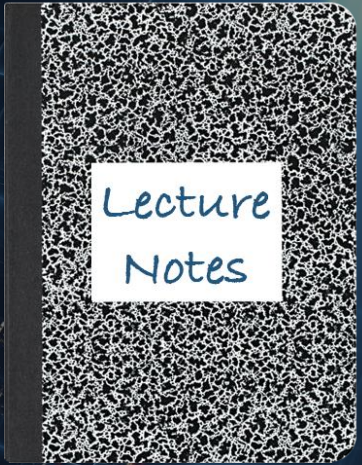
CS 417 – DISTRIBUTED SYSTEMS

Week 11: Large-Scale Data Processing

Part 3: Spark

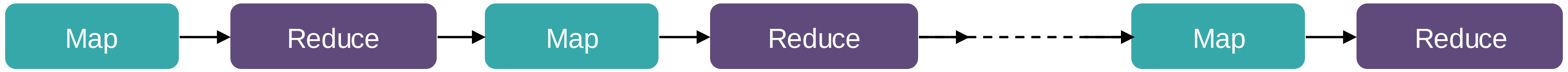


Paul Krzyzanowski

Generalizing MapReduce

MapReduce problems

- Not efficient when multiple passes needed
- Problems need to be converted to a series of Map & Reduce operations



- The next phase can never start until the previous has completed
- Output needs to be stored in the file system before the next step starts
  - Storage involves disk writes & replication
- Possibly unnecessary stages, such as when *map* simply passes <*key, value*> results from the previous *reduce*

Apache Spark Goals

- Generalize MapReduce
  - Similar shard-and-gather approach to MapReduce
  - Create multi-step pipelines based on directed acyclic graphs (DAGs) of data flows
- Create a general functional programming model
  - *Transformation* and *action*
  - In MapReduce, *transformation* = *map*, *action* = *reduce*
  - In Spark, support operations beyond *map* and *reduce*
- Add fast data sharing
  - In-memory caching
  - Different computation phases can use the same data if needed
- And generic data storage interfaces
  - Storage agnostic: use HDFS, Cassandra database, whatever
  - Resilient Distributed Data (RDD) sets
    - An RDD is a chunk of data that gets processed – a large collection of stuff

Spark Design: RDDs

RDD: **Resilient Distributed Datasets**

- Table that can be sharded (split) across many servers
- Holds any type of data
- Immutable: you can process the RDD to create a new RDD but not modify the original

Two operations on RDDs

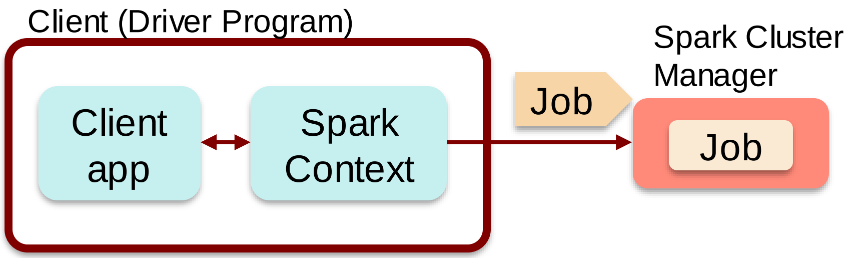
1. **Transformations**: transformation function takes RDD as input & creates a new RDD: *RDD→RDD'*
   - Examples: *map, filter, flatMap, groupByKey, reduceByKey, aggregateByKey*, ...
2. **Actions**: evaluates an RDD and creates a value: *RDD→result*
   - Examples: *reduce, collect, count, first, take, countByKey*, ...

Shared variables

- **Broadcast Variables**: define read-only data that will be cached on each system
- **Accumulators**: used for counters (e.g., in MapReduce) or sums
  - Only the driver program can read the value of the accumulator

High-level view

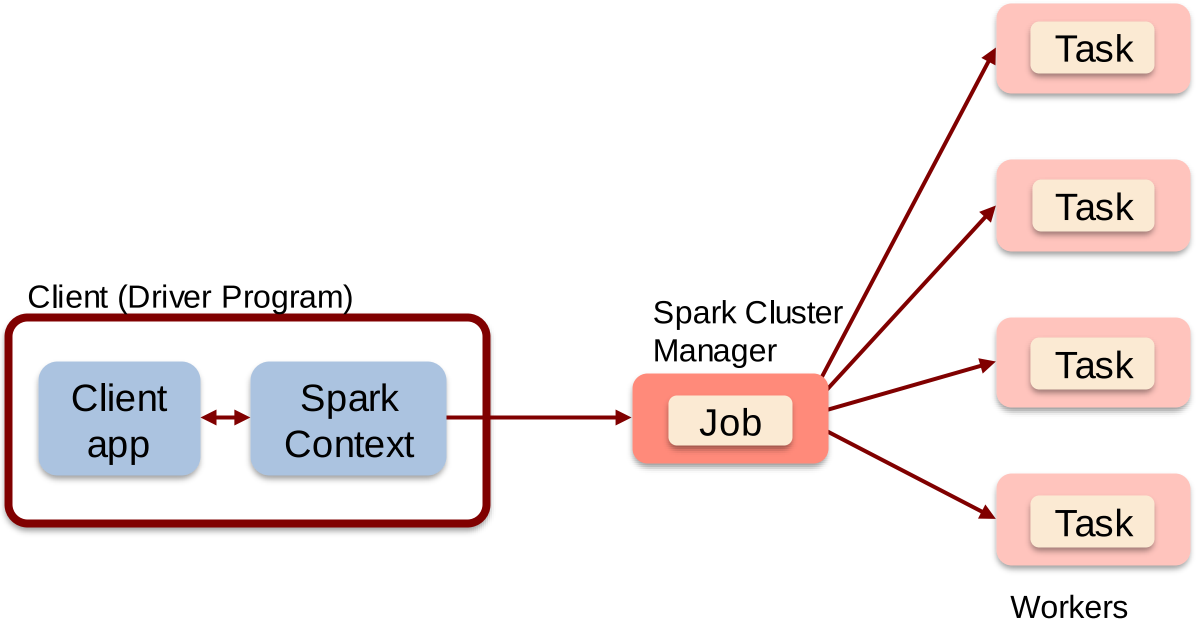
Job = bunch of transformations & actions on RDDs



High-level view

Cluster manager breaks the job into tasks

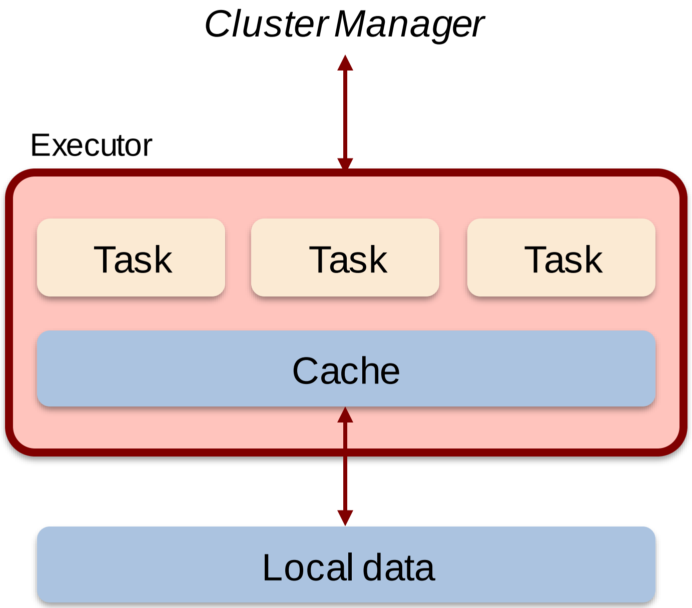
Sends tasks to worker nodes where the data lives



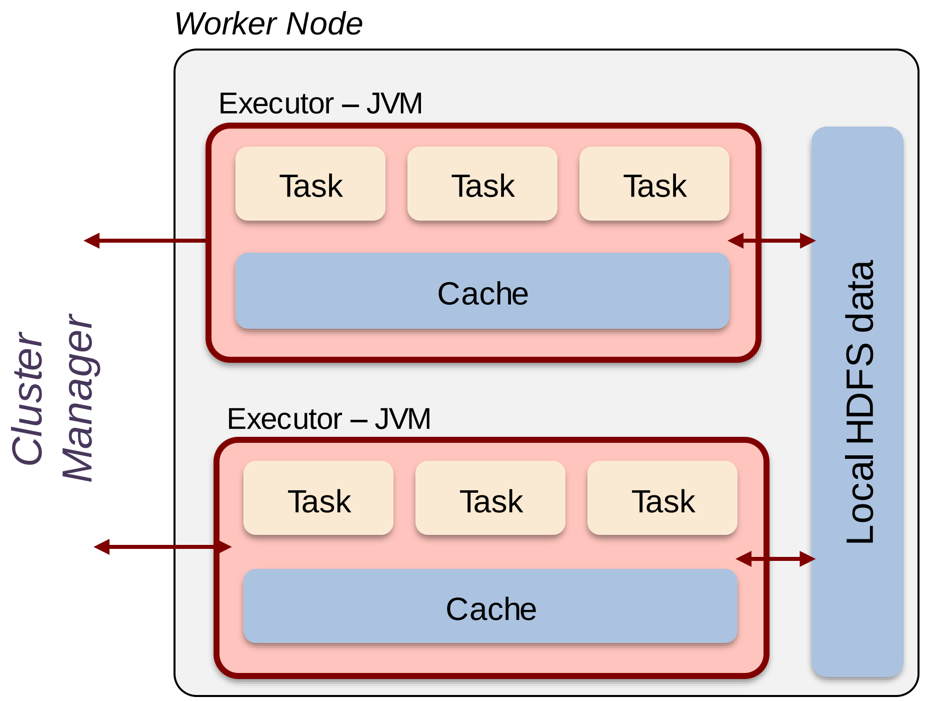
Worker node

One or more **executors**. Each executor:

- Runs as a JVM (Java Virtual Machine) process
- Talks with the Spark cluster manager
- Receives tasks
  - JVM code
    (e.g., compiled Java, Clojure, Scala, JRuby, …)
  - Task = **transformation** or **action**
- Gets data to be processed: the RDD
- Has its own cache
  - Stores results in memory
  - Key to high performance



Worker node


Worker Node
Executor – JVM
Task
Task
Task
Cache
Executor – JVM
Task
Task
Task
Cache
Local HDFS data
Cluster Manager

Data & RDDs

- Data organized into RDDs
  - One RDD may be partitioned across lots of computers
- How are RDDs created?
  - Create it from any file stored in HDFS or other storage supported in Hadoop (Amazon S3, HDFS, HBase, Cassandra, etc.)
    - Created externally (e.g., text files, SQL or NoSQL database)
    - Examples:
      - Query a database & make the query results into an RDD
      - Any Hadoop *InputFormat*, such as a list of files or a directory
  - Streaming sources (via *Spark Streaming*)
    - Fault-tolerant stream with a sliding time window
  - Output of a Spark *transformation* function
    - Example, filter out data, select key-value pairs

Properties of RDDs

Immutable	• You cannot change it – only create new RDDs • The framework will eventually collect unused RDDs
Partitioned	Parts of an RDD may go to different servers • Splits can be range-based or hash-based • For hash-based, default partitioning function = *hash(key) mod server_count*
Dependent	Created from – and thus **dependent** on – other RDDs • Either original source data or computed from one or more other RDDs
Fault tolerant	Original RDD in stable storage; other RDDs can be regenerated if needed
Persistent	Optional for intermediate RDDs • Original data is persistent. Intermediate data can be marked to be persistent
Typed	Contains some parsable data structure – e.g., a key-value set
Ordered (optional)	Elements in an RDD can be sorted

Operations on RDDs

Two types of operations on RDDs:

1. **Transformations**: create new RDDs
   - **Lazy**: computed when needed, not immediately
   - Transformed RDD is computed when an action is run on it
     - **Work backwards**:
       - What RDDs do you need to apply to get an action?
       - What RDDs do you need to apply to get the input to this RDD?
   - RDD can be persisted into memory or disk storage

2. **Actions**: create result values
   - **Finalizing** operations
     - *Reduce*, *count*, *grab samples*, *write to file*

Spark Transformations

Transformation	Description
map(func)	Pass each element through a function *func*
filter(func)	Select elements of the source on which *func* returns true
flatmap(func)	Each input item can be mapped to 0 or more output items
sample(withReplacement, fraction, seed)	Sample a *fraction* fraction of the data, with or without replacement, using a given random number generator seed
union(otherdataset)	Union of the elements in the source data set and *otherdataset*
intersection(otherdataset)	The elements that are in common to two datasets

Spark Transformations

Transformation	Description
groupByKey([numtasks])	When called on a dataset of (K, V) pairs, returns a dataset of (K, seq[V]) pairs
reduceByKey(func, [numtasks])	Aggregate the values for each key using the given *reduce* function
sortByKey([ascending], [numtasks])	Sort keys in ascending or descending order
join(otherDataset, [numtasks])	Combines two datasets, (K, V) and (K, W) into (K, (V, W))
cogroup(otherDataset, [numtasks])	Given (K, V) and (K, W), returns (K, Seq[V], Seq[W])
cartesian(otherDataset)	For two datasets of types T and U, returns a dataset of (T, U) pairs

Spark Actions

Action	Description
reduce(func)	Aggregate elements of the dataset using *func*.
collect(func, [numtasks])	Return all elements of the dataset as an array
count()	Return the number of elements in the dataset
first()	Return the first element of the dataset
take(n)	Return an array with the first *n* elements of the dataset
takeSample(withReplacement, fraction, seed)	Return an array with a random sample of *num* elements of the dataset

Spark Actions

Action	**Description**
saveAsTextFile(path)	Write dataset elements as a text file
saveAsSequenceFile(path)	Write dataset elements as a Hadoop SequenceFile
countByKey ()	For (K, V) RDDs, return a map of (K, Int) pairs with the count of each key
foreach(func)	Run *func* on each element of the dataset

Data Storage

- Spark does not care how source data is stored
  - RDD connector determines that
  - E.g.,
    read RDDs from tables in a Cassandra DB,
    write new RDDs to HBase tables

- **RDD Fault tolerance**
  - RDDs track the sequence of transformations used to create them
  - Enables recomputing lost RDDs
    - Go back to the previous RDD and apply the transforms again
    - Dependencies tracked by Spark in a **directed acyclic graph** (DAG)

Example: processing logs

- Transform (creates new RDDs)
  - Extract error message from a log
  - Parse out the source of error
- Actions: count mysql & php errors

```
// base RDD
val lines = sc.textFile("hdfs://...”)

// transformed RDDs
val errors = lines.filter(_.startsWith("ERROR"))
val messages = errors.map(_.split("\t")).map(r => r(1))
messages.cache()

// action 1
messages.filter(_.contains("mysql")).count()

// action 2
messages.filter(_.contains("php")).count()
```

Initial RDD – our data source

Extract only lines starting with ERROR

Split string by tabs.
Then extract string after the ERROR
Cache the results:
default is memory and disk as an overflow

Filter transformation to extract lines
Containing "mysql" – then count them

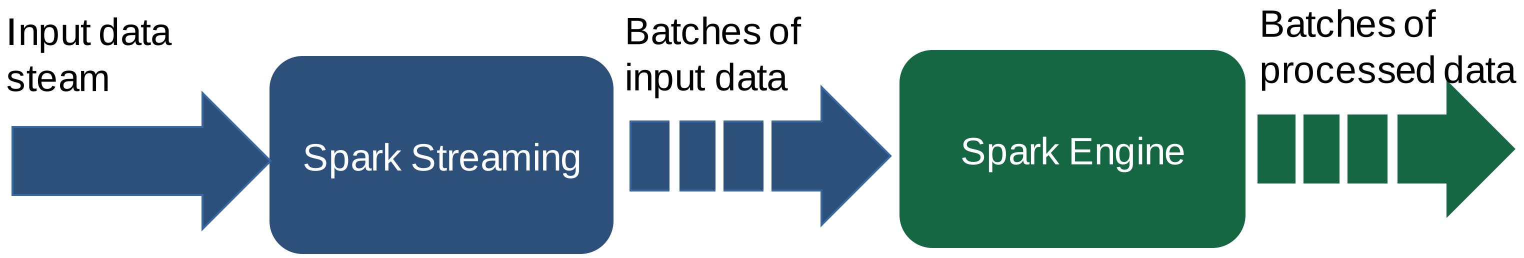
Filter transformation to extract lines
Containing "php" – then count them

Spark Ecosystem

- **Spark Streaming**: process real-time streaming data
  - Micro-batch style of processing
  - Uses DStreams: sequences of RDDs, each representing an interval of time
- **Spark SQL**: access Spark data over JDBC API
  - Use SQL-like queries on Spark data
- **Spark Mlib**: machine learning library
  - Utilities for classification, regression, clustering, filtering, ...
- **Spark GraphX**: graph computation
  - Adds Pregel API to Spark
  - Extends RDD by introducing a directed multi-graph with properties attached to each vertex & edge
  - Set of operators to create subgraphs, join vertices, aggregate messages, ...

Spark Streaming

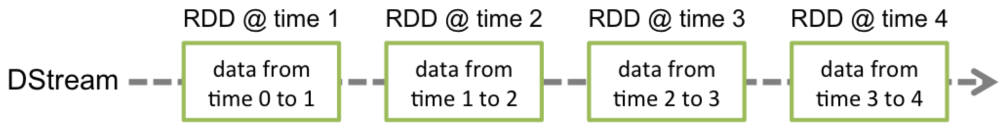
- MapReduce & Pregel expect static data
- **Spark Streaming** enables processing live data streams
  - Same programming operations
  - Input data is chunked into batches
    - Programmer specifies time interval



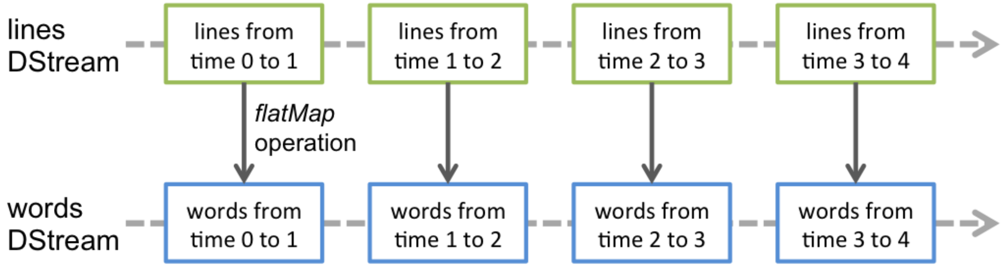
Spark Streaming: DStreams

Discretized Stream = DStream

– Continuous stream of data (from source or a transformation)
– Appears as a continuous series of RDDs, each for a time interval



– Each operation on a DStream translates to operations on the RDDs



– Join operations allow combining multiple streams

Spark Summary

- Fast
  - Often up to 10x faster on disk and 100x faster in memory than MapReduce
  - General execution graph model
    - No need to have ”useless” phases just to fit into the model
  - In-memory storage for RDDs
- Fault tolerant: RDDs can be regenerated
  - You know what the input data set was, what transformations were applied to it, and what output it creates
- Supports streaming
  - Handle continuous data streams via Spark Streaming

The End