ST446 Distributed Computing for Big Data

Lecture 3

Distributed computation models

Milan Vojnovic

https://github.com/lse-st446/lectures2021

Goals of this lecture

- Understand the main principles of some distributed computation models

- Learn about MapReduce computation model, and about
  - Apache Hadoop architecture

- Learn about Pregel computation model

- Learn about Resilient Distributed Sets (RDDs) and about
  - API provided by Apache Spark to perform operations on RDDs
  - Apache Spark architecture

Topics of this lecture

- MapReduce
- Pregel
- Resilient Distributed Datasets
- Resilient Distributed Datasets cont’d

MapReduce

MapReduce

- MapReduce: a programming model and an associated implementation for processing large datasets
  - Map: users specify a map function that processes each key/value pair to generate a set of intermediate key/value pairs
  - Reduce: users specify a reduce function that merges all intermediate values associated with the same intermediate key
- Developed by Google, first described to public in an OSDI 2004 paper

Word count example

Map:

```
map(String key, String value):
   // key: document name
   // value: document content
   for each word w in value:
      EmitIntermediate(w, "1")
```

Reduce:

```
reduce(String key, Iterator values):
   // key: a word
   // values: list of counts
   int result = 0
   for each v in values:
      result += ParseInt(v)
   Emit(AsString(result))
```

Use cases

- Distributed grep
  - Find all lines of text matching a supplied pattern
  - Map: emits a line if it matches the supplied pattern
  - Reduce: identity function copying the supplied intermediate data to the output

- Count of URL access frequency
  - For each URL return the number of times it was accessed by a user
  - Map: `(URL,1)`
  - Reduce: `(URL, total count)`

- Reverse web-link graph
  - For each URL (source) find all pages (URLs) pointing to it (target)
  - Map: `(target, source)`
  - Reduce: `(target, list(source))`

Use cases (cont'd)

- Inverted index
  - For each word, create a list of documents containing it
  - Map: `(word, document ID)`
  - Reduce: `(word, list(document ID))`

- Distributed sort
  - Sort (key, record) pairs with respect to key values
  - Map: `(key, record)`
  - Reduce: emits all pairs unchanged

  - Mapreduce guarantees that for each reduce task the assigned set of intermediate keys is processed in key order
  - If the keys are partitioned across reducers according to range partitioning, then we only need to concatenate outputs of reducers

Grep example

```
LSE021353:~ vojnovic$ man grep

GREP(1)                   BSD General Commands Manual                  GREP(1)

NAME
     grep, egrep, fgrep, zgrep, zegrep, zfgrep -- file pattern searcher

SYNOPSIS
     grep [-abcdDEFGHhIiJLlmnOopqRSsUVvwxZ] [-A num] [-B num] [-C[num]] [-e pattern] [-f file]
          [--binary-files=value] [--color[=when]] [--colour[=when]] [--context[=num]] [--label]
          [--line-buffered] [--null] [pattern] [file ...]

DESCRIPTION
     The grep utility searches any given input files, selecting lines that match one or more
     patterns.  By default, a pattern matches an input line if the regular expression (RE) in
     the pattern matches the input line without its trailing newline.  An empty expression
     matches every line.  Each input line that matches at least one of the patterns is written
     to the standard output.

     grep is used for simple patterns and basic regular expressions (BREs); egrep can handle
     extended regular expressions (EREs).  See re_format(7) for more information on regular
     expressions.  fgrep is quicker than both grep and egrep, but can only handle fixed patterns
     (i.e. it does not interpret regular expressions).  Patterns may consist of one or more
     lines, allowing any of the pattern lines to match a portion of the input.

     zgrep, zegrep, and zfgrep act like grep, egrep, and fgrep, respectively, but accept input
     files compressed with the compress(1) or gzip(1) compression utilities.
```

Grep example (cont'd)

```
LSE021353:StackExchange vojnovic$ grep 'Scala' * | more
Posts.xml:  <row Id="6947" PostTypeId="2" ParentId="6817" CreationDate="2008-08-09T23:26:30.570"
Score="2" Body="&lt;p&gt;As monty python would say &quot;and now for something completely differe
nt&quot; - you could try a language/environment that doesn't use threads, but processes and messa
ging (no shared state). One of the most mature ones is erlang (and this excellent and fun book: &
lt;a href=&quot;http://www.pragprog.com/titles/jaerlang/programming-erlang&quot; rel=&quot;nofoll
ow noreferrer&quot;&gt;http://www.pragprog.com/titles/jaerlang/programming-erlang&lt;/a&gt;). May
 not be exactly relevant to your circumstances, but you can still learn a lot of ideas that you m
ay be able to apply in other tools.&lt;/p&gt;&#xA;&#xA;&lt;p&gt;For other environments: &lt;/p&gt
;&#xA;&#xA;&lt;p&gt;.Net has F# (to learn functional programming).&#xA;JVM has Scala (which has a
ctors, very much like Erlang, and is functional hybrid language). Also there is the &quot;fork jo
in&quot; framework from Doug Lea for Java which does a lot of the hard work for you.&lt;/p&gt;&#x
A;" OwnerUserId="699" LastEditorUserId="116" LastEditorDisplayName="Mark Harrison" LastEditDate="
2008-08-09T23:56:46.657" LastActivityDate="2008-08-09T23:56:46.657" CommentCount="0" />
Posts.xml:  <row Id="8987" PostTypeId="1" AcceptedAnswerId="8993" CreationDate="2008-08-12T15:50:
49.460" Score="25" ViewCount="13795" Body="&lt;p&gt;Scalar-valued functions can be called from .N
ET as follows:&lt;/p&gt;&#xA;&#xA;&lt;pre&gt;&lt;code&gt;SqlCommand cmd = new SqlCommand(&quot;te
stFunction&quot;, sqlConn); //testFunction is scalar&#xA;cmd.CommandType = CommandType.StoredProc
edure;  &#xA;cmd.Parameters.Add(&quot;retVal&quot;, SqlDbType.Int);&#xA;cmd.Parameters[&quot;retV
```

Grep example (cont'd)

```
[LSE021353:StackExchange vojnovic$ grep 'Scala\|Python' * | more                                                ]
Posts.xml:  <row Id="307" PostTypeId="2" ParentId="260" CreationDate="2008-08-02T01:49:46.220" Sc
ore="32" Body="&lt;p&gt;&lt;a href=&quot;http://www.codeproject.com/Articles/8656/C-Script-The-Mi
ssing-Puzzle-Piece&quot; rel=&quot;noreferrer&quot;&gt;Oleg Shilo's C# Script solution (at The Co
de Project&lt;/a&gt;) really is a great introduction to providing script abilities in your applic
ation.&lt;/p&gt;&#xA;&#xA;&lt;p&gt;A different approach would be to consider a language that is s
pecifically built for scripting, such as &lt;a href=&quot;http://en.wikipedia.org/wiki/IronRuby&q
uot; rel=&quot;noreferrer&quot;&gt;IronRuby&lt;/a&gt;, &lt;a href=&quot;http://en.wikipedia.org/w
iki/IronPython&quot; rel=&quot;noreferrer&quot;&gt;IronPython&lt;/a&gt;, or &lt;a href=&quot;http
://en.wikipedia.org/wiki/Lua_%28programming_language%29&quot; rel=&quot;noreferrer&quot;&gt;Lua&l
t;/a&gt;.&lt;/p&gt;&#xA;&#xA;&lt;p&gt;IronPython and IronRuby are both available today.&lt;/p&gt;
&#xA;&#xA;&lt;p&gt;For a guide to embedding IronPython read&#xA;&lt;a href=&quot;http://blogs.msd
n.com/b/jmstall/archive/2005/09/01/howto-embed-ironpython.aspx&quot; rel=&quot;noreferrer&quot; t
itle=&quot;How to embed IronPython script support in your existing app in 10 easy steps&quot;&gt;
How to embed IronPython script support in your existing app in 10 easy steps&lt;/a&gt;.&lt;/p&gt;
&#xA;&#xA;&lt;p&gt;Lua is a scripting language commonly used in games. There is a Lua compiler fo
r .NET, available from CodePlex -- &lt;a href=&quot;http://www.codeplex.com/Nua&quot; rel=&quot;n
oreferrer&quot; title=&quot;Nua is Lua for .net&quot;&gt;http://www.codeplex.com/Nua&lt;/a&gt;&lt
;/p&gt;&#xA;&#xA;&lt;p&gt;That codebase is a great read if you want to learn about building a com
piler in .NET.&lt;/p&gt;&#xA;&#xA;&lt;p&gt;A different angle altogether is to try &lt;a href=&quo
t;http://en.wikipedia.org/wiki/Windows_PowerShell&quot; rel=&quot;noreferrer&quot;&gt;PowerShell&
lt;/a&gt;. There are numerous examples of embedding PowerShell into an application -- here's a th
orough project on the topic: &#xA;&lt;a href=&quot;http://code.msdn.microsoft.com/PowerShellTunne
l/Wiki/View.aspx?title=PowerShellTunnel%20Reference&quot; rel=&quot;noreferrer&quot; title=&quot;
PowerShell Tunnel&quot;&gt;Powershell Tunnel&lt;/a&gt;&lt;/p&gt;&#xA;" OwnerUserId="49" LastEdito
rUserId="2385" LastEditDate="2012-08-16T21:47:03.843" LastActivityDate="2012-08-16T21:47:03.843"
CommentCount="3" />
Posts.xml:  <row Id="337" PostTypeId="1" AcceptedAnswerId="342" CreationDate="2008-08-02T03:35:55
.697" Score="52" ViewCount="6615" Body="&lt;p&gt;I am about to build a piece of a project that wi
```

MapReduce: some key features

- Automatic parallelization: programs written in map-reduce (functional) style are automatically parallelized

- Run-time system: manages the execution of tasks such as
  - Partitioning the input data
  - Scheduling the program's execution across a set of machines
  - Handling machine failures
  - Managing the inter-machine communication

- Easy to use: allows programers without any experience in parallel and distributed systems to easily utilize the resources of a large distributed computing system

- Expressive model: many real-world tasks are expressible in this model

Parallel computing implementation

- Implementation: different implementation for different system environments
  - Ex. single-node multi-processor system vs a multi-node distributed system
  - Our focus: cluster computing systems with multiple machines

- Map invocations: distributed across multiple machines by automatically partitioning the input data to a set of M splits
  - M: a user configurable parameter

- Reduce invocations: distributed by partitioning the intermediate key space to R pieces using a partitioning function
  - Ex, using hash partitioning: `hash(key) mod R`
  - R: a user configurable parameter

- Node types: master and worker nodes
  - Master assigns tasks to workers (M map and R reduce tasks)

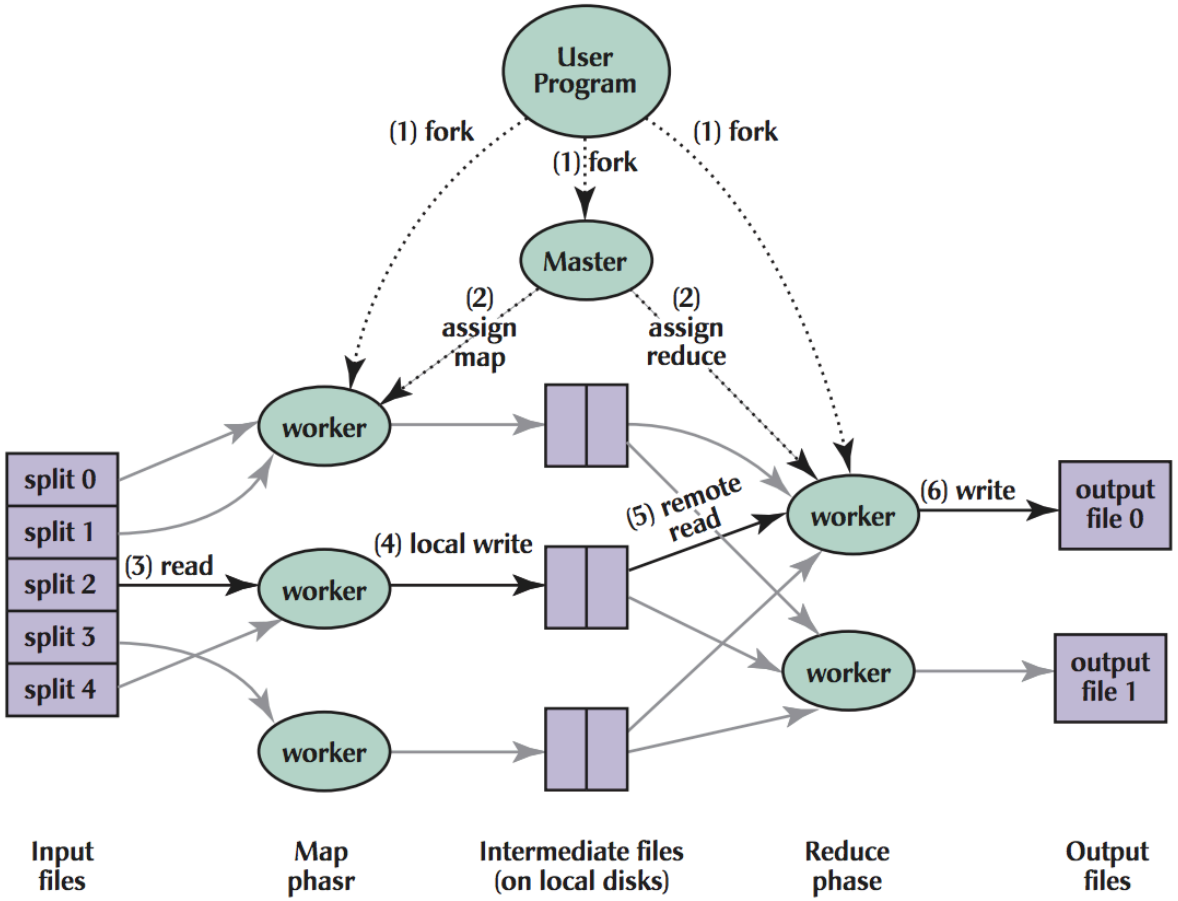
Workers assigned map tasks

- Read the content of the corresponding input split
- Parse key/value pairs in the input data and pass each pair to the user defined map function
- Buffer intermediate key/value pairs in memory
- Periodically write buffered pairs to local disk (accessible to the master node)

Workers assigned reduce tasks

- Read buffered data of the map workers
- When all intermediate key-value pairs are read, *sort* them by intermediate keys
  - Needed because different keys may be directed to the same reducer
- Iterate over the sorted intermediate data and for each intermediate key pass the key and the corresponding set of intermediate values to the user-defined reduce function
- Append the output of the reduce function to the output file of this reduce partition

MapReduce execution overview



- Source: original MapReduce paper

Data locality and parameter configuration

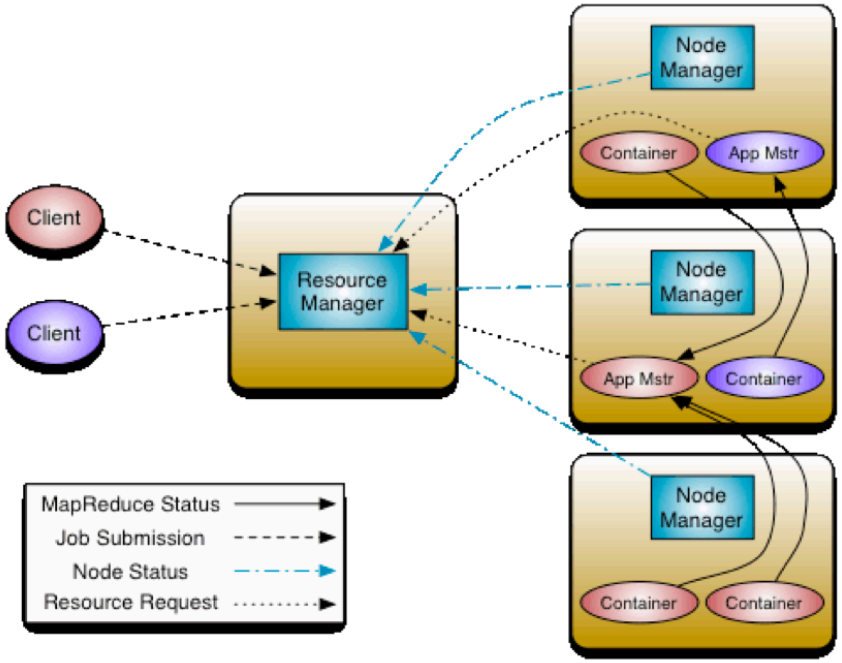
- Data locality: master node considers the location of input files to schedule each map task near to a machine that contains a replica of the input data

- M and R should be much larger than the number of worker machines
  - For load balancing and speed up recovery when a worker fails

- Master node must
  - Make O(M + R) scheduling decisions
  - Keep O(MR) state in memory

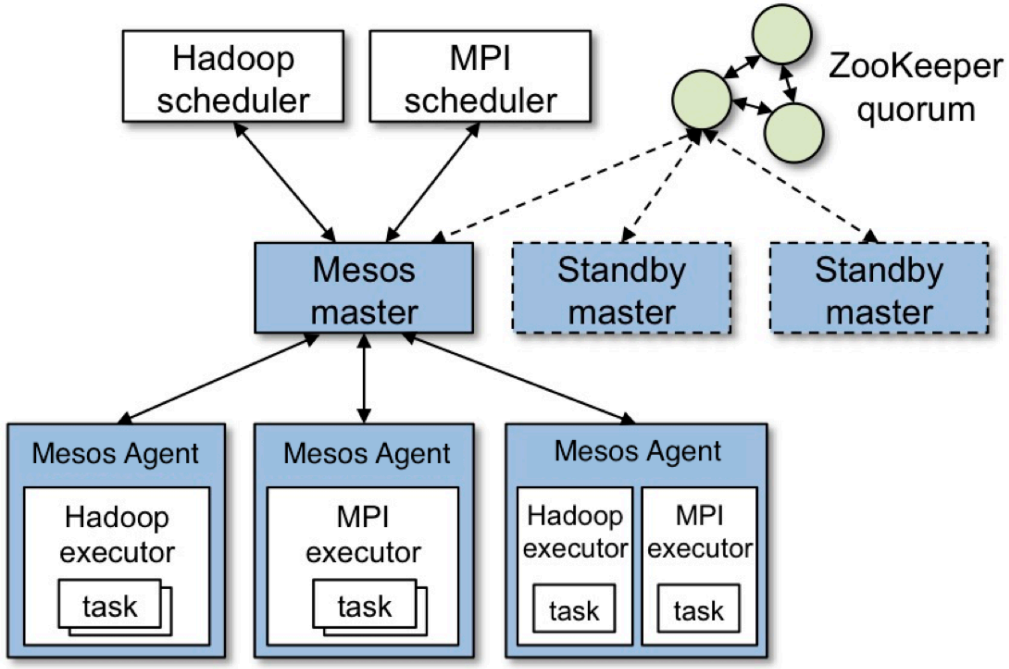
- A rule of thumb for setting M and R
  - M chosen such that each individual task processes roughly one file system block
    - So that the locality optimization is effective
  - R chosen to be a small multiple of the number of worker machines
  - Ex. M = 200,000 and R = 5,000 for 2,000 worker machines

Resource allocation: Hadoop YARN



- Source: https://hadoop.apache.org/docs/r2.7.3/hadoop-yarn/hadoop-yarn-site/YARN.html

Another resource allocator: Mesos



- Mesos: http://mesos.apache.org/documentation/latest/architecture/
- Mesos vs YARN: https://www.slideshare.net/mKrishnaKumar1/mesos-vs-yarn-an-overview
- More on Mesos vs YARN: https://www.oreilly.com/ideas/a-tale-of-two-clusters-mesos-and-yarn

Need for data persistence

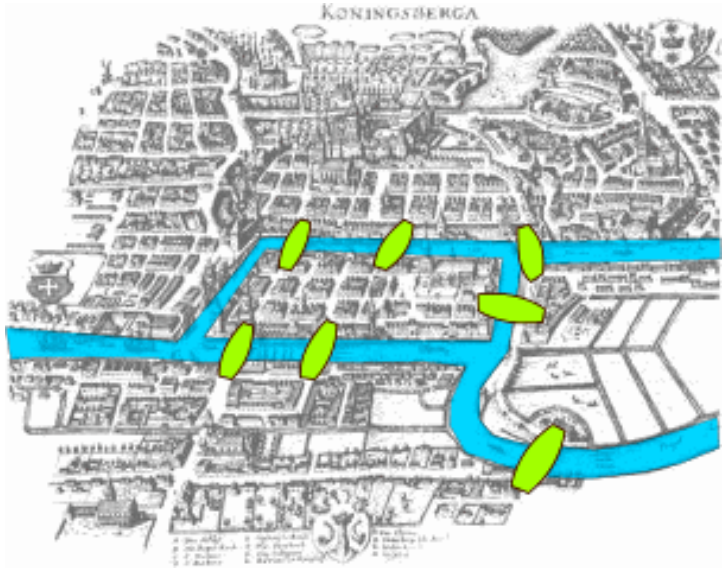
- MapReduce *does not* allow persisting data in memory
  - Requires writing to an external stable storage system
    - Ex. a distributed file system
  - Substantial overhead due to data replication, disk I/O and serialization

- Persisting data in memory speeds up iterative computations
  - Ex. PageRank computation

- Other computation models have been designed that persist data in memory
  - Ex. Pregel and Resilient Distributed Datasets (RDDs)

Pregel

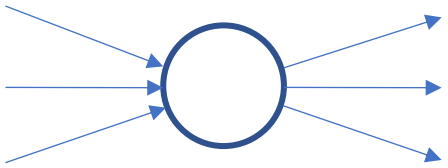
Origin of the name “Pregel”



- A river in Germany
- The name honors Leonhard Euler
- The seven Bridges of Königsberg: devise a walk through the city that would cross each of those bridges once and only once
  - Euler has shown that the problem has no solution (article)

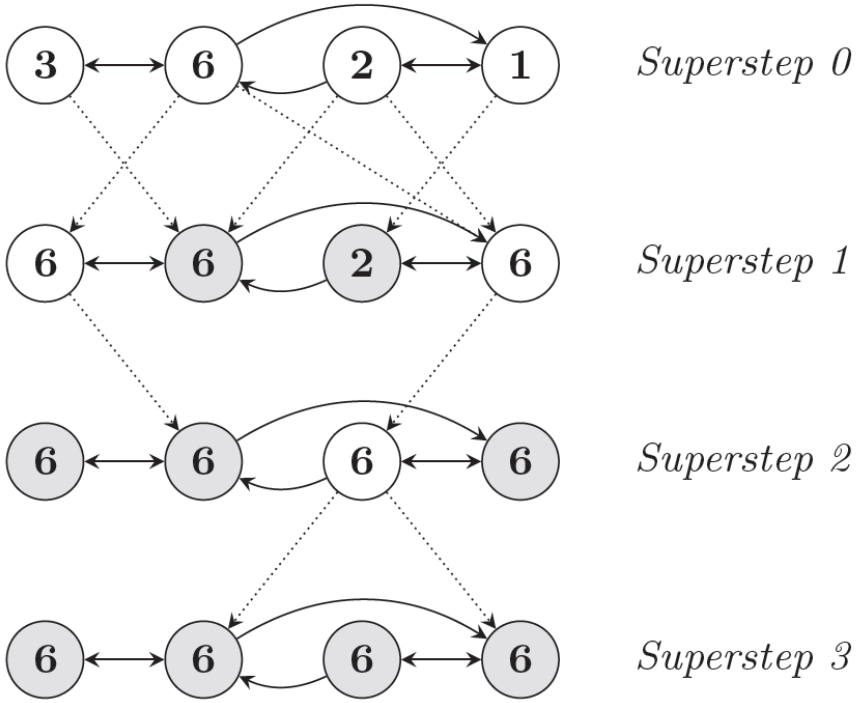
Pregel

- Pregel: a computation model for iterative graph processing
  - Developed by Google, first described in public in a SIGMOD 2010 paper
  - Apache cousin: Apache Giraph http://giraph.apache.org/
- Iterative computations: programs expressed as a sequence of iterations
- Vertex-centric model: in each iteration, a vertex can
  - Receive messages sent in previous iteration
  - Send messages to other vertices
  - Modify its own state and that of its out-going edges or mutate graph topology



Vertex v

Simple example: max value computation



Example use cases

- Shortest path computation
- Clustering algorithms
- PageRank type iterative algorithms
- Triangle counting
- Connected components

PageRank example

- PageRank: an algorithm for assigning numerical *relevance weights* to nodes of a graph with directed edges

- Originally proposed by Sergey Brin and Larry Page for hyperlinked sets of documents

  - Sergey Brin and Larry Page (1998). The anatomy of a large-scale hypertextual Web search engine. Computer Networks and ISDN Systems. 30 (1–7): 107–117

    http://infolab.stanford.edu/pub/papers/google.pdf

- Each node is assigned a relevance weight that reflects the relevance weights of the nodes that link to this node

PageRank example (cont'd)

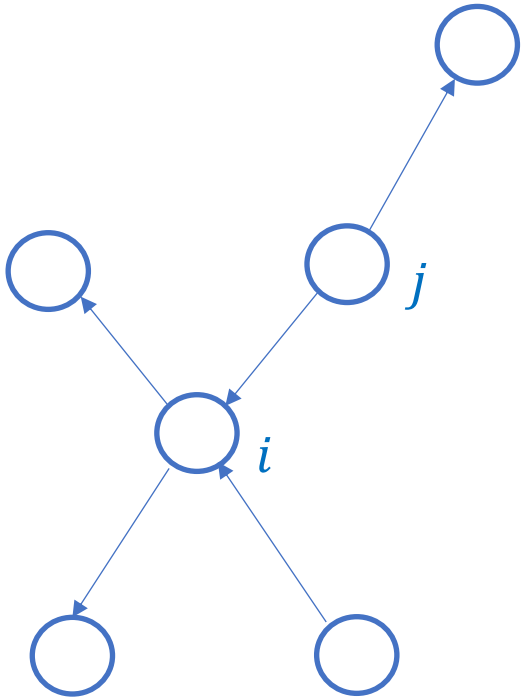
- Suppose given is a graph $G = (V, E)$ where V is the set of *nodes* and E is the set of *directed edges*
  - $(j, i) \in E$ is an edge from node j to node i
  - d_j := number of outgoing edges from node j

- PageRank *relevance weight* π_i of node i is defined by

$$\pi_i = (1 - \alpha) \sum_{j \in V : (j,i) \in E} \frac{1}{d_j} \pi_j + \alpha \frac{1}{|V|}$$

  where $\alpha \in (0,1]$ is a parameter

- $\pi = (\pi_1, \ldots, \pi_{|V|})^\top$ is the stationary distribution of a Markov chain (*random walk* on the graph following edges with *random teleportation* with probability α



PageRank example (cont'd)

- Power iteration method: assume arbitrary initial vector $\pi(0)$ and define

$$\pi_i(t+1) = (1-\alpha)\sum_{j \in V:(j,i)\in E} \frac{1}{d_j}\pi_j(t) + \alpha\frac{1}{|V|} \qquad \text{for } t = 0,1,\ldots$$

- Common initialization choice $\pi_i(0) = 1/|V|$ for $i \in V$

- In matrix notation

$$\pi^\top(t+1) = \pi^\top(t)P \qquad \text{for } t = 0,1,\ldots$$

  where $P = (p_{i,j})$ with $p_{i,j} = (1-\alpha)\frac{1}{d_i} + \alpha\frac{1}{|V|}$

- Guaranteed to converge to π for any graph G and $\alpha \in (0,1]$
  - This is because P is an irreducible transition matrix of a Markov chain
  - Matrix P irreducible means that all states communicate, i.e. for every node pair (i,j), the element in row i and column j of P^t is strictly positive for some $t \geq 1$

Pregel (cont'd)

- Expressive model: flexible model allowing to express a broad set of algorithms
  - Ex. iterative computation of a PageRank vector
- Design goals: efficient, scalable, and fault-tolerant computation in clusters of many commodity machines
- Synchronicity of computation: makes reasoning about programs easy
- Abstract API: hides the underlying distributed computation aspects

Pregel computation model

- Inspired by Bulk Synchronous Parallel (BSP) model (Leslie Valiant)

- Pregel computation elements:
  - Input graph initialization
  - Sequence of *supersteps* separated by *global synchronization points* until the algorithm terminates
  - Finishing output

- For each superstep, vertices compute in parallel, each executing the same user-defined function that expresses the logic of given algorithm

- Each vertex can modify its state or state of its outgoing edges, receive messages sent to it in the previous superstep, send messages to other vertices (to be received in the next superstep), or even mutate the topology of the graph

- Edges are second-class citizens: they don't have associated computation

BSP model

- Leslie Gabriel Valiant
- A British computer scientist and a computational theorist
- Professor of Computer Science and Applied Mathematics at Harvard University
- Turing Award Winner in 2010
- L. G. Valiant, A Bridging Model for Parallel Computation, Comm. of the ACM, Vol 3, No 8, August 1990

Pregel computation model (cont'd)

- Termination: algorithm termination is based on every vertex voting to halt
  - Superstep 0: every vertex is in active state
  - All active vertices participate in computation in every superstep
  - A vertex deactivates itself by voting to halt
    - Vertex has no further work to do, unless triggered externally
  - Pregel will not execute a deactivated vertex in subsequent supersteps unless it receives a message from this vertex

- Termination criteria: the algorithm terminates when all vertices are simultaneously inactive and there are no messages to transmit

- Output of the algorithm: set of values explicitly output by vertices
  - The output type is often isomorphic to the input type
  - Not necessarily always the case because vertices and/or edges can be added and removed during the computation

Message passing model

- Message passing model: distributed processing with nodes exchanging messages
  - No remote reads and other ways of emulating a shared memory
  - Shared memory: another computation model
    - A memory that can be simultaneously accessed by multiple processes with an intent to provide communication among them

- Sufficiently expressive

- Target applications that don't need remote reads

- Performance gains: reading a value from a remote machine incurs high latency

Pregel API

- Original implementation in C++
- Writing a Pregel program involves subclassing the predefined Vertex class
- The template arguments of Vertex class define three value types associated with vertices, edges and messages

- User overrides the virtual `Compute()` method
  - Executed by each active vertex in every superstep
  - Predefined Vertex methods allow `Compute()` to query information about the current vertex and its edges and to send messages to other vertices
  - `Compute()` can read the value associated with its vertex via `GetValue()` or modify it via `MutableValue()`
  - `Compute()` can inspect and modify value of out-edges using methods supplied by the out-edge iterator

- Persisted state: the values associated with a vertex and its edges
  - Persisted across supersteps

PageRank code example

Vertex class:

```
class PageRankVertex: public Vertex<double, void, double> {

    public:
        virtual void Compute(MessageIterator* msgs) {
            if (superstep() >= 1) {
                double sum = 0
                    for(; !msgs->Done(); msgs->Next()) sum += msgs->Value();
                    *MutableValue() = 0.15/NumVertices() + 0.85 * sum;

            if (superstep() < 30) {
                const int64 n = GetOutEdgeIterator().size();
                SendMessageToAllNeighbors(GetValue()/n);
            } else {
                VoteToHalt();
            }
        }
}
```

Note: implementation of $\pi_i(t+1) = (1-\alpha)\sum_{j \in V:(j,i) \in E} \frac{1}{d_j}\pi_j(t) + \alpha\frac{1}{|V|}$, for $t = 0,1,\ldots$

Resilient Distributed Datasets (RDDs)

Resilient Distributed Datasets

- Resilient distributed datasets: a distributed memory abstraction allowing programers to perform in-memory computations on a large clusters of machines in a fault-tolerant manner
  - Described in an NSDI 2012 paper

- Design motivated by the need for iterative algorithms and interactive data mining
  - Keeping data in memory improves performance

- Fault tolerance
  - RDDs provide a restricted form of shared memory
  - Based on coarse-grained transformations (ex. map, filter, join) rather than fine-grained updates to a shared state
  - Expressive enough for a wide class of computations, including specialized programing models for iterative jobs

Design rationale

- Main design challenge: defining a programing interface that can provide fault tolerance efficiently
- Solution based on coarse-grained transformations that apply the same operation repeatedly to many data items
- Efficient fault-tolerance by logging the transformations used to build a dataset (referred to as linage or provenance) rather than the actual data
- If a partition of an RDD is lost, the RDD has the information how it was derived from other RDDs to recompute the lost partition
- Suitable for applications that apply the same operation on multiple data items

RDD properties

- RDD: read-only, partitioned collection of elements

- RDD can only be created through deterministic operations on either
  - Data in stable storage (ex. local file system or HDFS)
  - Other RDDs

- These operations are called transformations: ex, map, filter, join

- Lazy computation: RDDs do always not need to be materialized

- User control over RDD's persistence and partitioning

Advantages of the RDD model

- RDDs can only be created through coarse-grained transformations
  - Restricts RDDs to applications that perform bulk writes
  - Allows for more efficient fault tolerance
  - No overhead of checkpointing
  - Only the lost partitions of an RDD need to be recomputed upon failure, which can be done in parallel on different nodes, without having to roll back the whole program
- Unlike to distributed shared memory systems that allow reads and writes to each memory location
- Mitigation of slow nodes (stragglers) by running backup copies of slow tasks as in MapReduce
- Allows to schedule tasks based on data locality
- Degrade gracefully when there is no enough memory
  - Data stored on disk if not fitting in RAM

Nonuse cases

- Applications that make asynchronous fine-grained updates to shared state
  - Ex. storage systems for web applications or an incremental web crawler

- For these applications, it is more efficient to use systems that perform traditional update logging and data check-pointing
  - Ex. RAMCloud, Percolator

- RDDs are best suited for batch applications that apply the same operation to all elements of a dataset

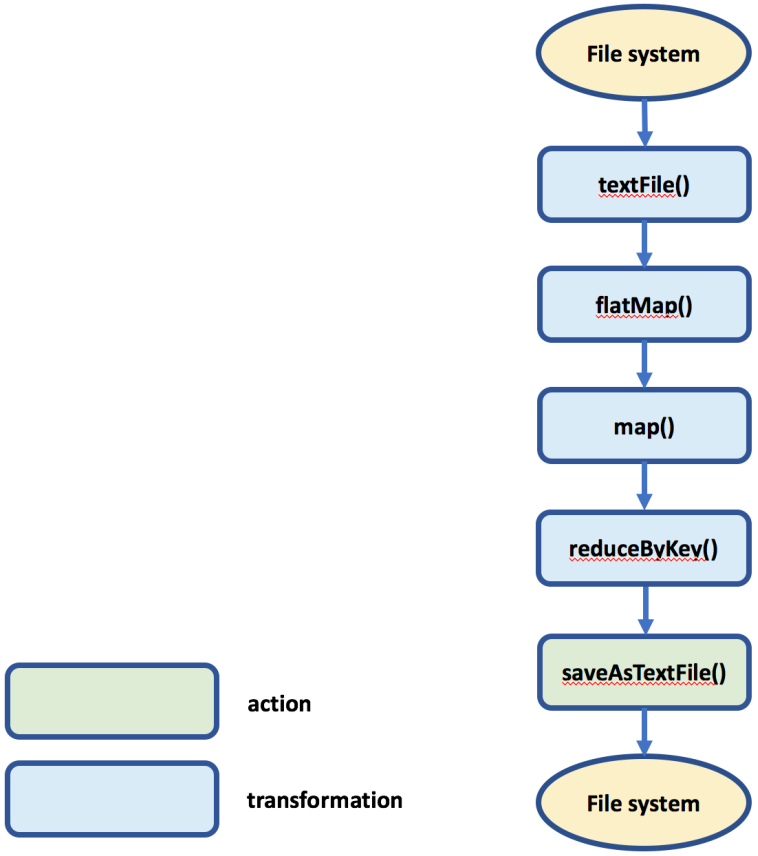
- Notes:
  - Checkpointing: saving a snapshot of the application's state
  - RAMCloud: see this morning paper, research paper
  - Percolator: research paper (Google's incremental web indexing)

PySpark word count example

```
text_file = sc.textFile("hdfs://...")
counts = text_file.flatMap(lambda line: line.split(" ")) \
             .map(lambda word: (word, 1)) \
             .reduceByKey(lambda a, b: a + b)
counts.saveAsTextFile("hdfs://...")
```

- Source: http://spark.apache.org/examples.html

DAG (lineage graph) for word count example



- DAG: Directed Acyclic Graph

RDD API

- RDDs are exposed through a language-integrated API
- Like some other systems, ex. DrayadLINQ and FlumeJava
- Integrated API for Scala, Java, Python, R, SQL APIs

Creating RDDs

- Two ways:
    - Parallelizing an existing collection
    - Referencing a dataset in an external storage system

RDD operations

- Transformations: operations on RDDs that return a new RDD
  - Many transformations are element-wise
  - Not true for all transformations

- Lazy computation: transformed RDDs are computed lazily
  - Only when they are used in an action

- Actions: operations that return a result to the driver program or write it to storage and kick off a computation

- Think of an RDD as a set of instructions how to compute the data by using transformations

Examples of transformations

Function name	Purpose	Example	Result
`map()`	Apply a function to each element, return an RDD as output	`d.map(lambda x: x - 1)`	`{0, 1, 2, 2}`
`flatMap()`	Return a new RDD by first applying a function to all element, and then flattening the results	`d.flatMap(lambda x: range(x))`	`{0, 0, 1, 0, 1, 2, 0, 1, 2}`
`filter()`	Return and RDD consisting of only elements that satisfy the condition passed to filter()	`d.filter(lambda x: x !=1)`	`{2, 3, 3}`
`distinct()`	Remove duplicates	`d.distinct()`	`{1, 2, 3}`
`sample(withReplacement, fraction, [seed])`	Sample with or without replacement	`d.sample(false, 0.5)`	Nondeterministic

- Input data: `d = sc.parallelize([1, 2, 3, 3])`

Common use flatMap: tokenization

- **map**:

```
lines = sc.parallelize(["hello world", "hi"])
words = lines.map(lambda line: line.split(" "))
words.collect() # returns {["hello", "world"], "hi"}
```

- **flatMap**:

```
lines = sc.parallelize(["hello world", "hi"])
words = lines.flatMap(lambda line: line.split(" "))
words.collect() # returns {"hello", "world", "hi"}
```

Actions

Function name	Purpose	Example	Result
`collect()`	Return all elements	`d.collect()`	`{1,2,3,3}`
`count()`	Number of elements	`d.count()`	`4`
`countByValue()`	Number of times each element occurs	`d.countByValue()`	`(1,1), (2,1), (3,2)`
`take(num)`	Return num elements	`d.take(2)`	`{1,2}`
`top(num)`	Return the top num elements	`d.top(2)`	`{3,3}`
`takeOrdered(num)(ordering)`	Return num elements based on provided ordering	`d.takeOrdered(2)(myOrdering)`	`{3, 3}`
`takeSample(withReplication, num, [seed])`	Return num elements at random	`d.takeSample(false, 1)`	Nondeterministic
`reduce(func)`	Combine the elements	`d.reduce(lambda x, y: x + y)`	9
`fold(zero)(func)`	Same as reduce() but with the provided zero value	`d.fold(0)(lambda x, y: x + y)`	`0`
`aggregate(zeroValue)(seqOp, combOp)`	Aggregate first for each partition and then results	`d.aggregate((0,0))(lambda x, y: (x[0] + y, x[1] + 1), lambda x, y: (x[0] + y[0], x[1] + y[1])`	`(9, 4)`
`foreach(func)`	Apply the provided function to each element	`d.foreach(func)`	Nothing

Spark I/O: textFile

- **`SparkContext.textFile`**`(path: String, minPartitions: Int = defaultMinPartitions)`

  - Reads text data from a file from a remote HDFS, a local file system (available on all nodes), or any Hadoop-supported file system URI (ex. sources in HBase or S3) at path
  - Automatically distributes the data across a Spark cluster as an RDD of strings

- Returns **`HadoopRDD`**: when using a textFile to read an HDFS folder with multiple files inside, the number of partitions are equal to the number of HDFS blocks

- path
  - `file://`
  - `hdfs://`
  - `s3://`
  - `gs://` (Google Cloud Storage connector)

https://cloud.google.com/dataproc/docs/concepts/connectors/cloud-storage

Spark I/O: saveAs*

- **saveAsNewAPIHadoopDataset**(self, conf, keyConverter=None, valueConverter=None)
  - Output a Python RDD of key-value pairs to any Hadoop file system, using the new Hadoop OutputFormat API (mapreduce package)

- **saveAsNewAPIHadoopFile**(self, path, outputFormatClass, keyClass=None, valueClass=None, keyConverter=None, valueConverter=None, conf=None)
  - Output a Python RDD of key-value pairs to any Hadoop file system, using the new Hadoop OutputFormat API (mapreduce package)

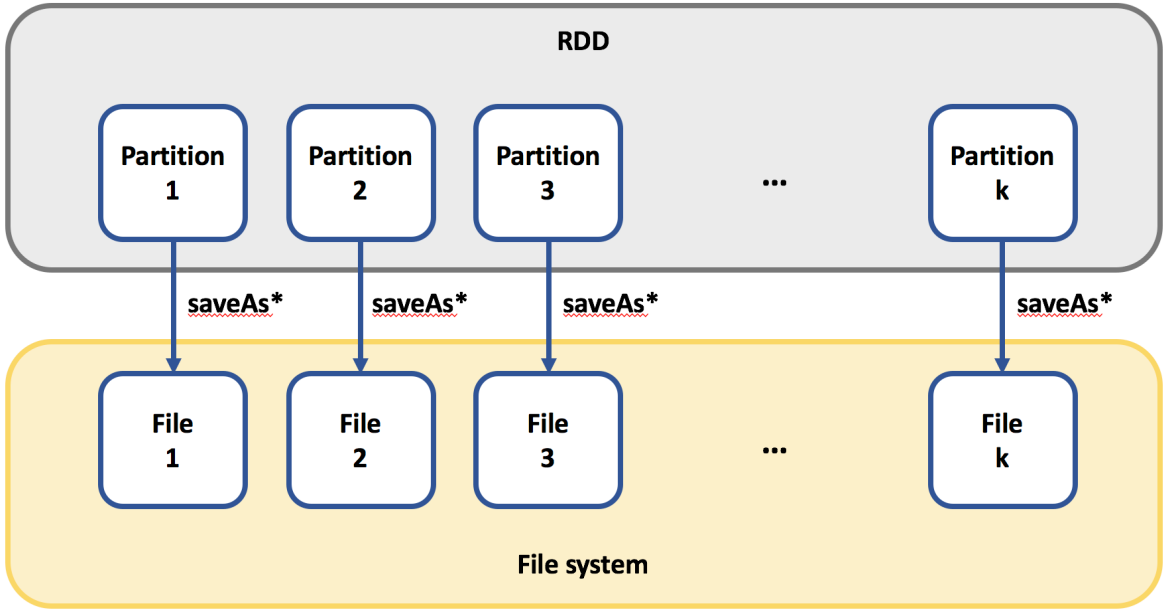
- **saveAsSequenceFile**(self, path, compressionCodecClass=None)
  - Output a Python RDD of key-value pairs to any Hadoop file system, using org.apache.hadoop.io.Writable types that we convert from the RDD's key and value types

- **saveAsPickleFile**(self, path, batchSize=10)
  - Save this RDD as a SequenceFile of serialized objects

- **saveAsTextFile**(self, path)
  - Save this RDD as a text file, using string representations of elements

- Note: saveAs* operations are actions

Spark I/O: saveAs* (cont'd)



- RDD is a set of partitions that make it
- Saving an RDD to a file saves the content of each partition to a file (one file per partition)
- To reduce the number of files, repartition the RDD to the target number of files
- More information here

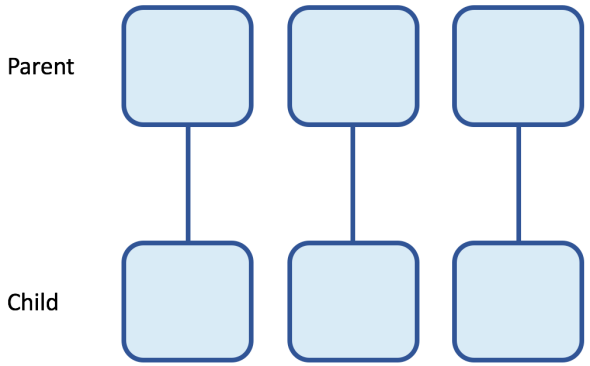
Narrow vs wide dependencies

- Narrow dependencies: each partition of a child RDD has simple, finite dependencies on partitions with the parent RDD
  - Narrow dependencies can be determined at the design time, irrespective of the values of the records in parent partitions
  - Each child partition depends only on a unique subset of parent partitions
    - No other child partitions depend on them
  - Spark automatically performs a pipelining operation on narrow dependencies
    - Performed in memory
  - Examples: map, filter, MapPartions, and flatMap

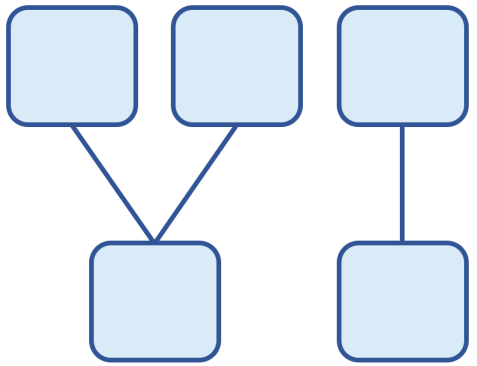
- Wide dependencies: input partitions contribute to multiple output partitions
  - Referred to as a shuffle when partitions are exchanged across the cluster
  - With a shuffle the results are written to a disk
  - Examples: reduceByKey, groupByKey, Join and any other transformation that calls the repartition function

Narrow dependencies

- Each child partition depends on a known subset of parent partitions

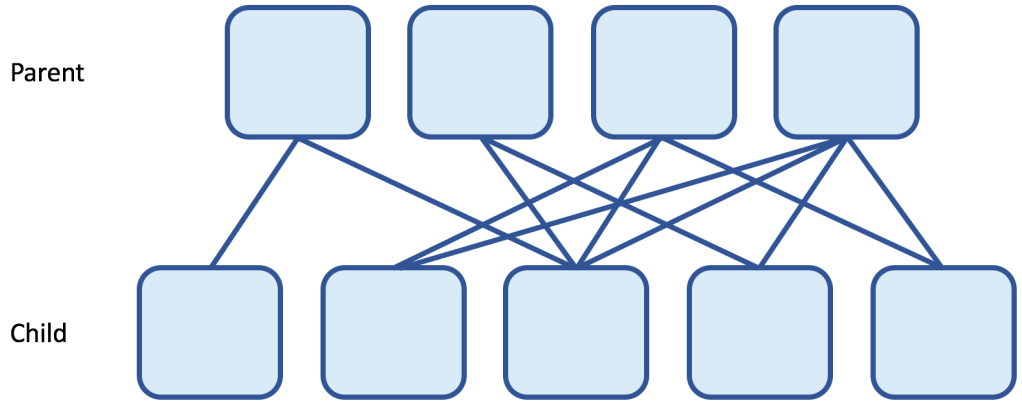


Ex. map, filter, mapPartitions, flatMap



Ex. coalesce

Wide dependencies



- Wide dependencies: child partitions depend on an arbitrary set of parent partitions
  - Not known fully before the data is evaluated
  - The dependency graph for any operations that cause a shuffle such as groupByKey, reduceByKey, sort and sortByKey follow this pattern

To probe further:

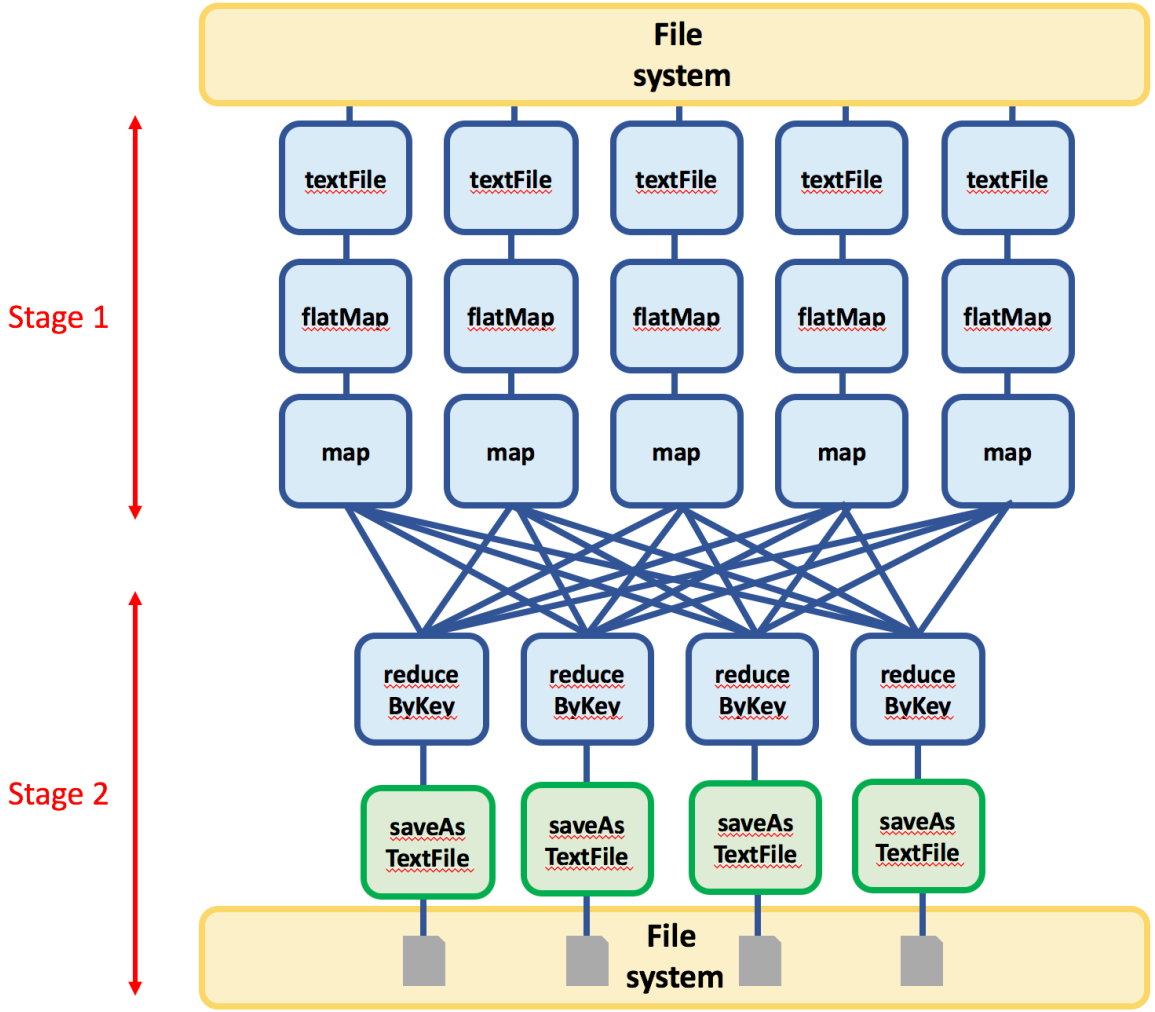
Wide and narrow dependencies: see here and here

See also Cloudera's doc

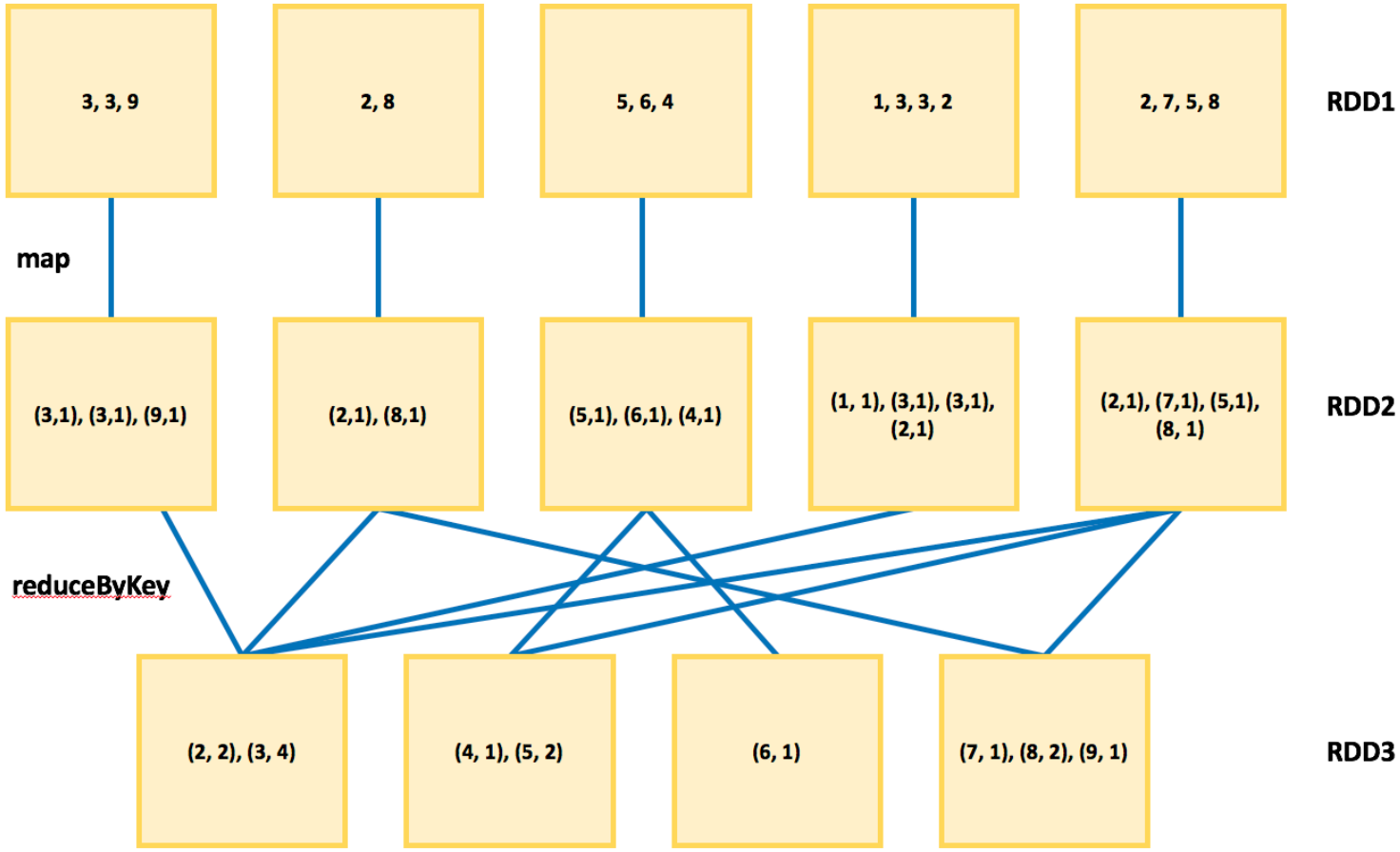
Shuffles

- Stages: evaluation of RDDs performed in one or more stages
- Stage boundaries: at each stage boundary, data is written to a disk by tasks in the parent stages and then fetched over the network by tasks in the child stage
- Shuffles are costly: incur high disk and network I/O
  - Stage boundaries can be expensive and should be avoided when possible
- Number of data partitions in a parent stage may be different than the number of partitions in a child stage
- Transformations that can trigger a stage boundary typically accept a `numPartitions` argument, which specifies how many partitions to split the data in the child stage

Word count example revisited



Word count example revisited: RDDs



RDDs cont’d

Pair RDD: working with key-value pairs

- Pair RDD: a dataset of elements, with each element being a pair of data items

```
(key, value)
```

- Used for storing and manipulating key-value datasets

- To probe more:

  https://spark.apache.org/docs/2.2.0/rdd-programming-guide.html#working-with-key-value-pair

Transformations on pair RDDs

Function name	Purpose	Example	Result
`reduceByKey(func)`	Combine values with the same key	`d.reduceByKey(lambda x, y: x + y)`	`{(1,2), (3,10)}`
`groupByKey()`	Group values with the same key	`d.groupByKey()`	`{(1, [2]), (3, [4, 6])}`
`combineByKey(createCombiner, mergeValue, mergeCombiner, partitioner)`	Combine values with the same key using a different result type		
`mapValues(func)`	Apply a function to each value without changing the key	`d.mapValues(lambda x: x+1)`	`{(1,3), (3,5), (3,7)}`
`flatMapValues(func)`	Apply a function that returns an iterator to each value, and for each element returned produce a key\value entry with the old key	`d.flatMapValues(lambda x: range(x,6))`	`{(1,2), (1,3), (1,4), (1,5), (3,4), (3,5)}`
`keys()`	Return keys	`d.keys()`	`{1,3,3}`
`values()`	Return values	`d.values()`	`{2,4,6}`
`sortByKey()`	Return sorted by the key	`d.sortByKey()`	`{(1,2), (3,4), (3,6)}`

- Input data: `d = {(1,2), (3,4), (3,6)}`

Transformations on pairs of pair RDDs

Function name	Purpose	Example	Result
`subtractByKey`	Remove elements with a key present in the other RDD	`d.subtractByKey(e)`	`{(1,2)}`
`join`	Perform an inner join between two RDDs	`d.join(e)`	`{(3, (4,9)), (3, (6,9))}`
`rightOuterJoin`	Perform a join between two RDDs where the key must be present in the other RDD	`d.rightOuterJoin(e)`	`{(3, (Some(4),9)), (3, (Some(6),9))}`
`leftOuterJoin`	Perform a join between two RDDs where the key must be present in the first RDD	`d.leftOuterJoin(e)`	`{(1, (2, None)), (3, (4, Some(9))), (3, (6, Some(9)))}`
`cogroup`	Group data from both RDDs sharing the same key	`d.cogroup(e)`	`{(1, ([2], None)), (3, ([4,6], [9]))}`

- Input data: `d = {(1,2), (3,4), (3,6)}` and `e = {(3,9)}`

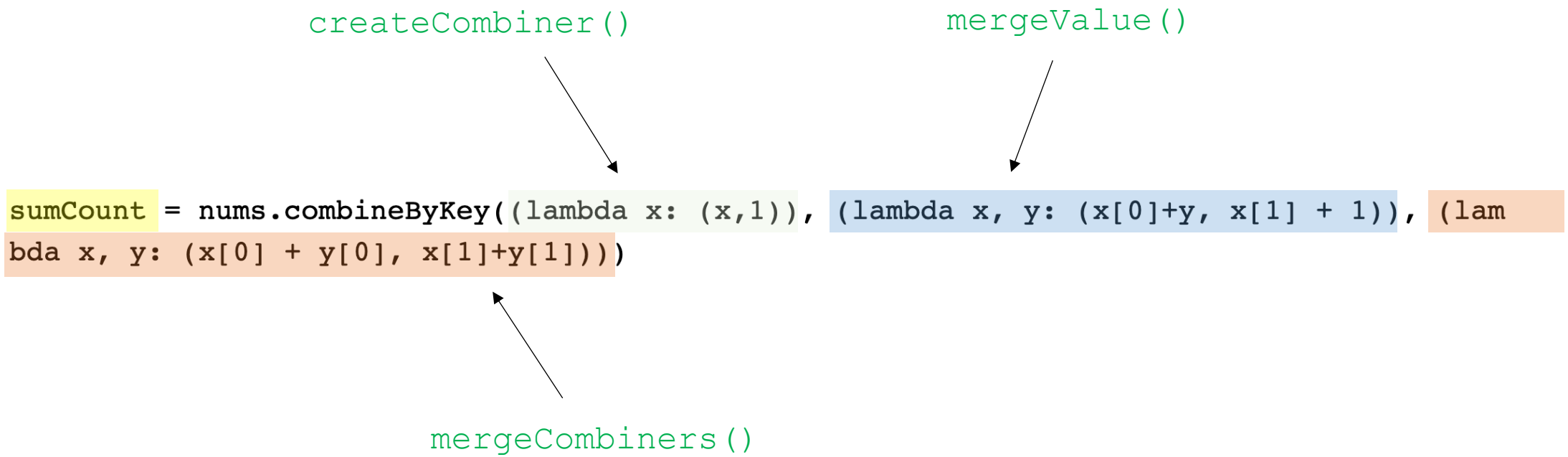
combineByKey

- `combineByKey()`: the most general per-key aggregation function
  - Most of the other per-key combiners are implemented by using it

- While `combineByKey()` goes through elements in a partition it performs:
  - If it is a new element, it uses `createCombiner()` to create initial value for the accumulator on that key
  - If it is a value seen before while processing the given partition, it uses the provided function `mergeValue()`

- When merging the results of different partitions, if two or more partitions have an accumulator of the same key, they may be merged using the provided function `mergeCombiners()`

combineByKey: example




```
r = sumCount.map(lambda key, xy: (key, xy[0]/xy[1])).collectAsMap()
```

Partitioning

- Partitioning: grouping of pair RDD elements based on a function of each key
  - Ensures that a set of keys is assigned to some worker node
  - Users cannot control to which worker node a set of keys is assigned
    - Handled by the system, and there may be worker node failures
- Defined by the Spark Partitioner object: an interface with two methods
  - `numPartitions`: defines the number of partitions in the RDD after partitioning
  - `getPartition`: defines the mapping from the key to the integer index of the partition to which records with that key should be sent

Spark partitioner object: HashPartitioner

- The default partitioner for pair RDD operations that determines the index of the child partition based on the hash value of the key

- Requires a parameter that determines the number of partitions in the output RDD (number of bins used by the hash function)

  - If unspecified, Spark uses the value of `spark.default.parallelism` variable in `SparkConf` to determine the number of partitions

  - If the default parallelism value is unset, Spark defaults to the largest number of partitions that the RDD has had in its lineage

Spark partitioner object: RangePartitioner

- Assigns records whose keys are in the same range to the same partition
- Sorting the records within each partition ensures that the entire RDD is sorted
- First need to determine the range boundaries for each partition by sampling, optimizing for an equal number of records per partition
- Then shuffles each record to the partition whose range includes the key
- Requires not only the number of partitions, but also the actual RDD to sample
- RDD must be a tuple and the keys must have an ordering defined
- Note: custom partitioning can be defined by the user
  - User needs to implement methods: numPartitions, getPartition, equals and hashcode

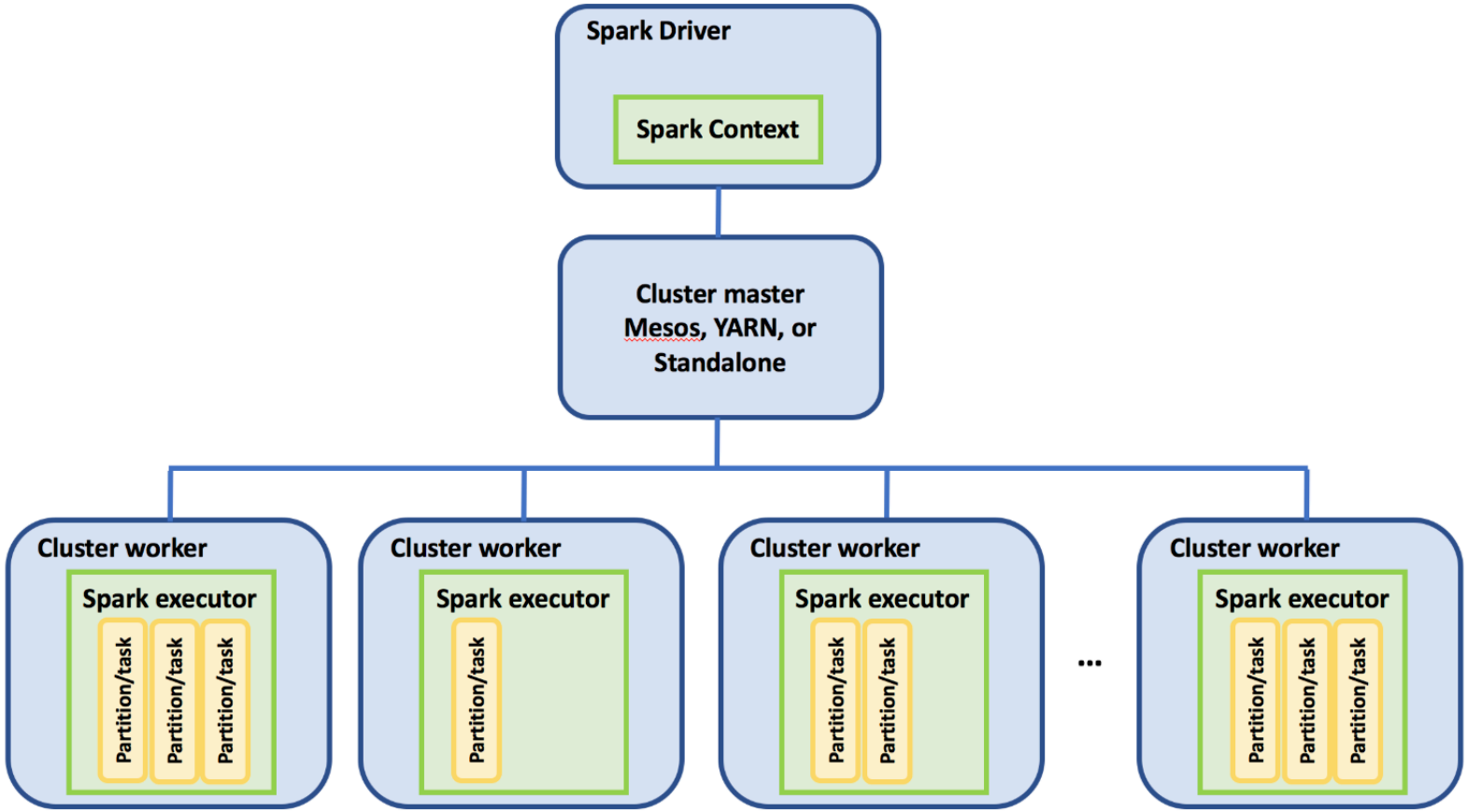
Operations that may benefit from partitioning

- Operations that involve shuffling data by key across the network
- Ex. `cogroup()`, `groupByKey()`, `reduceByKey()`, `combineByKey()`, `lookup()`, `join()`, `leftOuterJoin()`, `rightOuterJoin()`
- For operations on a single pre-partitioned RDD, all the values for each key may be computed locally on a single machine, requiring only the final, locally reduced value to be sent from each worker node back to the master
  - Ex. `reduceByKey()`

Persistence

- When an RDD needs to be used multiple times, the user can instruct to persist the data of given RDD
  - Nodes that compute the RDD store their partitions
  - Avoids recomputing the same RDD each time an action is called on this RDD
- There are several levels of persistence:
  - MEMORY_ONLY
  - MEMORY_ONLY_SER: store serialized representation
  - MEMORY_AND_DISK: spill to disk if there is too much data to fit in memory
  - MEMORY_AND_DISK_SER: store serialized representation in memory
  - DISK_ONLY

Spark runtime system architecture



- Cluster mode: master-slave architecture
  - One central coordinator and several distributed workers

Spark driver

- Spark driver is the central coordinator
  - Runs own Java process
  - User code creates a `SparkContext`, creates RDDs, and performs transformations and actions

- Responsibilities of a Spark driver:
  - Converting the user program into tasks
  - Coverts the logical graph of operations into a physical execution plan
  - Converts the execution graph into a set of stages each having multiple tasks

- Scheduling tasks on executors:
  - Schedules input tasks to executors accounting for data locality

Spark executors

- Spark executors: worker processes
  - Responsible for running individual tasks in each Spark job
  - Each executor run in its own Java process
  - Launched once at the beginning of a Spark application
  - Typically run for the entire lifetime of an application

- Responsibilities of a Spark executor:
  - Runs the tasks that make up the application and returns results to the driver
  - Provides in-memory storage for RDDs that are cached by user programs through the block manager service that lives within each executor

Spark application and cluster manageer

- Spark application: driver and executors together
  - Launched on a set of machines using an external service: cluster manager

- Cluster manager: a pluggable component that launches executors and in certain cases the driver

- Several different cluster managers:
  - Standalone cluster manager: Spark built-in cluster manager
  - Hadoop YARN
  - Apache Mesos

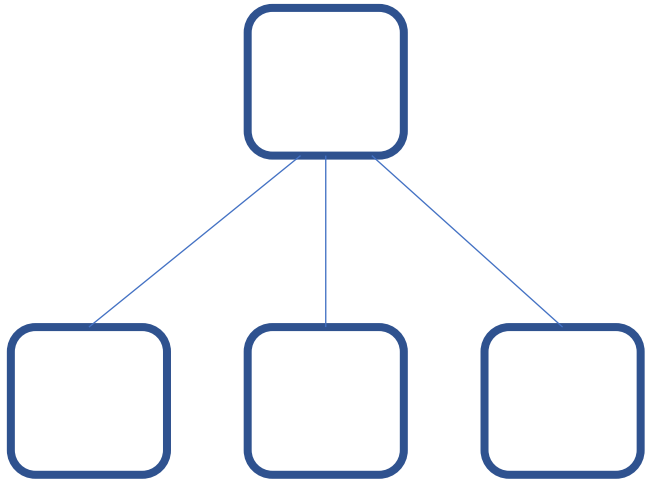
- Note: when Spark is run in local mode, the Spark driver runs with an executor in the same Java process

Launching a Spark program

- User submits an application using `spark-submit`
- `spark-submit` launches the driver program and invokes `main()` method specified by the user
- Driver program contacts cluster manager asking for resources to launch executors
- Cluster manager launches executors on behalf of the driver program
- Driver process runs through the user application, it sends work to executors in the form of tasks based on the RDD actions and transformations in the program
- Tasks are run on executor processes to compute and save results
- If the driver's `main()` method exists or it calls `SparkContext.stop()`, it will terminate the executors and release resources from the cluster manager

Seminar class 3: MapReduce and RDDs

- Run a MapReduce job in Hadoop
- Getting started with PySpark and RDDs
- Run a Spark Job using PySpark



https://github.com/lse-st446/lectures2021/blob/master/Week03/class/README.md

References

- MapReduce
  - Apache Hadoop docs: http://hadoop.apache.org/docs/r3.0.0/
  - Apache Hadoop docs: MapReduce Tutorial
  - Dean, J. and Ghemawat, S., Mapreduce: Simiplified Data Processing on Large Clusters, Comm. of the ACM, Vol 51, No 1, 2008; research paper OSDI 2004
  - Sort benchmark competition: http://sortbenchmark.org
- Pregel
  - Malewicz G. et al, Pregel: A System for Large-Scale Graph Processing, SIGMOD 2010

References (cont'd)

- Spark RDD programming guide
- Zaharia M. et al, Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing, NSDI 2012
- Chambers B. and Zaharia M., Spark: The Definitive Guide, Databricks, 2017
- Data Camp: PySpark RDD Basics Cheat Sheet
- Spark docs PySpark package
- Spark configuration
- PySpark interface / methods: see PySpark RDD source code, also docs

References (cont'd)

- Yu et al, DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing using a High-level Language, OSDI 2008
- Zhou et al, SCOPE: parallel databases meet MapReduce, The VLDB Journal 2012
- L. G. Valiant, A Bridging Model for Parallel Computation, Comm. of the ACM, Vol 3, No 8, Aug 1990

Books

- Karau, H., Konwinski, A., Wendell, P. and Zaharia, M., Learning Spark: Lightening-fast Data Analysis, O'Reilly, 2015
  - Chapter 3: Programming with RDDs
  - Chapter 4: Working with Key/Value Pairs
  - Chapter 7: Running on a Cluster

- Karau, H. and Warren, R., High Performance Spark: Best Practices for Scaling & Optimizing Apache Spark, O'Reilly, 2017
  - Chapter 2: How Spark Works
  - Chapter 5: Effective Transformations
  - Chapter 6: Working with Key/Value Pairs

- Drabas, T. and Lee D., Learning PySpark, Packt, 2016
  - Chapter 2: Resilient Distributed Datasets