

COMP5349 – Cloud Computing

Week 6: Distributed Execution

Dr. Ying Zhou
School of Information Technologies



Administrative

■ Assignment update

- ▶ Deadline is moved to mid day next week Thursday
- ▶ Wednesday is a public holiday.

Last Week

- Last week we cover Spark framework
- Spark is a data flow system
 - ▶ Data analytic workload is designed as data flowing along a sequence of operations
 - ▶ The original data structure and data flow operations closely follow the MapReduce design principles
 - RDD and transformations
- Difference to MapReduce framework
 - ▶ Spark is memory based
 - ▶ Spark has larger set of APIs
 - A number of map like operations: map, flatMap, mapToPair, filter, etc.
 - A number of reduce like operations: reduceByKey, groupByKey, aggregateByKey, join, etc....
 - ▶ Strong functional programming style

Outline

■ GFS and HDFS

- ▶ Overall Architecture
- ▶ Read/Write Operation
- ▶ HA Strategy
- ▶ Chunk Placement Strategy

■ MapReduce on YARN

■ Spark Execution

COMMONWEALTH OF AUSTRALIA

Copyright Regulations 1969

WARNING

This material has been reproduced and communicated to you by or on behalf of the **University of Sydney** pursuant to Part VB of the Copyright Act 1968 (the Act).

The material in this communication may be subject to copyright under the Act. Any further reproduction or communication of this material by you may be the subject of copyright protection under the Act.

Do not remove this notice

Distributed File System Basics

- A distributed file systems allow clients to access files on remote servers “transparently”.
- We are using at least two popular ones in SIT
 - ▶ Your home directory is mounted on all Linux servers using NFS
 - Physically located in somewhere, used to be soitlabhomes01.it.usyd.edu.au
 - Mounted on all Linux work station and servers: lab workstation, ucpu1.ug.it.usyd.edu.au, ucpu2.ug.it.usyd.edu.au, soit-ucpu-pro-1.ucc.usyd.edu.au, etc...
 - ▶ Your home directory is mapped as U drive in Windows using Samba
- Constrains
 - ▶ No sufficient mechanism in terms of reliability and availability
 - Replica, migration, etc
 - You may have encountered login hiccups like your U drive is not mapped when you login to one of the lab machine...

GFS Design Constraints

- Component failures are the norm
 - ▶ 1000s of components
 - ▶ Bugs, human errors, failures of memory, disk, connectors, networking, and power supplies
 - ▶ Monitoring, error detection, fault tolerance, automatic recovery
- Files are huge by traditional standards
 - ▶ Multi-GB files are common
 - ▶ Billions of objects
- Workload features (Co-design of application and the file system)
 - ▶ Most files are mutated by appending new data rather than overwriting existing data.
 - ▶ Files are often read sequentially rather than randomly.

Master/Slave Architecture

■ A GFS cluster

▶ A **single master**

■ Maintains all **metadata**

- Name space, access control, file-to-chunk mappings, garbage collection, chunk migration

■ Periodically communicates with chunkservers in *HeartBeat* messages.

▶ **Multiple chunkservers** per master

■ Store actual file data

▶ Running on commodity Linux machines

Master/Slave Architecture (cont'd)

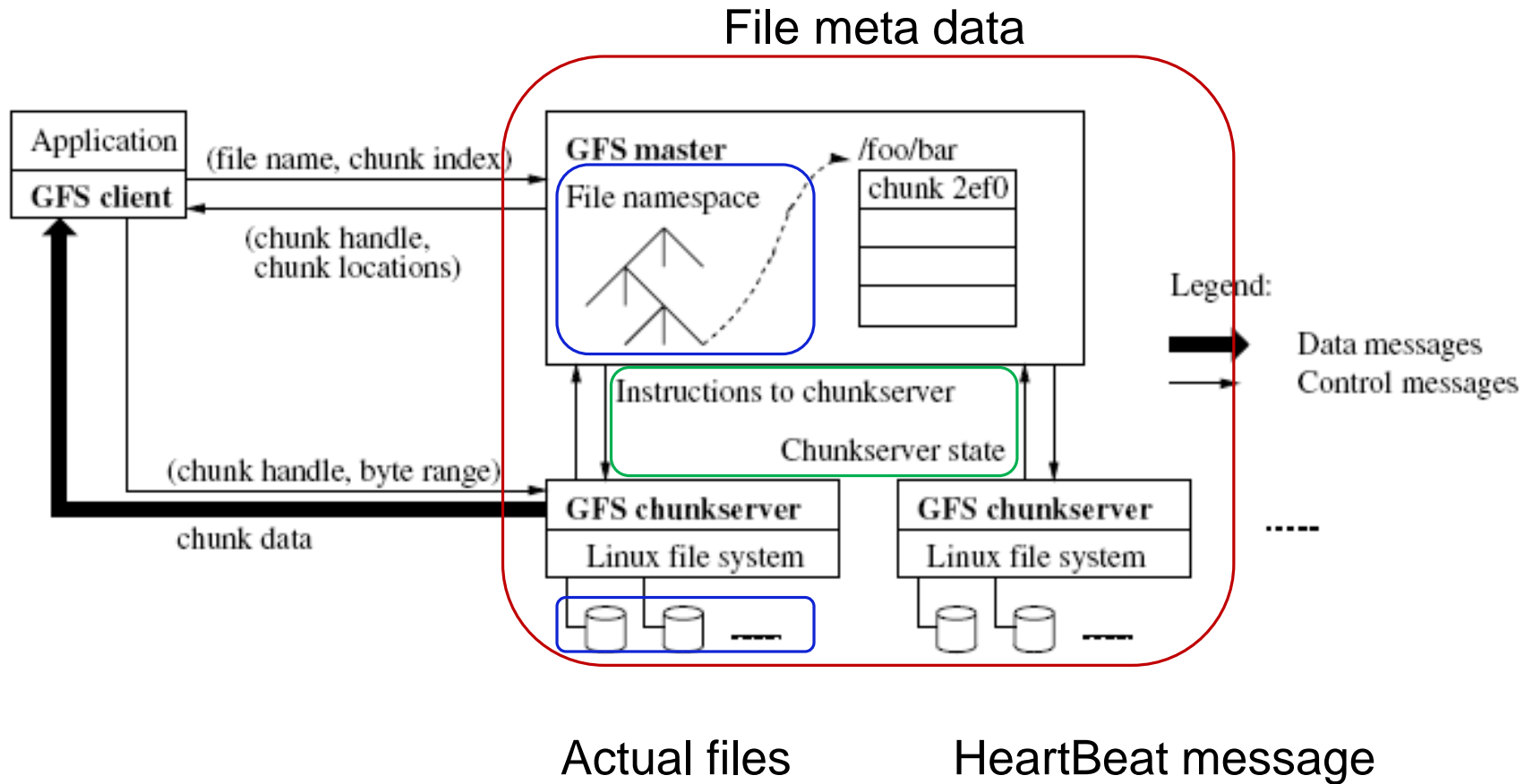
■ A file

- ▶ Divided into fixed-sized *chunks*
 - Default chunk size is 64 MB
 - Labeled with 64-bit unique global IDs (chunk handle)
 - Stored at chunkservers
 - Chunk is replicated on multiple chunkservers (by default 3 replicas
 - Files that needs frequent access will have a high replication factor

■ GFS clients

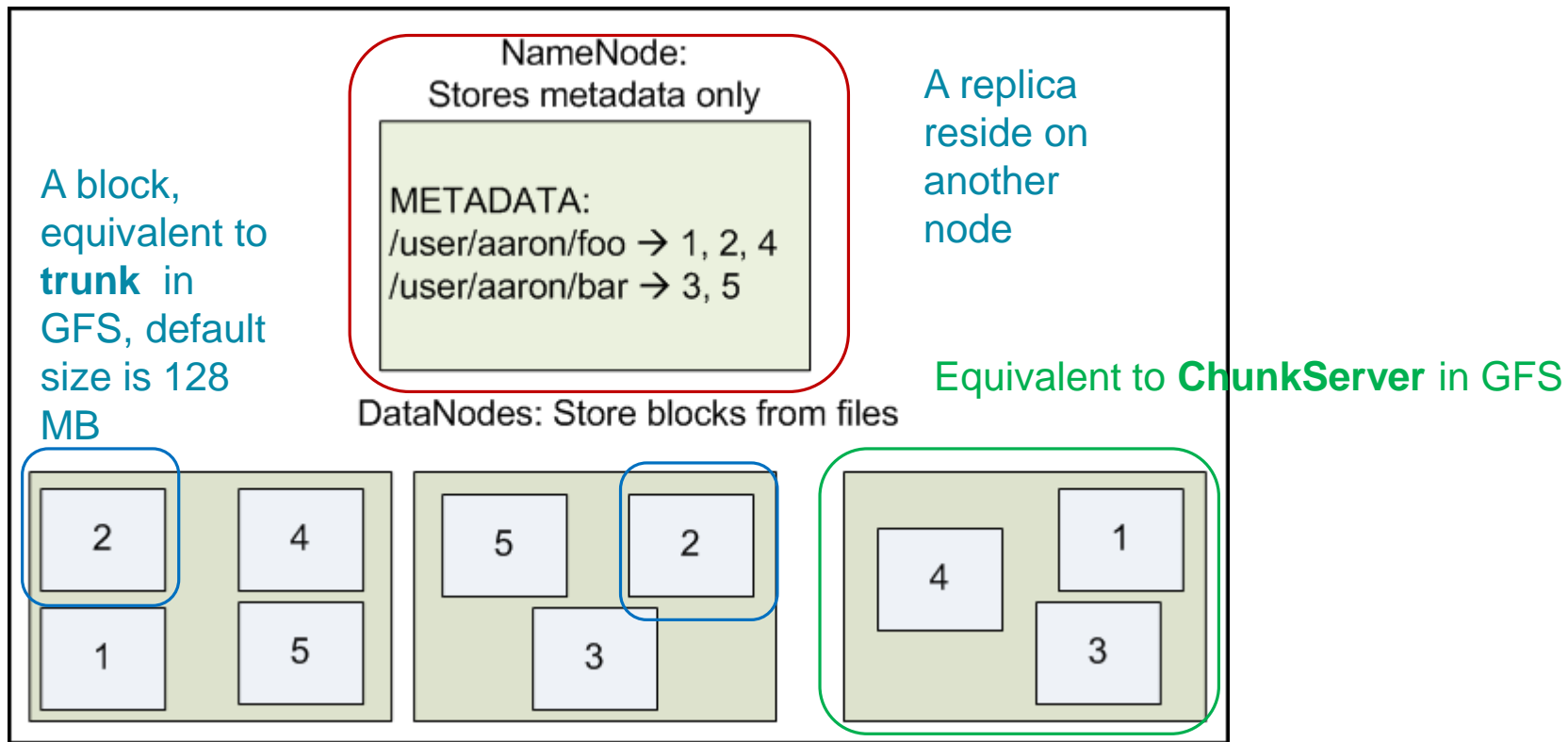
- ▶ Consult master for **metadata**
- ▶ Access data from **chunk servers**
- ▶ The equivalent in HDFS is the hdfs shell script
 - `hdfs dfs -put <source> <destination>`

Architecture diagram



HDFS

- Hadoop Distributed File System is the open source implementation of GFS
- It is part of the Hadoop framework Equivalent to **Master** in GFS



<http://developer.yahoo.com/hadoop/tutorial/module2.html>

GFS v.s. HDFS

GFS	HDFS
Naming	
Master	NameNode
Chunk Server	DataNode
Chunk	Block
Functionality	
Support random write and record append. In practical workload, the number of random write is very small	Support Write-Once-Read-Many workload
HA strategy requires an external monitoring services, such as Chubby lock service for Master recovery	HA configuration is only available in the latest version Secondary name node does part of the job GFS master is supposed to do. It is not a shadow master

Single-Master Design

■ Benefits:

- ▶ Simple, master can make sophisticated chunk placement and replication decisions using **global** knowledge

■ Possible disadvantage:

- ▶ Single point of failure (availability)
- ▶ Bottleneck (scalability)

■ Solution

- ▶ Replicate master states on multiple machines; Chubby is used to point a master among several shadow masters
- ▶ Fast recovery;
- ▶ Clients use master only for metadata, not reading/writing actual file

Metadata

■ Three types:

- ▶ File and chunk namespaces
- ▶ Mapping from files to chunks
- ▶ Locations of chunk replicas

■ All metadata is in memory.

- ▶ Large chunk size (64MB) ensures small meta data size
- ▶ First two are kept persistent in an *operations log* for recovery.
- ▶ Third is obtained by querying chunkservers at startup and periodically thereafter.

- Chunkservers may come and go or have partial disk failure

■ You may noticed in the lab that sometimes you are able to create directory but cannot upload files, why?

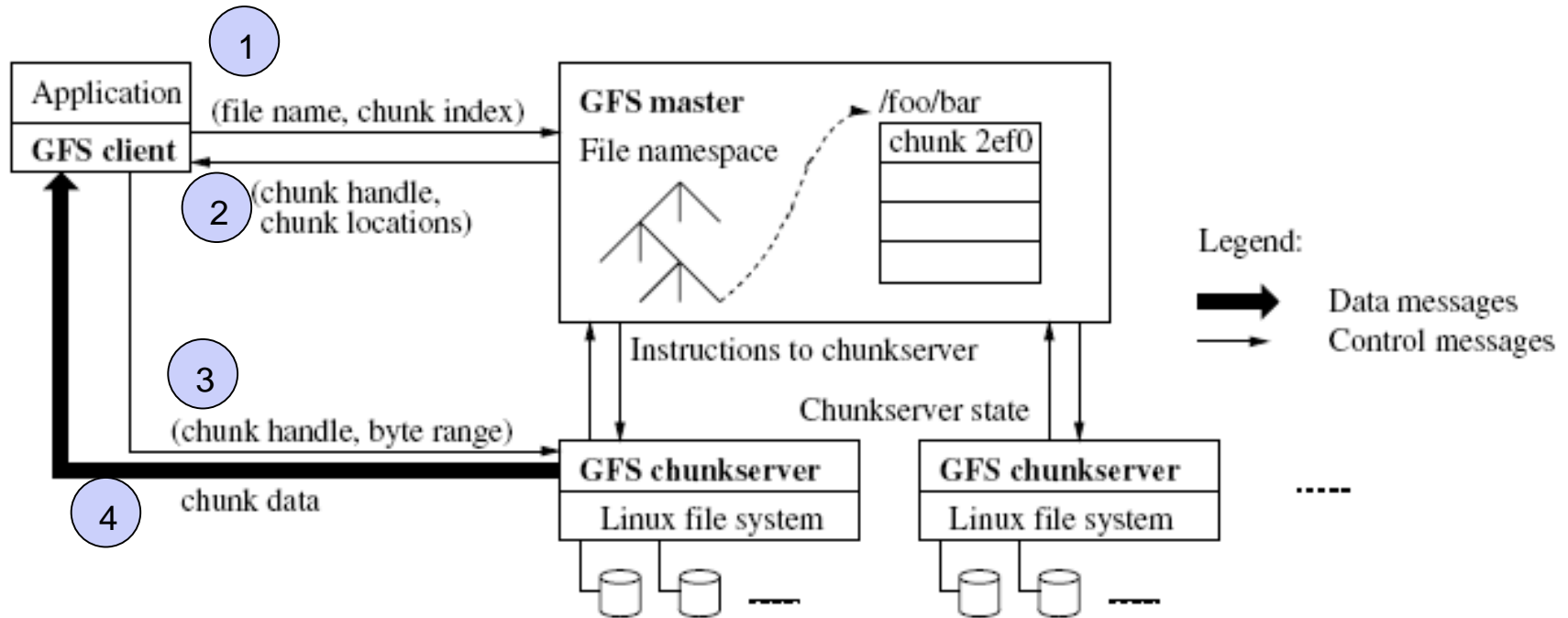
Operation Log and Master Recovery

- Metadata updates are logged
 - ▶ Create new directory, new files
 - ▶ Log replicated on remote machines
- Take global snapshots (checkpoints) to truncate logs
 - ▶ Memory mapped (no serialization/deserialization)
 - ▶ Checkpoints can be created while updates arrive
- Recovery
 - ▶ Latest checkpoint + subsequent log files

Scalability

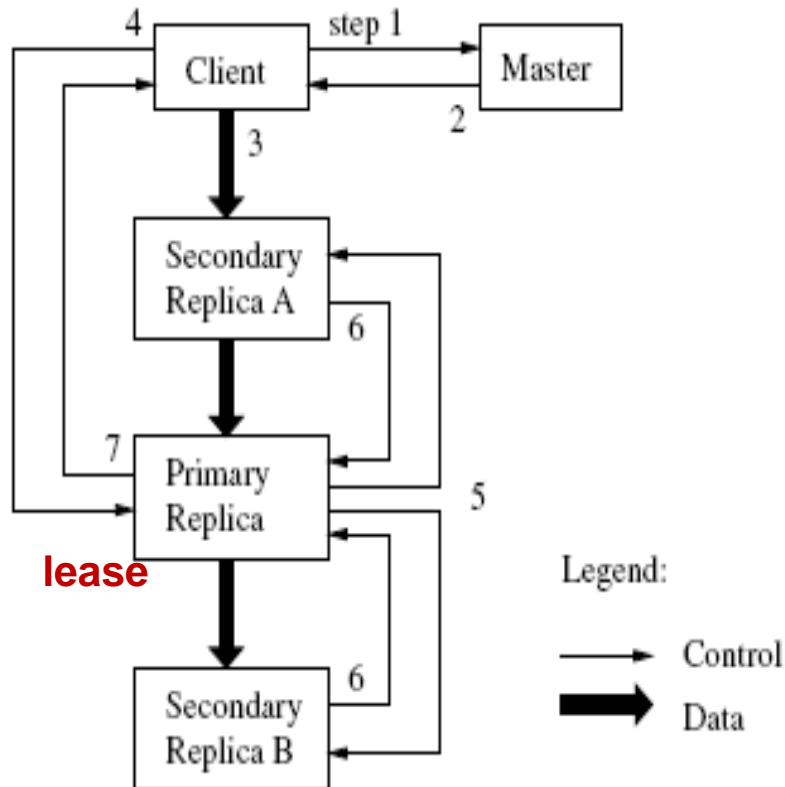
- Single Master is the only possible bottleneck
 - ▶ Minimize Master storage requirement for each file
 - ▶ Minimize Master involvement in read/write operation
 - Number of messages
 - Size of messages
- Design
 - ▶ Master is only involved at the *beginning* of Read/Write operation
 - ▶ Only limited chunk information is transferred
 - ▶ Actual data movement does not involve master
- A single master is able to server thousands of chunk servers

GFS Read



- › Client translates file name and byte offset to chunk index.
- › Sends request to master.
- › Master replies with chunk handle and location of replicas.
- › Client caches this info.
- › Client sends request to a close replica, specifying chunk handle and byte range.
- › The chunk server sends chunk data to the client

GFS write



1. The client asks the master which chunkserver holds the current **lease** for the chunk and the locations of the other replicas.

Grant a new lease if no one holds one

2. The master replies with the **identity of the primary** and the locations of the other (*secondary*) replicas

Cached in the client

3. The client **pushes the data** to all the replicas

4. Once all the replicas have acknowledged receiving the data, the client sends a **write request** to the primary. The primary **assigns consecutive serial numbers** to all the mutations it receives, possibly from **multiple clients**, which provides the necessary serialization.

5. The Primary forwards the write request to all secondary replicas.

6. Secondaries signal completion.

7. Primary replies to client. Errors handled by retrying.

Data Flow

- Data flow is decoupled from control flow
- Data is pushed from the client to a chain of chunkservers
 - ▶ Each machine selects the “closest” machine to forward the data
 - Client pushes data to chunkserver **S1** that has **replica A** and tells it there are two more chunkservers: **S2** and **S3** in the chain
 - **S1** pushes the data to **S2**, which is closer than **S3** and tells it **S3** is in the chain. **S2** has primary replica.
 - **S2** pushes the data to **S3** and tells it you are the last one. **S3** has **replica B**
- The network topology is simple enough that “distances” can be accurately estimated from IP addresses
- Data is pushed in a pipelined fashion
 - ▶ Each trunk server starts forwarding immediately after it receives some data



Lease and Mutation Order

- A mutation is an operation that changes the contents or metadata of a chunk
- The master grants a chunk lease to a replica
- The replica holding the lease becomes the **primary replica**; it determines the order of updates to all replicas
- Lease
 - ▶ 60 second timeouts
 - ▶ Can be extended indefinitely
 - ▶ Extension request are piggybacked on heartbeat messages
 - ▶ After an old lease expires, the master can grant new leases
- Any replica may become primary to coordinate the mutation process

High Availability (HA) Strategies

- Any server may be down/unavailable at a given time
- HA of the overall system relies on two simple strategies
 - ▶ fast recovery
 - Restore states quickly when a server (master or chunk) is back to life
 - ▶ Replication
 - Putting extra copies of data

Fast Recovery

- Small meta data
- Chunkservers maintain
 - ▶ Checksums for 64kb blocks of data for **data integrity check**
 - ▶ Replica/chunk version number for **data freshness check**
- Master maintain
 - ▶ Data information (owner, permission, mapping, etc)
 - ▶ Memory image and short operation log enable fast recovery
- It takes only a few seconds to read this metadata from disk before the server is able to answer queries.
- Master may wait a little longer to receive all chunk location information.

Master Replication

- Master operation log and checkpoints are replicated on multiple machine.
- If the master fails, monitoring infrastructure outside GFS starts a new master process elsewhere (Chubby lock service which utilizes Paxos algorithms to elect a new master among a group of living shadow masters).
- Shadow masters provides read-only access when the primary master is down.

Chunk replica

- Chunk replica is created within the cluster
- Master is responsible for chunk replica creation and placement
 - ▶ Decide where to put replica
 - ▶ Decide if a chunk needs new replica
 - ▶ Decide if a replica needs to migrate to a new location

Chunk Replica Placement

■ Goal

- ▶ Maximize data reliability and availability
- ▶ Maximize network bandwidth

■ Need to spread chunk replicas across machines and racks

- ▶ Read can exploit the aggregate bandwidth of multiple racks (read only needs to contact one replica)
- ▶ Write has to flow through multiple racks (write has to be pushed to all replicas)

Creating, Re-replication, Rebalancing

- Replicas created for three reasons:
 - ▶ Chunk creation, Re-replication, Load balancing
- Creation location consideration
 - ▶ Balance disk utilization
 - ▶ Balance creation events
 - After a creation, lots of traffic, especially write
 - ▶ Spread replicas across racks
- Re-replication
 - ▶ Occurs when number of replicas falls below a user-specified number
 - Replica corrupted, chunkserver down, replication number increased.
 - ▶ Replicas prioritized, then rate-limited (to reduce impact on client traffic).
 - ▶ Placement heuristics similar to that for creation.
- Rebalancing
 - ▶ Periodically examines distribution and moves replicas around.

Outline

■ GFS and HDFS

■ MapReduce on YARN

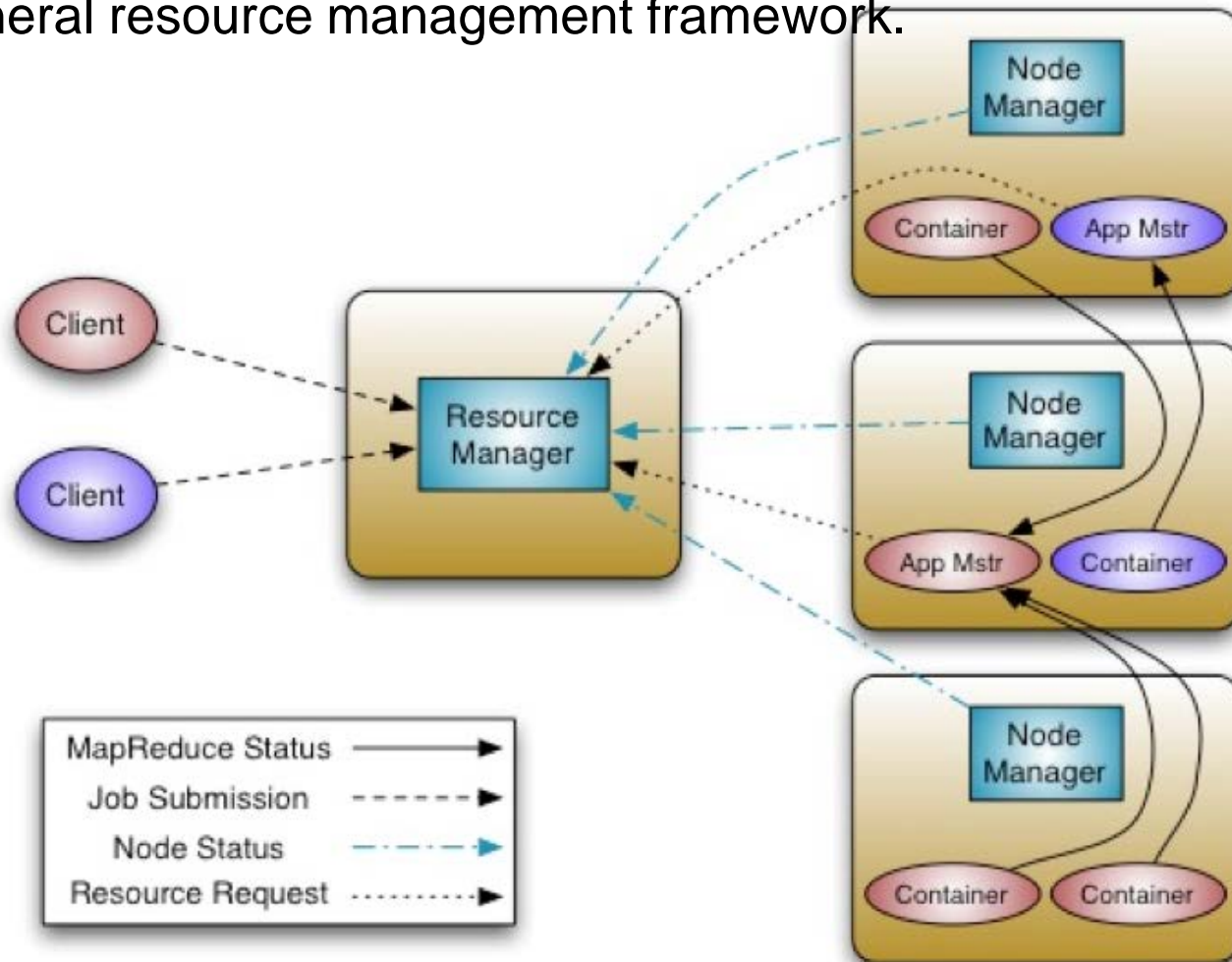
- ▶ Revisit basic component of YARN
- ▶ YARN scheduling
- ▶ Data locality constraints
- ▶ Hadoop Straggler handling mechanism

■ Spark Execution



YARN Framework

- Yet Another Resource Negotiator
 - ▶ A general resource management framework.



YARN Concepts

- YARN's world view consists of applications requesting resources
- Resource Manager (one per cluster)
 - ▶ “Is primarily, a *pure scheduler*. In essence, it's strictly limited to arbitrating available resources in the system among the competing applications”
- Node Manager (one per node)
 - ▶ Responsible for managing resources on individual node
- ApplicationMaster (one per application)
 - ▶ Responsible for requesting resources from the cluster on behalf of an application
 - ▶ Monitor the execution of application as well
 - ▶ Framework specific
 - MapReduce application's AM is different of Spark application's AM



YARN Resource Concepts

■ Resource Request

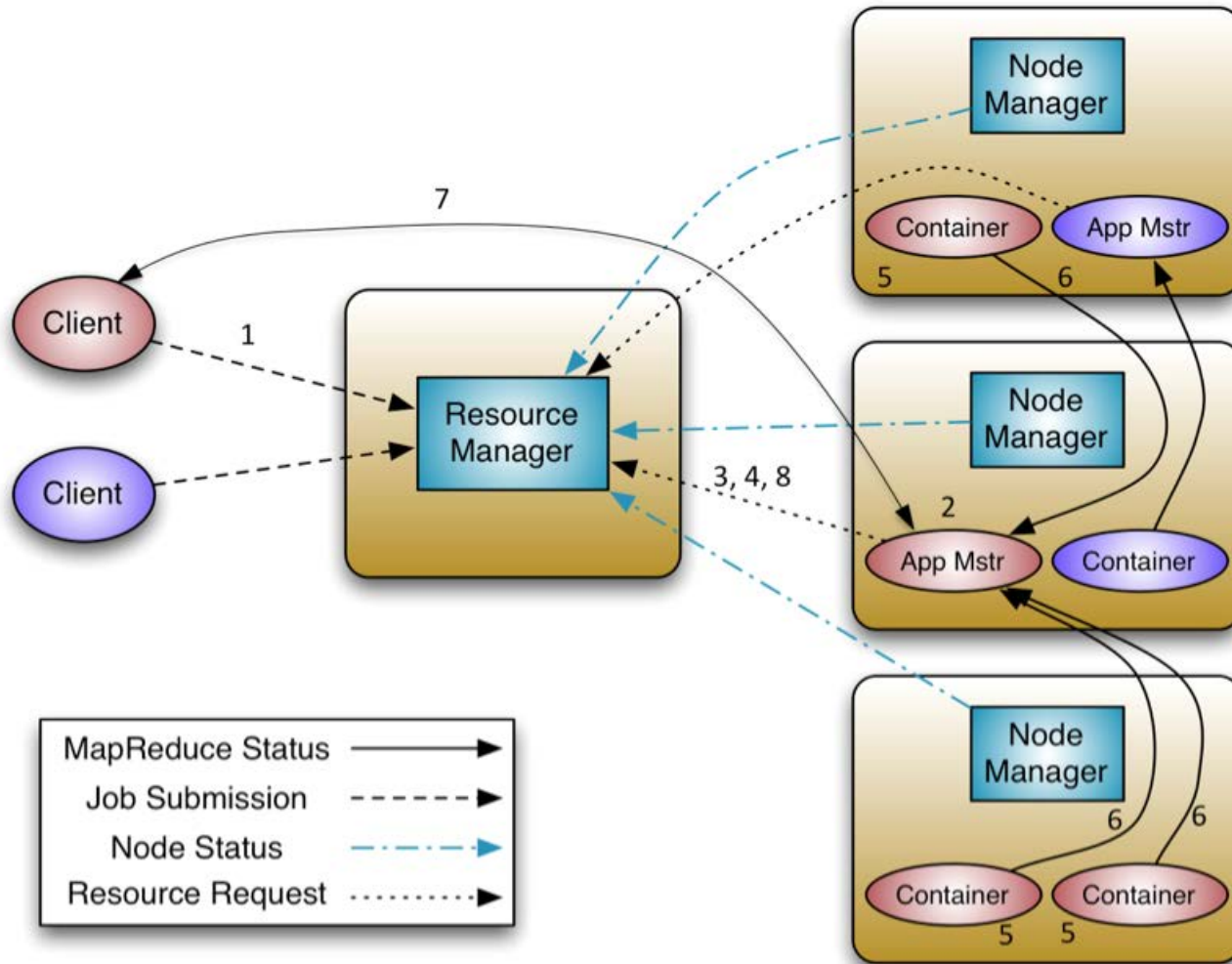
- ▶ In the current model, resource requirements are expressed mainly in Memory and/or CPU cores
- ▶ Depends on configured scheduler

■ Container

- ▶ Resource allocation is in the form of container
- ▶ A Container grants rights to an application to use a specific amount of resources (memory, cpu etc.) on a specific host.
- ▶ MR framework uses containers to run map or reduce tasks
- ▶ Container is launched by NodeManager

- A MapReduce Job with 10 map tasks and 3 reduce tasks would need 14 containers. The first container runs its AM, which would request 10 containers for map tasks and 3 containers for reduce tasks.

YARN Walkthrough



YARN and MR memory configuration

- YARN needs to know
 - ▶ The total amount of memories allocated to YARN
 - ▶ How to break up the total resources into containers
 - Minimum container size
- ApplicationMaster for MapReduce application needs to know
 - ▶ Memory requirement for Map and Reduce tasks
 - We usually assume Reduce tasks need more memory
 - ▶ Each task is running on a JVM, the JVM heap size needs to be set as well

Memory Configuration Example

■ Node capacity

- ▶ 48G Ram, 12 core

■ Memory allocation

- ▶ Reserve 8G Ram for OS and others (e.g. HBase)
- ▶ YARN can use up to 40G
- ▶ Set Minimum container size to 2G
 - Each node may have at most 20 containers
- ▶ Allow each map task to use 4G and reduce task to use 8G
 - Each node may run 10 map tasks or 5 reduce task or a combination of the two

Resource Scheduling

- Resource Scheduling deals the problem of which requests should get the available resources when there is a queue of resources
- Early MRv1 uses FIFO as default scheduling algorithm
 - ▶ Large job that comes first may starve small jobs
 - ▶ Users may experience long delay as their jobs are waiting in queue
- YARN is configured to support shared multi-tenant cluster
 - ▶ Capacity Scheduler
 - ▶ Fair Scheduler
 - ▶ FIFO Scheduler

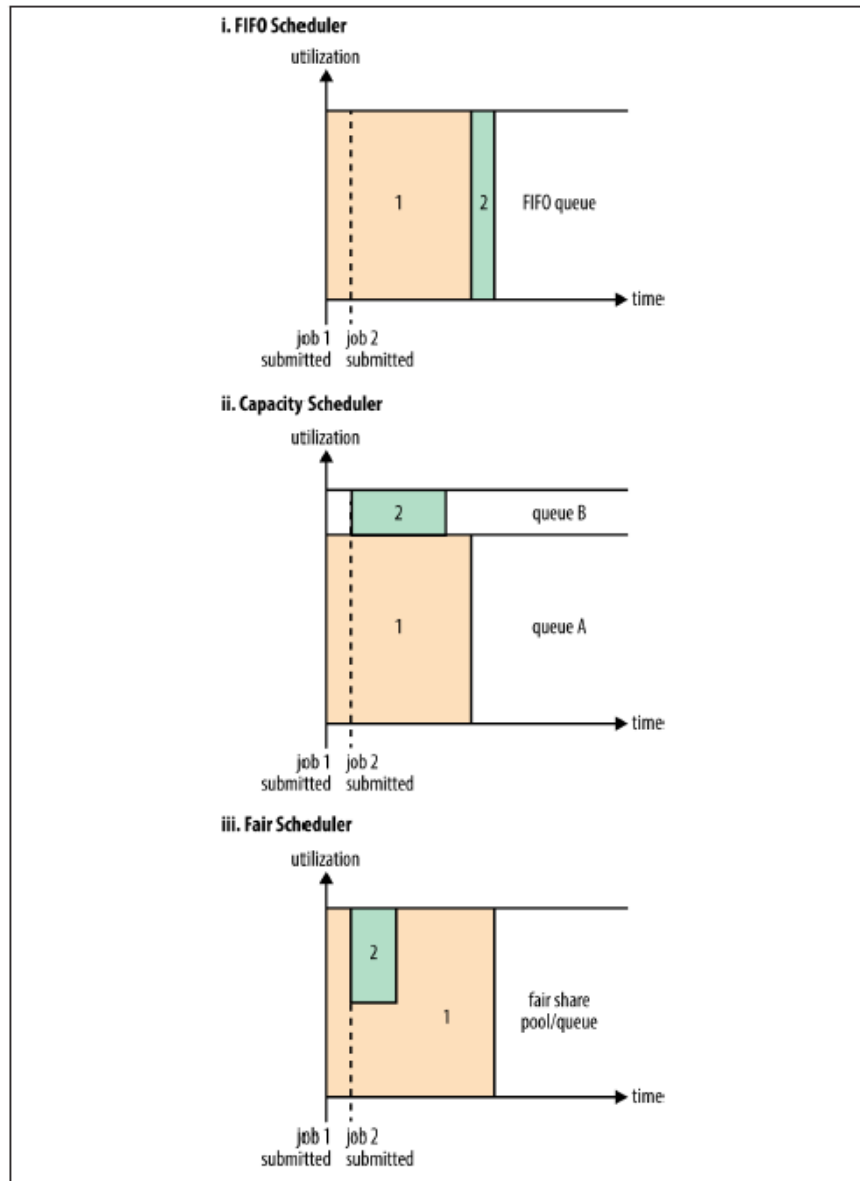


Figure 4-3. Cluster utilization over time when running a large job and a small job under the FIFO Scheduler (i), Capacity Scheduler (ii), and Fair Scheduler (iii)

Capacity Scheduler

- Capacity Scheduler is designed for multi-tenancy situation
 - ▶ Each organization has a dedicated queue with a given fraction of the cluster capacity
 - Queues may be further divided to set up capacity allocation within organization
 - A single job does not use more resources than the queue capacity, but if there are more than one job in the queue, and there are idle resources, spare resources can be allocated to jobs in the queue
 - How much more resources can be allocated to a given queue is configurable
 - Which users can submit jobs to which queue is configurable

<http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/CapacityScheduler.html>



Fair Scheduler

- The principle of fair scheduling is to allocate resources so that all running applications get similar share of resources
 - ▶ E.g. if there are R units of resources and J jobs, each job should get around R / J units of resources.
- YARN fair scheduler works between queues: each queue gets a fair share of the cluster resources
 - ▶ Within queue, the scheduling policy is configurable: FIFO or Fair

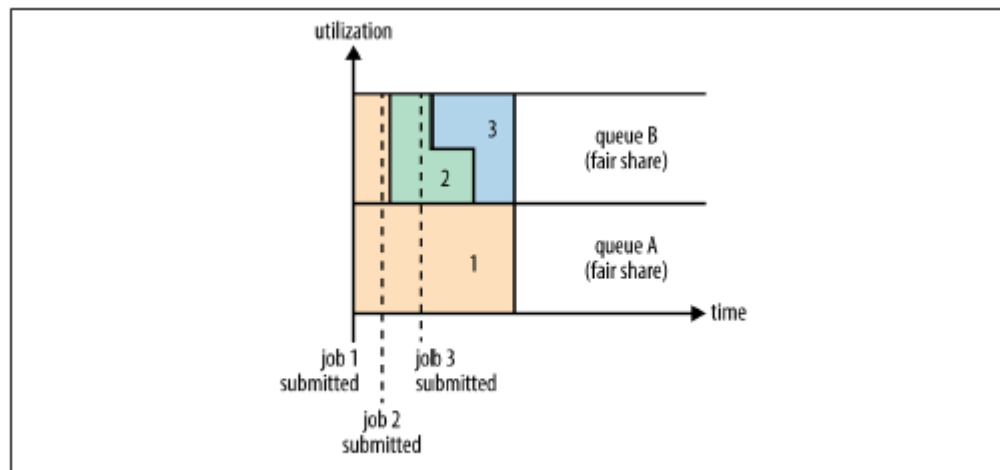


Figure 4-4. Fair sharing between user queues

Hadoop: the definitive guide, 4th edition, page 91

Fair Scheduler – Queue Placement

- Fair scheduler uses a rule based system to determine which queue an application should be placed in
 - ▶ Example queue placement policy

```
<queuePlacementPolicy>  
  <rule name="specified" />  
  <rule name="user" />  
</queuePlacementPolicy>
```
- All queues in capacity scheduler needs to be specified beforehand by cluster administrator; Queues in fair scheduler can be created on the fly

Data Locality Constrains

- Data intensive workloads (MapReduce/Spark)
 - ▶ have **storage attached** to computers.
 - ▶ Scheduling **tasks near data** improves performance.
- The requirements of fairness and locality often conflict
 - ▶ A strategy that achieves optimal data locality will typically delay a job until its ideal resources are available
 - ▶ Fairness benefits from allocating the best available resources to a job as soon as possible after they are requested
- General third party resource management system, e.g. YARN, might not follow the data locality preference

Hadoop's straggler handling mechanism



■ Speculative execution

- ▶ If a node is available but is performing poorly, this is called a straggler
- ▶ MapReduce has a build-in mechanism to run a speculative copy of its task on another machine to finish the computation faster.
- ▶ Speculative task attempts could be successful or just a waste

■ Who manages this speculative mechanism

- ▶ YARN resource manager
- ▶ YARN node manager
- ▶ Application Master?

A successful speculative task attempt

Task Attempts	Machine	Status	Progress	Start Time	Finish Time
attempt_201108241404_4015_m_000030_0	Task attempt: /default-rack/dm1.cs.usyd.edu.au Cleanup Attempt: /default-rack/dm1.cs.usyd.edu.au	KILLED	100.00% 	23-Oct-2011 09:17:20	23-Oct-2011 09:17:57 (37sec)
attempt_201108241404_4015_m_000030_1	/default-rack/gpu1.cs.usyd.edu.au	SUCCEEDED	100.00% 	23-Oct-2011 09:17:32	23-Oct-2011 09:17:50 (17sec)

Input Split Locations

/default-rack/dm2.cs.usyd.edu.au
/default-rack/gpu1.cs.usyd.edu.au
/default-rack/gpu0.cs.usyd.edu.au

A not useful speculative task attempt

Attempt	State	Status	Node	Logs	Start Time	Finish Time	Elapsed Time	Note
attempt_1522231243385_0346_m_000000_0	SUCCEEDED	map > sort	/default-rack/soit-hdp-pro-27.ucc.usyd.edu.au:8042	logs	Mon Apr 16 14:51:09 +1000 2018	Mon Apr 16 14:51:33 +1000 2018	24sec	
attempt_1522231243385_0346_m_000000_1	KILLED		/default-rack/soit-hdp-pro-3.ucc.usyd.edu.au:8042	logs	Mon Apr 16 14:51:18 +1000 2018	Mon Apr 16 14:51:33 +1000 2018	14sec	Speculation: attempt_1522231243385_0346_m_000000_0 succeeded first!

http://soit-hdp-pro-1.ucc.usyd.edu.au:19888/jobhistory/task/task_1522231243385_0346_m_000001

Progress score

- Hadoop monitors task progress using a *progress score* to select speculative tasks
 - ▶ Map task's progress score is the fraction of input data read
 - ▶ Reduce task's execution is divided into three phases, each of which account for 1/3 of the score. In each phases, the score is the fraction of data process
- When a task's progress score is less than the average for its category minus 0.2 and the task has run for at least one minute, it is marked as a straggler

Task Attempts	Machine	Status	Progress	Start Time	Shuffle Finished	Sort Finished	Finish Time
attempt_201108241404_1214_r_000000_0	/default-rack/gpu1.cs.usyd.edu.au	SUCCEEDED	100.00% 	6-Oct-2011 15:21:48	6-Oct-2011 15:23:36 (1mins, 47sec)	6-Oct-2011 15:23:36 (0sec)	6-Oct-2011 15:23:39 (1mins, 51sec)

For a **reduce** task, the execution is divided into three phases, each of which accounts for 1/3 of the score progress score is the fraction of input data read

Copy phase sort phase reduce phase

Hadoop job execution profile

```
hadoop jar userTag.jar usertag.TagSmartDriver /share/photo/n08.txt userN08Out
```

http://soit-hdp-pro-1.ucc.usyd.edu.au:19888/jobhistory/tasks/job_1522231243385_0348/m

	Replicas	Map Task Running on
Block 0	soit-hdp-pro-[4,8,27].ucc.usyd.edu.au	soit-hdp-pro-15.ucc.usyd.edu.au
Block 1	soit-hdp-pro-[3,21,2].ucc.usyd.edu.au	soit-hdp-pro-3.ucc.usyd.edu.au
Block 2	soit-hdp-pro-[4,5,12].ucc.usyd.edu.au	soit-hdp-pro-[5,15].ucc.usyd.edu.au
Block 3	soit-hdp-pro-[4,13,21].ucc.usyd.edu.au	soit-hdp-pro-5.ucc.usyd.edu.au

Data Locality preference is only observed in two tasks

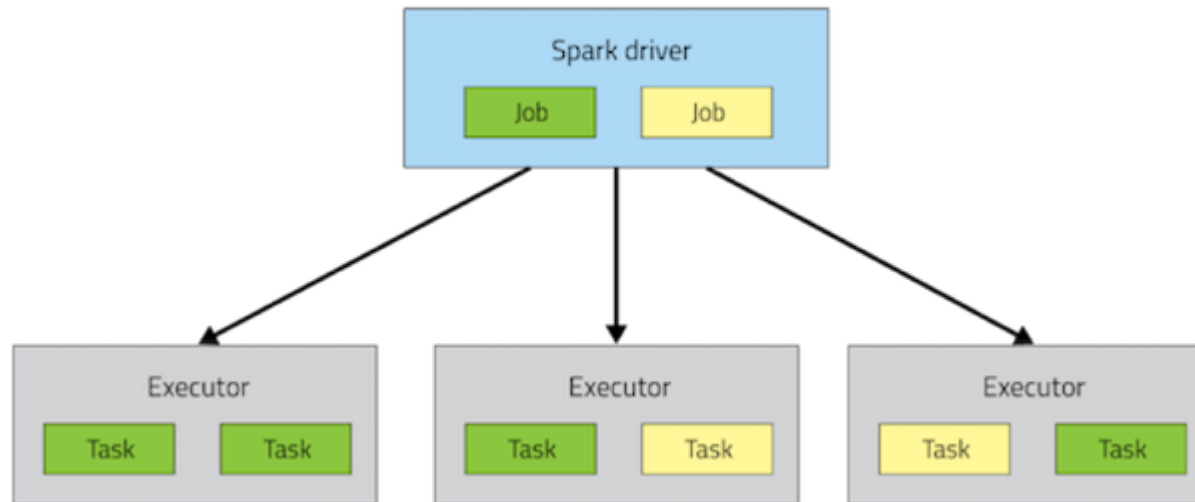
Speculative map tasks are always a waste with certain input size

Outline

- GFS and HDFS
- **MapReduce on YARN**
- Spark Execution
 - ▶ Execution Environment



How Spark Execute Application



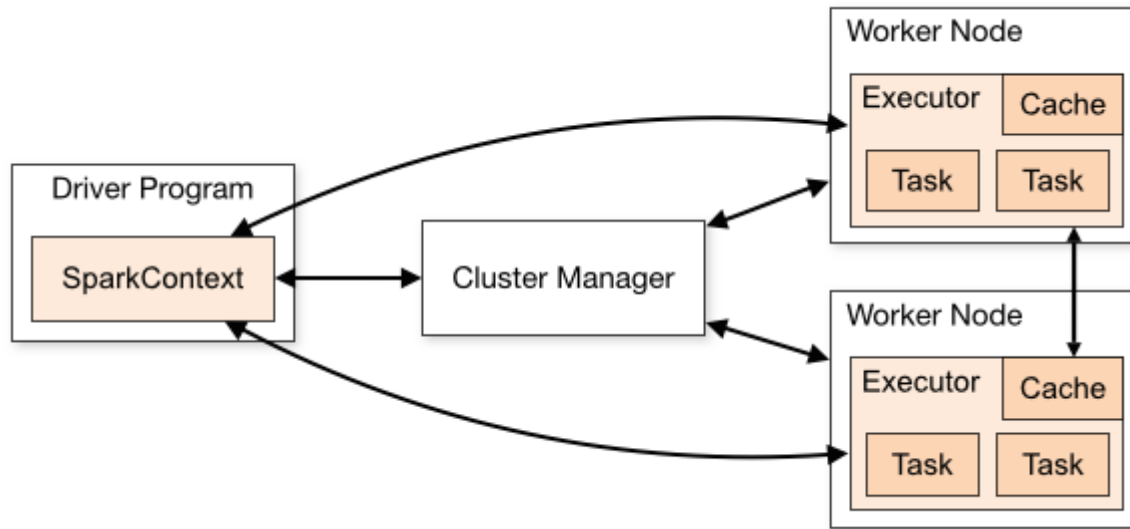
“The **driver** is the process that is in charge of the high-level control flow of work that needs to be done. The **executor** processes are responsible for executing this work, in the form of **tasks**, as well as for storing any data that the user chooses to cache. Both the **driver** and the **executors** typically stick around for the entire time the application is running”

Based on <http://blog.cloudera.com/blog/2015/03/how-to-tune-your-apache-spark-jobs-part-1/>
Section 5 of the original Spark paper published on NSDI'12 by Zaharia, Matei, et al



Cluster Deployment Modes

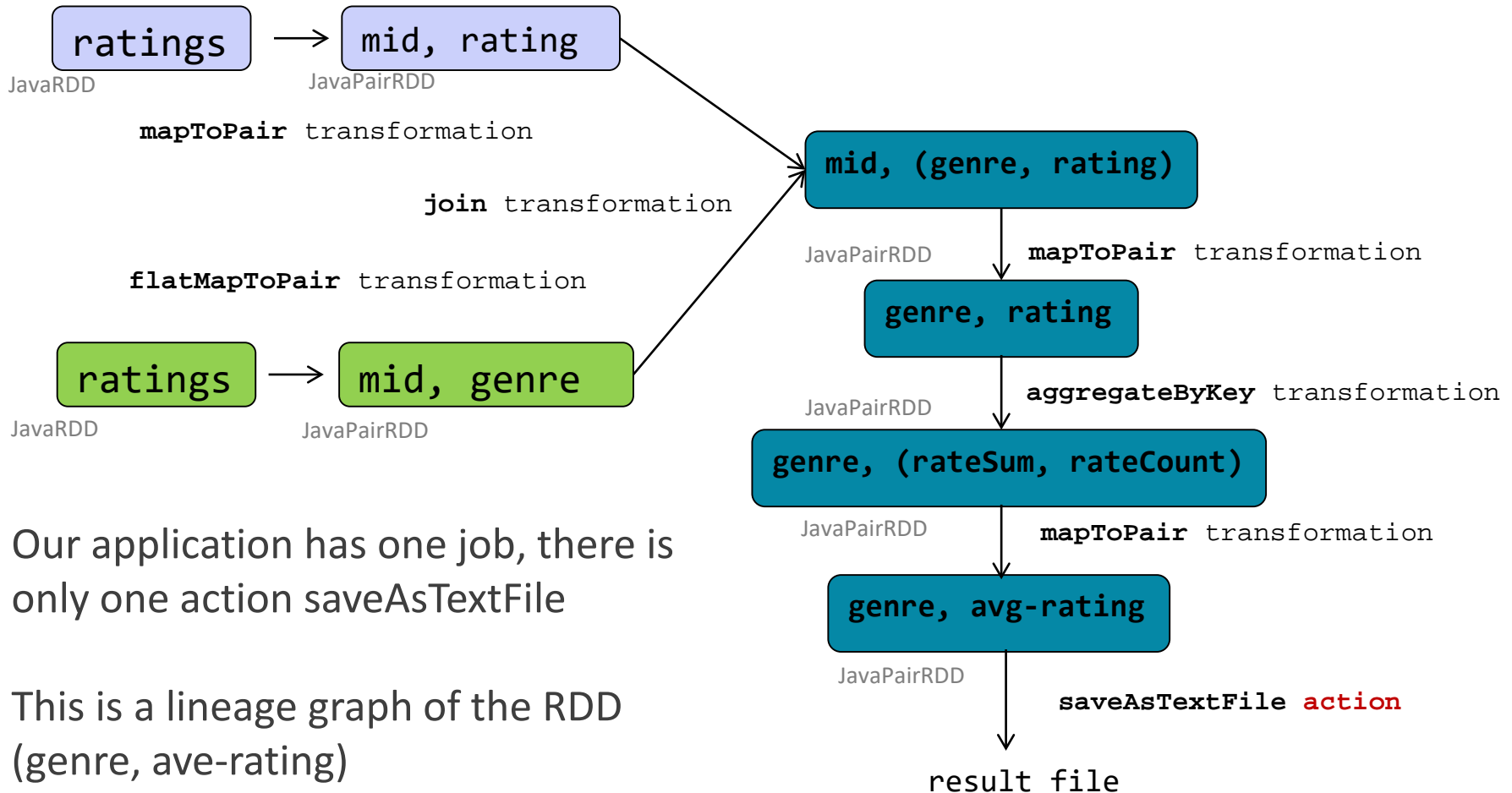
YARN/Mesos/Standalone



Depending on where the driver is running, spark application can be submitted in either cluster or client mode

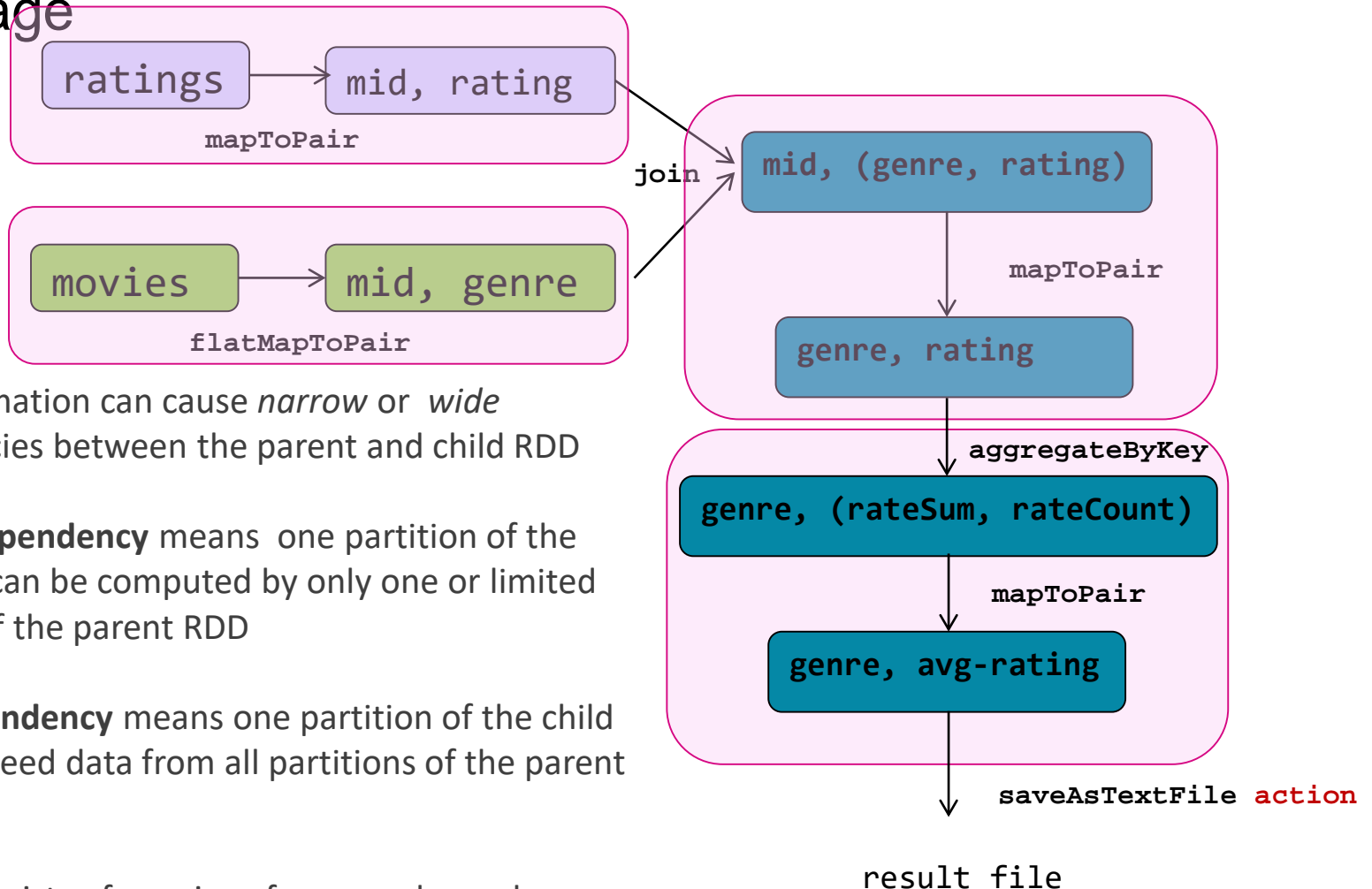
Job, Stage and Task

- Job is triggered by action such as **count**, **save**, etc



Job, Stage and Task

- A job can have many stages as a DAG based on the RDD's lineage



A transformation can cause *narrow* or *wide* dependencies between the parent and child RDD

Narrow dependency means one partition of the child RDD can be computed by only one or limited partition of the parent RDD

Wide dependency means one partition of the child RDD may need data from all partitions of the parent RDD

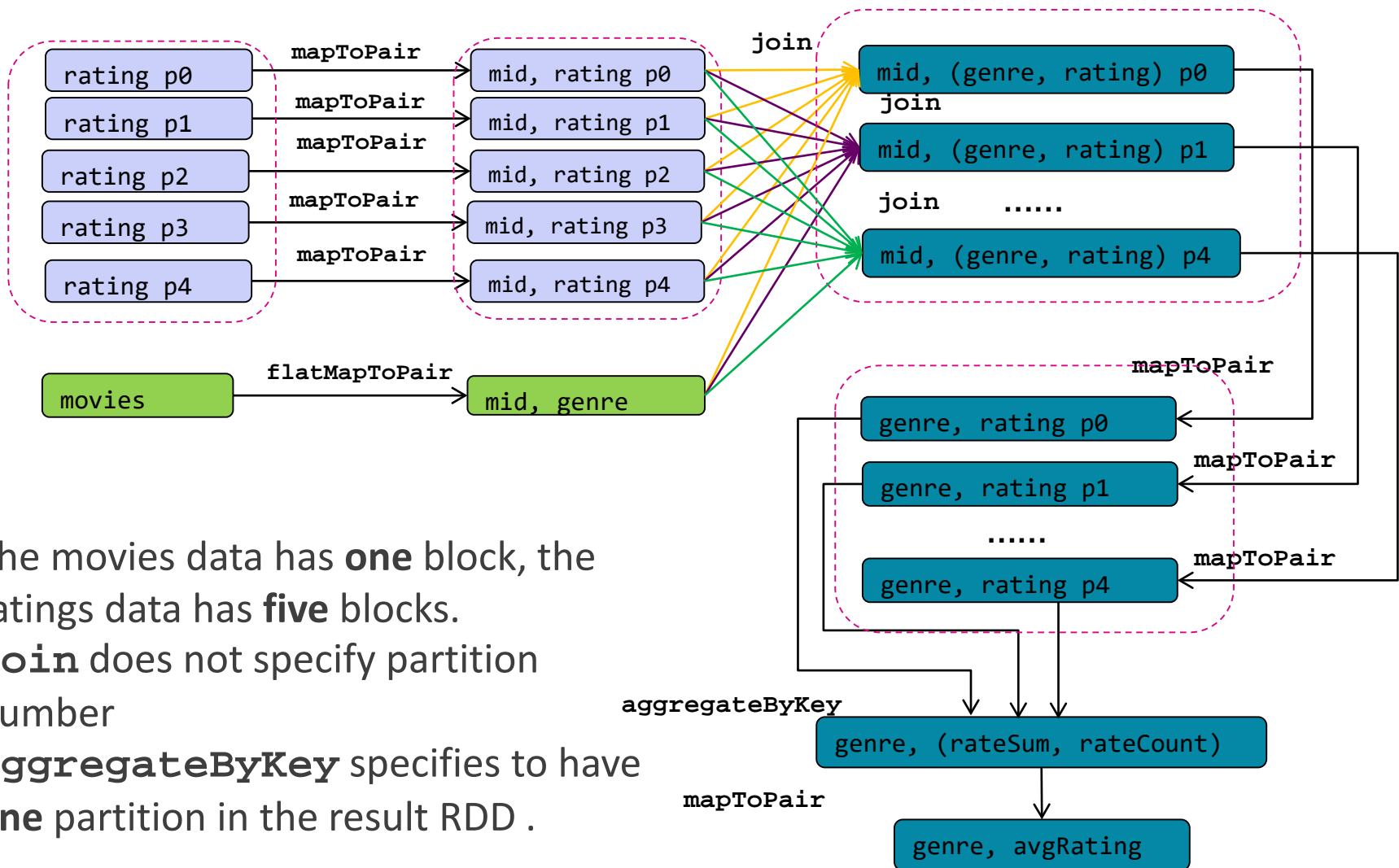
A **stage** consists of a series of narrow dependency transformations

Job, Stage and Task

- Transformations inside a stage can execute in a pipeline style to improve efficiency
 - ▶ There is no global data shuffle inside a stage
- There will be data shuffling across stages
 - ▶ Similar to shuffle in MapReduce framework
- The pipelined transformations inside a stage represent a task
 - ▶ Task is the actual execution unit of an application
 - ▶ The actual number of tasks of an application depends on the number of partitions the parent RDD has
 - ▶ Wide dependency transformation can take a number of partition parameter

Job, Stage and Task

5 join + mapToPair tasks



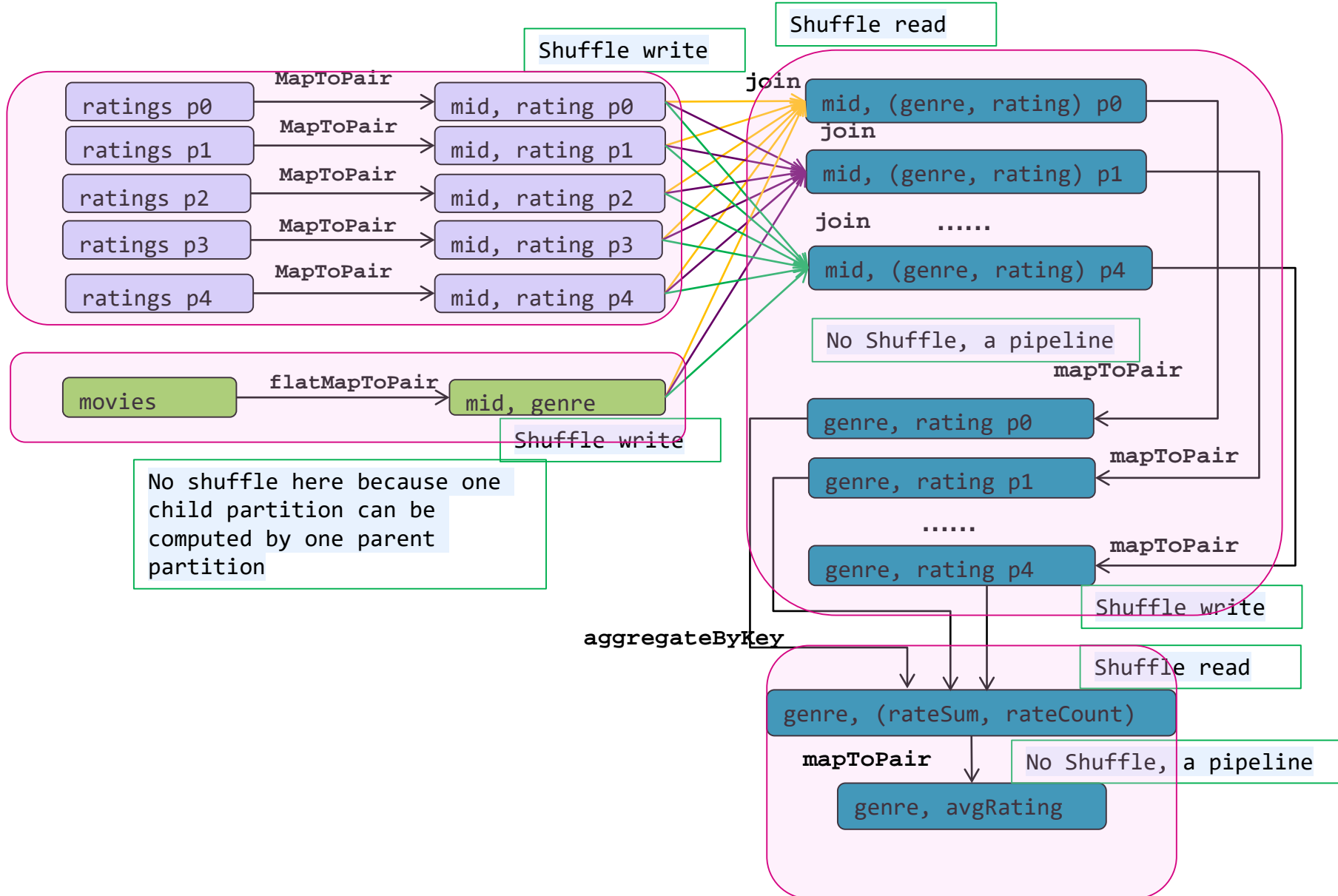
The movies data has **one** block, the ratings data has **five** blocks.
join does not specify partition number
aggregateByKey specifies to have **one** partition in the result RDD.

1 aggregateByKey + mapToPair tasks

Shuffle and its impact

- Shuffles are fairly expensive; all shuffle data must be written to disk and then transferred over the network.
- Design and choose your transformations carefully to avoid shuffling too much data
- Transformations causing shuffle also stress memory if not designed properly
 - ▶ Any **join**, ***ByKey** operation involves holding objects in hashmaps or in-memory buffers to group or sort.
 - ▶ It is preferred to have more tasks to reduce memory stress in individual node

When shuffles happen

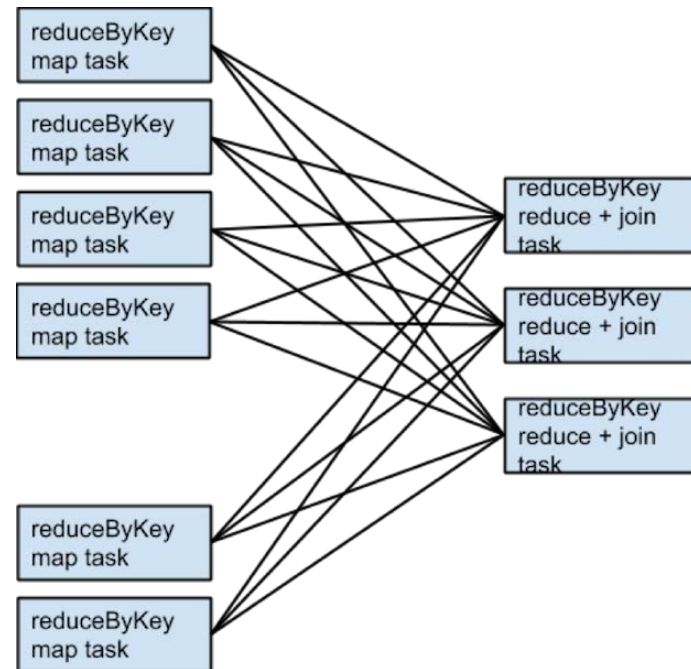


When shuffles do not happen

- There are cases where **join** or ***ByKey** operation does not involve shuffle and would not trigger stage boundary
 - ▶ If child and parent RDDs are partitioned by the same partitioner and/or have the same number of partition

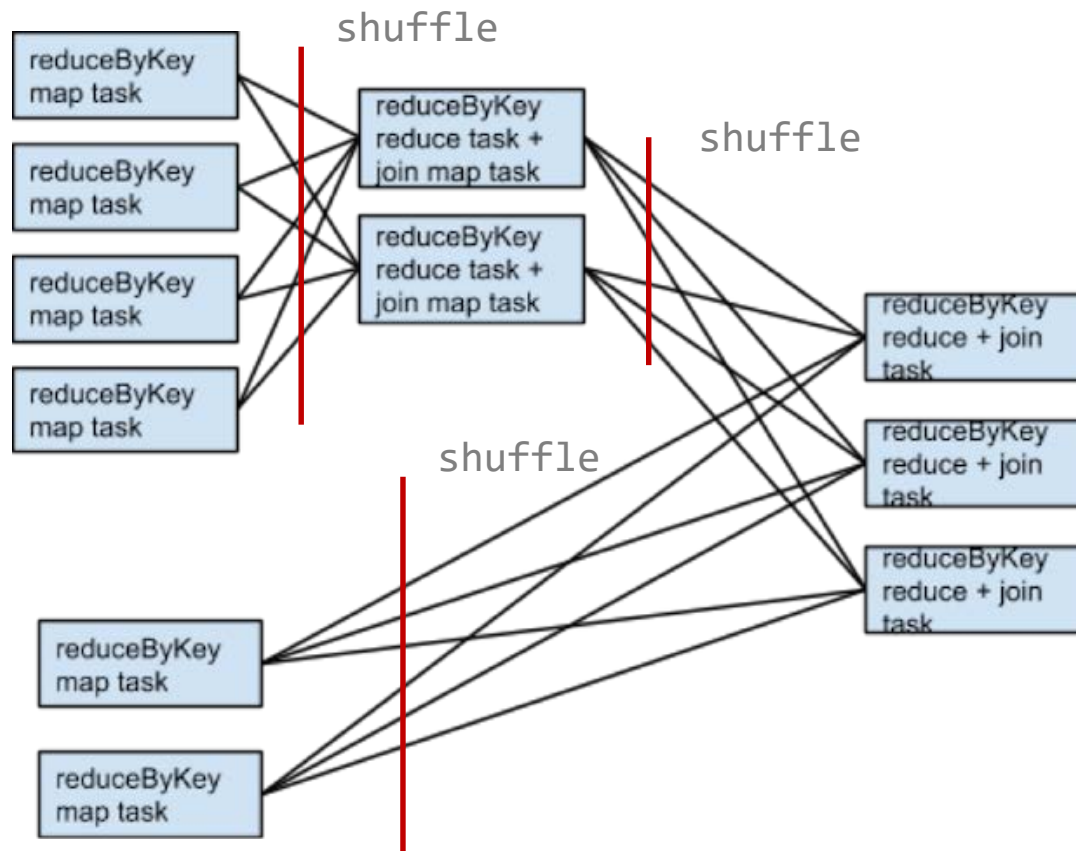
Join with input co-partitioned

```
rdd1 = someRdd.reduceByKey(...)  
rdd2 = someOtherRdd.reduceByKey(...)  
rdd3 = rdd1.join(rdd2)
```



When shuffles do not happen (cont'd)

- If the parent RDD uses the same partitioner, but result in different number of partitions. Only one parent RDD needs to be re-shuffled for the join operation



How to submit Spark Application

- Spark applications are usually submitted using **spark-submit** script
 - ▶ Specify a few important parameters
- For debugging on local installation, you can set all important parameters in **SparkContext** and run the program as a java application inside an IDE such as Eclipse.

```
spark-submit \  
  --class ml.MovieLensLarge \  
  --master yarn-cluster \  
  --num-executors 2 \  
  --num-cores 2 \  
  sparkML.jar \  
  /share/movie/ \  
  week6-out-java
```

Application History Screen Shot

Completed Jobs (1)

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
0	saveAsTextFile at MovieLensLarge.java:153	2015/04/10 06:46:29	33 ms	4/4	13/13

<http://soit-hdp-pro-1.ucc.usyd.edu.au:18080/>
<http://soit-hdp-pro-1.ucc.usyd.edu.au:8080/>
<http://soit-hdp-pro-1.ucc.usyd.edu.au:8088/>

Completed Stages (4)

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
3	saveAsTextFile at MovieLensLarge.java:153 +details	2015/04/10 06:45:11	1.0 s	1/1		579.0 B		
2	mapToPair at MovieLensLarge.java:112 +details	2015/04/10 06:44:27	44 s	5/5			58.8 MB	3.7 KB
1	flatMapToPair at MovieLensLarge.java:87 +details	2015/04/10 06:43:53	3 s	2/2	1386.9 KB			357.4 KB
0	mapToPair at MovieLensLarge.java:76 +details	2015/04/10 06:43:53	34 s	5/5	542.2 MB			128.5 MB

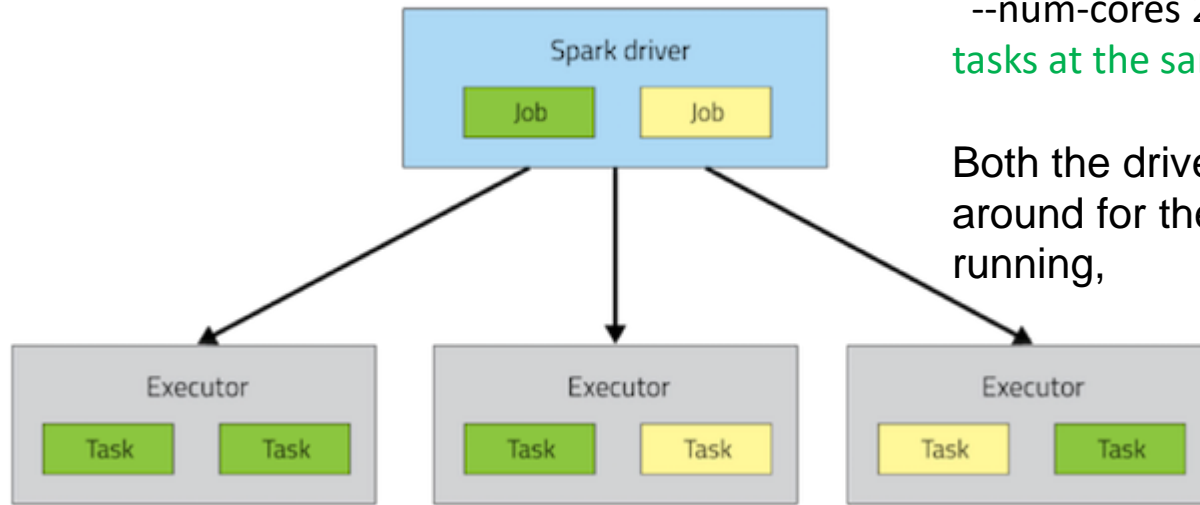
Spark starts two tasks for file with only one block

Application History Screen Shot

Executor ID	Address	Status	RDD Blocks	Storage Memory	Complete Tasks	Total Tasks	Task Time (GC Time)	Input	Shuffle Read	Shuffle Write
driver	172.16.176.37:51666	Active	0	0.0 B / 384.1 MB	0	0	0 ms (0 ms)	0.0 B	0.0 B	0.0 B
1	soit-hdp-pro-6.ucc.usyd.edu.au:50824	Active	0	0.0 B / 956.6 MB	6	6	28 s (1 s)	270.3 MB	33 MB	64 MB
2	soit-hdp-pro-5.ucc.usyd.edu.au:42232	Active	0	0.0 B / 956.6 MB	7	7	38 s (2 s)	351.9 MB	38.4 MB	82.4 MB

--num-executors 2 // there are two executors to run this application

--num-cores 2 // each executor can run maximum 2 tasks at the same time, , e.g. two threads



Both the driver and the executors typically stick around for the entire time the application is running,

Spark Data Locality

- Data locality means how close data is to the code processing it
 - ▶ Both MapReduce and Spark work on the principle of shipping code instead of data
- Spark Data Locality Levels
 - ▶ PROCESS_LOCAL data is in the same JVM as the running code. This is the best locality possible
 - ▶ NODE_LOCAL data is on the same node. Examples might be in HDFS on the same node, or in another executor on the same node. This is a little slower than PROCESS_LOCAL because the data has to travel between processes
 - ▶ NO_PREF data is accessed equally quickly from anywhere and has no locality preference
 - ▶ RACK_LOCAL data is on the same rack of servers. Data is on a different server on the same rack so needs to be sent over the network, typically through a single switch
 - ▶ ANY data is elsewhere on the network and not in the same rack
- Again data locality preference is not always observed if spark is scheduled by YARN

Inspecting Data Locality

■ Check 'stage information' on spark history server

Details for Stage 1 (Attempt 0)

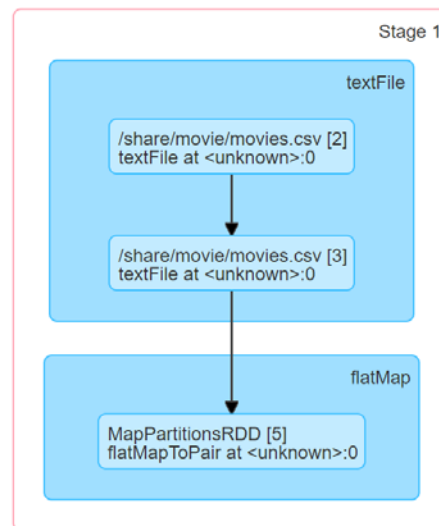
Total Time Across All Tasks: 2 s

Locality Level Summary: Rack local: 2

Input Size / Records: 1740.6 KB / 34208

Shuffle Write: 425.4 KB / 66668

▼ DAG Visualization



Index	ID	Attempt	Status	Locality Level	Executor ID / Host	Launch Time	Duration	GC Time	Input Size / Records	Write Time	Shuffle Write Size / Records
0	1	0	SUCCESS	RACK_LOCAL	1 / soit-hdp-pro-6.ucc.usyd.edu.au stdout stderr	2018/04/16 16:57:28	1 s	0.2 s	896.0 KB / 17052	27 ms	221.1 KB / 35415
1	2	0	SUCCESS	RACK_LOCAL	1 / soit-hdp-pro-6.ucc.usyd.edu.au stdout stderr	2018/04/16 16:57:30	0.2 s		844.6 KB / 17156	3 ms	204.3 KB / 31253

Inspecting Data Locality

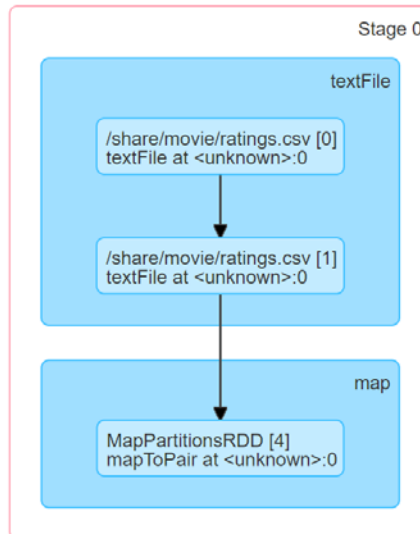
Total Time Across All Tasks: 30 s

Locality Level Summary: Node local: 1; Rack local: 4

Input Size / Records: 591.7 MB / 22884377

Shuffle Write: 139.2 MB / 22884377

▼ DAG Visualization



Index	ID	Attempt	Status	Locality Level	Executor ID / Host	Launch Time	Duration	GC Time	Input Size / Records
0	3	0	SUCCESS	RACK_LOCAL	1 / soit-hdp-pro-6.ucc.usyd.edu.au stdout stderr	2018/04/16 16:57:32	6 s	43 ms	128.1 MB / 5091193
1	0	0	SUCCESS	NODE_LOCAL	2 / soit-hdp-pro-5.ucc.usyd.edu.au stdout stderr	2018/04/16 16:57:28	8 s	0.2 s	128.1 MB / 5023431
2	4	0	SUCCESS	RACK_LOCAL	2 / soit-hdp-pro-5.ucc.usyd.edu.au stdout stderr	2018/04/16 16:57:37	6 s	19 ms	128.1 MB / 4872155
3	5	0	SUCCESS	RACK_LOCAL	1 / soit-hdp-pro-6.ucc.usyd.edu.au stdout stderr	2018/04/16 16:57:38	6 s	18 ms	128.1 MB / 4871736
4	6	0	SUCCESS	RACK_LOCAL	2 / soit-hdp-pro-5.ucc.usyd.edu.au stdout stderr	2018/04/16 16:57:43	4 s	10 ms	79.5 MB / 3025862



References

- Sanjay Ghemawat, Howard Gobioff and Shun-Tak Leung, *The Google File System*. In Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP'03), 2003
- Tom White, Hadoop The Definitive Guide, 4th edition, O'Reilly, 2015
 - ▶ Chapter 4, YARN
 - ▶ Chapter 9, MapReduce Features
- Apache Hadoop Yarn Introduction
 - ▶ <http://hortonworks.com/blog/introducing-apache-hadoop-yarn/>
 - ▶ <http://hortonworks.com/blog/apache-hadoop-yarn-background-and-an-overview/>
- Determine YARN and MapReduce Memory configuration Settings
 - ▶ http://docs.hortonworks.com/HDPDocuments/HDP2/HDP-2.0.6.0/bk_installing_manually_book/content/rpm-chap1-11.html
- How to plan and configure YARN and MapReduce 2 in HDP
 - ▶ <http://hortonworks.com/blog/how-to-plan-and-configure-yarn-in-hdp-2-0/>



References

- Job Scheduling
 - ▶ FairScheduler: <http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/FairScheduler.html>
 - ▶ CapacityScheduler: <http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/CapacityScheduler.html>
- Sandy Ryza, How-to: Tune Your Apache Spark Jobs (part 1) published on March 09, 2015
 - ▶ <http://blog.cloudera.com/blog/2015/03/how-to-tune-your-apache-spark-jobs-part-1/>
- Sandy Ryza, How-to: Tune Your Apache Spark Jobs (part 2) published on March 30, 2015
 - ▶ <http://blog.cloudera.com/blog/2015/03/how-to-tune-your-apache-spark-jobs-part-2/>
- Spark cluster mode overview
 - ▶ <http://spark.apache.org/docs/latest/cluster-overview.html>
- Spark job scheduling
 - ▶ <http://spark.apache.org/docs/latest/job-scheduling.html>

