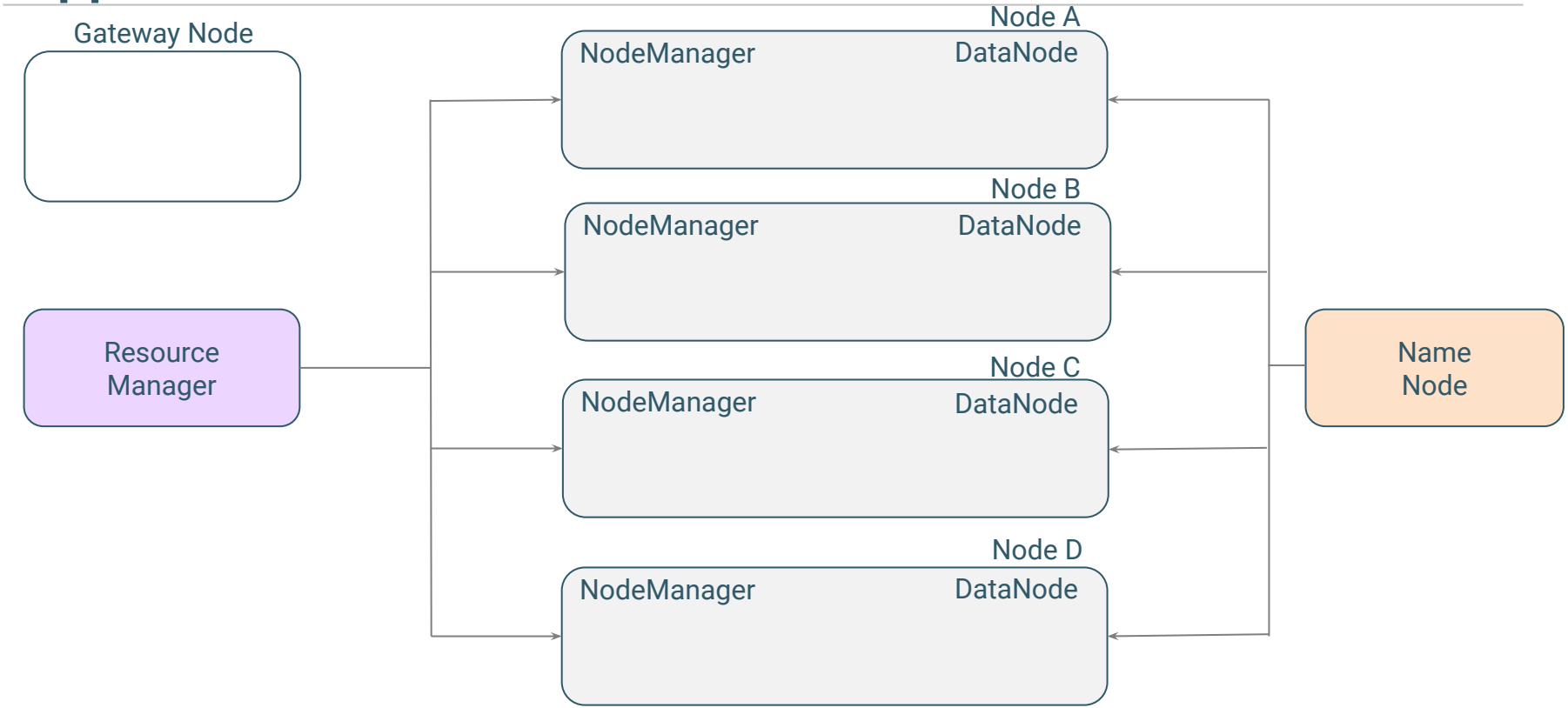
Static Resource Allocation (4): Application Granted Resources



Static Resource Application (5): Application Uses Resources



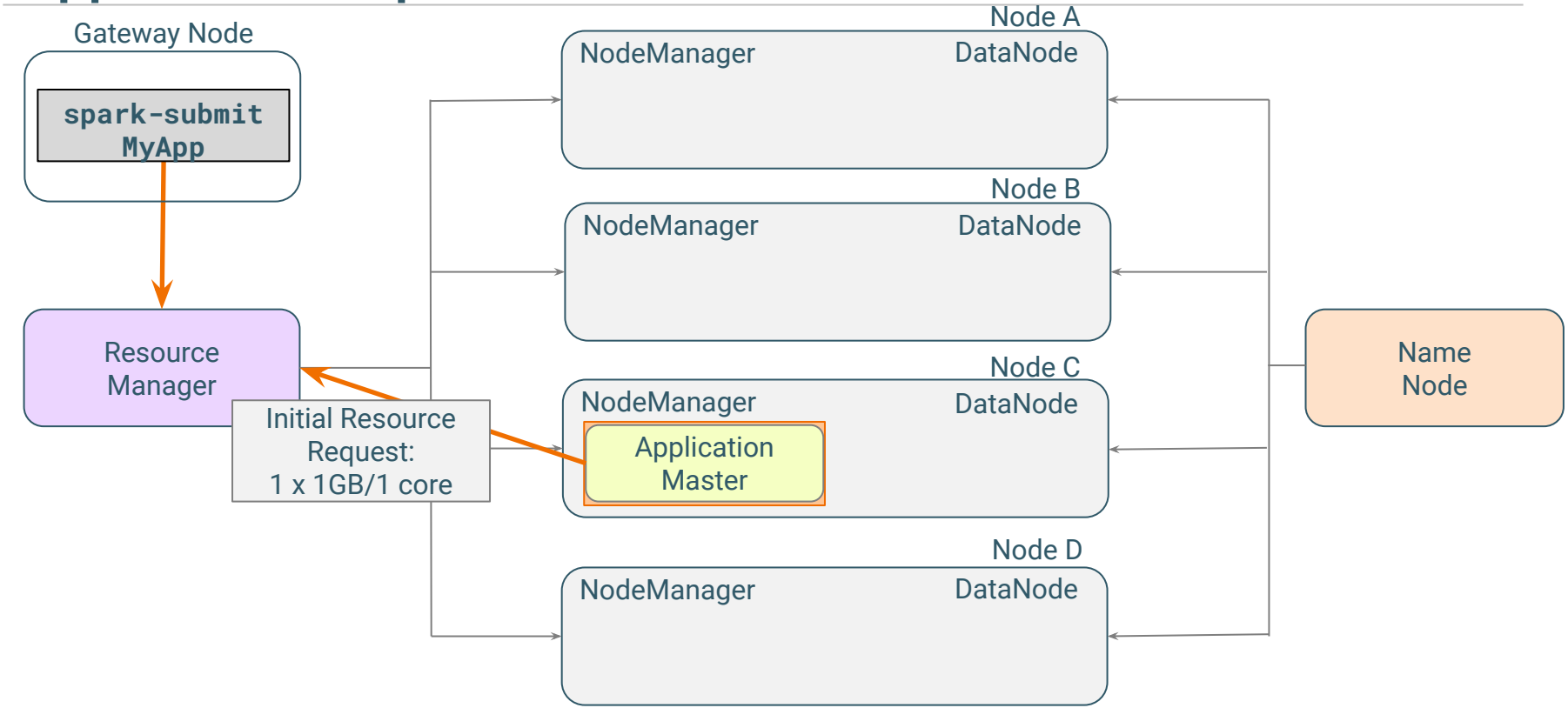
Static Resource Allocation (6): Application Terminates



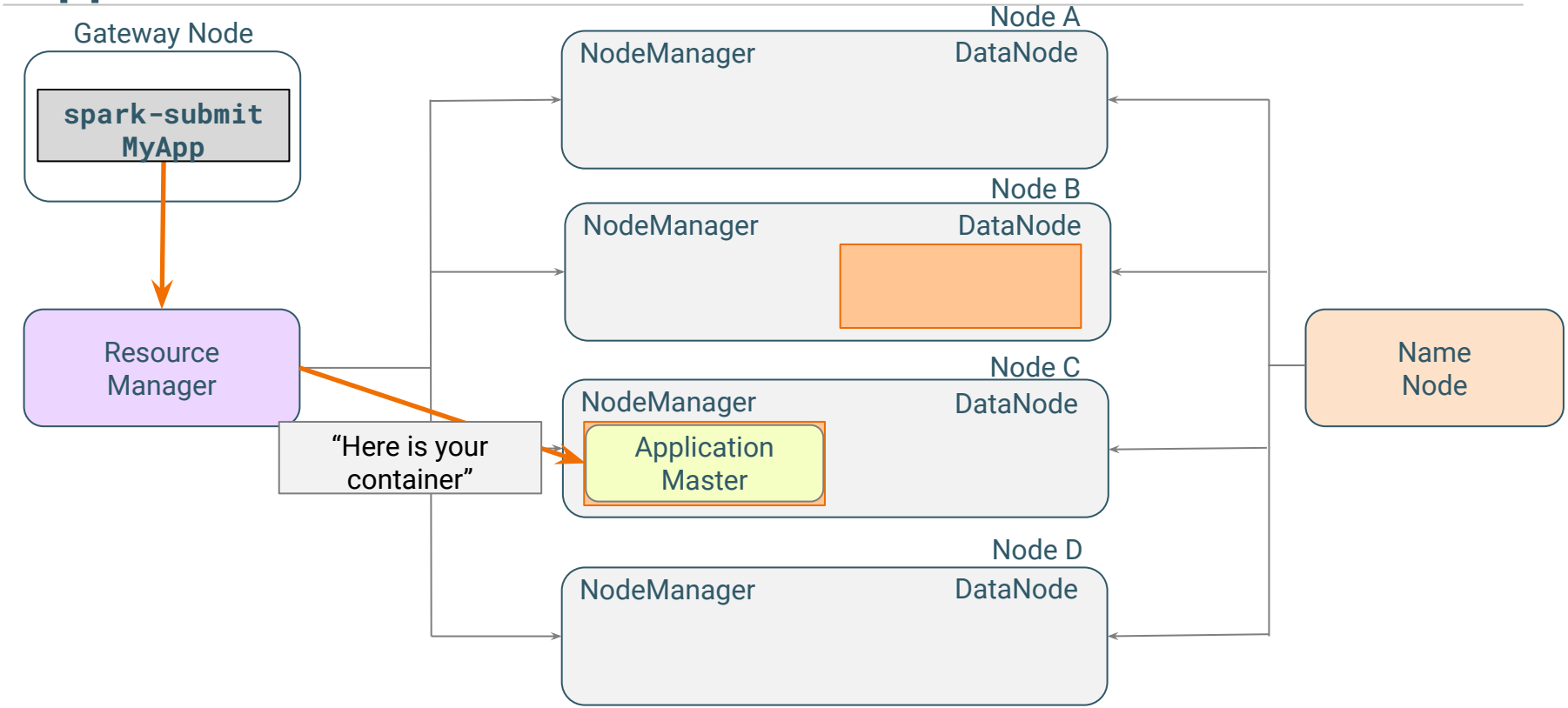
Spark Dynamic Allocation of Resources in YARN

- Application is submitted to cluster for execution
- Resource Manager creates Application Master to represent application
- Application Master reads configured properties for *initial*, *minimum* and *maximum* resources
- Application Master requests the initial resources
- YARN schedules the request
- Once granted, the application runs
  - If the application needs more resources, it requests them
  - Locality considered – wants containers where HDFS blocks are stored
- Containers can time out and be returned to YARN
  - By default, cached data keeps containers alive
- When the application terminates, it returns containers to YARN

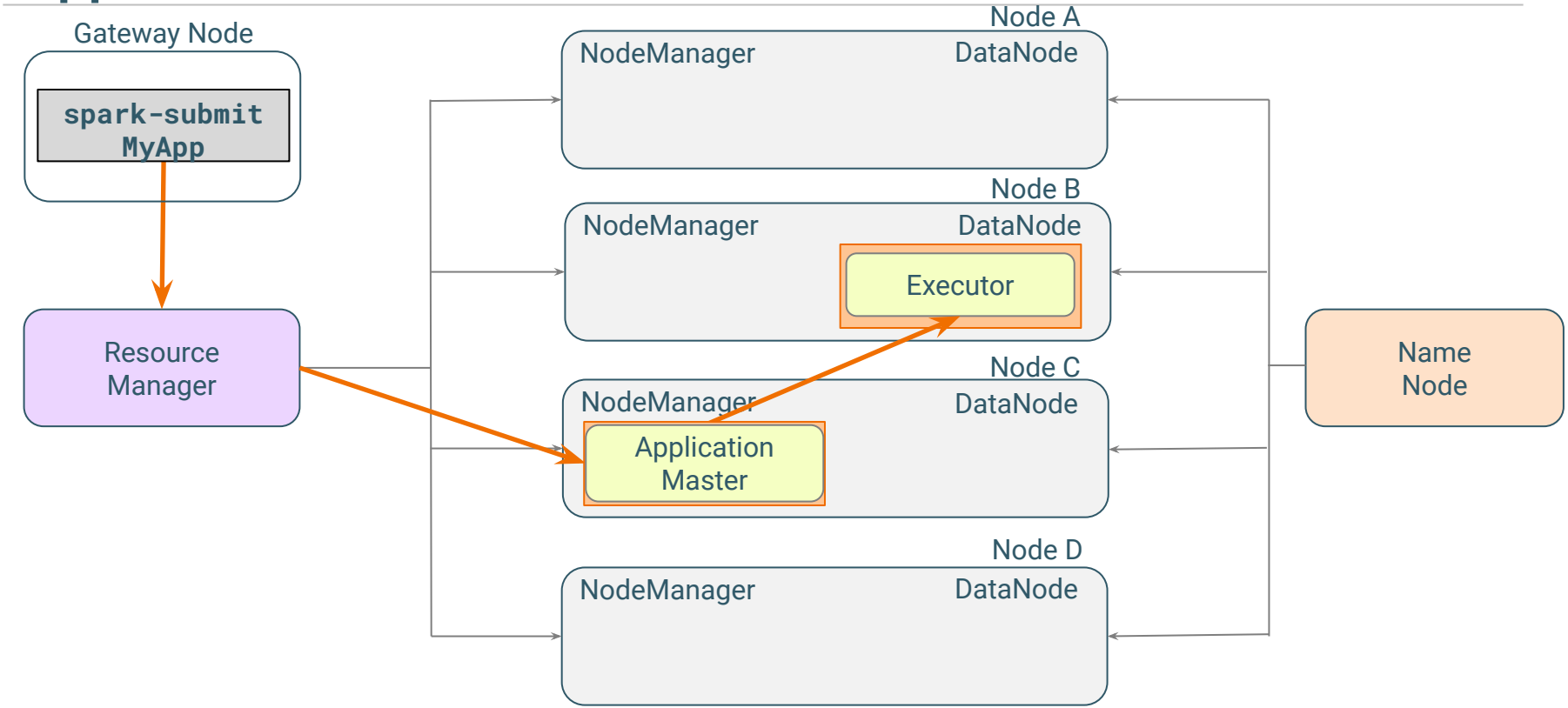
Dynamic Resource Allocation (1): Application Requests Initial Resources



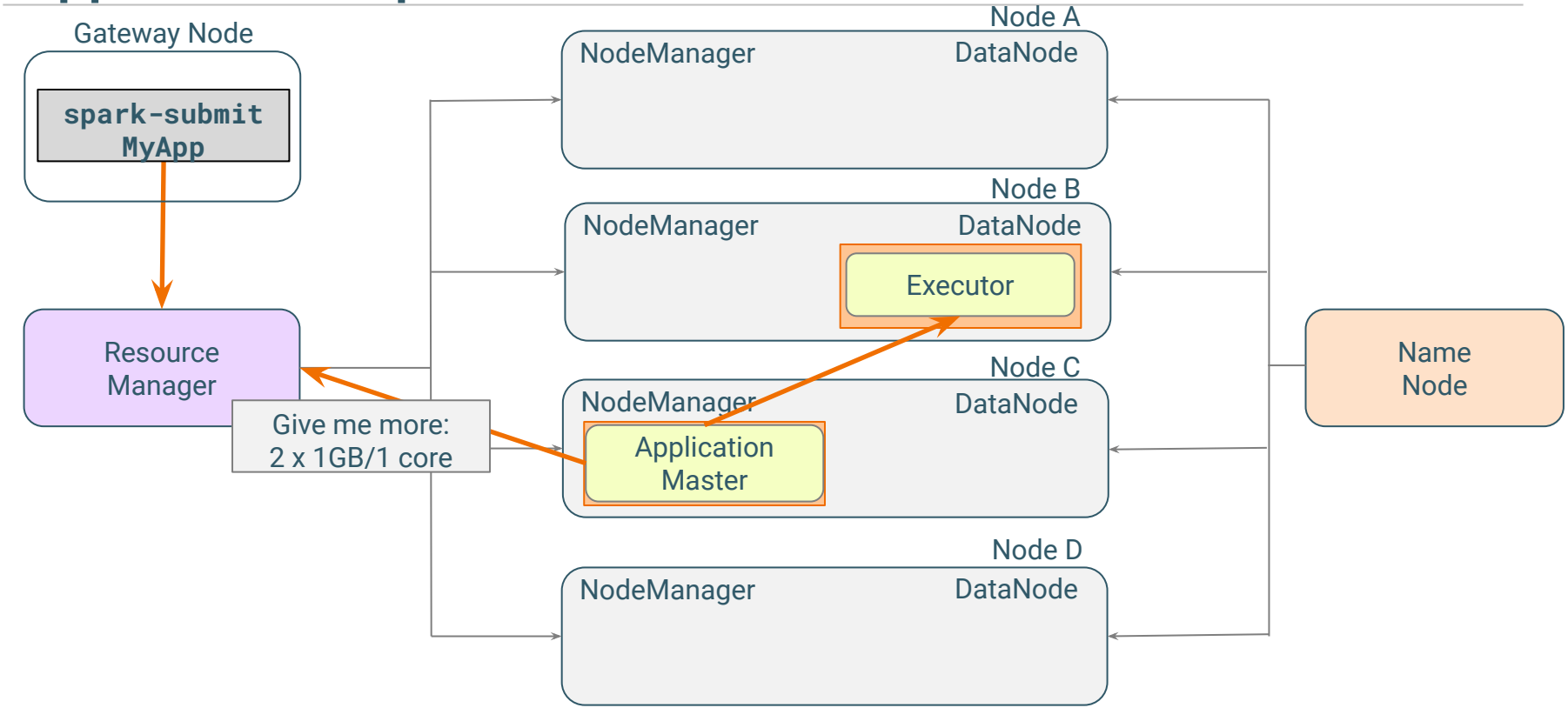
Dynamic Resource Allocation (2): Application is Granted Initial Resources



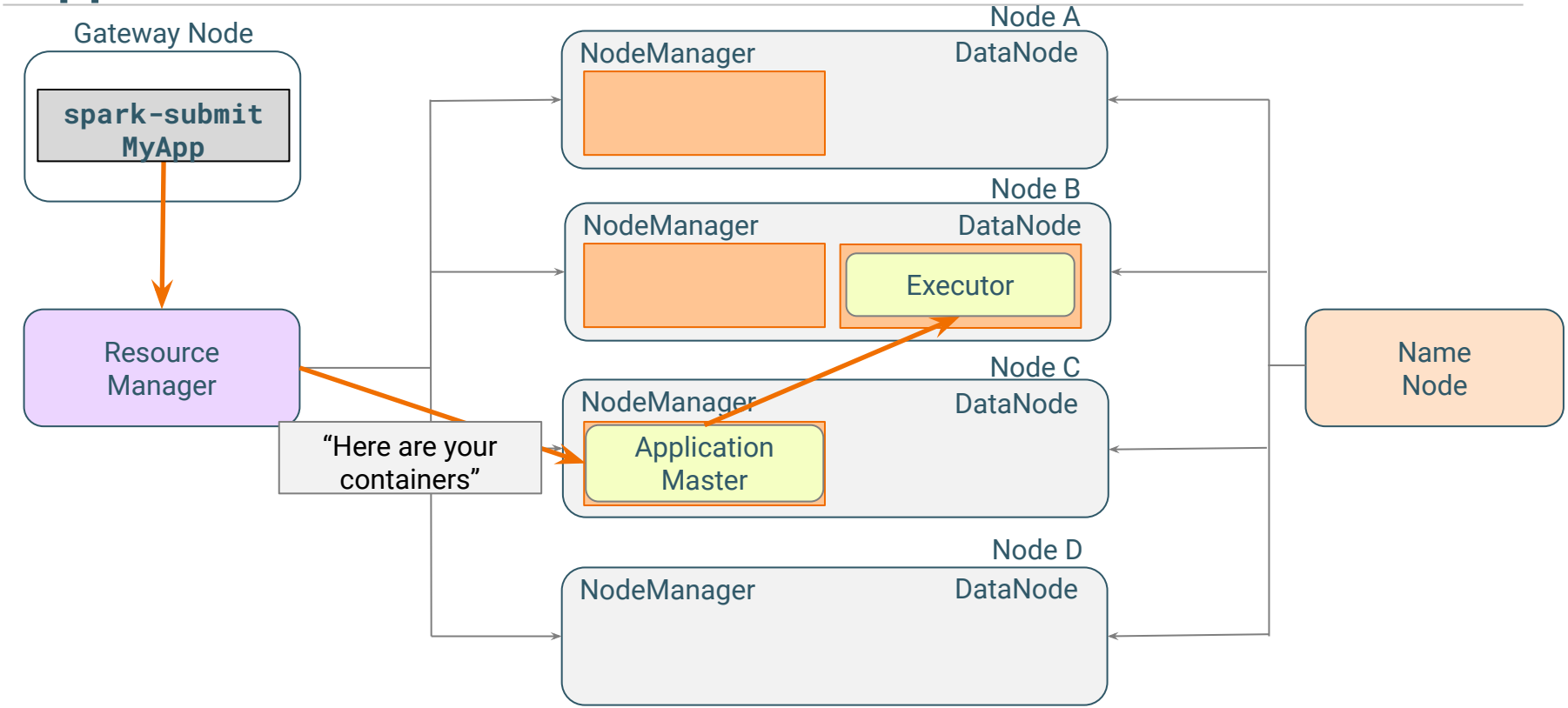
Dynamic Resource Allocation (3): Application Uses Initial Resources



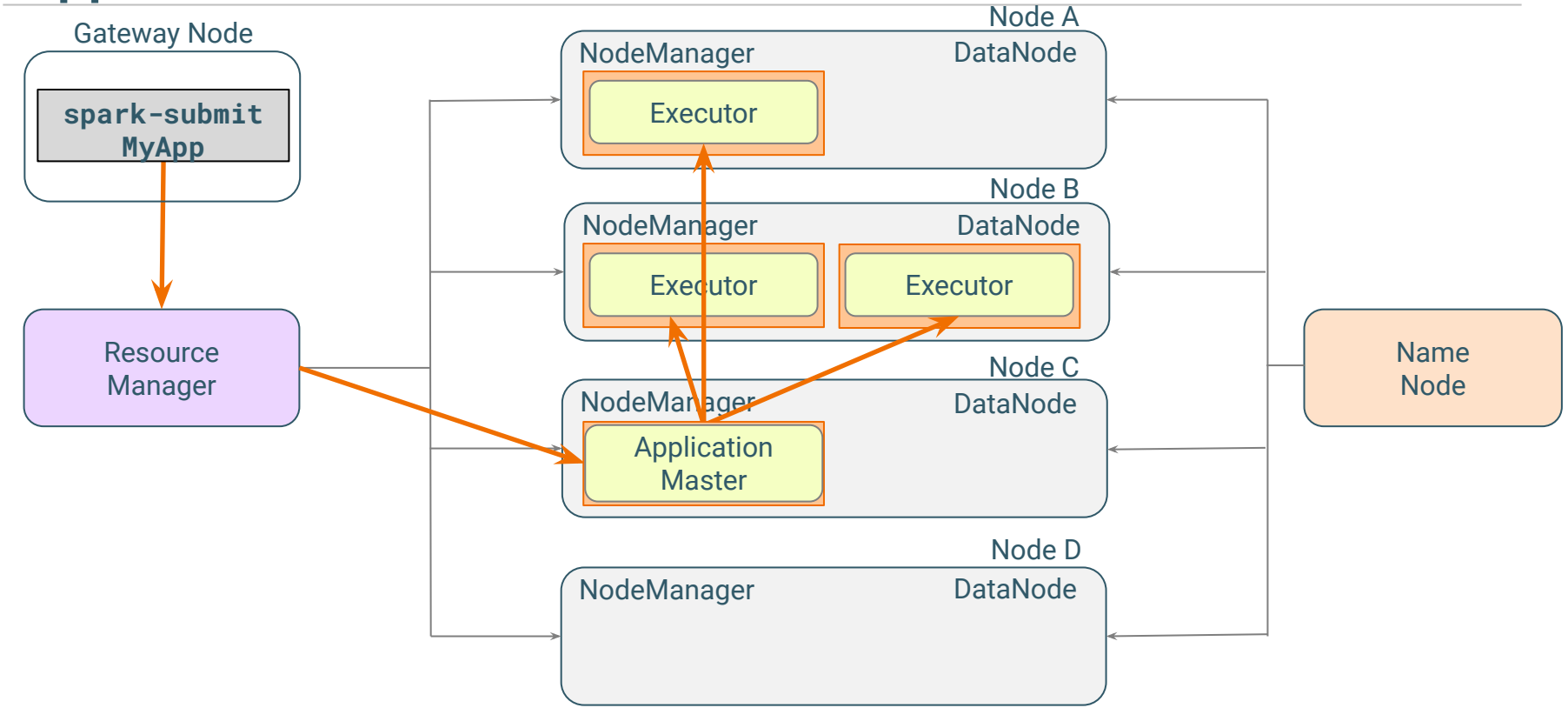
Dynamic Resource Allocation (4): Application Requests More Resources



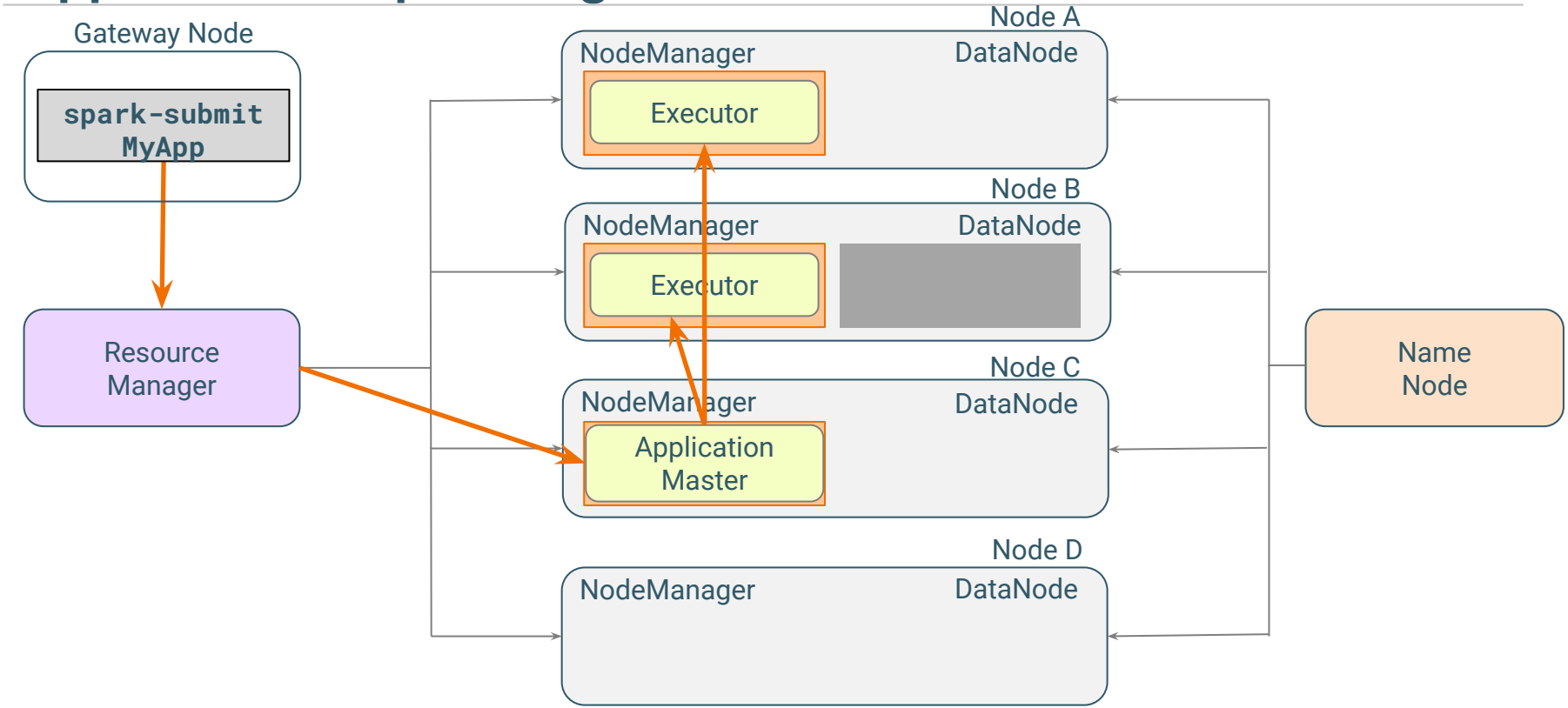
Dynamic Resource Allocation (5): Application is Granted More Resources



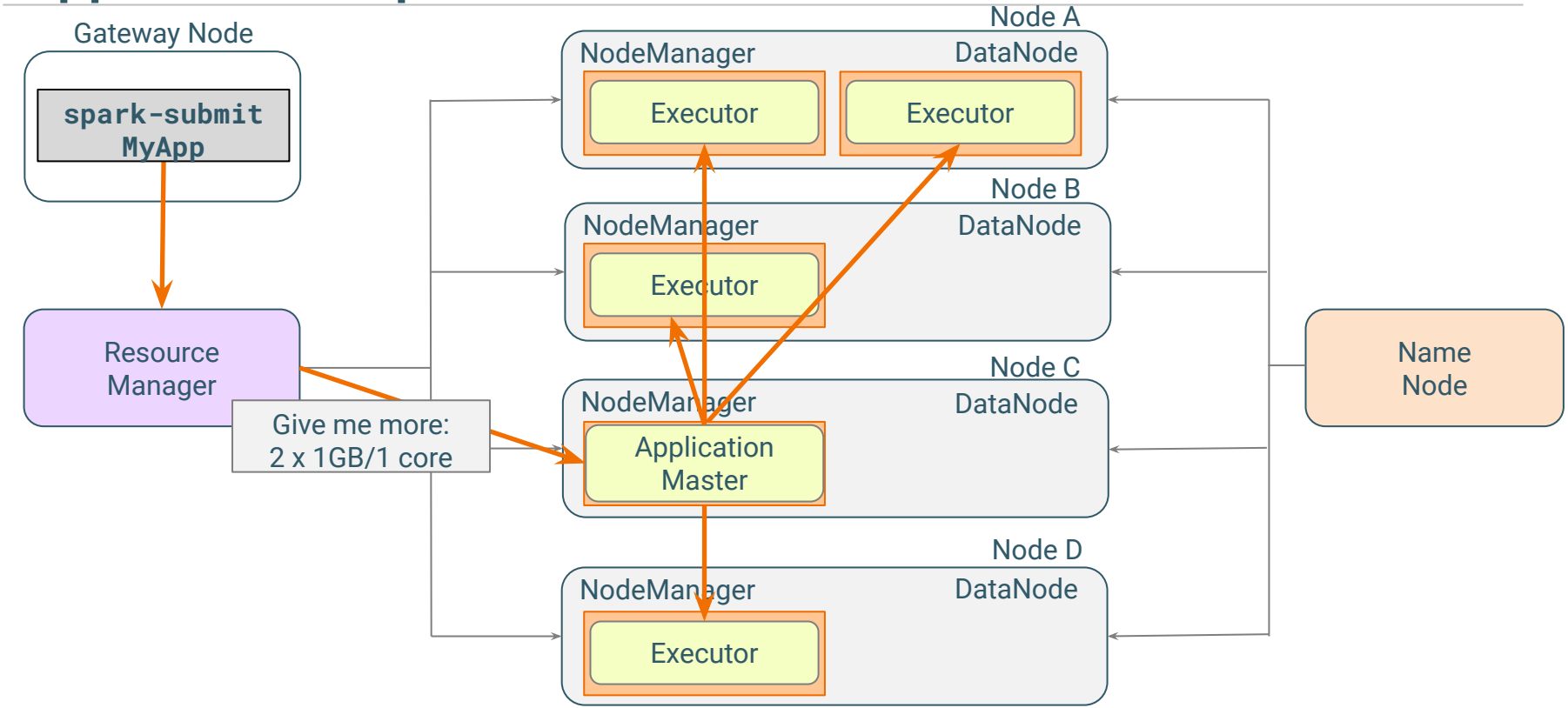
Dynamic Resource Allocation (6): Application Uses Resources



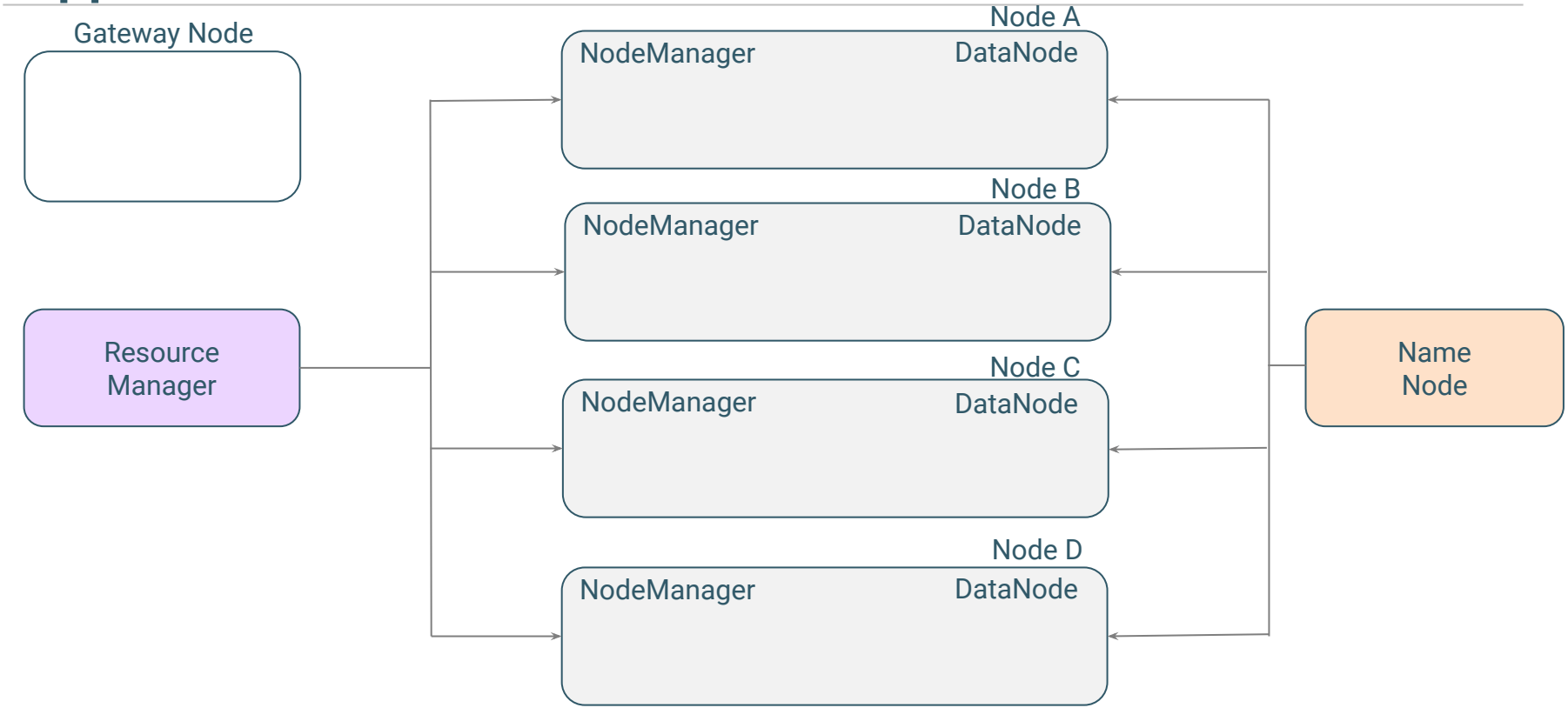
Dynamic Resource Allocation (7): Application Stops Using a Container. It times out



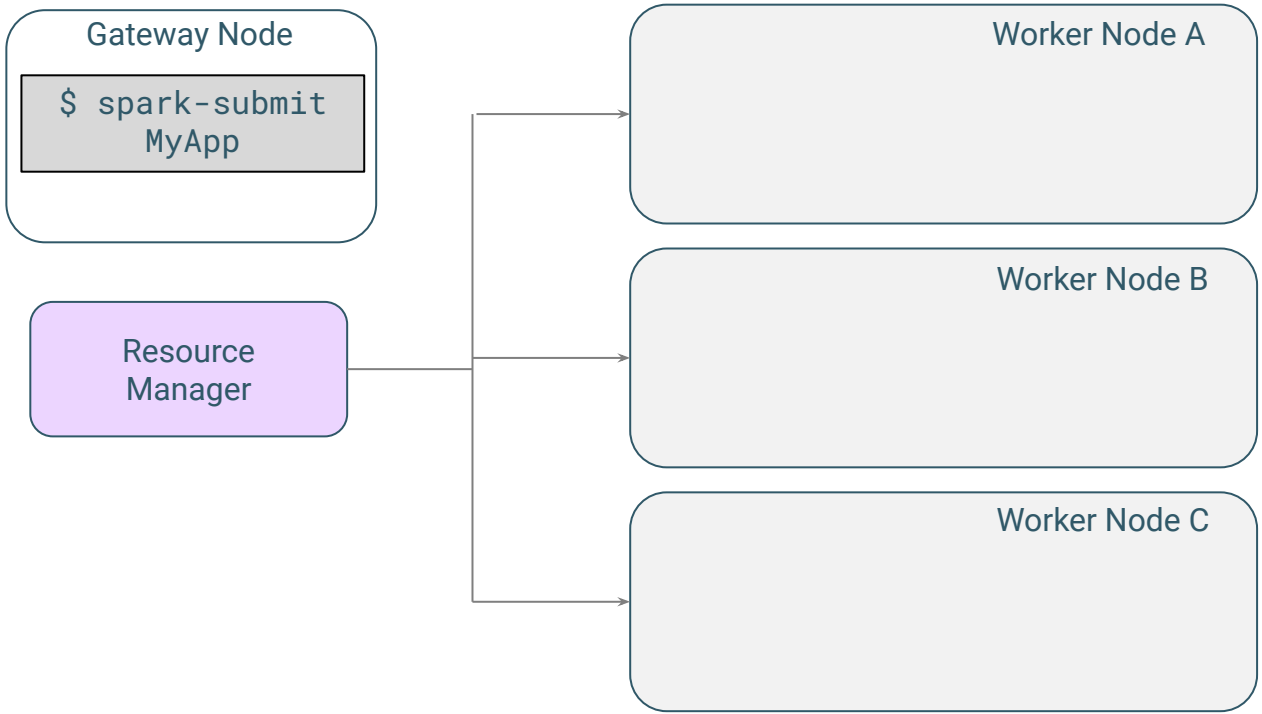
Dynamic Resource Allocation (8): Application Requests and Uses Additional Resources



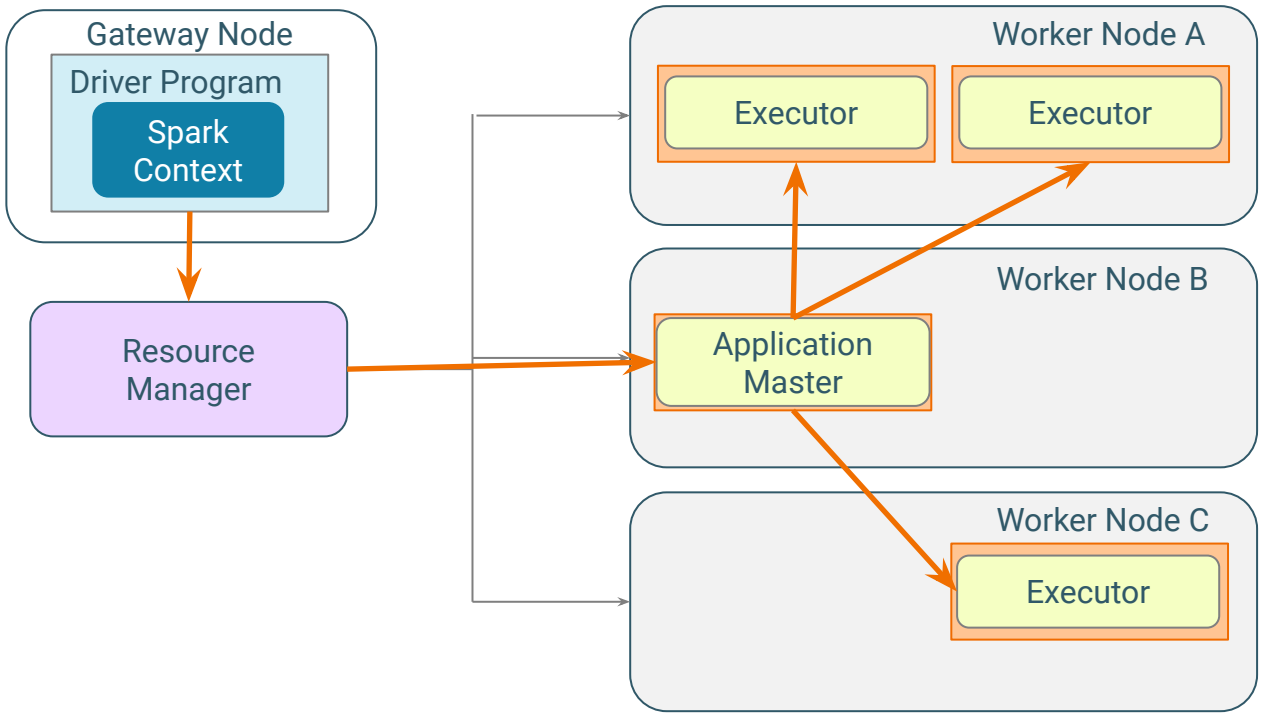
Dynamic Resource Allocation (9): Application Terminates



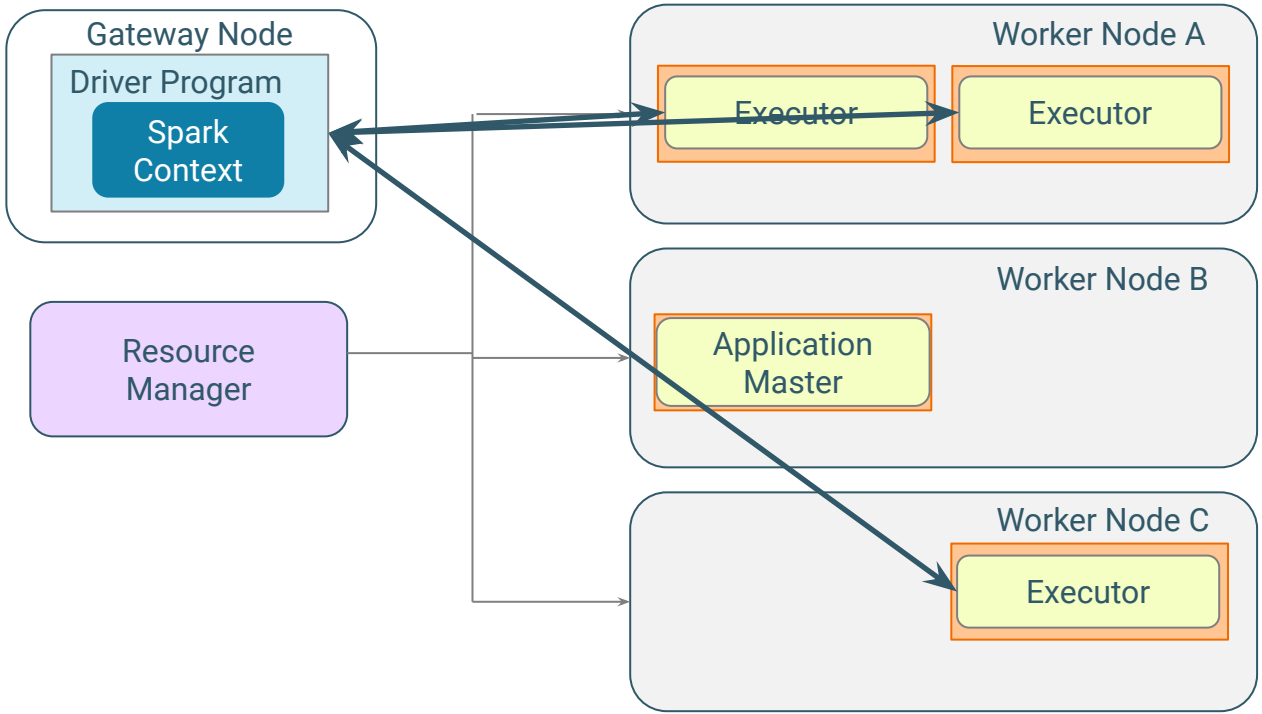
Spark Deployment Mode on YARN: Client Mode (1)



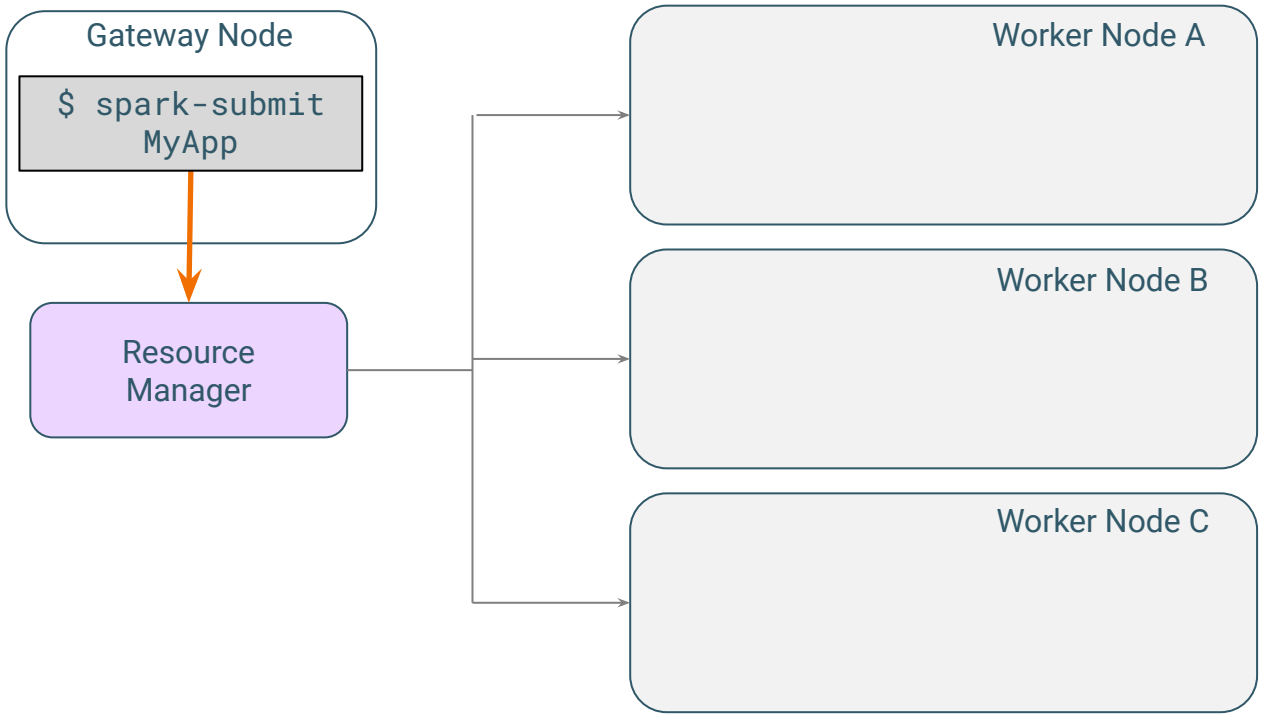
Spark Deployment Mode on YARN: Client Mode (2)



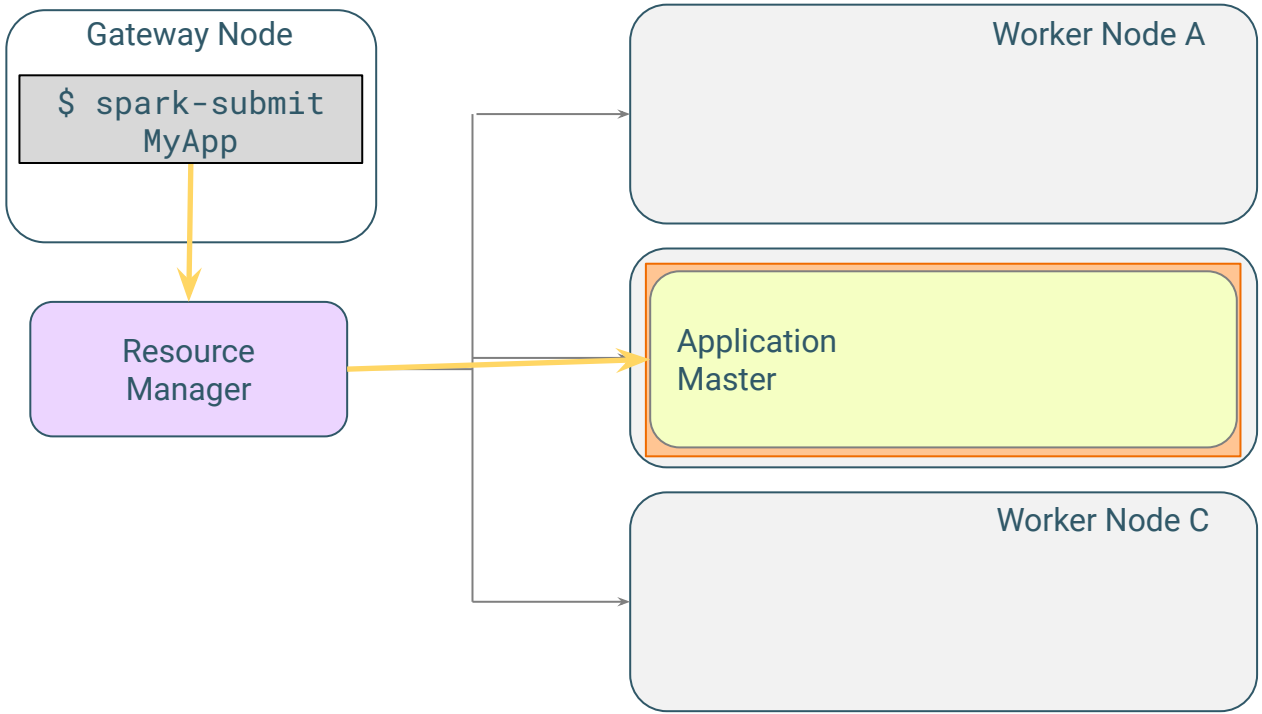
Spark Deployment Mode on YARN: Client Mode (3)



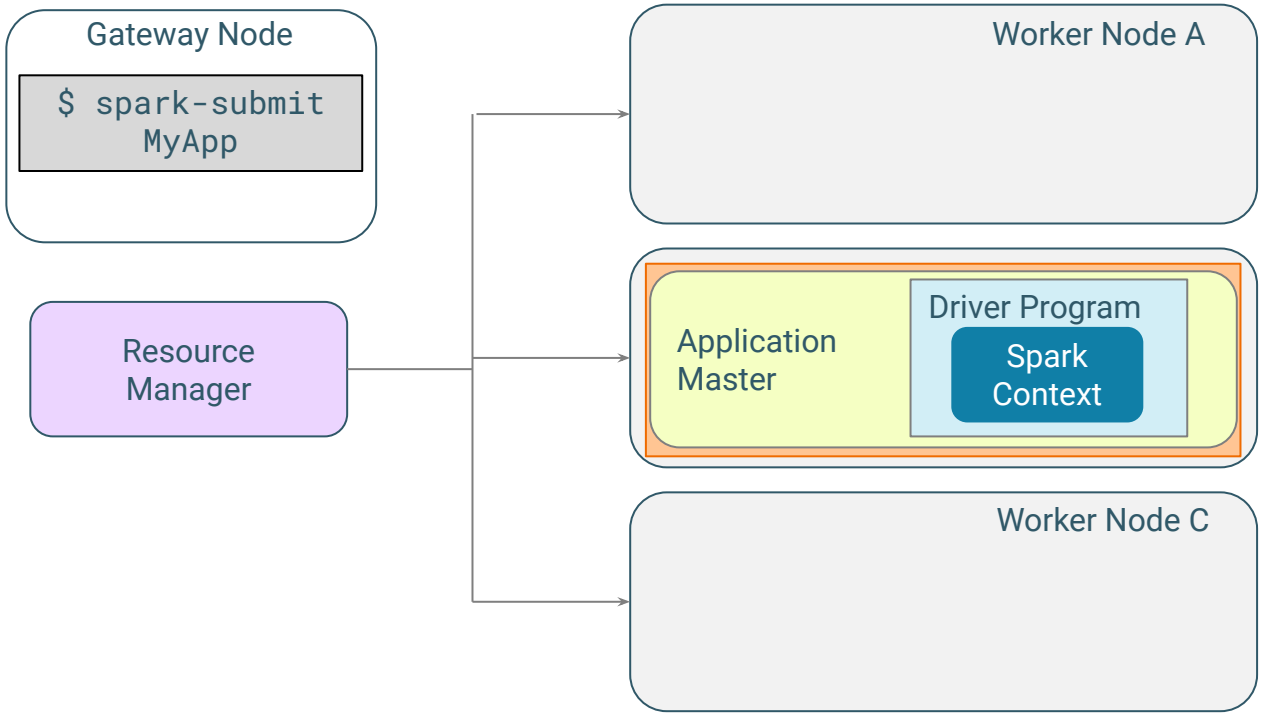
Spark Deployment Mode on YARN: Cluster Mode (1)



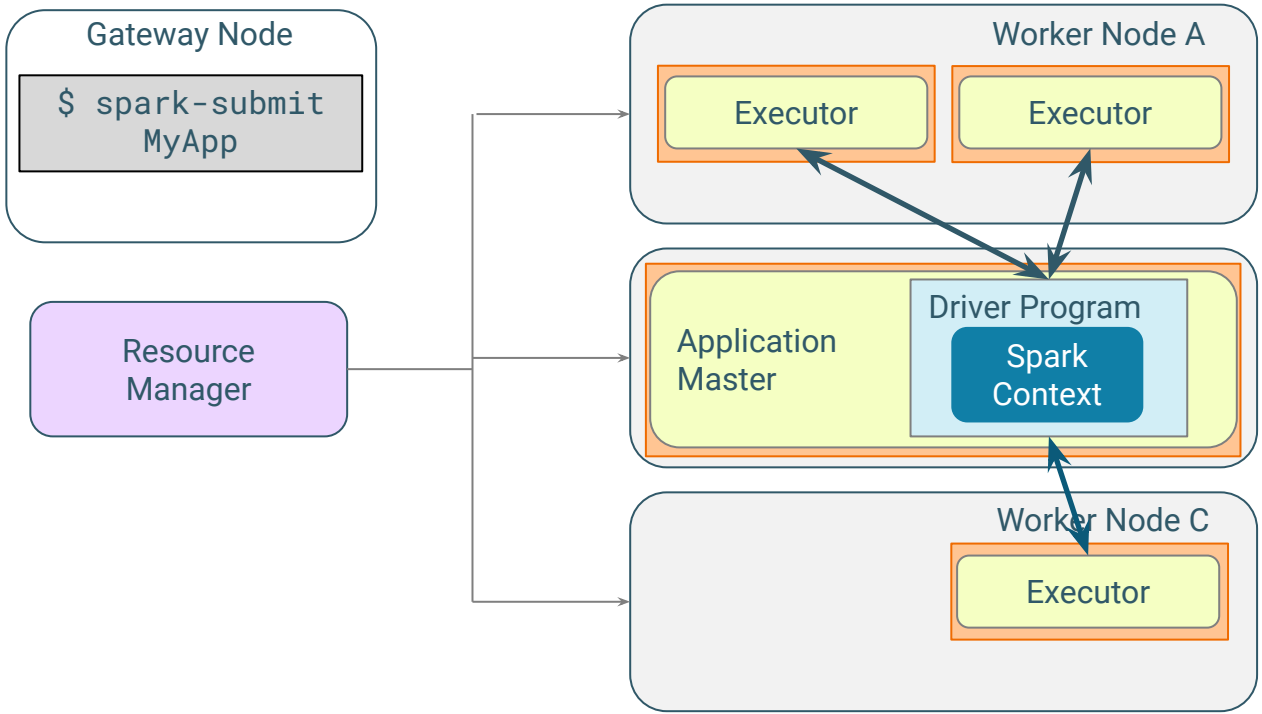
Spark Deployment Mode on YARN: Cluster Mode (2)



Spark Deployment Mode on YARN: Cluster Mode (3)



Spark Deployment Mode on YARN: Cluster Mode (4)



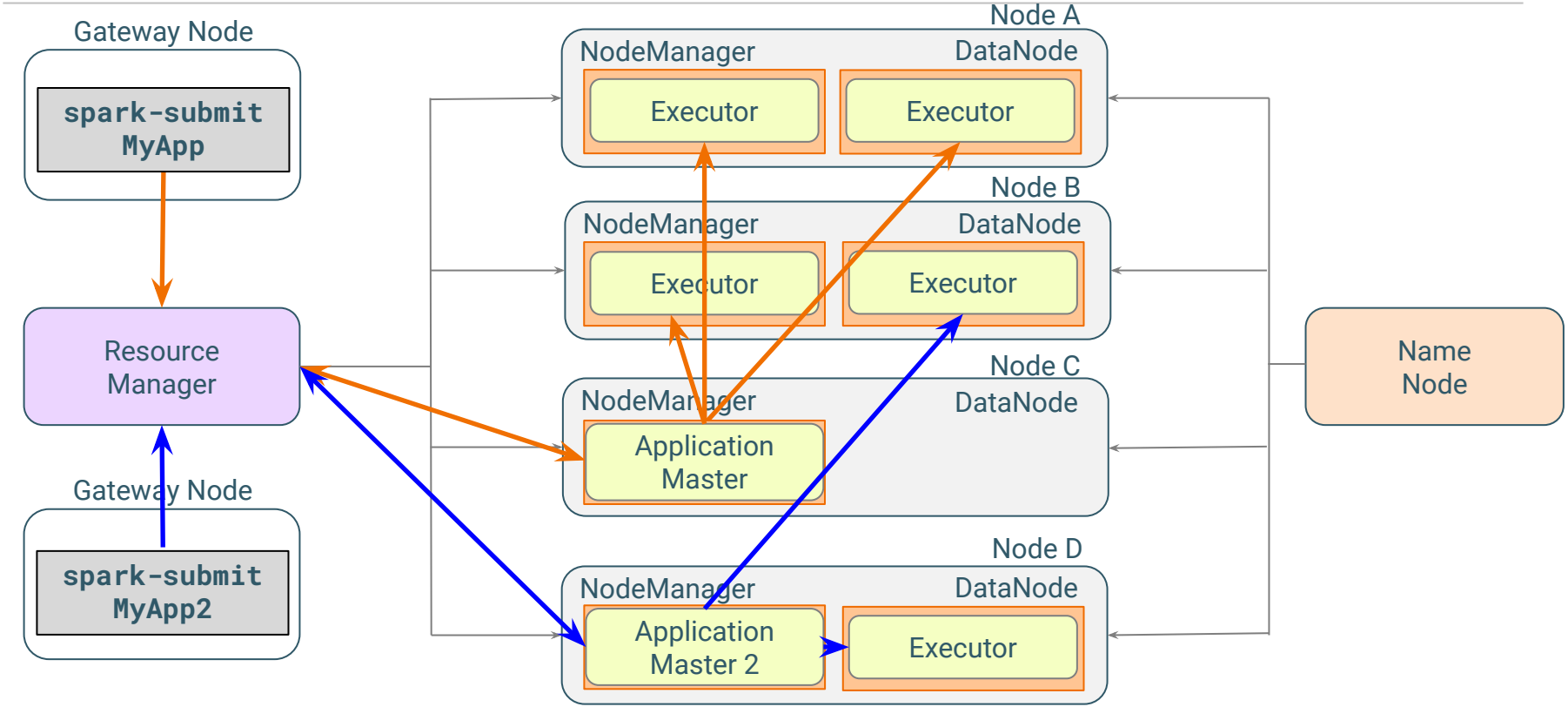
Supply and Demand of Resources

- The *supply* of resources are the actual hardware resources of the cluster
  - Number of nodes
  - RAM
  - CPU cores
- YARN properties define the supply
  - Nodes need not be *homogeneous*
  - Clusters typically have multiple generations of nodes
  - Cloudera Manager supports this with *role groups*
- Spark properties define the demand, including
  - Number of executors (static allocation)
  - Executor RAM
  - Executor cores
  - Separate requests for executors and driver containers

Scheduling

- Scheduling is the process of allocating the *supply* of resources to meet the *demand* for resources from multiple applications
- In YARN, the Resource Manager does the scheduling
  - If there is more supply than demand, jobs run in parallel
  - If there is more demand than supply, some jobs wait
- The Resource Manager schedules jobs according to a *scheduling policy*
- YARN offers the choice of two schedulers:
  - Fair Scheduler
  - Capacity Scheduler

Applications Competing for Resources



Dynamic Resource Pools

- By default, all applications are equal and compete for resources
- Organizations need some applications to have priority over others
  - "Production" applications for example
- YARN supports *resource queues*, aka *resource pools*
  - Cloudera Manager calls them "Dynamic Resource Pools"
- A Dynamic Resource Pool can be configured with:
  - Minimum and maximum amount of resources
  - A maximum number of jobs in the pool
  - A weight
  - Ability to preempt other jobs, not in the pool
- When submitting a Spark application to YARN, you can specify a resource pool
  - If you don't, your application runs in a pool named by your userid

Performance Tuning Rules of Thumb

There are rules of thumb for some params:

- # of partitions: 3x the number of cores in the cluster
- # of cores per executor: 4-8
- Memory per executor: 85% * (node memory / # executors by node)

Most of the impact come from a few parameters

- # and type of instances for exec and driver
- Executor and driver size (memory, # of cores)
- # of partitions

Resource Requests: Memory

Main takeaway: YARN memory used is always more than Spark memory requested

Property	Description	Example
`spark.executor.memory`	Specifies maximum Java heap size of executors. Affects how much memory YARN reserves for each executor process.	1 GB (default)
`spark.executor.memoryOverhead`	Off-heap memory to be allocated per executor. Default is 10% of spark.executor.memory with a minimum of **384MB**	384MB
`spark.executor.pyspark.memory`	Amount of memory to be added to executor resource requests. Used when Spark Apps are written in Python	0 MB
`spark.driver.memory`	Specifies maximum Java heap size of driver. Affects YARN memory used when Spark app is ran in cluster mode	1 GB (default)
`spark.driver.memoryOverhead`	Default is 10% of spark.driver.memory with a minimum of **384MB**	384MB

Resource Requests: Example

Example Scala Application

Property	Value
`spark.executor.memory + spark.executor.memoryOverhead`	1.384 GB
`Number of executors: (see later)`	3
`spark.driver.memory + spark.driver.memoryOverhead`	1.384 GB
`Total YARN memory used:`	Executors * 3 =: 4.5 GB + Driver: 1.384 = ~ 6 GB of YARN memory

Resource Requests: Cores

Main takeaway: Virtual cores much simpler than memory

Property	Description
`spark.executor.cores`	Number of YARN cores to reserve for each executor. Roughly equates to number of simultaneous tasks performed by executors.
`spark.driver.cores`	Number of YARN cores to reserve for ApplicationMaster/driver in YARN cluster mode. Not used frequently

An Aside: Spark really wants to run

The features described in this chapter are governed by the following settings

Property	Description
`spark.scheduler.maxRegisteredResourcesWaitingTime`	Maximum amount of time to wait for resources to register before (Spark) scheduling begins.
`spark.scheduler.maxRegisteredResourcesResourcesRatio`	Minimum ratio of registered resources (registered resources / total expected resources) (resources are executors in yarn mode to wait for before (Spark) scheduling begins.

- A Spark application will start scheduling tasks according to the settings above
- Do not be surprised if developers specify they want X executors and tasks begin executing with less than X executors

Important Configuration Settings: Dynamic Resource Allocation

All settings are configurable *per application*

Property	Description
`spark.dynamicAllocation.enabled`	Whether to use dynamic resource allocation - default is false
`spark.dynamicAllocation.minExecutors`	Lower bound for the number of executors
`spark.dynamicAllocation.maxExecutors`	Upper bound for the number of executors
`spark.dynamicAllocation.initialExecutors`	Initial number of executors to run if dynamic allocation is enabled. **Note:** maxRegisteredResourcesWaitingTime and maxRegisteredResourceRatio apply. Do not be surprised when Spark apps start running tasks before initialExecutors are granted.