Spark

Lecture 6

Recap

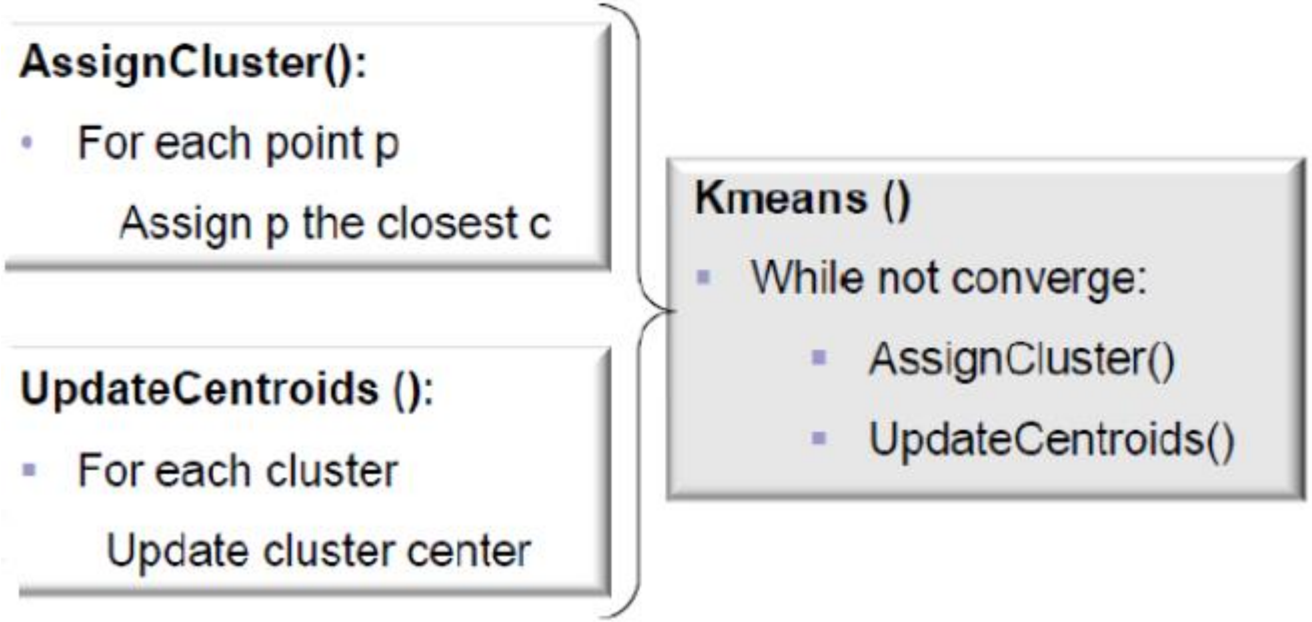
- MapReduce introduced by Google
  - Simple programming model for building distributed applications that process vast amounts of data
  - Runtime for executing jobs on large clusters in a reliable, fault-tolerant manner

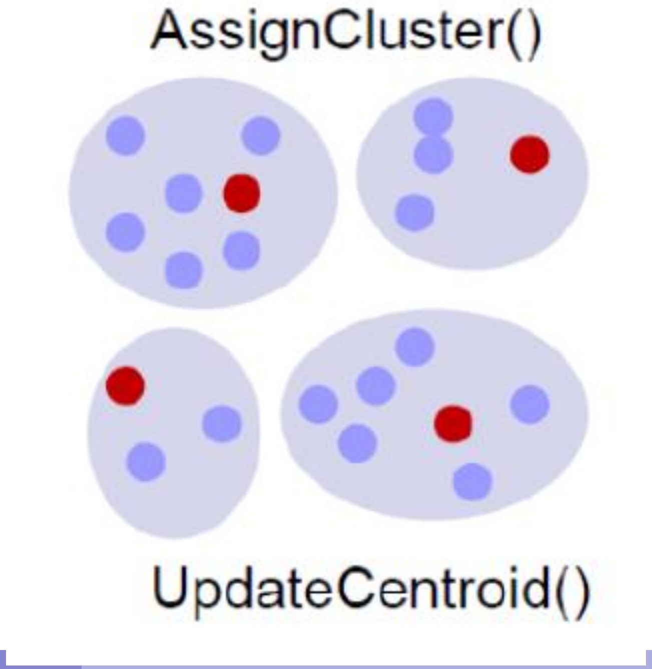
- Hadoop makes MapReduce broadly available
  - HDFS becomes central data repository
  - Becomes Defacto standard for batch processing

- Stonebraker & Dewitt: Mapreduce a major step backwards

New applications, new workloads

- Iterative computations
  - Ex: More and more people aiming to get insights from data
  - Apache Mahout becomes popular framework for ML over Hadoop
- How would we implement K-Means with MapReduce?
- Traditional k-means





K-means MapReduce algorithm

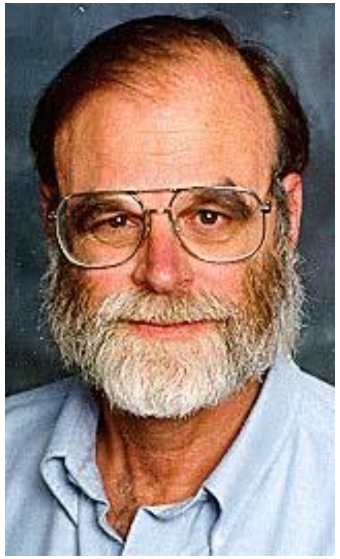
- Configure: A single file containing cluster centers
- **Mapper**
  - Input: Input data points
  - Compute: Distance of a point from each centroid to identify a cluster
  - Output: (cluster id, data id)
- **Reducer**
  - Input: (cluster id, data id)
  - Compute: New cluster centroid based on data points assigned
  - Output: (cluster id, cluster centroid)
- **Driver**
  - Each iteration produces new cluster centroids
  - Run multiple iteration jobs using mapper + reducer until convergence: <u>*High overhead leading to poor performance*</u>

MapReduce & Iterative Computations

- MapReduce is built for batch processing
  - Entirely disk based: Input and output sit on HDFS
- Let us look at **k-means algorithm, 1 iteration**
  - HDFS Read
  - **Map**(Assign sample to closest centroid)
  - NETWORK Shuffle
  - **Reduce**(Compute new centroids)
  - HDFS Write
- Each iteration reads and writes data from disk-based HDFS
  - To understand why this is bad, let us look at the memory hierarchy

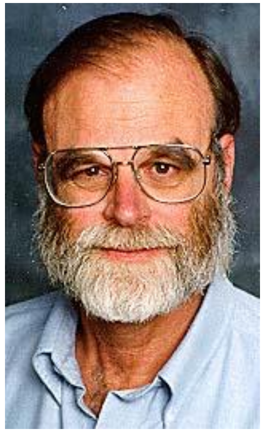
Understanding Memory Hierarchy: How Far Away is the Data?

10^9	Tape	Andromeda	2,000 Years
10^6	Disk	Pluto	2 Years
100	Memory	St. Tropez	1.5 hr
10	On Board Cache	This Building	10 min
1	Registers	This Room	1 min

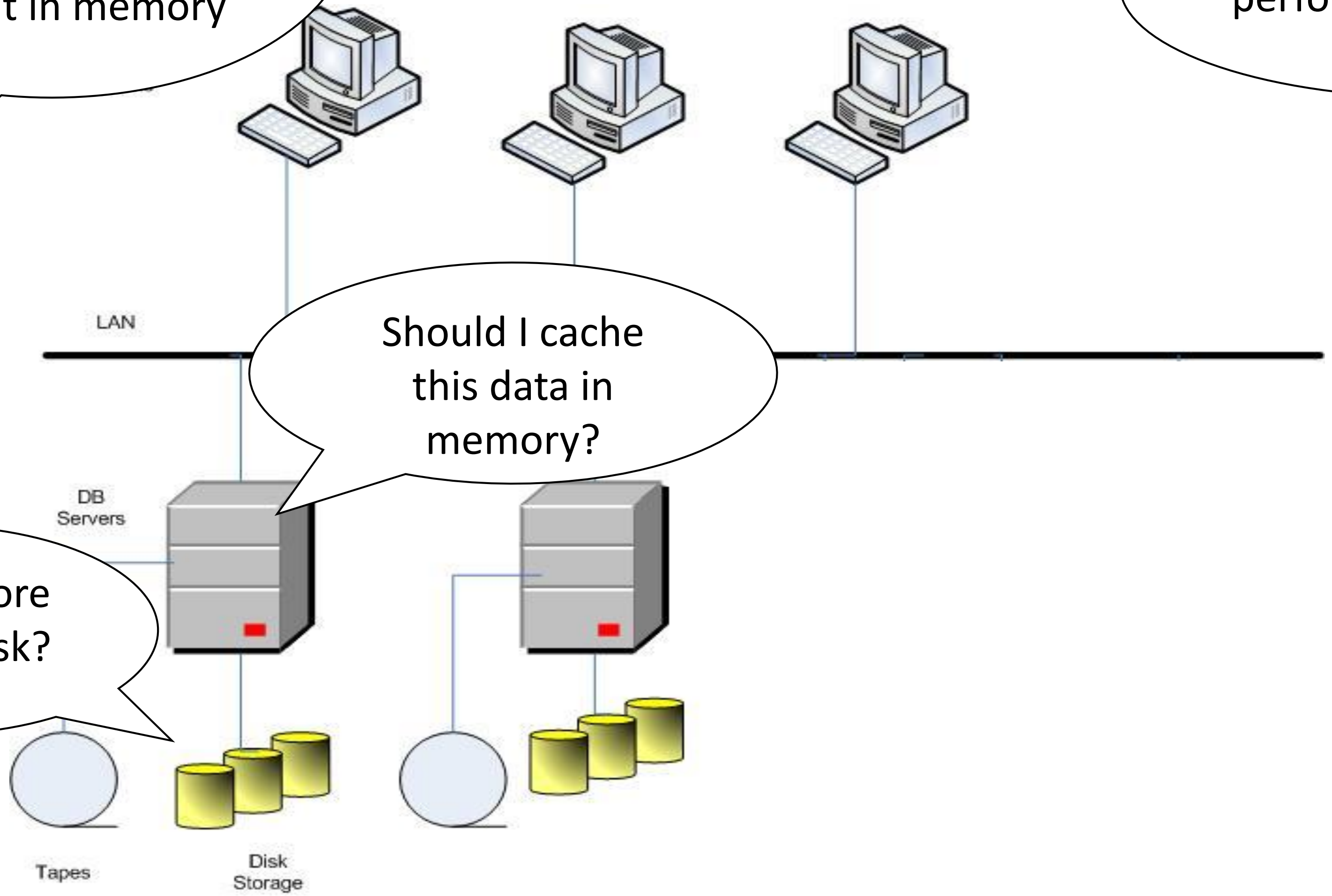


Jim Gray

1980s: Database Administrators Dilemma



If a data is accessed more than once within 5 minutes, cache it in memory



Should I cache this data in memory?

Should I store data on disk?

How do I improve the performance of my DB server?

Tandem Computers: Price/performance

- Tandem disk: **$1k/access**
  - cost: $15k for 180MB
  - performance: 15 accesses / second
- Tandem CPU + supporting hardware: **$1k/access**
  - Cost: $15,000
- Cost of accessing data from disk: **$2k/access**

- Memory cost: $5k for 1MB => **$5/KB**

Five-minute rule

- Cost of accessing data from disk: **<u>$2k/access,</u>** Memory cost: **<u>$5/KB</u>**
- If we keep 1KB in memory, assuming we have 1 access/sec
  - We save $2k of disk i/o by paying $5 for memory
- If we have 1 access every 10 secs => 0.1 access/sec
  - We save $200 of disk i/o by paying $5 for memory

.

.

.

- Break even point : 1 access every 400 secs

400 seconds ~ 5 minutes

Five-minute rule: then and now

Page size (4KB)	1987	Now
RAM-HDD	5 mins	5 hours

- RAM-HDD break-even 60x higher due to drop in DRAM price
  - Take away: Never ever go to disk!
- See “Five minute rule” CACM paper for more details
  - https://cacm.acm.org/magazines/2019/11/240388-the-five-minute-rule-30-years-later-and-its-impact-on-the-storage-hierarchy/fulltext

MapReduce/Hadoop and memory hierarchy

- Hadoop misaligned with five-minute rule
  - All data is stored in disk
  - Does not cache data in memory even if workload can fit

- Hadoop unfit for new classes of workloads
  - Interactive and iterative applications are bottlenecked by disk

- MapReduce was also too simple a computational model
  - Algorithm design with just map and reduce functions is non trivial

Hadoop: Fractured ecosystem

General Batching	Specialized systems			
	Streaming	Iterative	Ad-hoc / SQL	Graph
MapReduce	Storm	Mahout	Pig	Giraph
	S4		Hive	
	Samza		Drill	
			Impala	

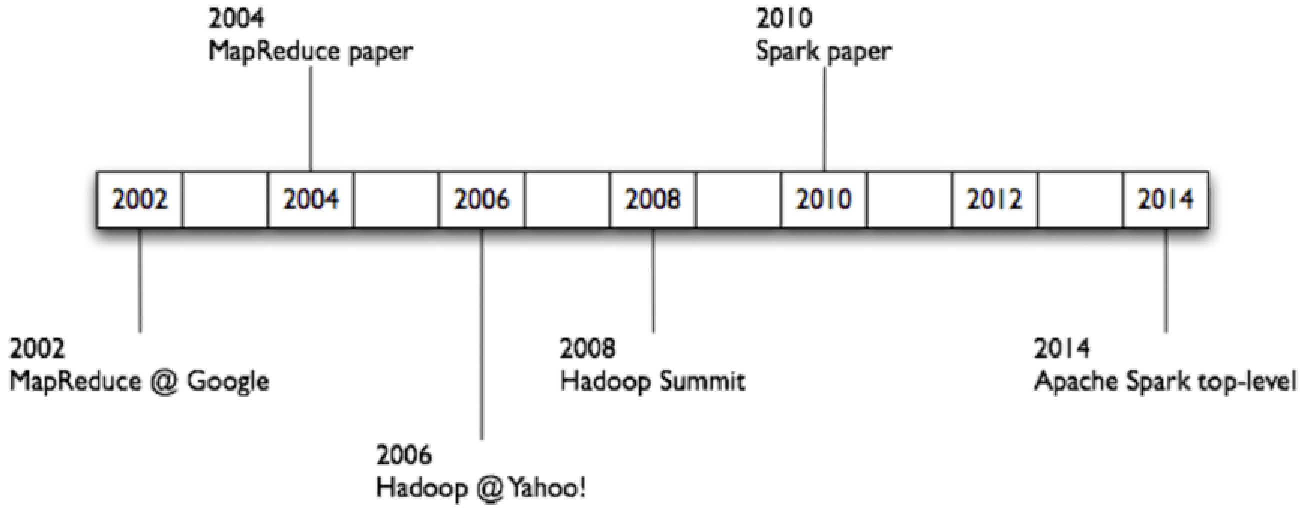
- Specialized systems emerged with no unified vision
  - Diverse APIs, sparse modules, high operational costs
  - MapReduce runtime replaced with more optimized ones

Lighting a Spark

Flexible, in-memory data processing framework written in Scala

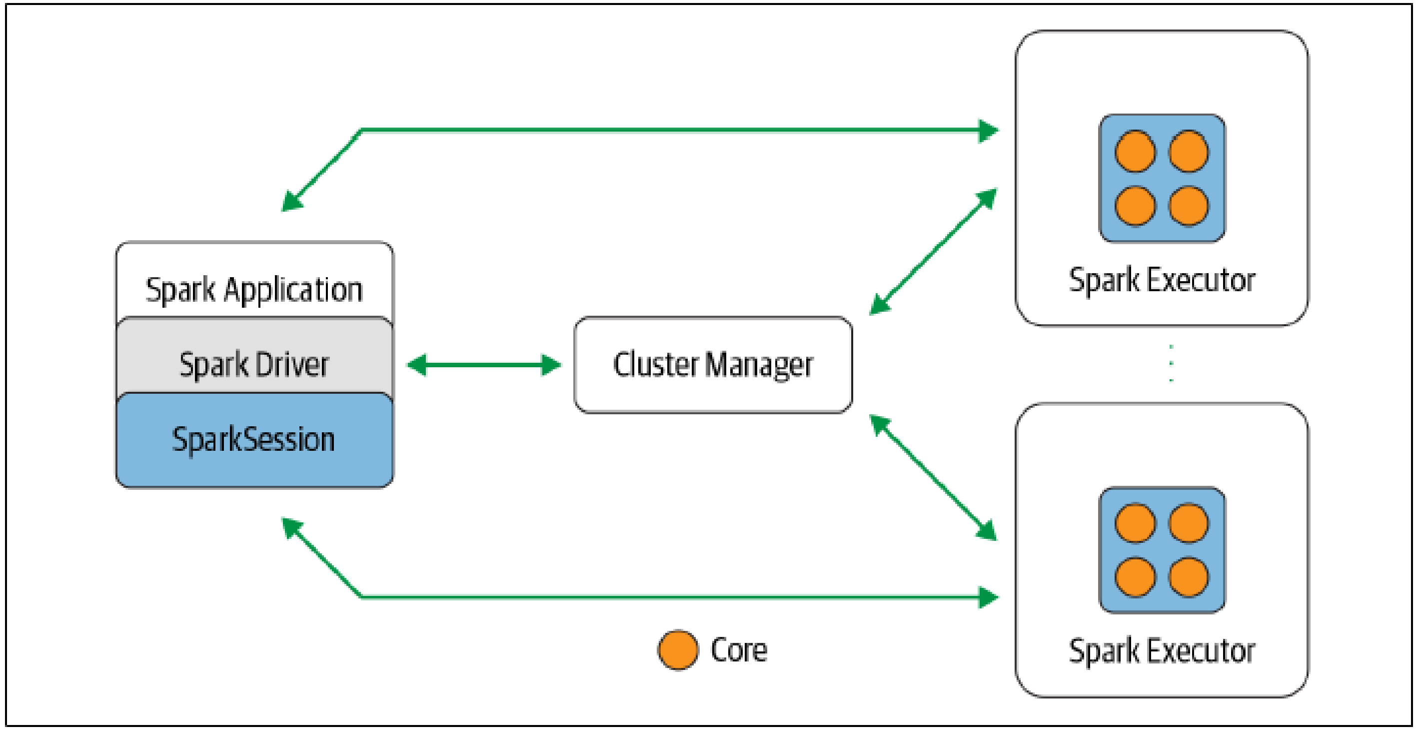
Central Ideas

- Exploit memory by caching data to enable fast data sharing
- Generalize the two-stage computational model of mapreduce to a Directed Acyclic Graph-based one that can support a richer API



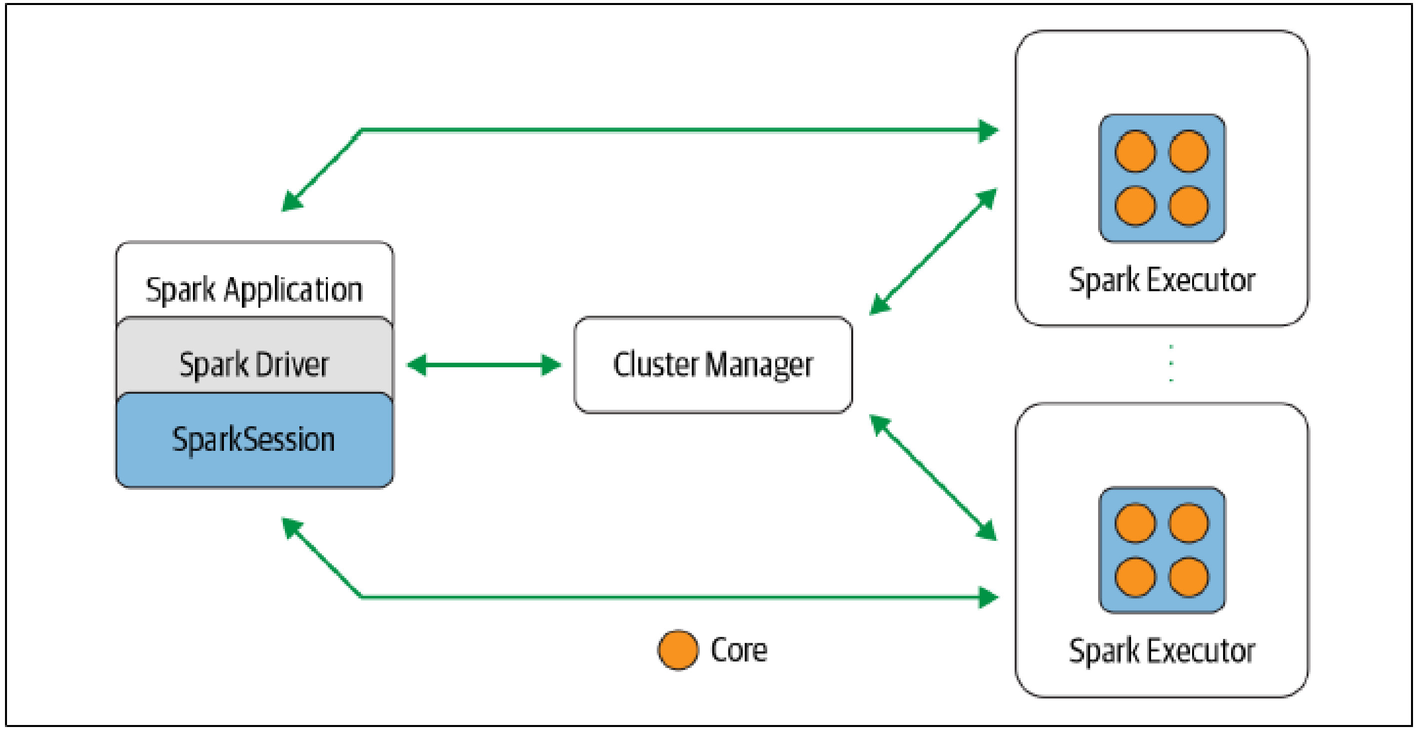
Spark Distributed Architecture

- Spark Application
  - User program built on Spark
  - Contains the Spark driver
- Spark Driver
  - Transforms all the Spark operations into DAG computations
  - Communicates with the cluster manager & requests resources (CPU, memory, etc.) for Spark executors
  - Schedules computations
  - Instantiates SparkSession



Spark Distributed Architecture

- SparkSession
  - A unified conduit to all Spark operations and data
- Cluster Manager
  - Responsible for managing and allocating resources
  - 4 supported: Standalone, YARN, Mesos, Kubernetes
- Executor
  - Responsible for executing tasks
  - Usually one per node, but depends on deployment mode



RDD: Need for a new abstraction

- Need an efficient way to share data stored in memory
- Traditional way: Distributed shared memory abstraction
  - General purpose, extends single-node shared memory to a cluster
  - Applications can make fine-grained updates to any data in memory
  - Can be used to build very efficient applications
- Problem: Fault tolerance
  - Need to replicate data across nodes or log updates which is 10-100x slower than memory write
  - Too expensive for data-intensive apps
- Goal: In-memory abstraction that provides fault-tolerance and efficiency

Resilient Distributed Dataset

RDD (Resilient Distributed Dataset): Restricted form of DSM

- An immutable, partitioned collection of objects
- Can only be built through coarse-grained deterministic transformations

- **RDD are data structures that:**
  - Either point to a direct data source (e.g. HDFS)
  - Apply some transformations to its parent RDD(s) to generate new data elements

RDD

- An abstraction that encapsulates 3 things
  - Dependencies, Partitions, Compute function
- Dependencies
  - Instruct Spark how an RDD is constructed
  - To reproduce results, Spark can recreate an RDD from these dependencies and replicate operations on it => resiliency
- Partitions
  - Split the work to parallelize computation on partitions across executors
  - Exploit data locality
- Compute function: Partition -> Iterator[t]
  - A function that produces an iterator for data stored in RDD

RDD: Example

- Query: Find average age for each name
  - aggregate all the ages for each name
  - group by name
  - average the ages

```python
# In Python
# Create an RDD of tuples (name, age)
dataRDD = sc.parallelize([("Brooke", 20), ("Denny", 31), ("Jules", 30),
  ("TD", 35), ("Brooke", 25)])
# Use map and reduceByKey transformations with their lambda
# expressions to aggregate and then compute average

agesRDD = (dataRDD
  .map(lambda x: (x[0], (x[1], 1)))
  .reduceByKey(lambda x, y: (x[0] + y[0], x[1] + y[1]))
  .map(lambda x: (x[0], x[1][0]/x[1][1])))
```

RDD Transformations

- Set of operations that define how to transform an RDD
  - Examples: map(), filter(), select(), join(), orderby(), …
- As in relational algebra, the application of a transformation to an RDD yields a new RDD
  - RDD are immutable
- Transformations are lazily evaluated
  - Computation that performs the transformation is not performed immediately

RDD Actions

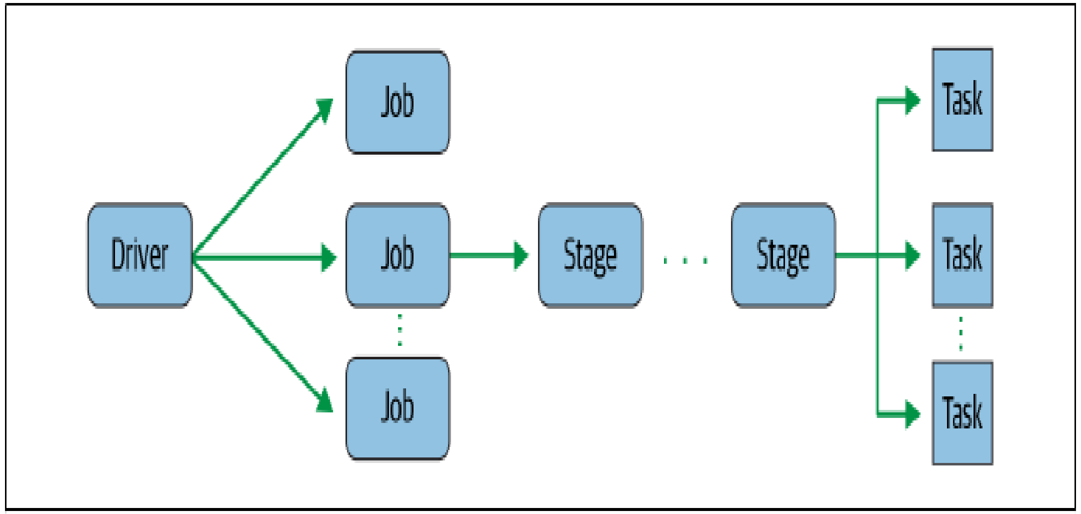
- Actions trigger computation of the chain of transformations
- Some actions only store data to an external data source (e.g. HDFS)
  - Ex: show(), take(), count(), collect(), ..
- Others fetch data from the RDD (and its transformation chain) upon which the action is applied, and convey it to the driver
  - count() – return the number of elements
  - take(*n*) – return an array of the first *n* elements
  - collect()– return an array of all elements
  - …

RDD & Lazy Execution Demo

- https://mediaserver.eurecom.fr/permalink/v12641880f54fqht30dc/iframe/#start=4615

Spark Execution

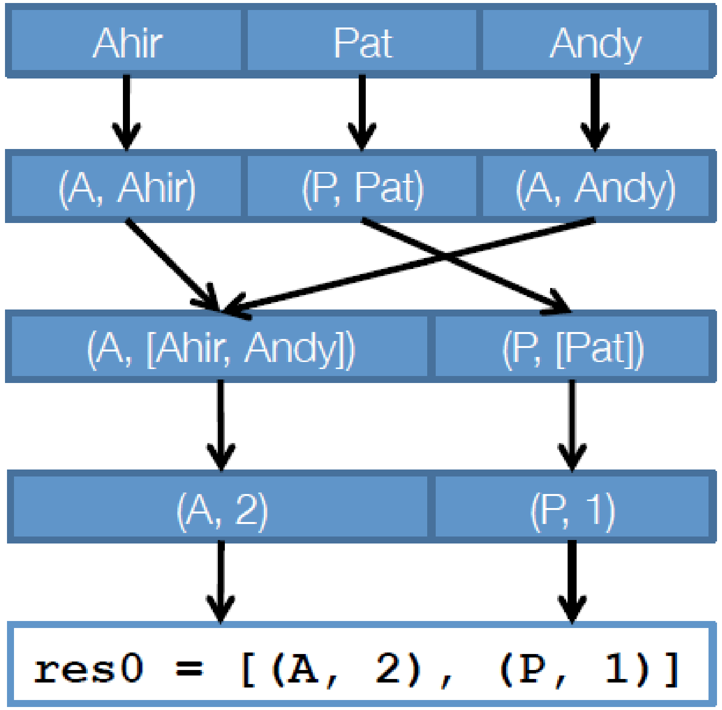
- Spark Driver
  - Converts your Spark application into one or more Spark jobs
  - Transforms each job into a DAG – execution plan
- Job broke into stages
  - Stages are created based on what operations can be performed in parallel
  - Dictate data transfer among Spark executors.
- Stages composed of tasks
  - Task - a unit of execution
  - Maps to a single core
  - Maps to 1 partition of data



Spark DAG execution: An Example

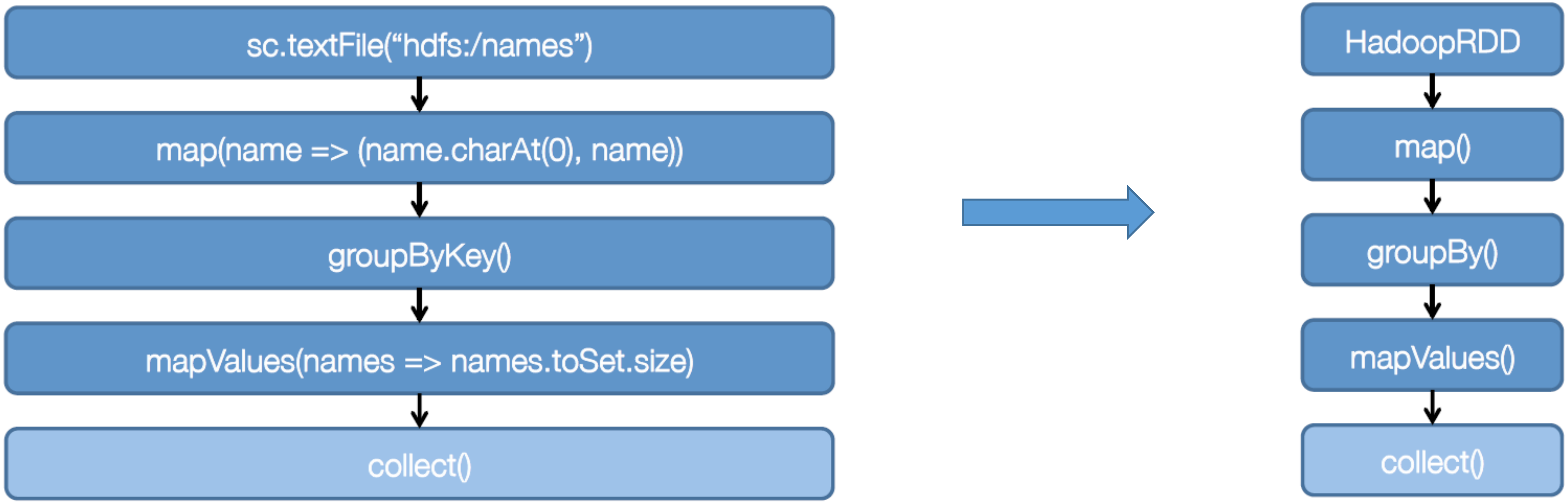
- Goal: Find the number of distinct names per first letter

```
sc.textFile("hdfs:/names")
    .map(name => (name.charAt(0), name))
    .groupByKey()
    .mapValues(names => names.toSet.size)
    .collect()
```



Spark Execution (1)

1. Create a DAG of RDDs to represent computation



Spark Execution (2)

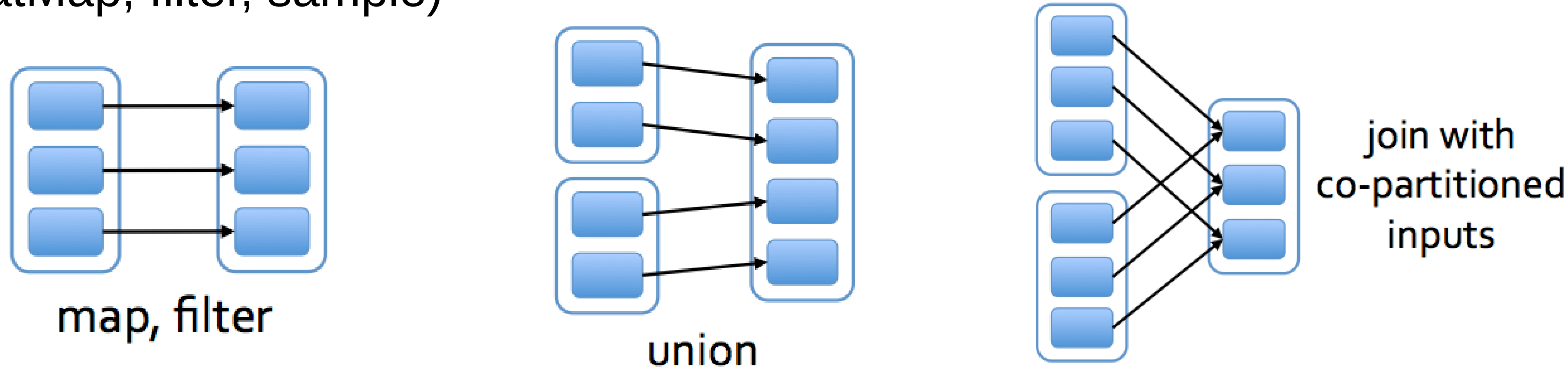
1. Create a DAG of RDDs to represent computation
2. Create logical execution plan for the DAG
   - Split DAG into “stages” based on dependencies
   - Pipeline as much as possible

RDD: Data Set vs Partition Views

Much like in Hadoop MapReduce, each RDD is stored physically in multiple nodes as input partitions

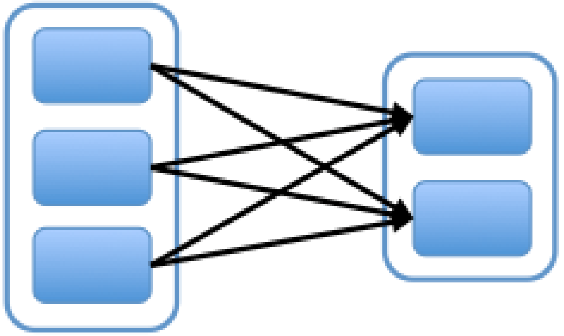
A word about dependencies (1)

- Dependencies determine the need to shuffle data
  - Two types: Narrow and wide
- Narrow dependencies
  - Each partition of the parent RDD is used by at most one partition of the child RDD
  - Task can be executed locally and we don’t have to shuffle. (E.g. map, flatMap, filter, sample)

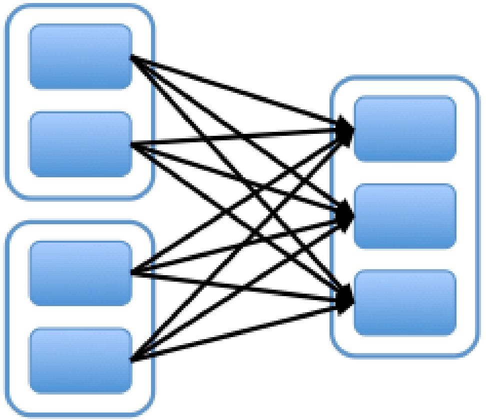


A word about dependencies (2)

- Wide dependencies
  - Multiple child partitions may depend on one partition of the parent RDD
  - We have to shuffle data (E.g. sortByKey, reduceByKey, groupByKey, cogroupByKey, join, cartesian)



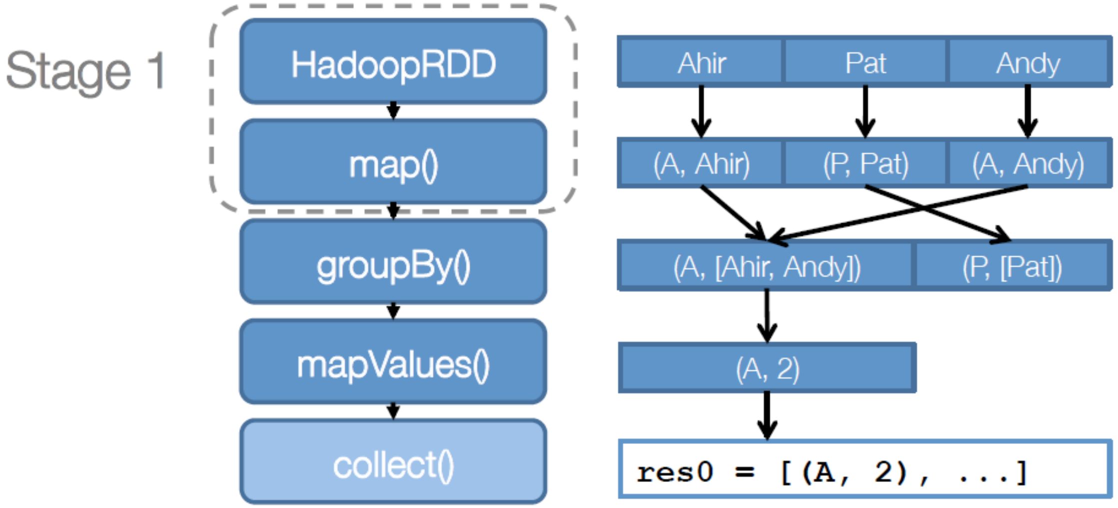
groupByKey

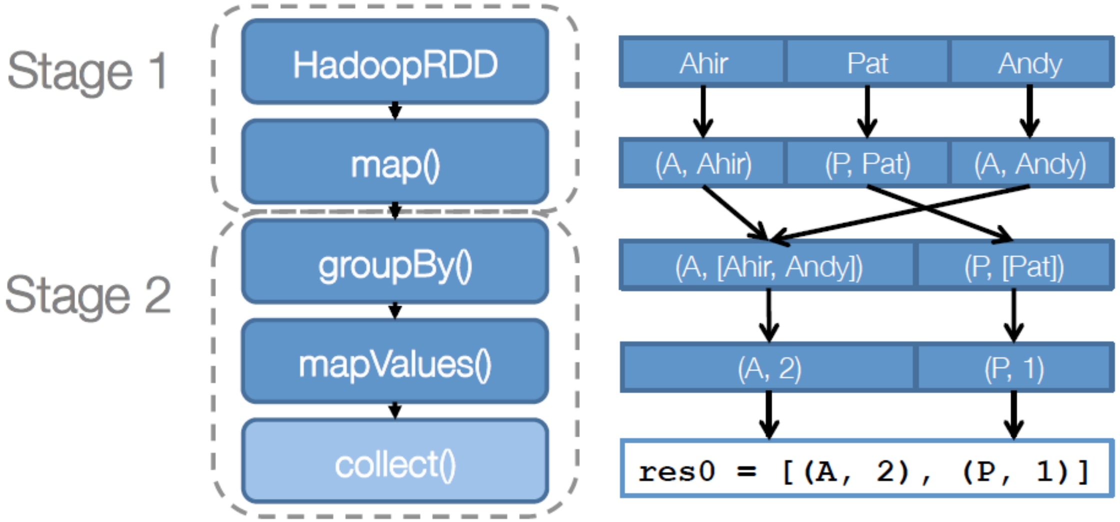


join with inputs not co-partitioned

How does Spark execute this job?

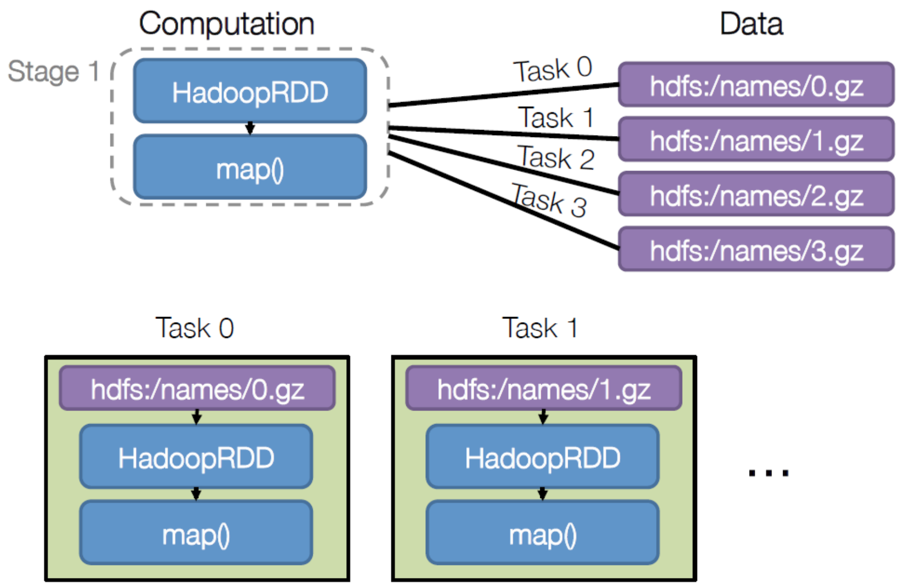
1. Create a DAG of RDDs to represent computation
2. Create logical execution plan for the DAG
   - Pipeline as much as possible
   - Split DAG into “stages” based on need to shuffle data





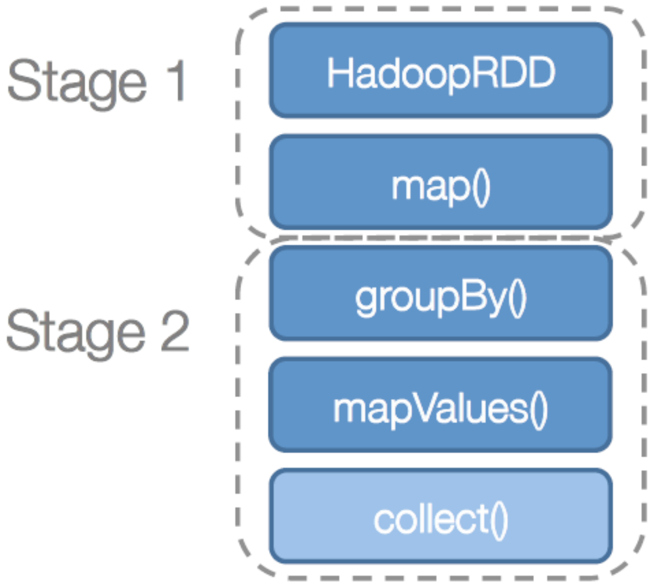
Spark Execution (3)

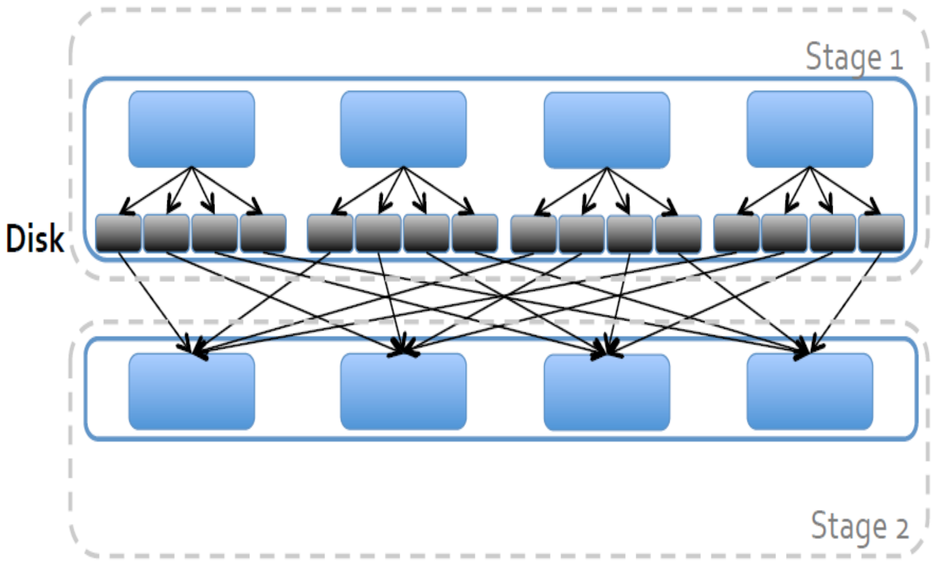
1. Create a DAG of RDDs to represent computation
2. Create logical execution plan for the DAG
3. Split each stage into tasks and execute tasks stage by stage
   - Task = Data + Computation
   - In this example, all tasks from stage 1 would be executed together first



Spark Execution (3)

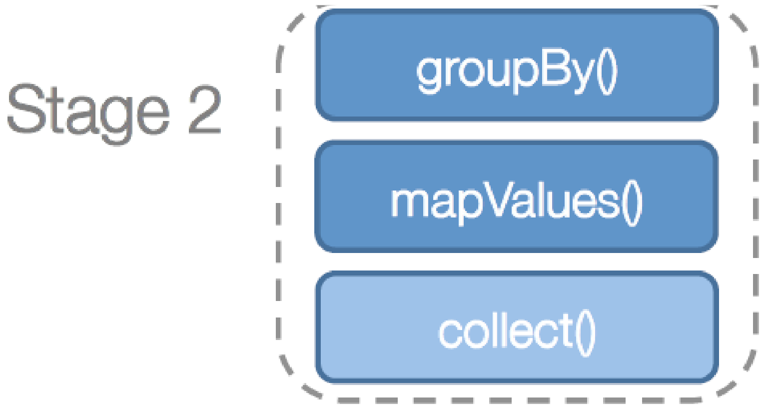
1. Create a DAG of RDDs to represent computation
2. Create logical execution plan for the DAG
3. Split each stage into tasks and execute tasks stage by stage
   - In this example, all tasks from stage 1 would be executed together first
   - After stage 1, pull-based shuffle occurs (intermediates written to files and pulled)





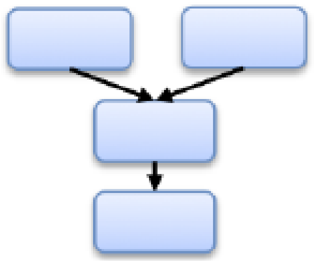
Spark Execution (3)

1. Create a DAG of RDDs to represent computation
2. Create logical execution plan for the DAG
3. Split each stage into tasks and execute tasks stage by stage
   - In this example, all tasks from stage 1 would be executed together first
   - After stage 1, pull-based shuffle occurs (intermediates written to files and pulled)
   - Now, tasks from stage 2 are executed (operators pipelined in each task)



Putting it all together

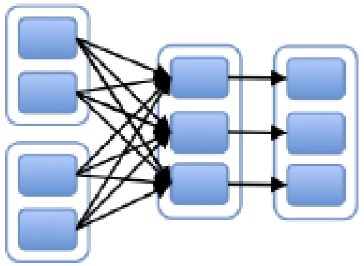
RDD Objects



```
rdd1.join(rdd2)
.groupBy(...)
.filter(...)
```

Build the operator DAG

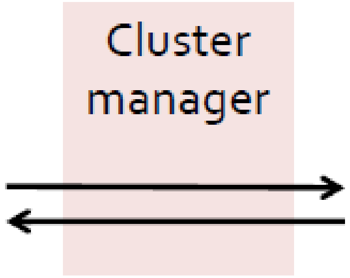
DAG Scheduler



Split the DAG into *stages* of *tasks*

Submit each stage and its tasks as ready

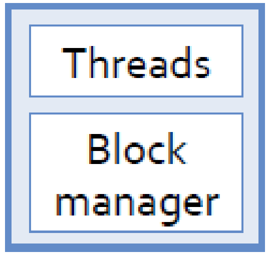
Task Scheduler



Launch tasks via Master

Retry failed and strag-gler tasks

Worker



Execute tasks

Store and serve blocks

RDD to Structured API

- Spark can support interactive workloads
  - But working with RDD is procedural
- SQL as a high-level programming language
  - Offers expressiveness, succinctness
  - Enables compatibility with existing tools, e.g. Business Intelligence using JDBC
  - Large pool of engineers proficient in SQL

```python
# In Python
# Create an RDD of tuples (name, age)
dataRDD = sc.parallelize([("Brooke", 20), ("Denny", 31), ("Jules", 30),
  ("TD", 35), ("Brooke", 25)])
# Use map and reduceByKey transformations with their lambda
# expressions to aggregate and then compute average

agesRDD = (dataRDD
  .map(lambda x: (x[0], (x[1], 1)))
  .reduceByKey(lambda x, y: (x[0] + y[0], x[1] + y[1]))
  .map(lambda x: (x[0], x[1][0]/x[1][1])))
```

```sql
SELECT name, avg(age)
FROM people
GROUP BY name
```

DataFrame: Schema

- **General idea borrowed from Python Pandas**
  - Tabular data with an API

- **Schema to the rescue**
  - A distributed collection of rows organized into named, typed columns
  - Basic types and Structured/Complex types supported
  - Schema defines column names and associated types
  - 3 ways to get schema definition (demo here: https://mediaserver.eurecom.fr/permalink/v12641880f54fqht30dc/iframe/#start=6211)
    - (i) Define with struct type, (ii) define with DDL, (iii) auto infer
  - Columns and Rows are objects with APIs

APIs

- DataSource API
  - Enables you to read/write data from/to a DataFrame from myriad data sources in formats such as JSON, CSV, Parquet, Text, Avro, ORC, etc.

- Tranformations and Actions
  - Relational Projections: done with select() method
  - Relational Selection: done with filter() or where() method
  - Aggregations: groupBy, orderBy, count, …
  - Descriptive stats: min, max, sum, avg

- Demo: https://mediaserver.eurecom.fr/permalink/v12641880f54fqht30dc/iframe/#start=6495
  - time taken to infer schema vs predeclare
  - Transformations, actions

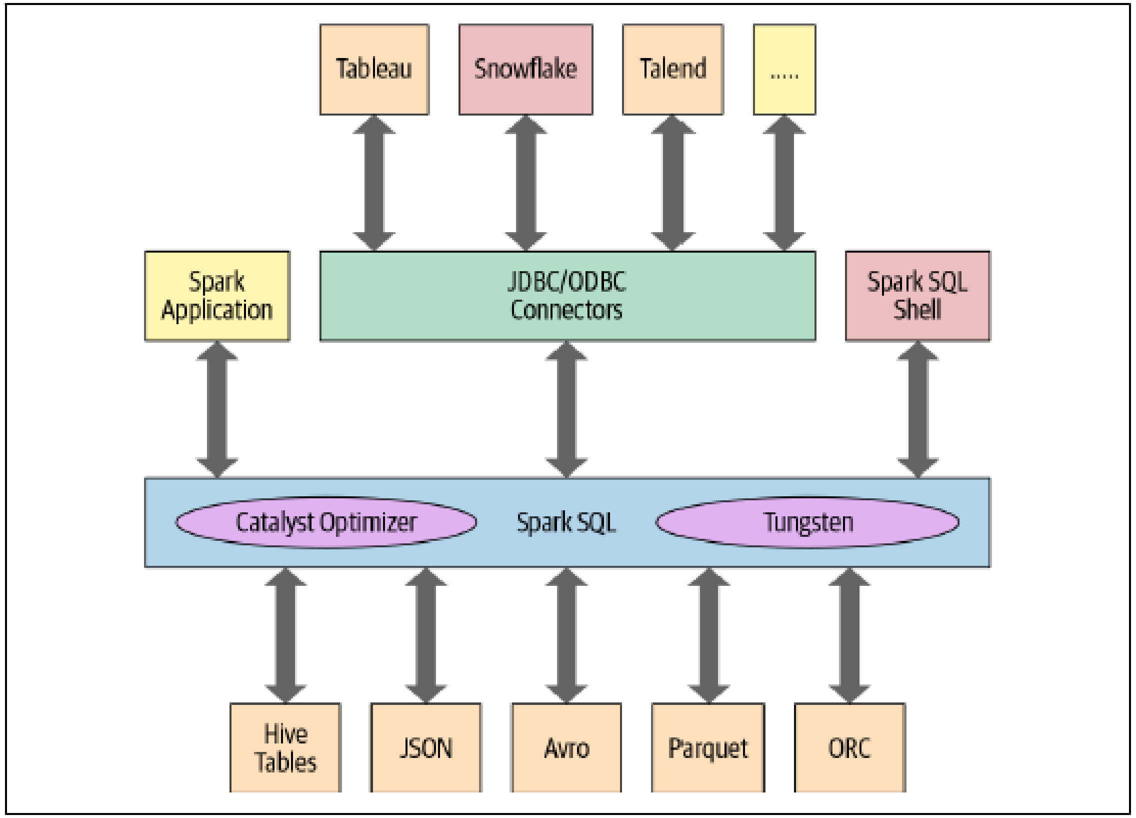
RDD vs DF

- Use RDD when
  - Want to precisely instruct Spark *how to do* a query
  - Can forgo the code optimization, efficient space utilization, and performance benefits available with DataFrames and Datasets!
  - You can get rdd from df: df.rdd

- Basically, save yourself some time and use DF

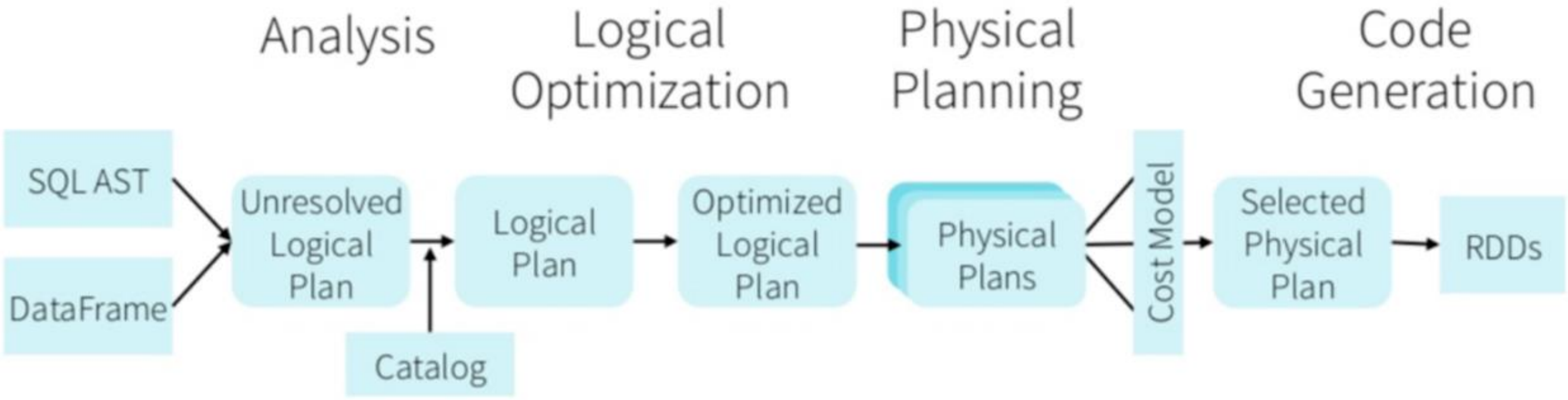
SparkSQL engine

- The substrate on which structured APIs are built
- Core components
  - Catalyst and Tungsten



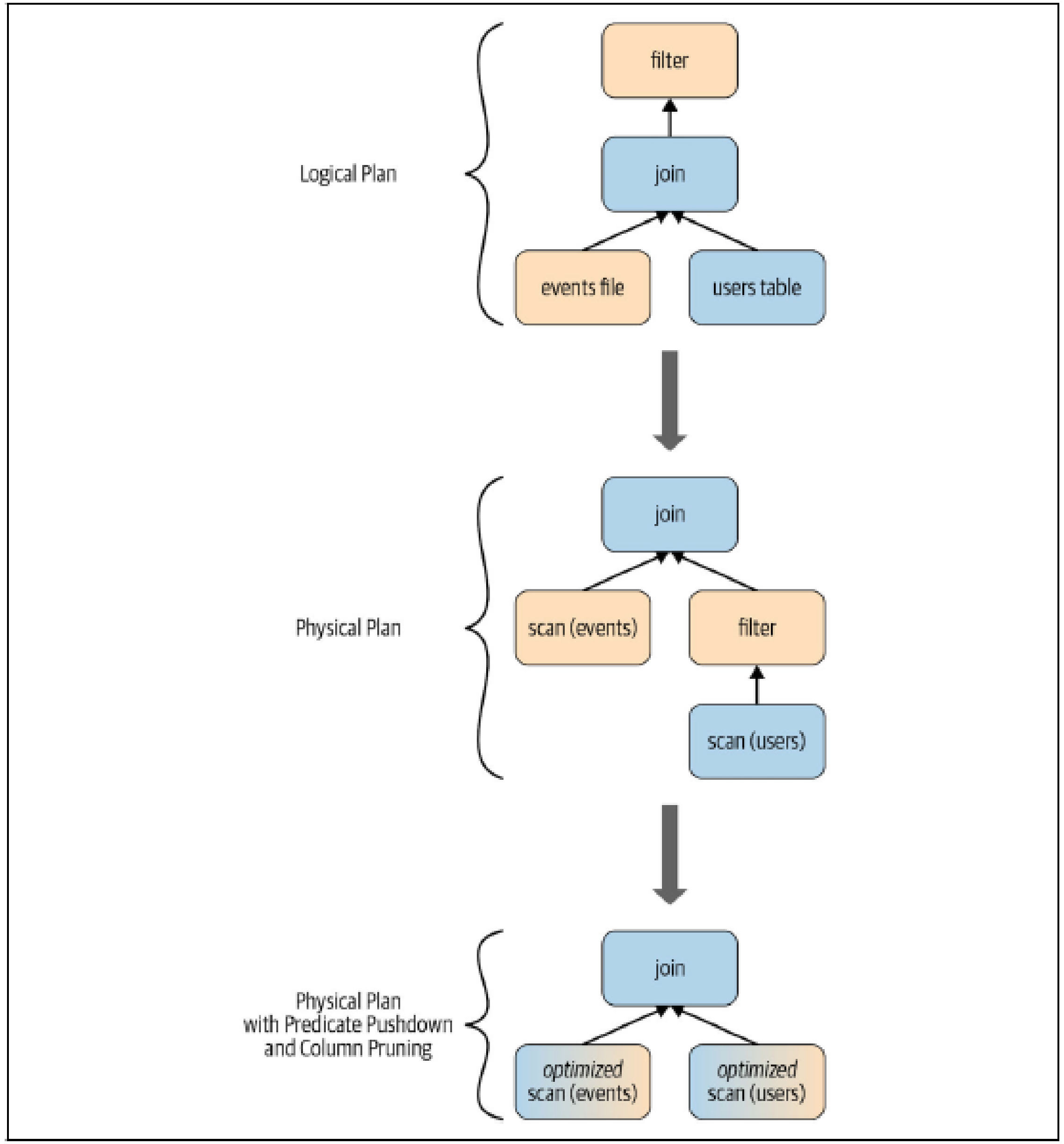
Catalyst Optimizer

- **Reminiscent of traditional database systems**
  - Analysis: SQL to Plan Abstract Syntax Tree
  - Logical & physical optimization: Use cost-based optimization to pick optimal plan



Catalyst Example

```
// In Scala
// Users DataFrame read from a Parquet table
val usersDF  = ...
// Events DataFrame read from a Parquet table
val eventsDF = ...
// Join two DataFrames
val joinedDF = users
  .join(events, users("id") === events("uid"))
  .filter(events("date") > "2015-01-01")
```



Tungesten & Code Generation

- Take optimized physical plan and do “Full stage code generation”
  - collapses the whole query into a single function
  - getting rid of virtual function calls
  - employing CPU registers for intermediate data

- Demo of Catalyst and Tungsten: https://mediaserver.eurecom.fr/permalink/v12641880f54fqht30dc/iframe/#start=7410

Spark & Caching

- DataFrame.cache()
  - store as many of the partitions read in memory across Spark executors as memory
- Dataframe.persist(StorageLevel)
  - Control how your data is cached
- Unpersist
  - Remove any cached data
- Cache/persist are hints
  - DataFrame is not fully cached until you invoke an action
- Demo of caching perf: https://mediaserver.eurecom.fr/permalink/v12641880f54fqht30dc/iframe/#start=8042

StorageLevel	Description
MEMORY_ONLY	Data is stored directly as objects and stored only in memory.
MEMORY_ONLY_SER	Data is serialized as compact byte array representation and stored only in memory. To use it, it has to be deserialized at a cost.
MEMORY_AND_DISK	Data is stored directly as objects in memory, but if there's insufficient memory the rest is serialized and stored on disk.
DISK_ONLY	Data is serialized and stored on disk.
OFF_HEAP	Data is stored off-heap. Off-heap memory is used in Spark for storage and query execution; see "Configuring Spark executors' memory and the shuffle service" on page 178.
MEMORY_AND_DISK_SER	Like MEMORY_AND_DISK, but data is serialized when stored in memory. (Data is always serialized when stored on disk.)

Spark & RDBMS: Summary

- Spark: unified analytics engine
  - Quickly adopted RDBMS concepts to optimize SQL analytics
  - Other libraries developed for machine learning (Mlib), graph analytics(GraphX),…
  - RDD: an underlying abstraction that supports several libraries

- DBMSs have also evolved
  - Disk-based to in-memory to NVM
  - One-size-fits-all “OldSQL” DBMS to customized “NewSQL” engines
    - column stores for Business Intelligence
    - highly parallel transaction engines for OLTP
    - Array databases for scientific applications
    - …
  - NewSQL still king for structured data management