CMPE 282 Cloud Services
Hadoop

Instructor: Kong Li

Content

- MapReduce
- Apache Hadoop
- HDFS
  - Name space
  - Data access
  - HA – Active-standby namenode
  - HA - Data Replication
- Daemons
- YARN
- MapReduce Job
- Fault Tolerance
- Speculative Execution
- User application
- Spark
- Storm

Big Data

- Characteristics
  - Volume: MB, GB, TB, PB, etc
  - Variety: different forms or types
  - Velocity: batch, near realtime, realtime
- Search for actionable insights
  - Regardless of structured, semi-structured, or unstructured data
  - Q: How to analyze structured, semi-structured, and unstructured data?
- Evolution: Batch ➔ real time ➔ prediction
- Tools
  - Generic: NoSQL, SQL, search
  - Batch: MapReduce, Hive, Pig, etc.
  - Real time / streaming: Spark (streaming), Storm, etc
  - Machine learning: Mahout, Spark ML, etc
- Q: how to use the right tool for the job?
  - http://www.slideshare.net/AmazonWebServices/aws-november-webinar-series-architectural-patterns-best-practices-for-big-data-on-aws

MapReduce

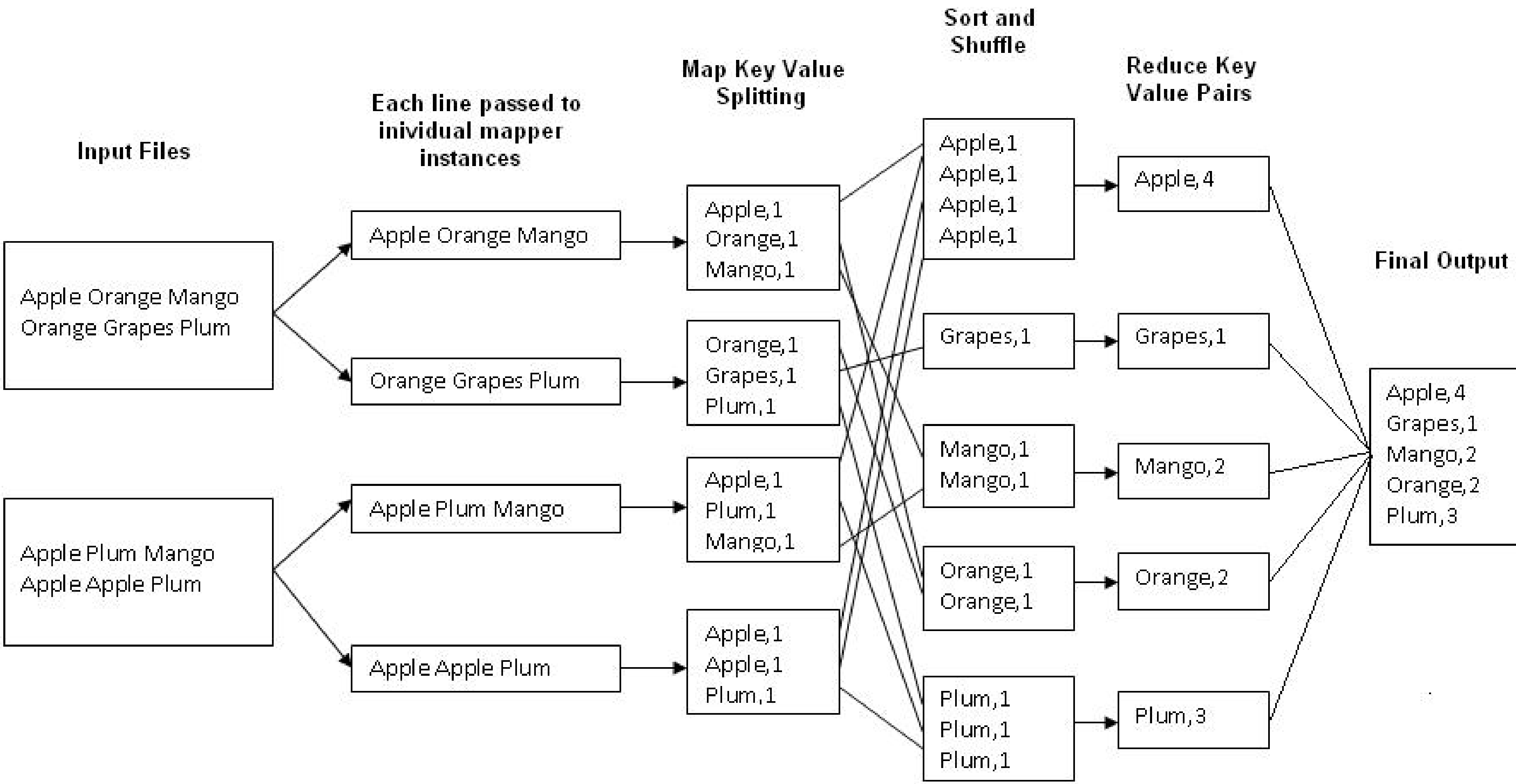
- New programming paradigm
  - Purpose: mine large datasets
    - Structured, semi-structured, unstructured
  - Implementation: computation cluster
- Two phases/stages
  - Map: applies to all the members of the dataset and returns a list of results, executed in parallel
  - Reduce: collates and resolves the results from one or more mapping operations, executed in parallel

Google calls it:	Hadoop equivalent:
MapReduce	Hadoop
GFS	HDFS
Bigtable	Hbase
Chubby	Zookeeper

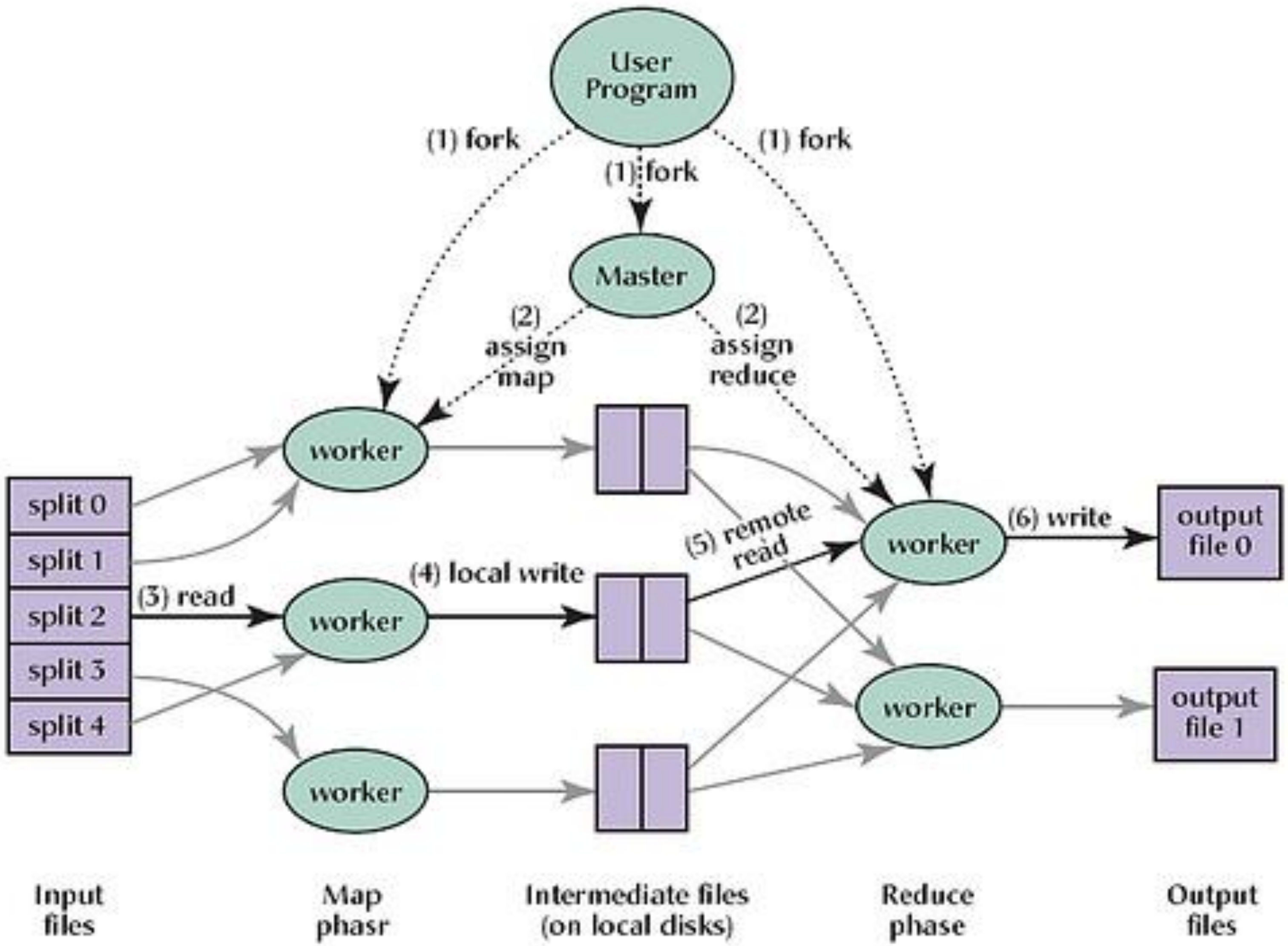
MapReduce (cont'd)

- Details
  - Large datasets are split into subsets called splits
  - Map: parallelized operation: splits → intermediate results
    - map(k1, v1) applied to each split → list of (k2, v2)
  - Shuffle/sort intermediate results into new splits
    - Group list of (k2, v2)'s by k2 → list of (k2, list(v2))
  - Reduce: parallelized operation: intermediate results → final result
    - reduce(k2, list(v2)) → v3
  - All values are processed *independently*
  - final results: same as sequential execution on the entire dataset (assuming deterministic map/reduce)
- Separate app (business logic) from multi-processing logic
  - MapReduce framework: dispatching, scalability, locking, and logic flow
  - App: the business logic w/o worrying about infrastructure or scalability issues

MapReduce: Word Count

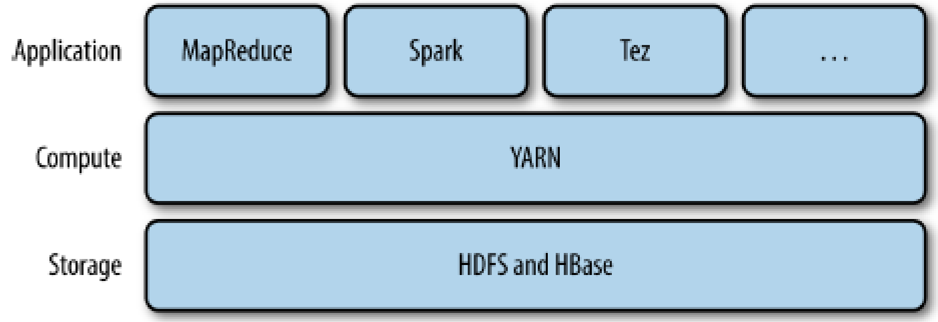


MapReduce (cont'd)

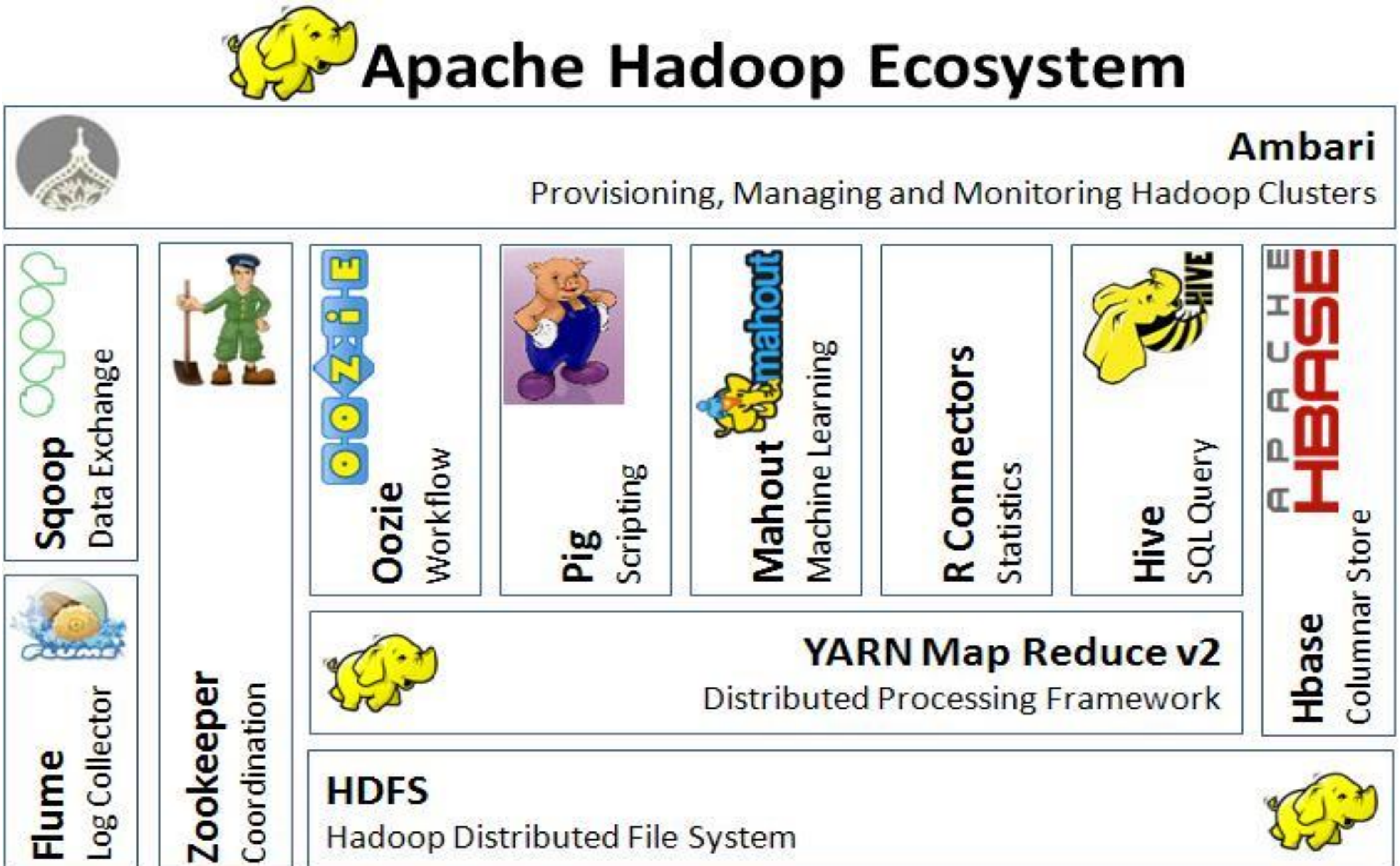


Hadoop

- Distributed computing framework
- Large scale: HDFS + YARN + MapReduce
  - HDFS: scalable storage layer
  - YARN: cluster compute layer, cluster resource mgmt
  - MapReduce app: map/reduce tasks on cluster of machines
    - New programming model: computation on top of set of key/value pair
    - Batch processing: A job runs until a given set of data has been processed
- For semi-structured, unstructured, as well as structured data
  - Can use RDBMS (w/ SQL) and NoSQL together
- Built-in resiliency, fault tolerance
- Batch → streaming/real time?
  - Apache Spark
  - Apache Storm



Apache Hadoop Ecosystem



Apache Hadoop Ecosystem (cont'd)

- **HDFS**: scalable distributed file system
- **YARN**: framework for job scheduling and cluster resource mgmt
- **MapReduce**: YARN-based system for parallel processing of large data sets
- HBase: scalable, distributed NoSQL DB on HDFS for large tables
- Hive: data warehouse infrastructure providing data summarization and ad hoc querying
- Cassandra: scalable, multi-master, HA, NoSQL DB
- Pig: high-level data-flow language and execution framework for parallel computation
- ZooKeeper: a coordination service for distributed apps
- Oozie: workflow scheduler system for MR jobs using DAGs
- Ambari: provisioning, managing, and monitoring Apache Hadoop clusters
- Mahout: scalable machine learning and data mining library
- **Spark**: distributed framework for in-memory analytics on large data sets; co-exist w/ Hadoop - SQL, machine learning, *stream* processing, and graph computation
- **Storm**: distributed real-time computation framework for event stream processing – real-time analytics, online machine learning, continuous computation, ETL

HDFS

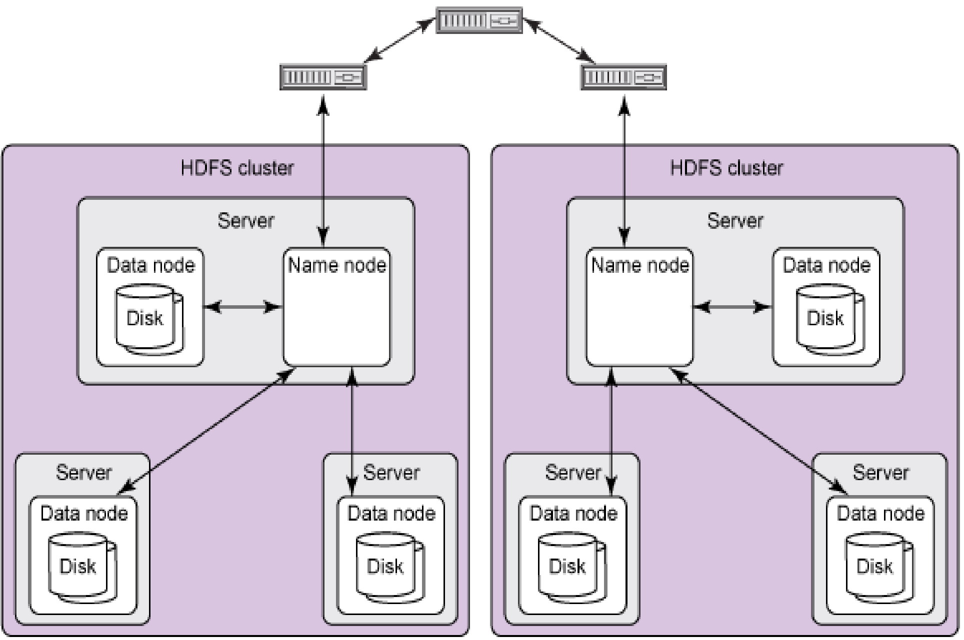
- A distributed file system: makes tradeoffs that are good for MapReduce
  - Single virtual file system spread over many machines
  - Traditional hierarchical file organization: directory, file
- Goals:
  - Simple Coherency Model: write-once-read-many access model for files
  - Large data sets
  - Moving computation is cheaper than moving data
- Good for:
  - Very large read-only or append-only files (individual file size GB or TB)
  - Sequential access patterns
- Not so good for:
  - Storing lots of small files
  - Low-latency access
  - Multiple writers
  - Writing to arbitrary offsets in the file
- Optimized for sequential read and local accesses

HDFS: namespace

- Name space: separated from FS of OS
  - Files are stored as sets of (large) blocks on top of FS of OS
    - Blocks are replicated for durability and availability (discussed later)
  - Files in HDFS are not visible in the normal FS
    - `hadoop fs -ls`
  - Only the blocks and the block metadata are visible in FS of OS
  - Also support 3rd party FS e.g., CloudStore and Amazon S3

```
[ahae@carbon ~]$ ls -la /tmp/hadoop-ahae/dfs/data/current/
total 209588
drwxrwxr-x 2 ahae ahae     4096 2013-10-08 15:46 .
drwxrwxr-x 5 ahae ahae     4096 2013-10-08 15:39 ..
-rw-rw-r-- 1 ahae ahae 11568995 2013-10-08 15:44 blk_-3562426239750716067
-rw-rw-r-- 1 ahae ahae    90391 2013-10-08 15:44 blk_-3562426239750716067_1020.meta
-rw-rw-r-- 1 ahae ahae        4 2013-10-08 15:40 blk_5467088600876920840
-rw-rw-r-- 1 ahae ahae       11 2013-10-08 15:40 blk_5467088600876920840_1019.meta
-rw-rw-r-- 1 ahae ahae 67108864 2013-10-08 15:44 blk_7080460240917416109
-rw-rw-r-- 1 ahae ahae   524295 2013-10-08 15:44 blk_7080460240917416109_1020.meta
-rw-rw-r-- 1 ahae ahae 67108864 2013-10-08 15:44 blk_-8388309644856805769
-rw-rw-r-- 1 ahae ahae   524295 2013-10-08 15:44 blk_-8388309644856805769_1020.meta
-rw-rw-r-- 1 ahae ahae 67108864 2013-10-08 15:44 blk_-9220415087134372383
-rw-rw-r-- 1 ahae ahae   524295 2013-10-08 15:44 blk_-9220415087134372383_1020.meta
-rw-rw-r-- 1 ahae ahae      158 2013-10-08 15:40 VERSION
```

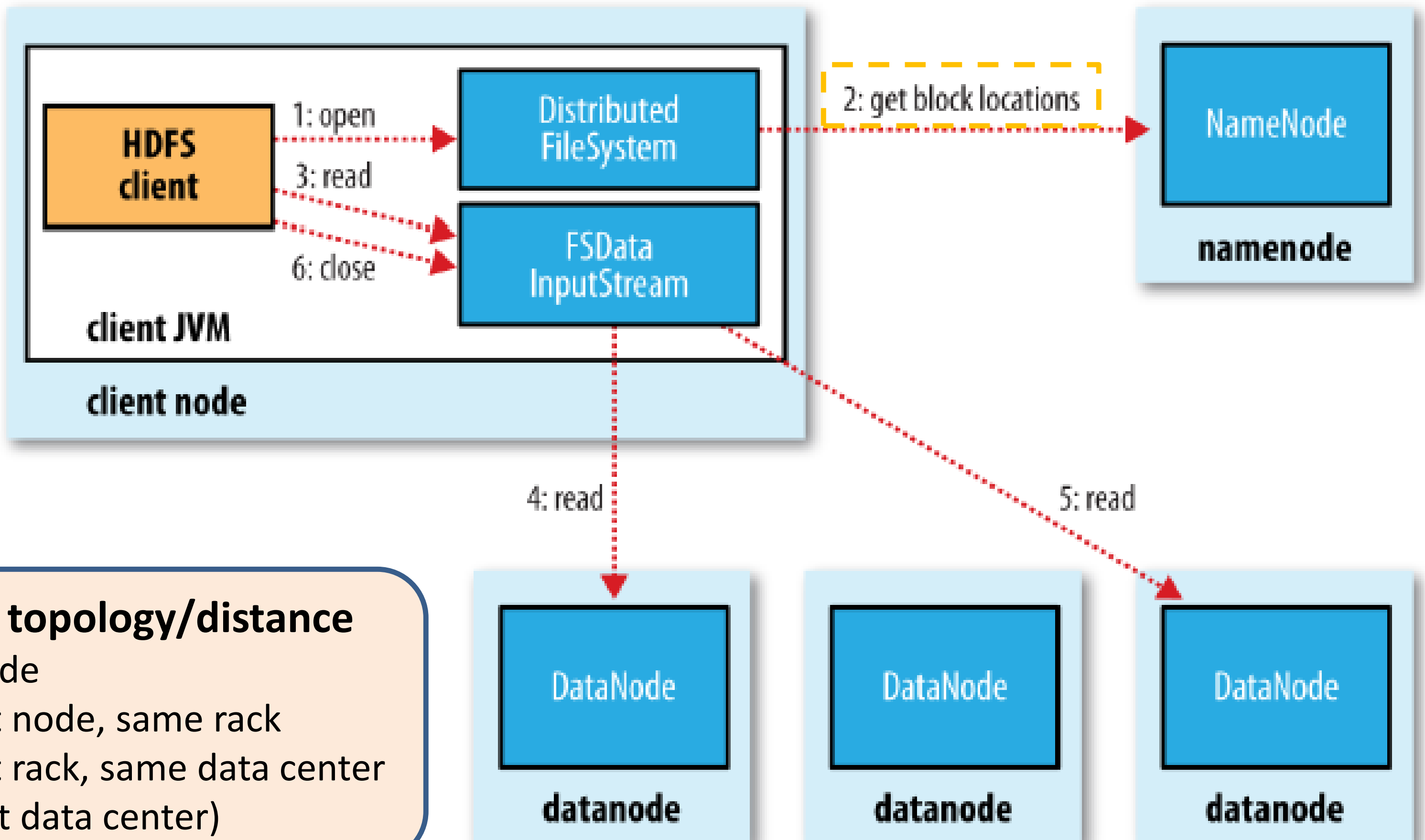
HDFS: Data Access



- File == multiple blocks
  - By default, block size = 128 MB
  - Blocks of a given file spread across multiple datanodes
  - `hdfs fsck / -files -blocks`
- Namenode (master)
  - manages FS namespaces: FS tree, metadata of all files/directories
    - persist info in namespace image, edit log
  - Caches datanodes containing which blocks for files
- Datanode (worker)
  - Read/write blocks: requests from clients and namenode
  - Actual data transfer is directly between client & datanode
  - Periodically report blocks to namenode

Secondary namenode
- Not a *real* namenode
- Not for HA
- periodically merge namespace image w/ edit log to prevent the edit log from becoming too large

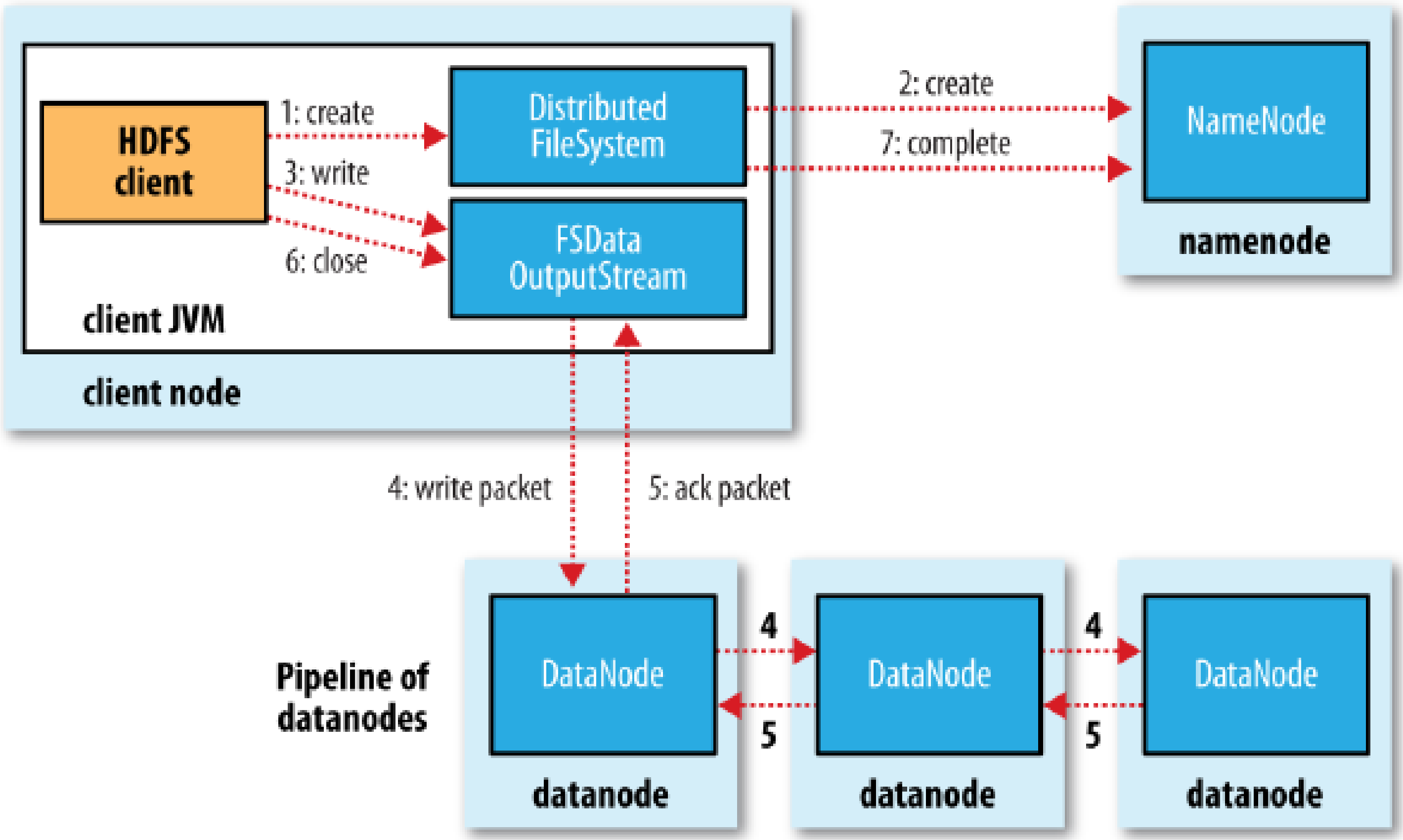
HDFS: File Read



Network topology/distance
- Same node
- Different node, same rack
- Different rack, same data center
- (different data center)

- Step 2
  - Response: datanodes are sorted according to their proximity to the client
  - Client reads local datanode if given blocks are stored locally, or from **closest** datanode

HDFS: File Write



(Assume replication level = 3)

Rack-aware replica placement (later)

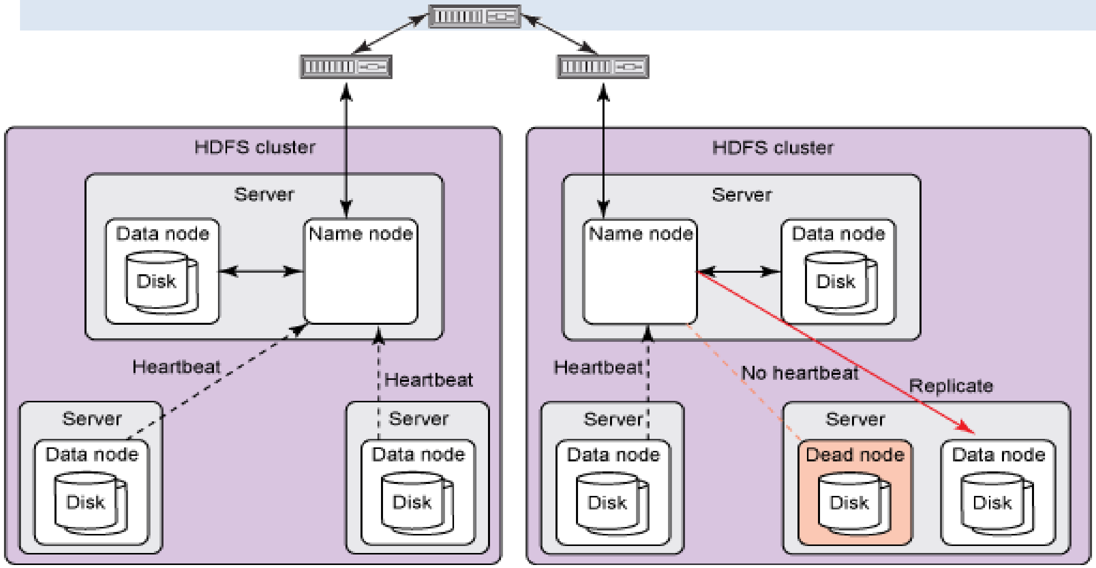
HDFS: Data Access from Java

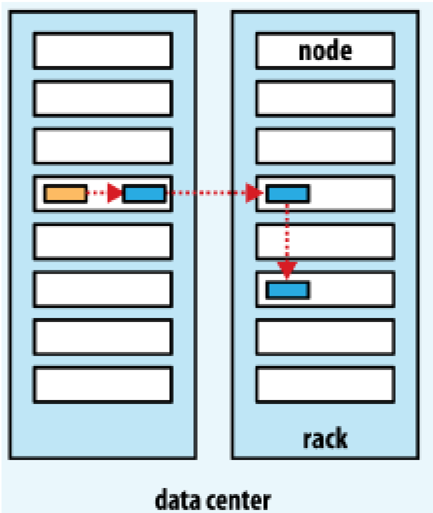
- Programs can read/write HDFS files directly
  - Not needed in MapReduce; I/O is handled by the framework
- Files: represented as URIs
  - Example: hdfs://localhost/foo/bar/example.txt
- Access is via the FileSystem API
  - To get access to the file: FileSystem.get()
  - For reading, call open() -- returns InputStream
  - For writing, call create() -- returns OutputStream

HDFS: HA - Active-Standby NN

- Namenode – single point of failure
- HDFS HA
  - Active-standby namenodes: since Hadoop 2
  - Data Replication
- Active-standby namenodes
  - Two namenodes in an active-standby configuration
  - Namenodes use highly available shared storage to share the edit log
  - Block mappings stored in namenode's memory (not on disk) → datanodes send block reports to both namenodes
  - Clients configured to handle namenode failover transparently
  - The secondary namenode's role is subsumed by the standby namenode
  - Failover controlled by ZooKeeper

HDFS: HA - Data Replication





- Namenode
  - receives periodical heartbeats from datanodes
  - makes all decisions regarding replication of blocks

Tradeoffs

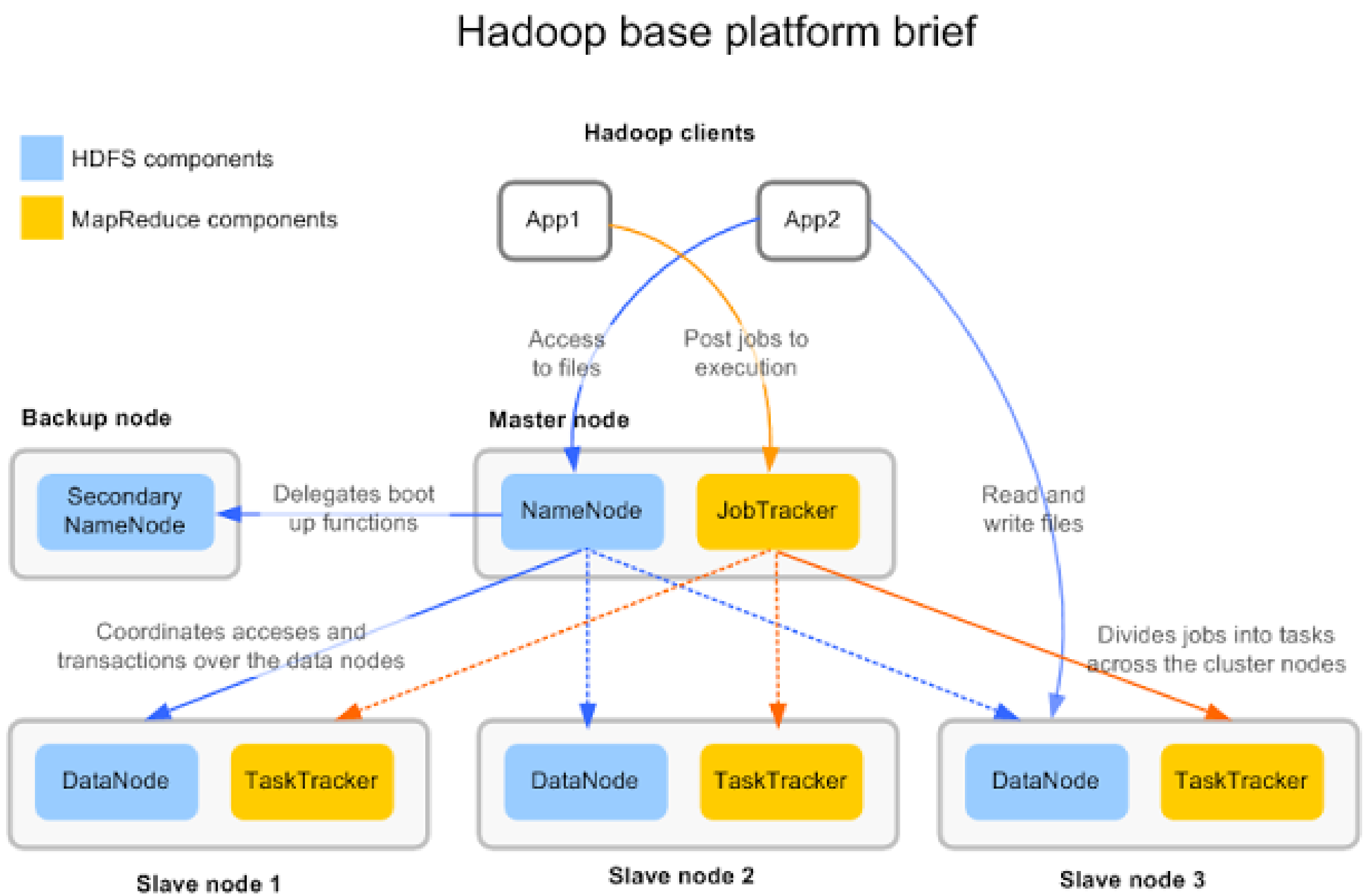
- Rack-aware replica placement policy: fault tolerance, locality/performance
  - 1st replica: same node as the client
  - 2nd replica: a off-rack node
  - 3rd replica: the same rack as the second, but on a different node
  - Others: random nodes in the cluster, but avoid placing too many replicas on the same rack

Daemons

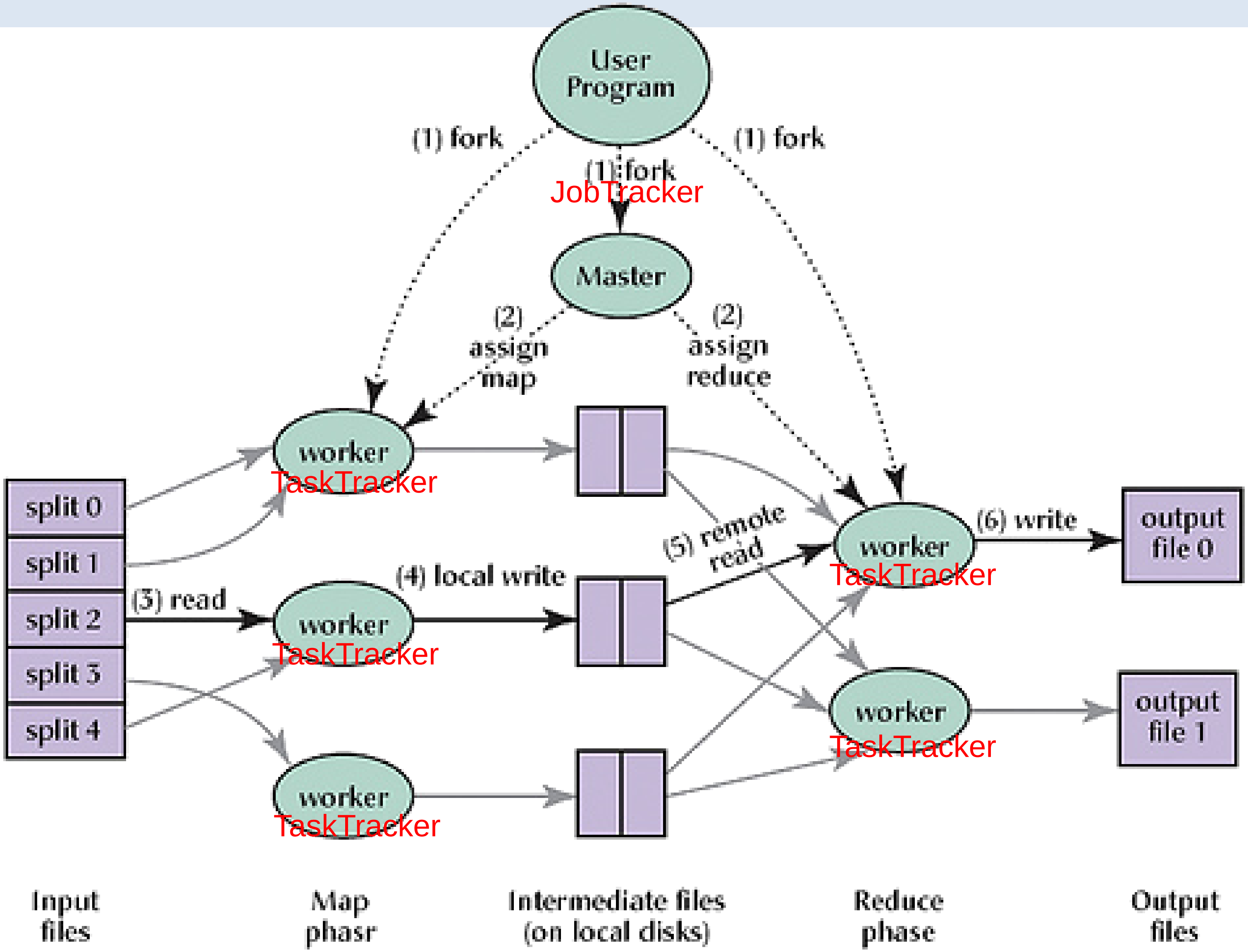
- HDFS
  - Namenode: manages FS namespaces
  - Datanode: stores HDFS blocks
  - v1: Secondary namenode (not for HA); v2: standby namenode
- MapReduce
  - JobTracker: one per cluster
    - Coordinates all jobs; schedules tasks to TaskTrackers
    - Monitors progress of TaskTrackers; restarting failed or slow tasks (discussed later)
  - TaskTracker: one per node
    - Runs map tasks and reduce tasks
    - Sends progress report to jobTracker
- YARN or MapReduce 2
  - JobTracker → global resource manager, per-app application master, timeline server
  - TaskTracker → node manager
  - Multi-tenancy: MapReduce app, Spark app, etc.
- A server may host multiple daemons

Hadoop Cluster

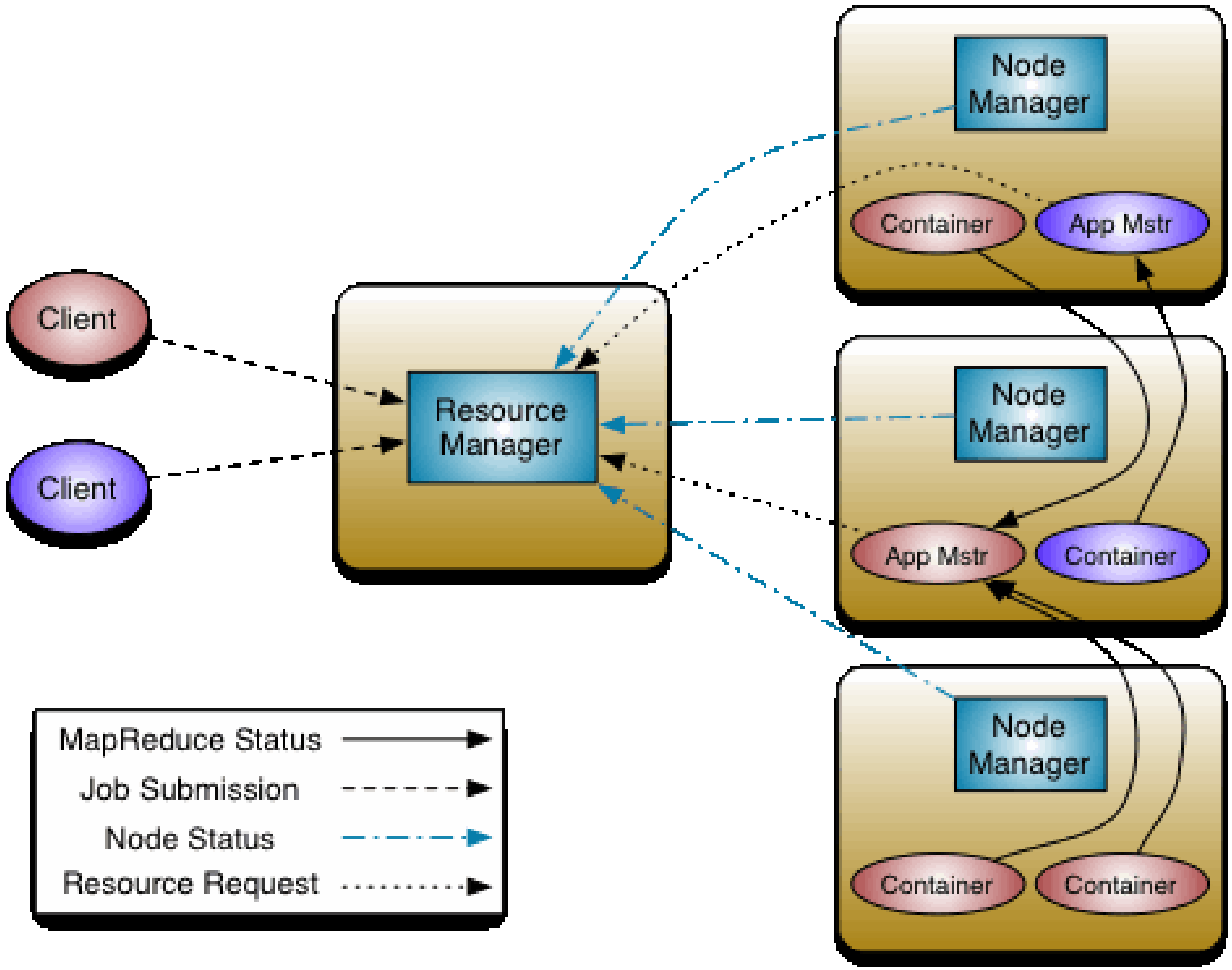
For more information visit me at
www.hadooper.blogspot.com



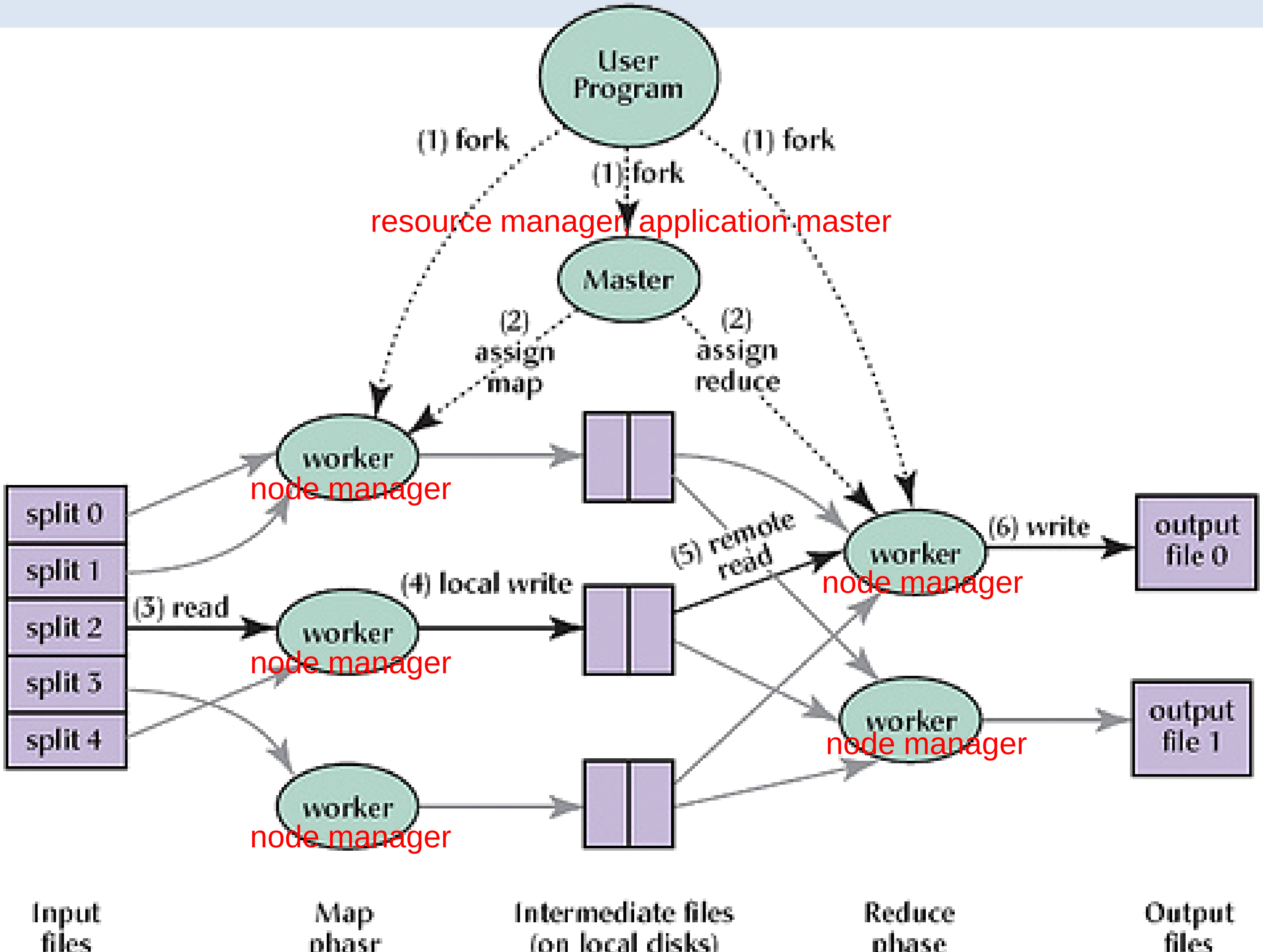
MapReduce


User Program
(1) fork
(1) fork
(1) fork
JobTracker
Master
(2) assign map
(2) assign reduce
worker
TaskTracker
worker
TaskTracker
worker
TaskTracker
worker
TaskTracker
worker
TaskTracker
split 0
split 1
split 2
split 3
split 4
(3) read
(4) local write
(5) remote read
(6) write
output file 0
output file 1
Input files
Map phasr
Intermediate files (on local disks)
Reduce phase
Output files

Hadoop Cluster (YARN)

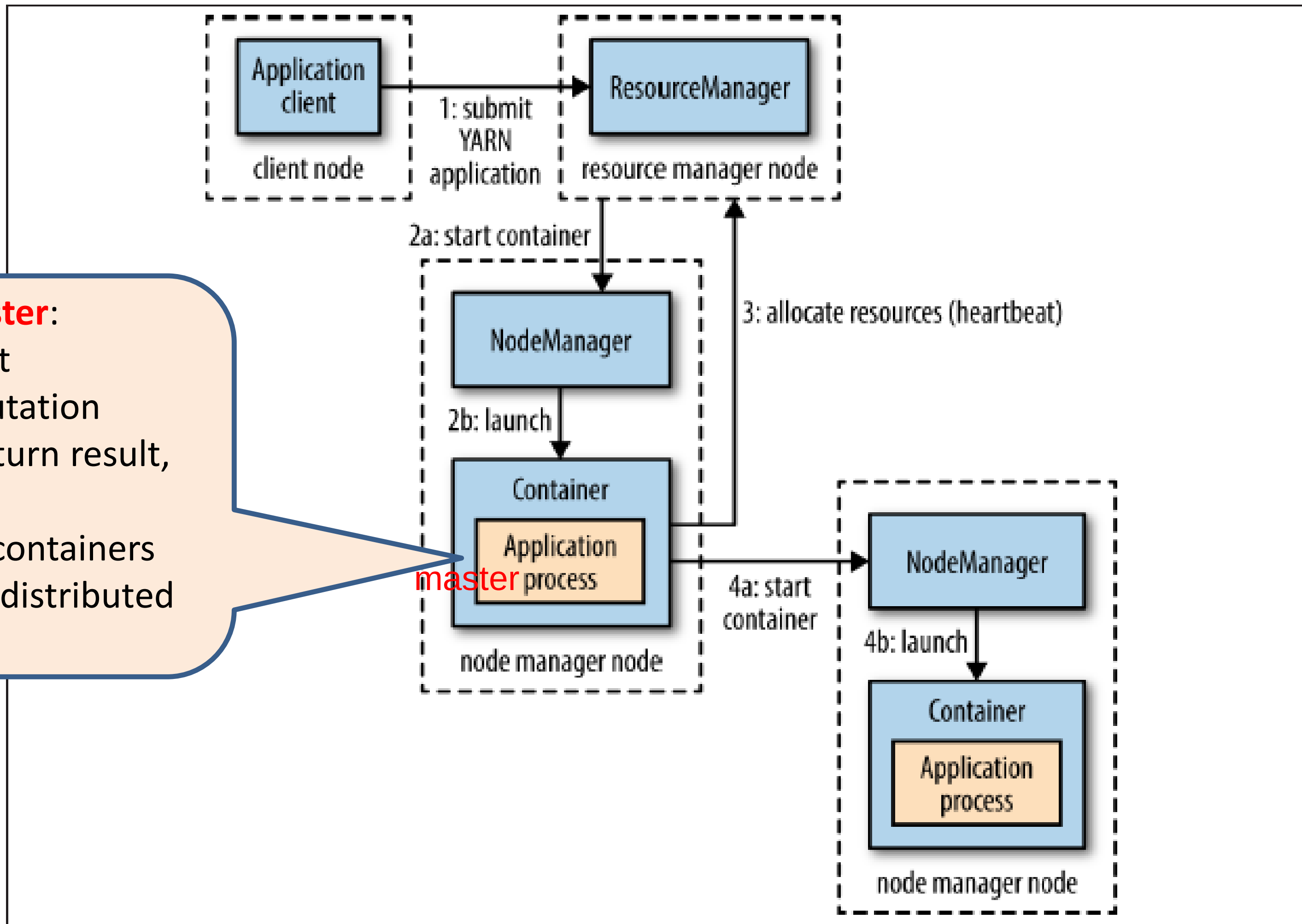


MapReduce2 / YARN



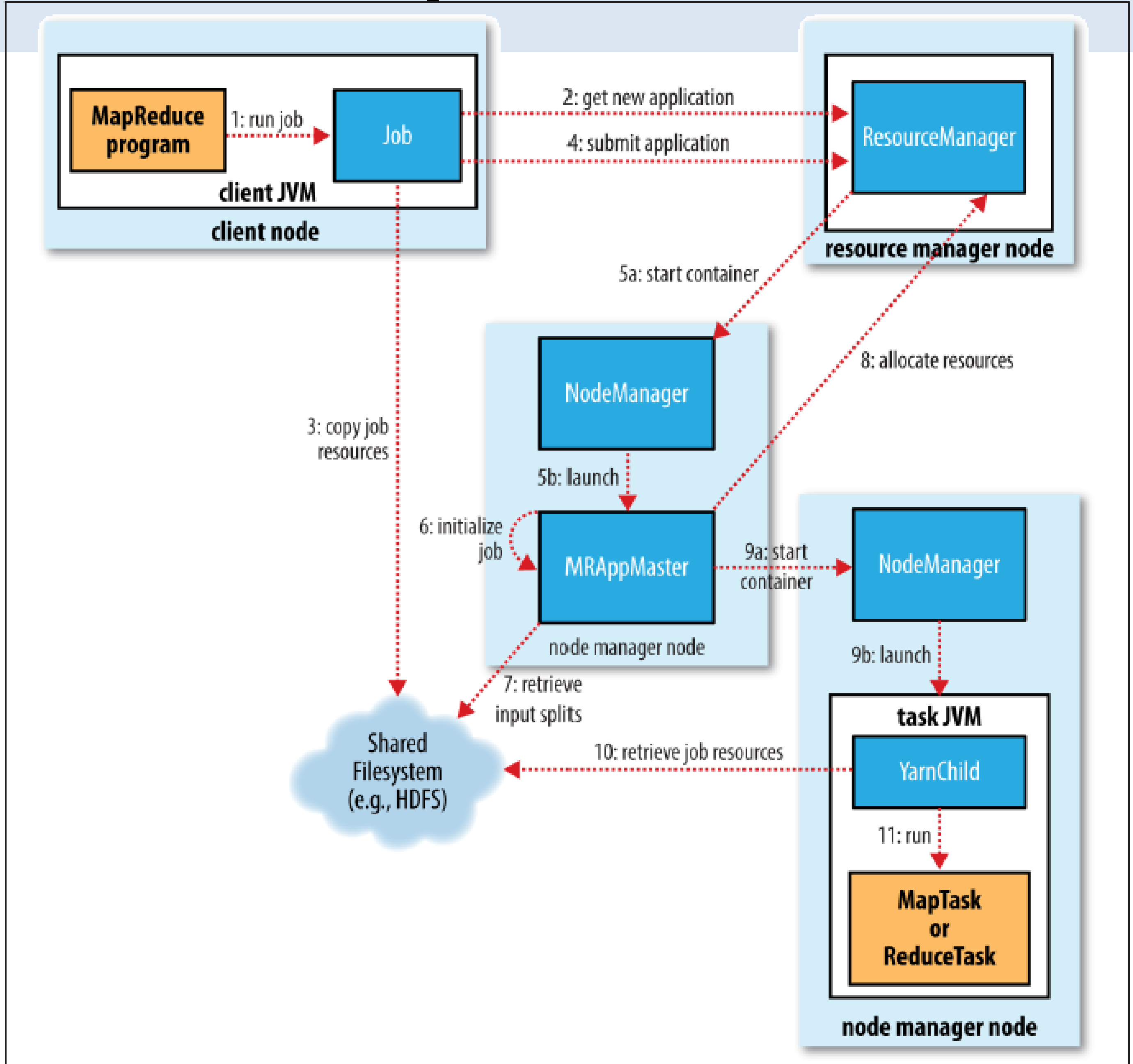
YARN: Application Run

- For any YARN application: include MapReduce, Spark, etc.

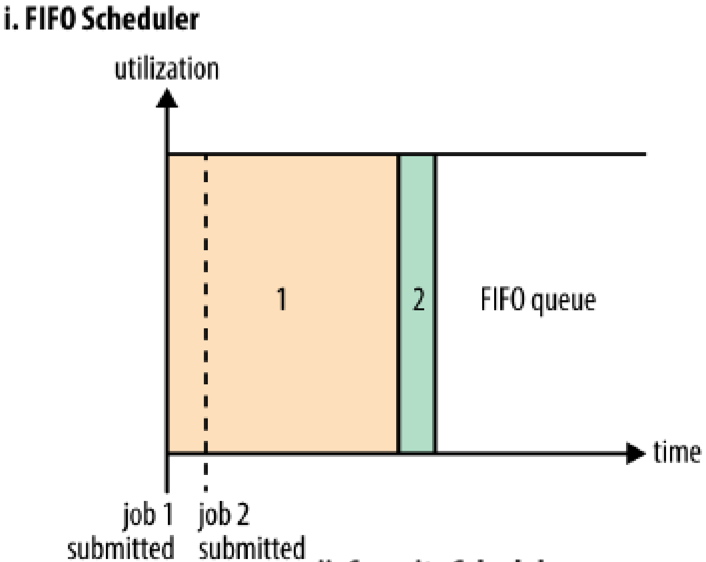


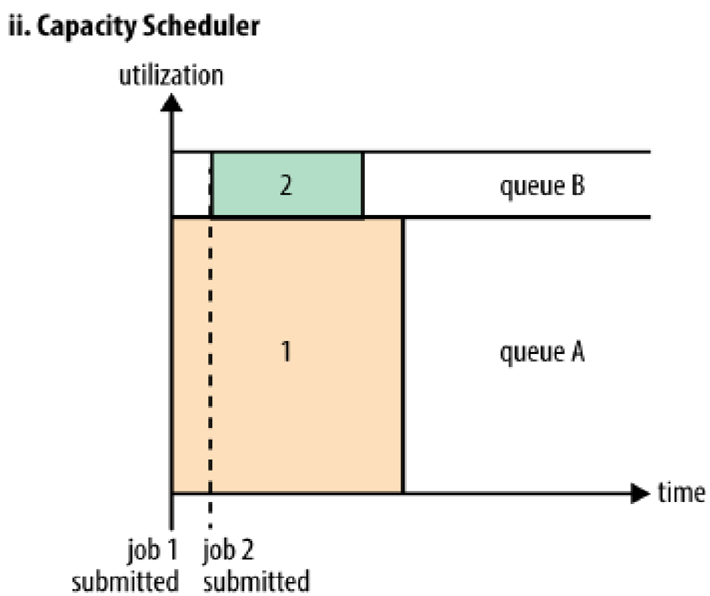
Application master:
- App dependent
- Can run computation directly and return result, OR
- Can req more containers (step 3) to run distributed app (step 4)

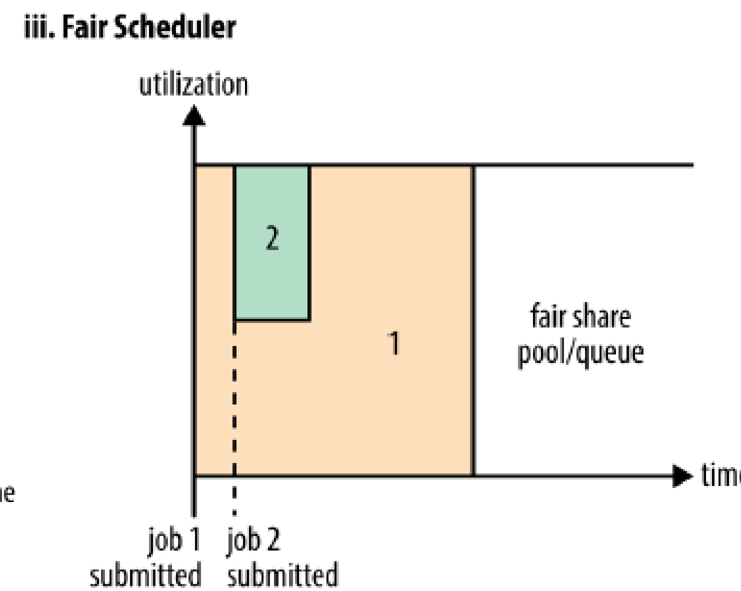
YARN: MapReduce Job Run



YARN: Schedulers

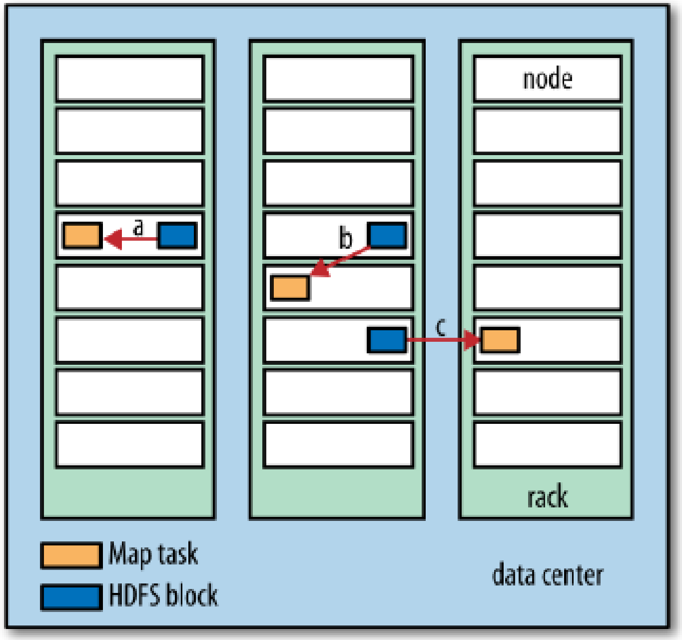






MapReduce Job

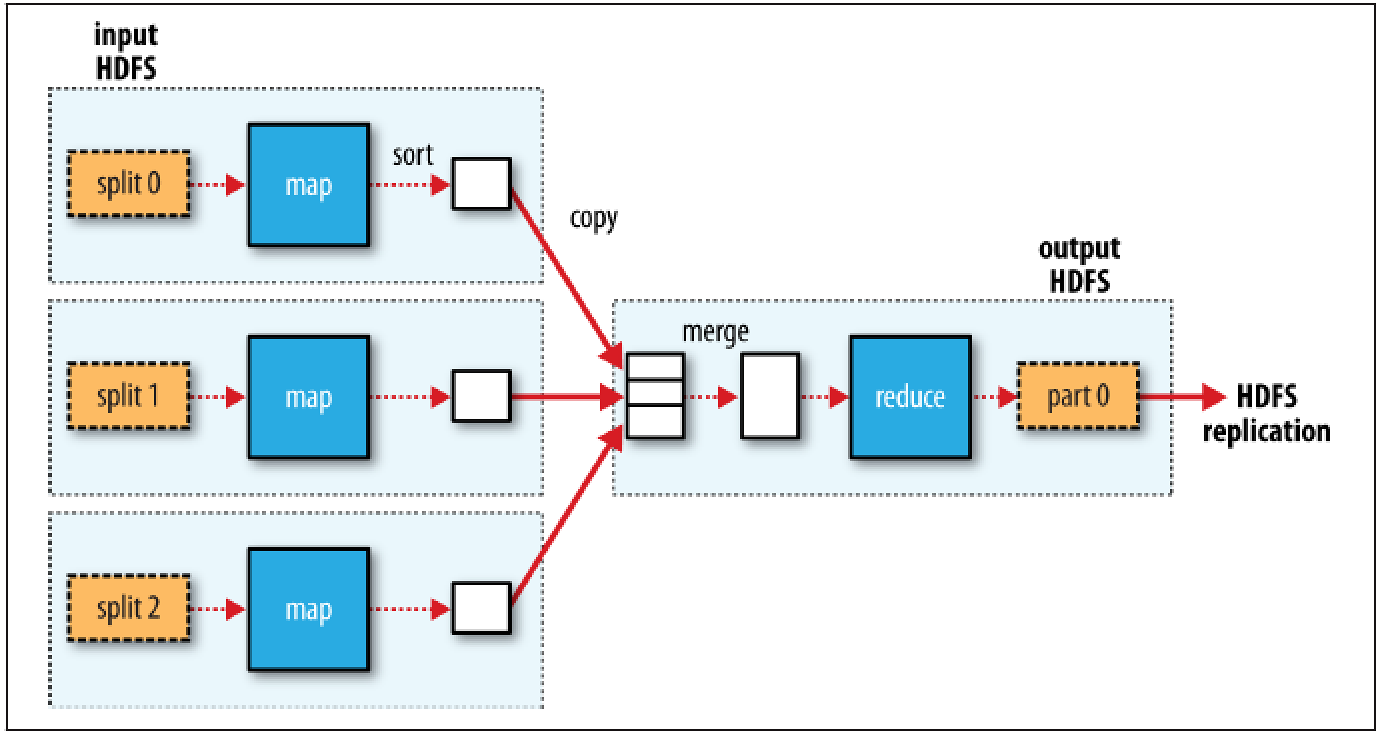
- Input data: divided into multiple *splits*
  - By default, 128 MB (HDFS block size)
- Map task: (K1, V1) → list(K2, V2)
  - Input: HDFS
    - Data locality: data-local, rack-local, off-rack
  - Output: local disk (why?)
  - One map task per split
    - Run map function for each *record in the split*
- Reduce task: (K2, list(V2)) → list(K3, V3)
  - Input: shuffle-and-sorted map output's intermediate key-value pairs by key
    - Each reduce task can be fed by many map tasks
  - Output: HDFS
  - # of reduce tasks: specified independently
    - Nothing to do with input data size
    - In driver by default, job.setNumReduceTasks(1);
    - OK to have 0 reduce task



Trade-offs:
More splits → more map tasks, more parallelism, higher overhead
Less splits → less map tasks, more sequentiality, less overhead

Data Flow: Combiner Function

- Combiner function: (K2, list(V2)) → list(K2, V2)
  - Optional, part of map phase
  - Often the combiner and reduce functions are the same
  - Why? minimize the data transferred between map and reduce tasks
  - Same MapReduce output even when no combiner function



Data Flow: Combiner Function (cont'd)

- Example: Find highest temperature in 1950s

	Without Combiner Function	With Combiner Function
Map output (two map tasks)	• (1950, 0), (1950, 20), (1950, 10) • (1950, 25), (1950, 15)	• (1950, 20) • (1950, 25)
Reduce input (one reduce task)	(1950, [0, 20, 10, 25, 15])	(1950, [20, 25])
Reduce output	25	25

  – max(0, 20, 10, 25, 15) = max(max(0, 20, 10), max(25, 15)) = max(20, 25) = 25

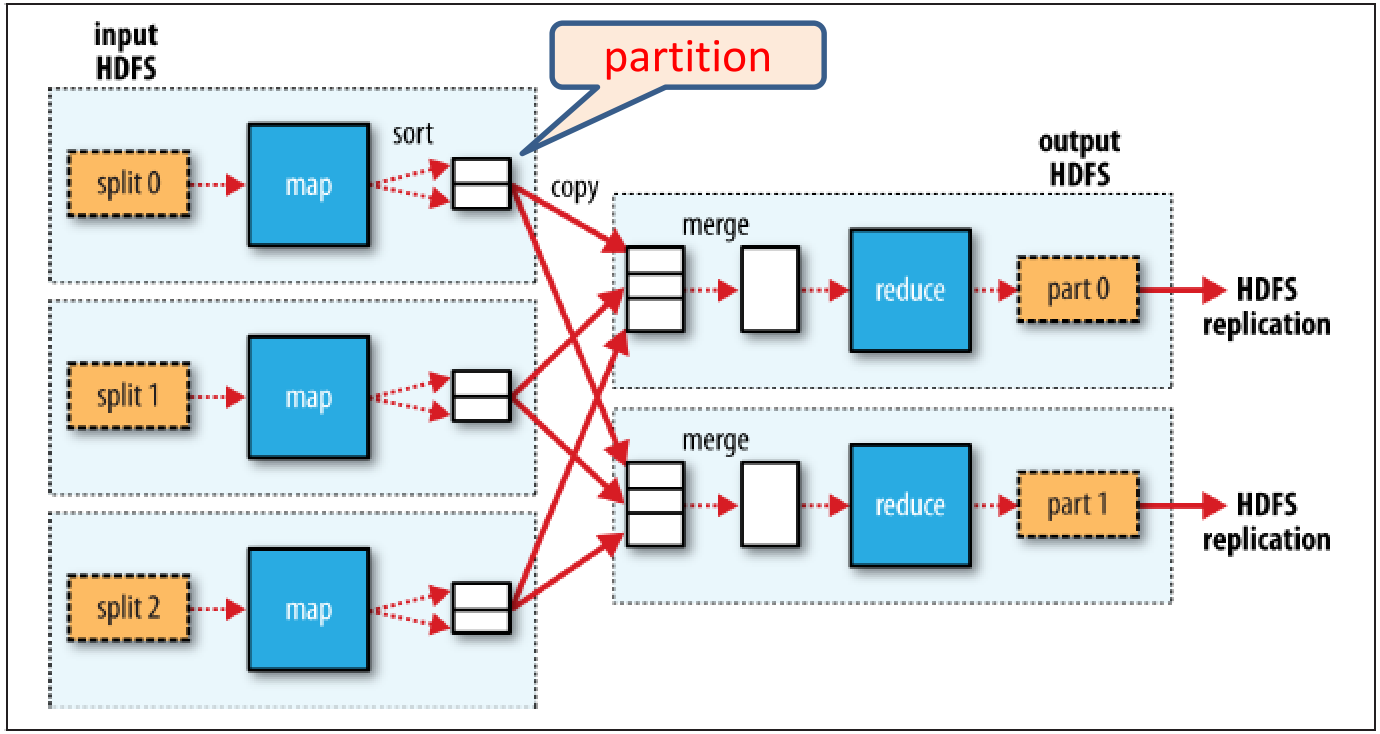
- Note: not all functions can be done in this way
  – Ex: mean(0, 20, 10, 25, 15) = 14

    But mean(mean(0, 20, 10), mean(25, 15)) = mean(10, 20) = 15

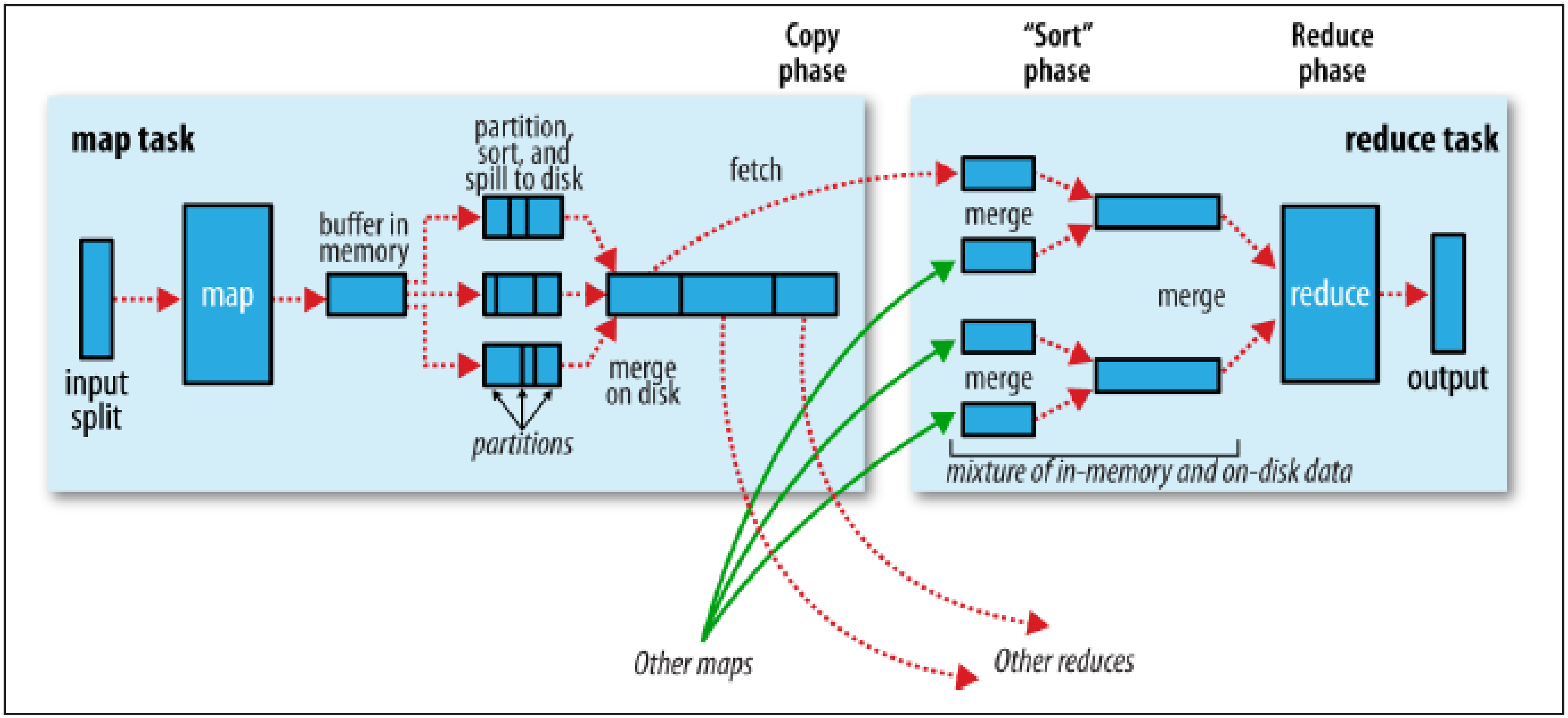
  – Workaround?

Data Flow: Partition Function

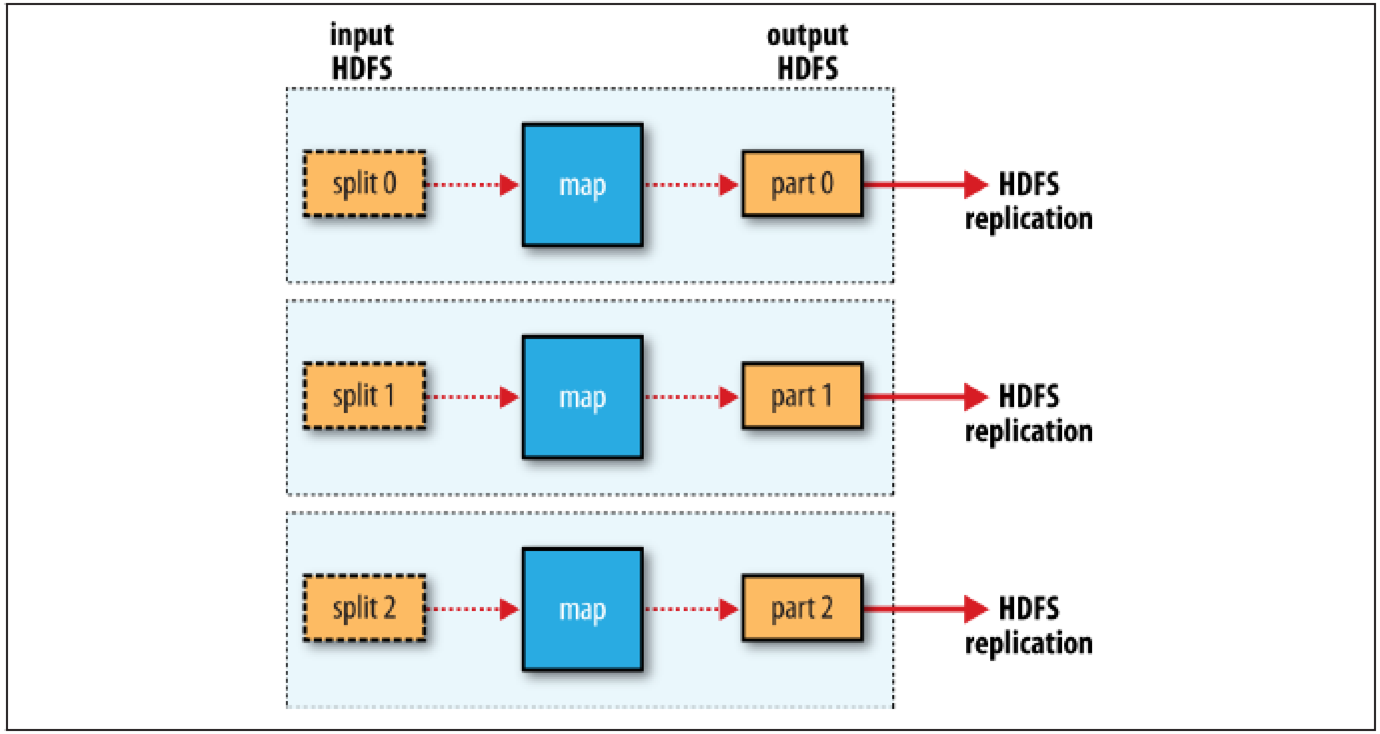
- Partition function: (K2, V2) → integer
  - For > 1 reduce tasks
  - Optional – can overwrite the default one
  - Partition a single map task output to multiple reduce tasks
  - Records of any given key are in a single partition



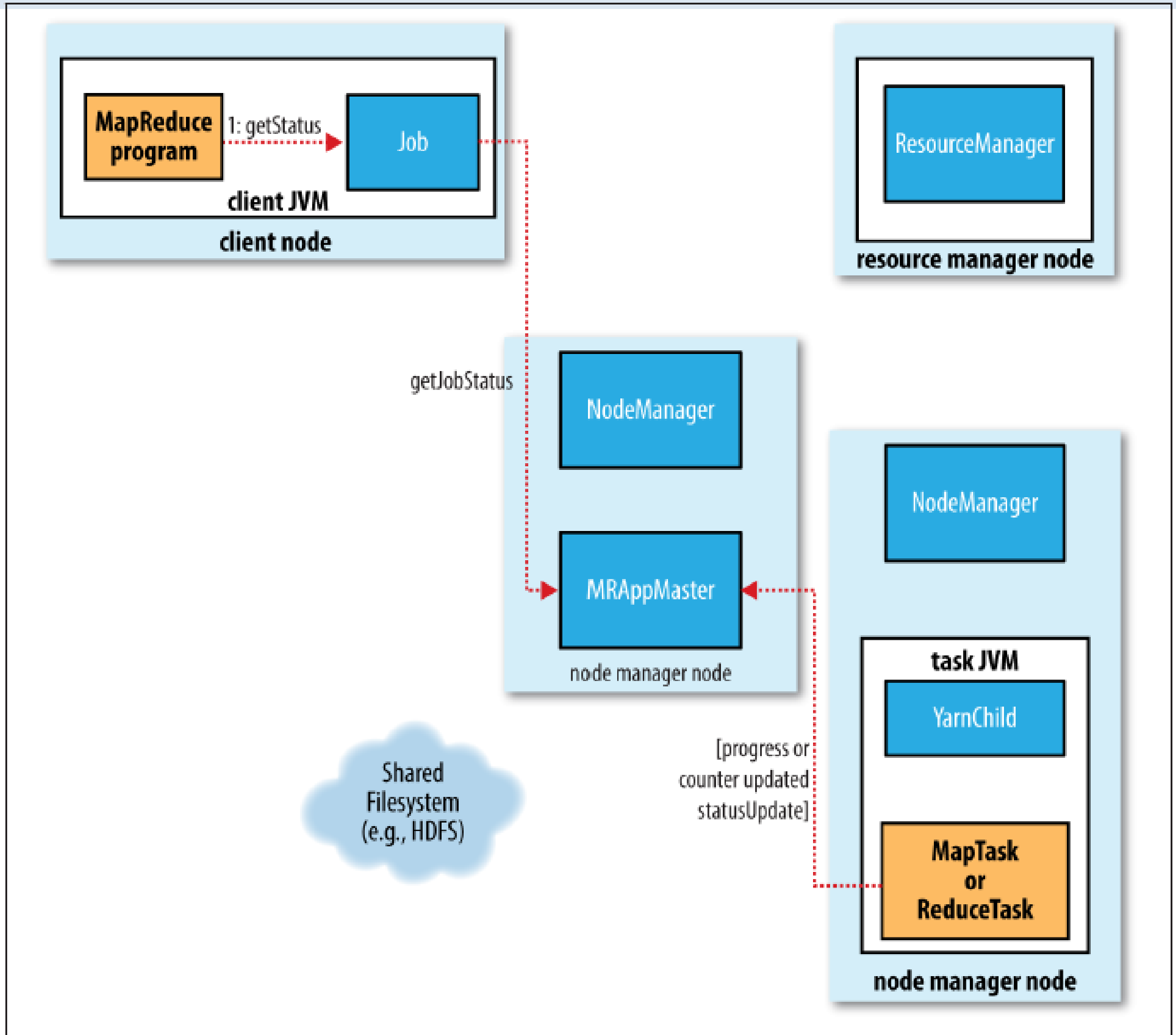
Data Flow: Shuffle-and-Sort



Data Flow: no reduce task



MapReduce Job Status Update



FT – MR Task Failure

- Problem: what if a MapReduce task fails during a job
- Solution:
  - JobTracker (application master) notices that runtime exception, JVM exit, or timeout of progress update from the node's TaskTracker (node manager)
  - JobTracker (application master) re-schedules the failed node's task
    - Map: re-execute *all* map tasks assigned to the node
    - Reduce: re-execute *all* reduce tasks assigned to the node
      - Good enough?
      - May also need to re-execute *all* map tasks on the failed node
        - » Failed node may also have completed map tasks, and other nodes may not have finished copying out the results

FT – Application Master Failure

- Problem: what if an application master fails
- Solution:
  - Application master periodically heartbeats resource manager
  - When timeout on heartbeat, resource manager starts a new instance of the failed master in a new container
    - If it is MapReduce master failure, use the job history (from timeline server) to recover the state of tasks; no need to rerun the entire app

FT – Node Manager Failure

- Problem: what if an node manager fails
- Solution:
  - Node manager periodically heartbeats resource manager
  - When timeout on heartbeat, resource manager
    - Removes the failed node manager from the pool of nodes to schedule containers on
    - Recover any task or application master running on the failed node manager
      - For incomplete MapReduce job, application master re-runs map tasks that were completed successfully on the failed node manager (**why?**)

FT – Resource Manager Failure

- Problem: what if a resource manager fails
- Solution:
  - Run two resource managers in active-standby configuration
  - Standby resource manager
    - recovers info of all apps from HA state store (backed by ZooKeeper or HDFS)
    - restarts all apps in the cluster
  - Clients and node managers must be configured to handle resource manager failover

Speculative Execution

- Problem: a slow task would delay the entire job
- Tasks are executed *in isolation* from one another → the same input split can be processed by multiple nodes *in parallel*
- Solution:
  - When a job is almost done, schedule *redundant* copies of remaining tasks across several *idle* nodes
  - The one finishes first becomes the definitive copy; the others' tasks are killed (results are discarded)
- Note
  - Is an optimization, not required to make it more reliable
  - Can independently apply to map tasks and reduce tasks
  - Different from “running duplicated tasks at the same time”
    - Waste cluster resource

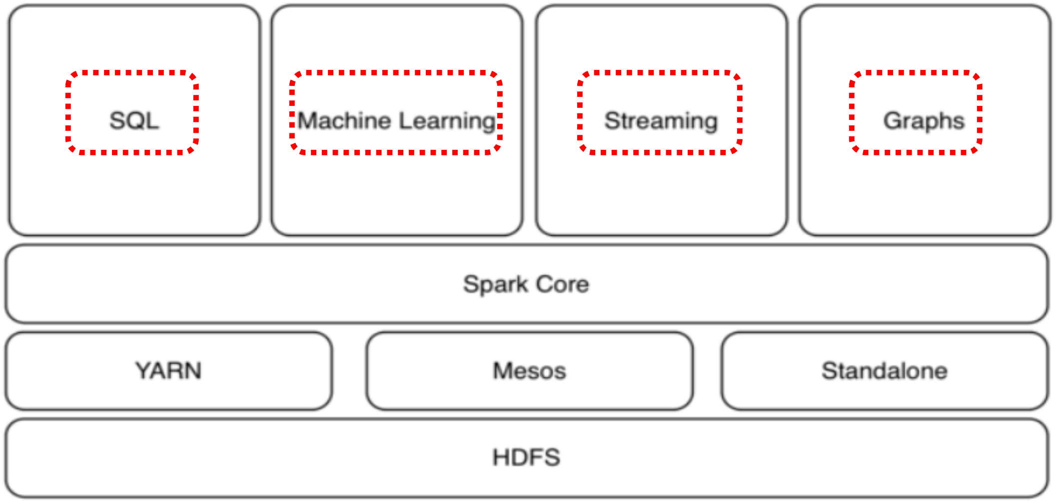
MapReduce User Applications

- User provides the codes (Java, scripts, etc) for mapper, reducer, and a driver
- The rest is handled by the framework

- What problems are MapReduce good for?
  - Good for single-pass algorithm w/ filter/aggregate stages
  - Complicated one may need multi-map/reduce stages
- Are there problems one cannot solve efficiently with MapReduce?
- Are there problems it can't solve at all?

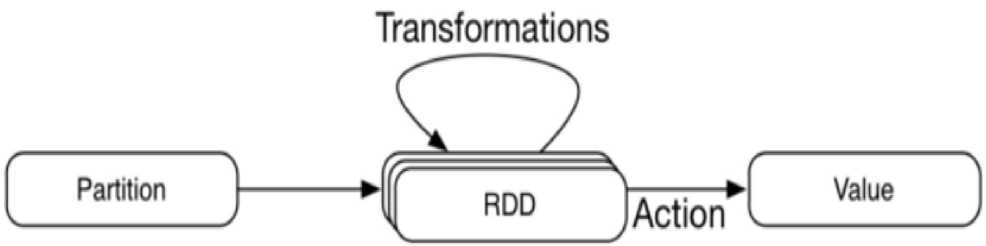
Apache Spark

- In-memory analytics on large data sets (vs. read/write files in MapReduce)
- Good for
  - interactive analysis - series of ad hoc exploratory queries by a user on dataset
  - Iterative algorithms – functions applied to data set repeatedly
- Alternative data processing engine: Directed Acyclic Graph (DAG) engine
  - Data in RAM: better performance than Hadoop MapReduce
  - Does *not* replace Hadoop: standalone, or inside Hadoop cluster (a YARN app)
  - Does *not* have own storage / file system: can use Hadoop (HDFS, HBase, etc), or other (cloud) data platforms

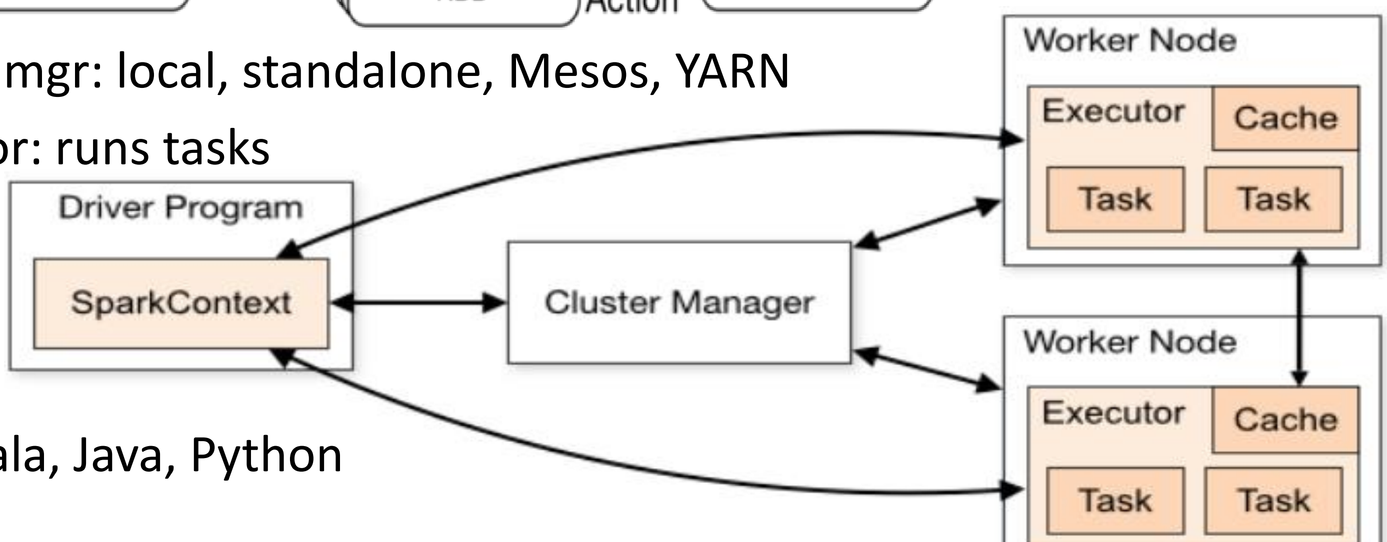


Apache Spark (cont’d)

- Resilient Distributed Datasets (RDD)
  - partitioned in multiple nodes; in-memory *between* jobs
  - Immutable (read-only): can generate new one easily; remember dependencies
  - Lazy transformation: no results computed until an action requires the results
  - recovery capability: re-apply same transformation to RDD



- Cluster mgr: local, standalone, Mesos, YARN
- Executor: runs tasks



- API: Scala, Java, Python

Apache Spark (cont'd)

```
$ spark-shell
> val lines = sc.textFile("input/ncdc/micro-tab/sample.txt")
> val records = lines.map(_.split("\t"))
> val filtered = records.filter(rec => (rec(1) != "9999“ && rec(2).matches("[01459]")))
> val tuples = filtered.map(rec => (rec(0).toInt, rec(1).toInt))
> val maxTemps = tuples.reduceByKey((a, b) => Math.max(a, b))
> maxTemps.foreach(println(_))
> maxTemps.saveAsTextFile("output")
$ hadoop fs -cat output/*
$ spark-shell
>:paste
sc.textFile("input/ncdc/micro-tab/sample.txt")
  .map(_.split("\t"))
  .filter(rec => (rec(1) != "9999" && rec(2).matches("[01459]")))
  .map(rec => (rec(0).toInt, rec(1).toInt))
  .reduceByKey((a, b) => Math.max(a, b))
  .saveAsTextFile("output2")
^D (i.e., control-D)
$ hadoop fs -cat output2/*
```

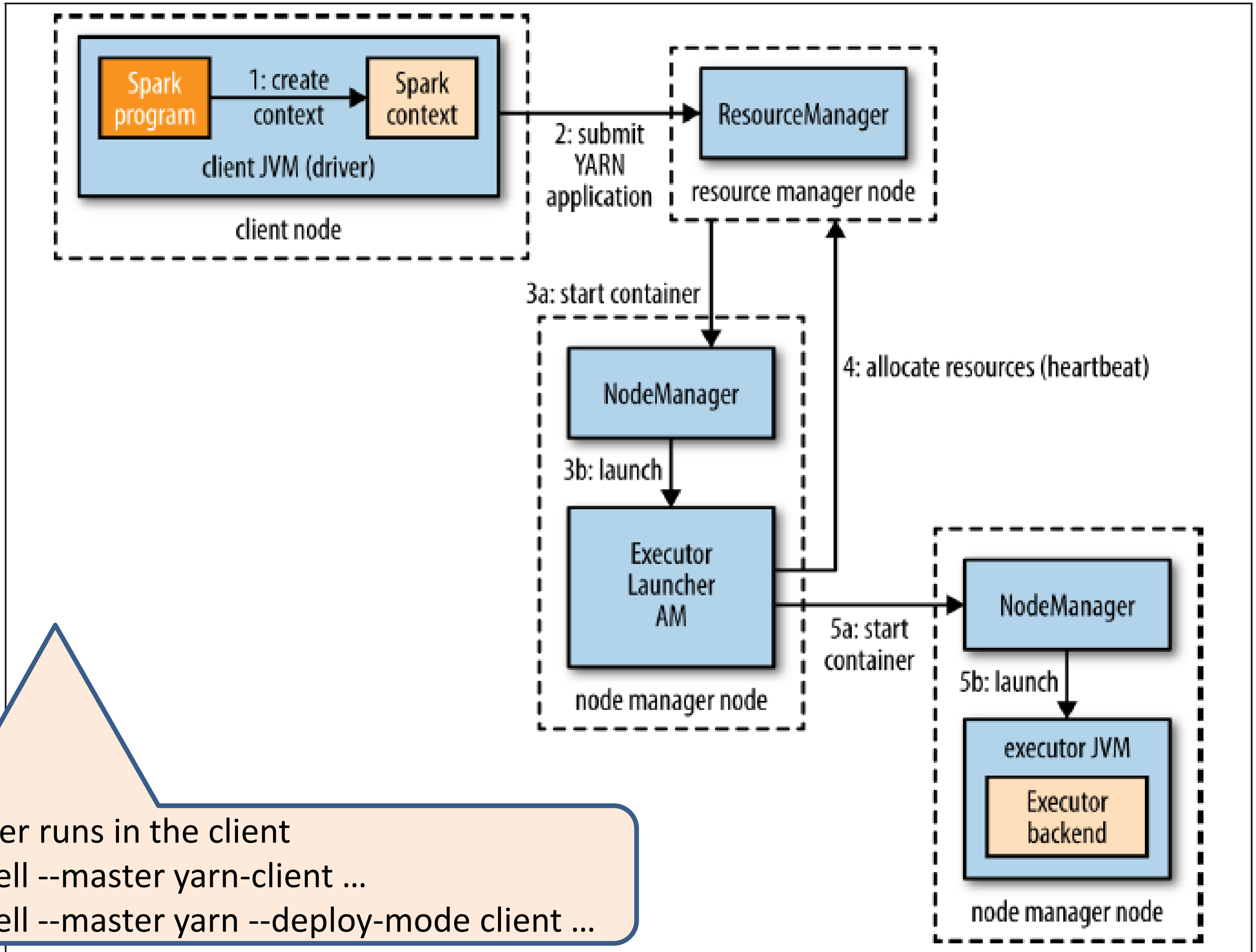
transformation

RDD

action

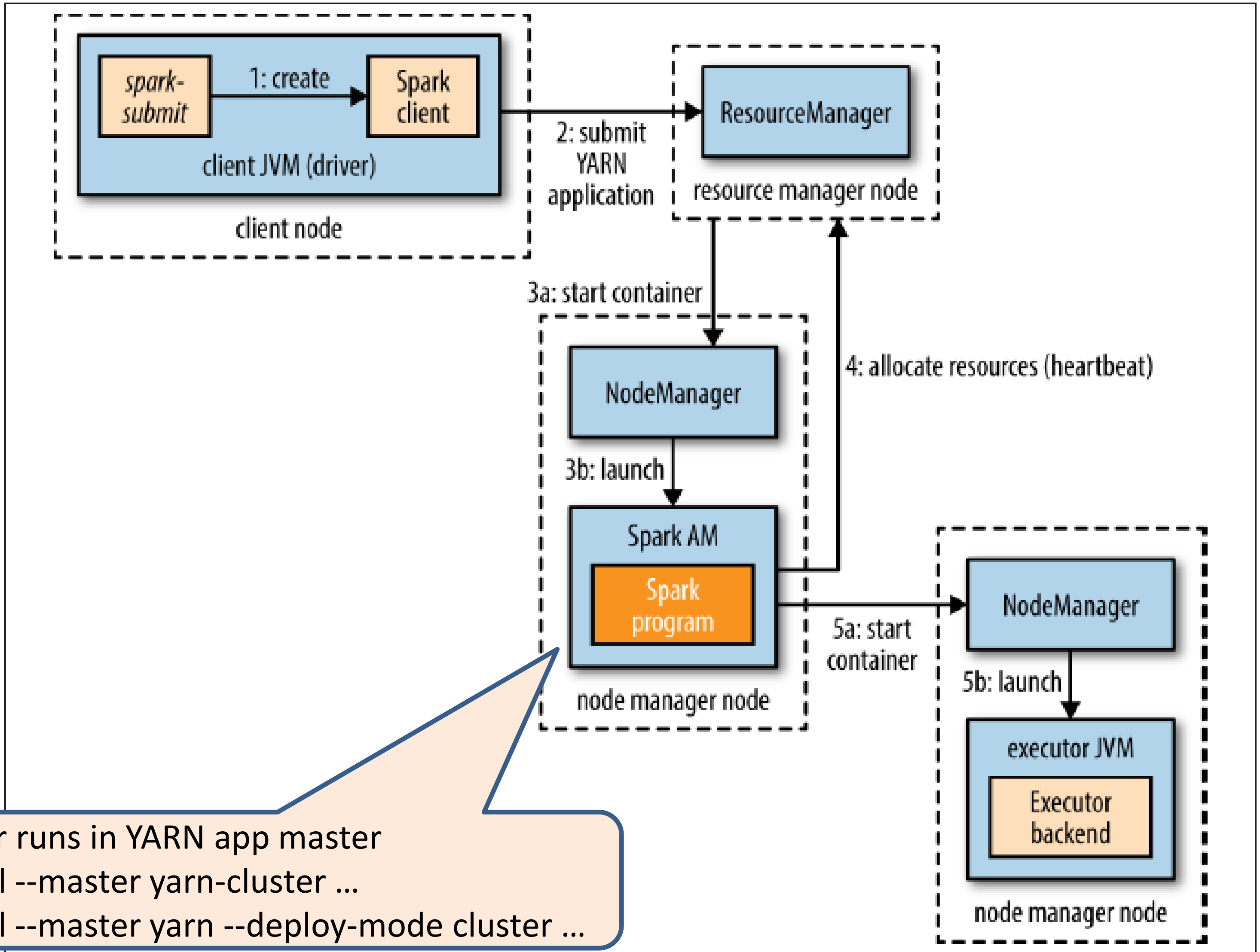
```
$ spark-submit --class MaxTemperature --master local \
spark-examples.jar input/ncdc/micro-tab/sample.txt output3
```

Apache Spark: YARN Client Mode



User's driver runs in the client
$ spark-shell --master yarn-client …
$ spark-shell --master yarn --deploy-mode client …

Apache Spark: YARN Cluster Mode



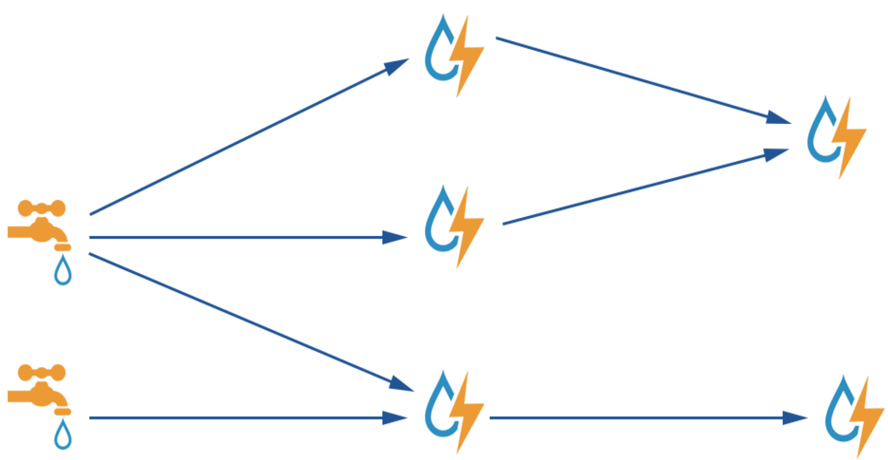
User's driver runs in YARN app master
$ spark-shell --master yarn-cluster …
$ spark-shell --master yarn --deploy-mode cluster …

Apache Storm

- Distributed computation framework for event stream processing
  - Scalable
  - Real-time computations on continuous streams of data
  - Incremental computation (e.g., rolling avg, etc.)
  - running forever until they are explicitly killed
  - Fault tolerance: restart failed worker (perhaps on different node)
- No dependency on MapReduce/Hadoop; it may
  - Adapter: interop w/ HDFS

Hadoop & Spark: process a batch at a time
Storm: process an event (microbatch) a time

- Storm *topology*: collection of wired spouts and bolts
  - *Spout*: filter/process input data stream to output *tuples*
  - *Bolt*: process input *tuples*, can be fed to other bolts
  - Instances of spout and bolt distributed on multi nodes in cluster

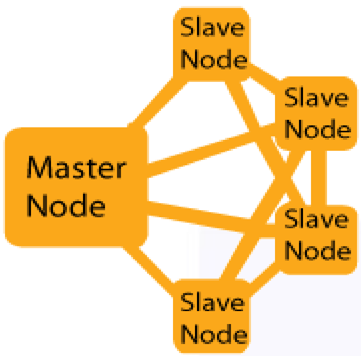


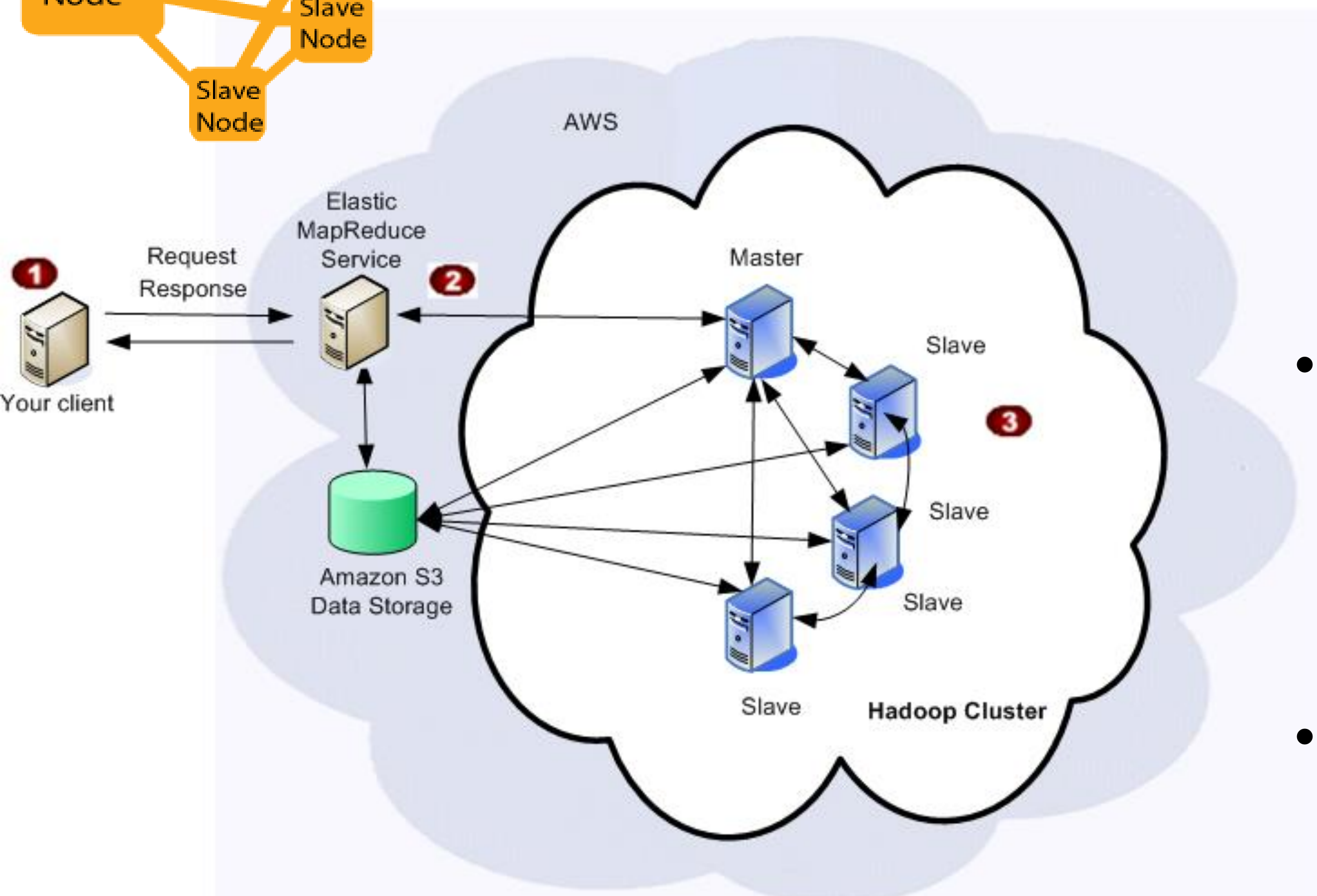
MapReduce Services

- AWS EMR: http://aws.amazon.com/elasticmapreduce/
- Azure HDInsight: http://azure.microsoft.com/en-us/services/hdinsight/
- Google Cloud DataProc: https://cloud.google.com/dataproc/
- Etc.

- Hadoop sandbox (VM on VMware player or VirtualBox, Docker container image): free
  - CDH: http://www.cloudera.com/content/www/en-us/downloads/quickstart_vms.html
  - HDP: https://hortonworks.com/downloads/#data-platform
  - https://www.mapr.com/products/mapr-sandbox-hadoop/download

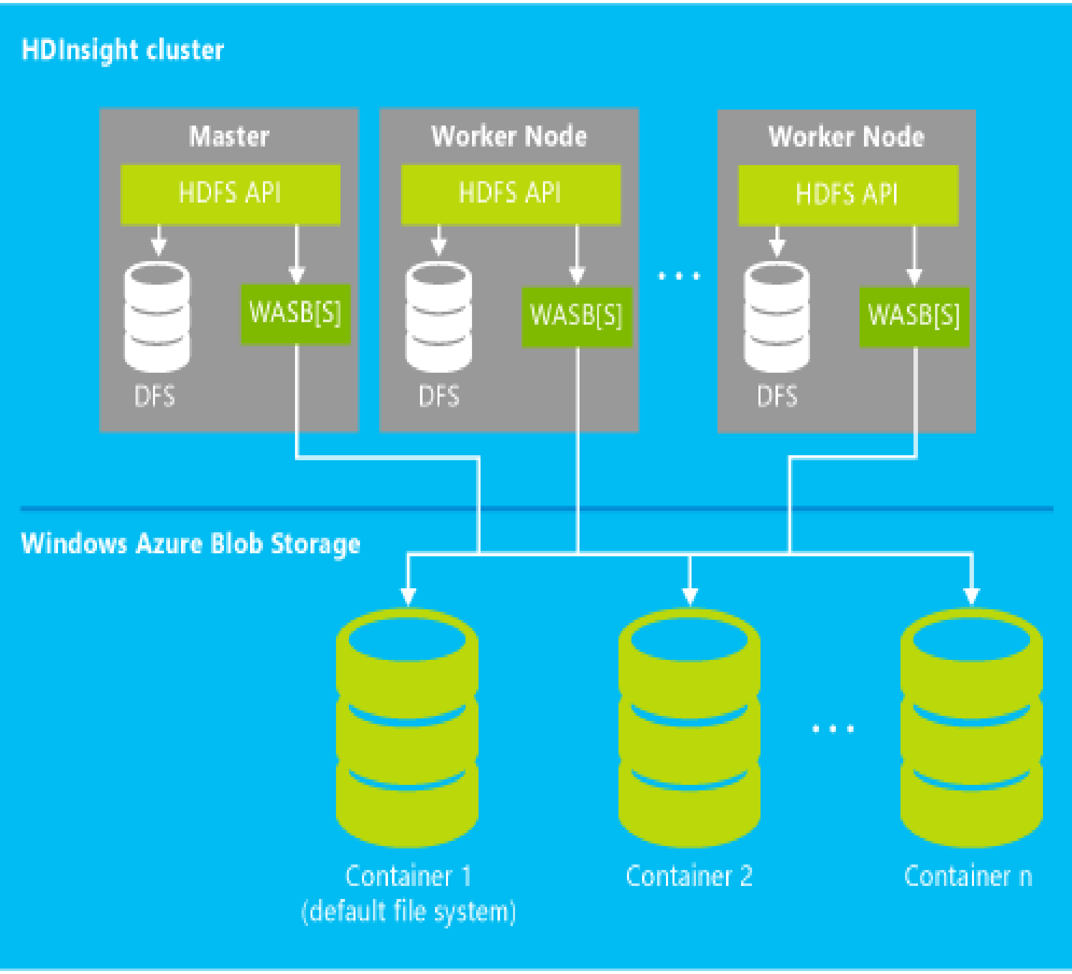
AWS - Elastic MapReduce (EMR)





- Use Hadoop to distribute data and process across a cluster of EC2 instances
  - Integrated w/ S3 and CloudWatch
- Scaling up / down
  - Deploy multiple clusters
  - Resize a running cluster
- Apache Spark on EMR

Azure - HDInsight



- Hadoop-based service
  - MapReduce
  - Various Hadoop related tools
- HA: HDInsight cluster
- query on-premises and cloud-based Hadoop clusters
  - Local to node: hdfs://<namenode>/<path>
  - Azure Blob: wasb[s]://<containername>@<accountname>.blob.core.windows.net/<path>
- Include
  - Apache Spark
  - Apache Hbase: NoSQL
  - Apache Storm: real-time stream processing

References

- Tom White, *Hadoop: The Definite Guide, 4/E*
  - http://bit.ly/hadoop_tdg_4e
  - https://github.com/tomwhite/hadoop-book/
- Hadoop, HDFS
  - http://hadoop.apache.org
  - http://hadooper.blogspot.com
  - http://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html
  - www.ibm.com/developerworks/library/wa-introhdfs/
- Hadoop sandbox (VM on VMware player or VirtualBox, docker container image): free
  - CDH: http://www.cloudera.com/downloads/quickstart_vms.html
  - HDP: https://hortonworks.com/downloads/#data-platform
  - https://www.mapr.com/products/mapr-sandbox-hadoop/download
- Spark: http://spark.apache.org/
- Storm: http://storm.apache.org/