# Distributed Systems

22. Spark

Paul Krzyzanowski
Rutgers University
Fall 2016



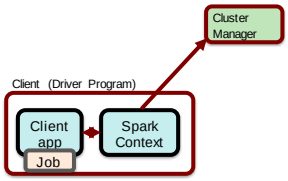
## Apache Spark

- Goal: generalize MapReduce
  - Similar shard-and-gather approach to MapReduce
  - Add fast data sharing & general DAGs (graphs)
- 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
  - In-memory caching
- More general functional programming model
  - *Transformations* and *actions*
  - In Map-Reduce, *transformation* = *map*, *action* = *reduce*



## High-level view

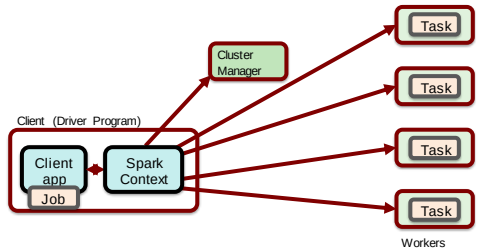
- Job = bunch of transformations & actions on RDDs
- Cluster manager: Allocates worker nodes






## High-level view

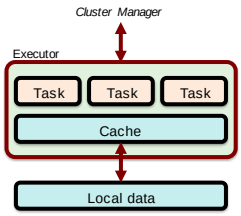
- Driver breaks the job into tasks
- Sends tasks to worker nodes where the data lives






## Worker node

- One or more executors
  - JVM process
  - Talks with cluster manager
  - Receives tasks
    - JVM code (e.g., compiled Java, Clojure, Scala, Jruby, …)
    - Task = **transformation** or **action**
  - Data to be processed (RDD)
    - Local to the node
  - Cache
    - Stores frequently-used data in memory
    - Key to high performance






## Data & RDDs

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

## Properties of RDDs

Main Properties

- Immutable
  - You cannot change it – only create new RDDs
  - The framework will eventually collect unused RDDs
- Partitioned – parts of an RDD go to different servers
  - Default partitioning function = *hash(key) mod server_count*

Optional Properties

- Typed: they're not BLOBs
  - Embedded data structure – e.g., key-value set
- Ordered
  - Elements in an RDD can be sorted



## Operations on RDDs

Two types of operations on RDDs

- Transformations
  - Lazy – not computed immediately
  - Transformed RDD is recomputed 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
- Actions
  - 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* |
| **distinct**([numtasks]) | The distinct elements of the source dataset |



## 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 Cassandra tables
- RDD Fault tolerance
  - RDDs track the sequence of transformations used to create them
  - Enables recomputing of lost data
    - Go back to the previous RDD and apply the transforms again



## Example: processing logs

- Transform (creates new RDDs)
  - Grab error message from a log
  - Grab only ERROR messages & extract 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()
```



## Spark Streaming

- Map-Reduce & 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

- Supports streaming
  - Handle continuous data streams via Spark Streaming
- 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



## The end