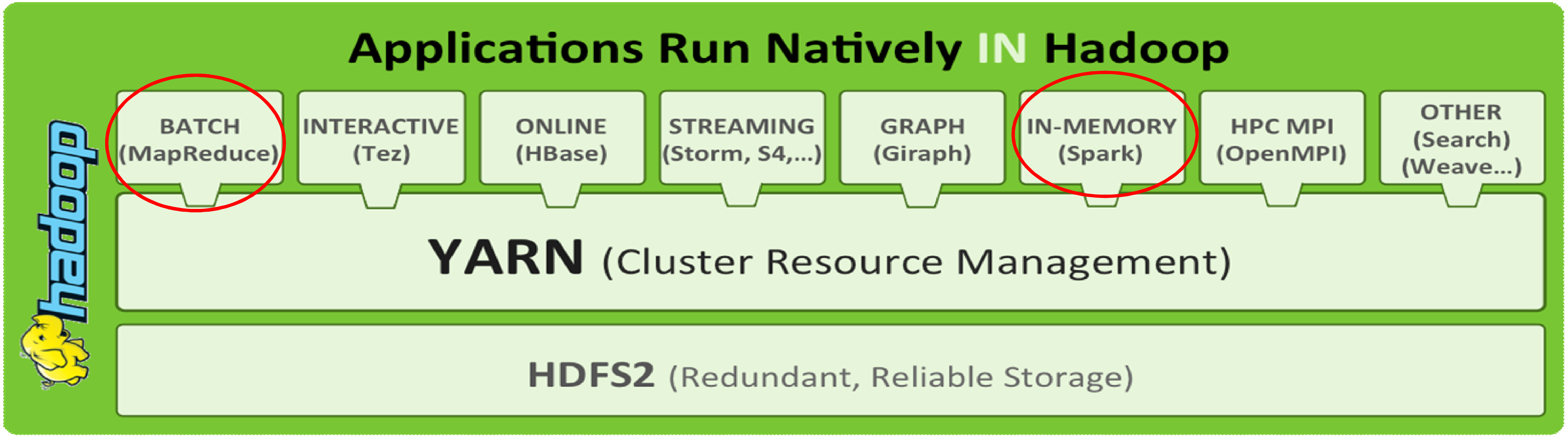
MapReduce和Spark探讨

史巨伟

2015-5

Hadoop 2.0 stack

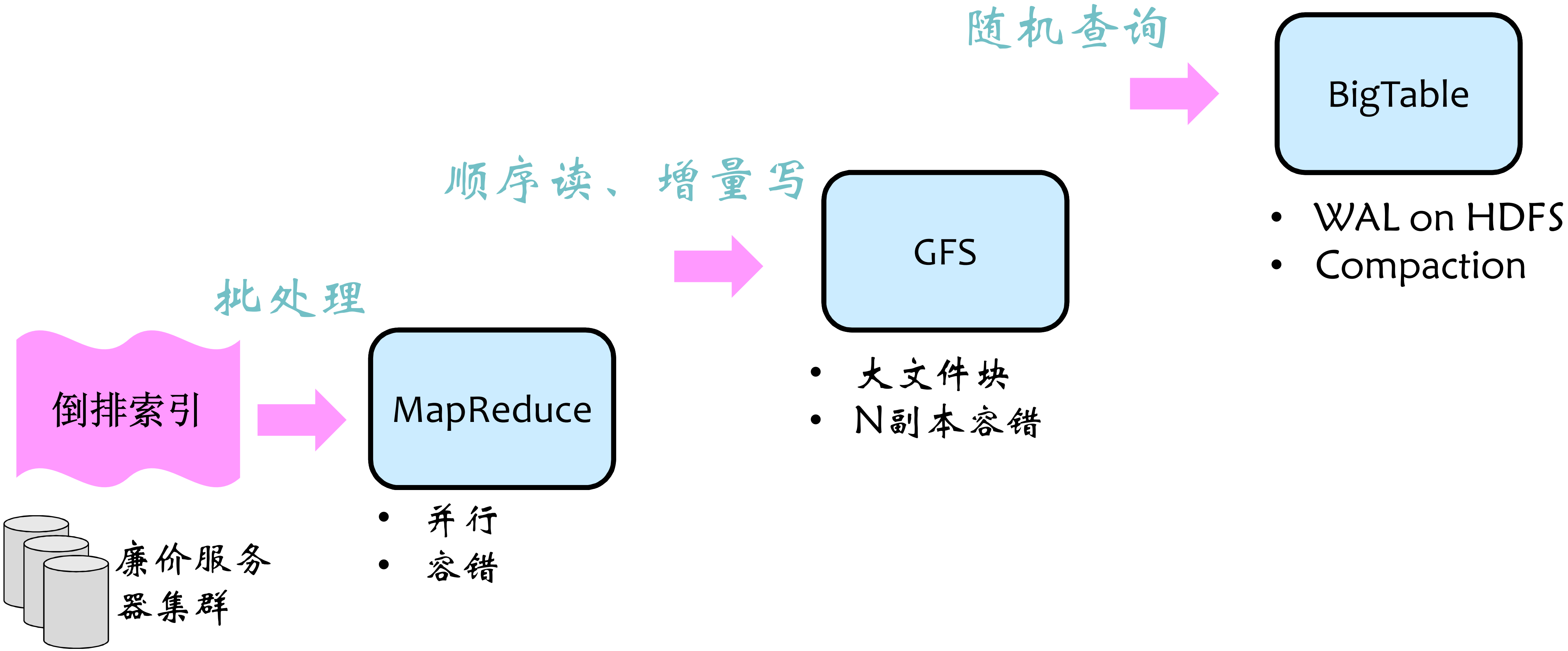


议程表

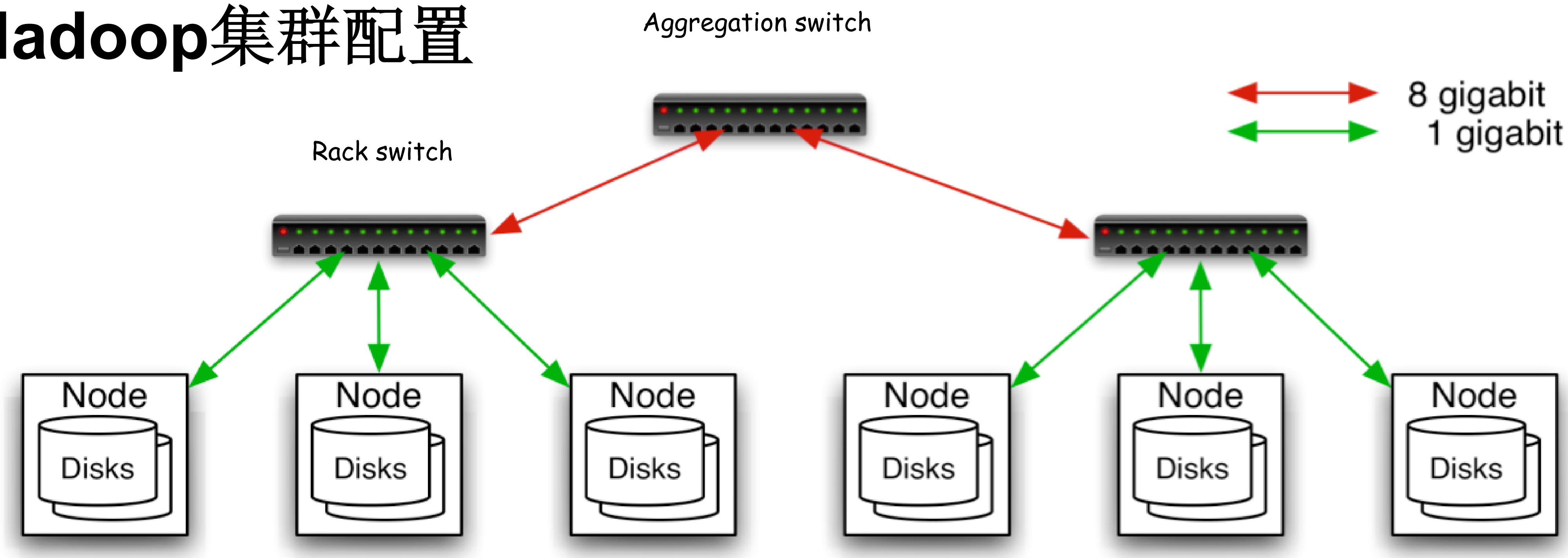
- Hadoop MapReduce简介
- Spark简介
- Hadoop vs. Spark 性能比较

Hadoop起源

- Google File System (GFS), MapReduce, BigTable



Hadoop集群配置



- 8 cores (16 hw thread)
- 64 GB RAM
- 4 disk drives @ 7.2k RPM with 1 TB
- 1 Gbps Ethernet switch

- 1 hw thread / task
- 4 GB RAM for JVM / task
- 250 GB (1/4 TP) for MR temp and DFS / task (e.g., 100 GB temp and 50 GB DFS)
- 64 Mbps Ethernet switch / task

MapReduce 编程模型

- 数据类型：key-value records

- Map
  - $(K_{in}, V_{in}) \rightarrow list\ (K_{inter}, V_{inter})$

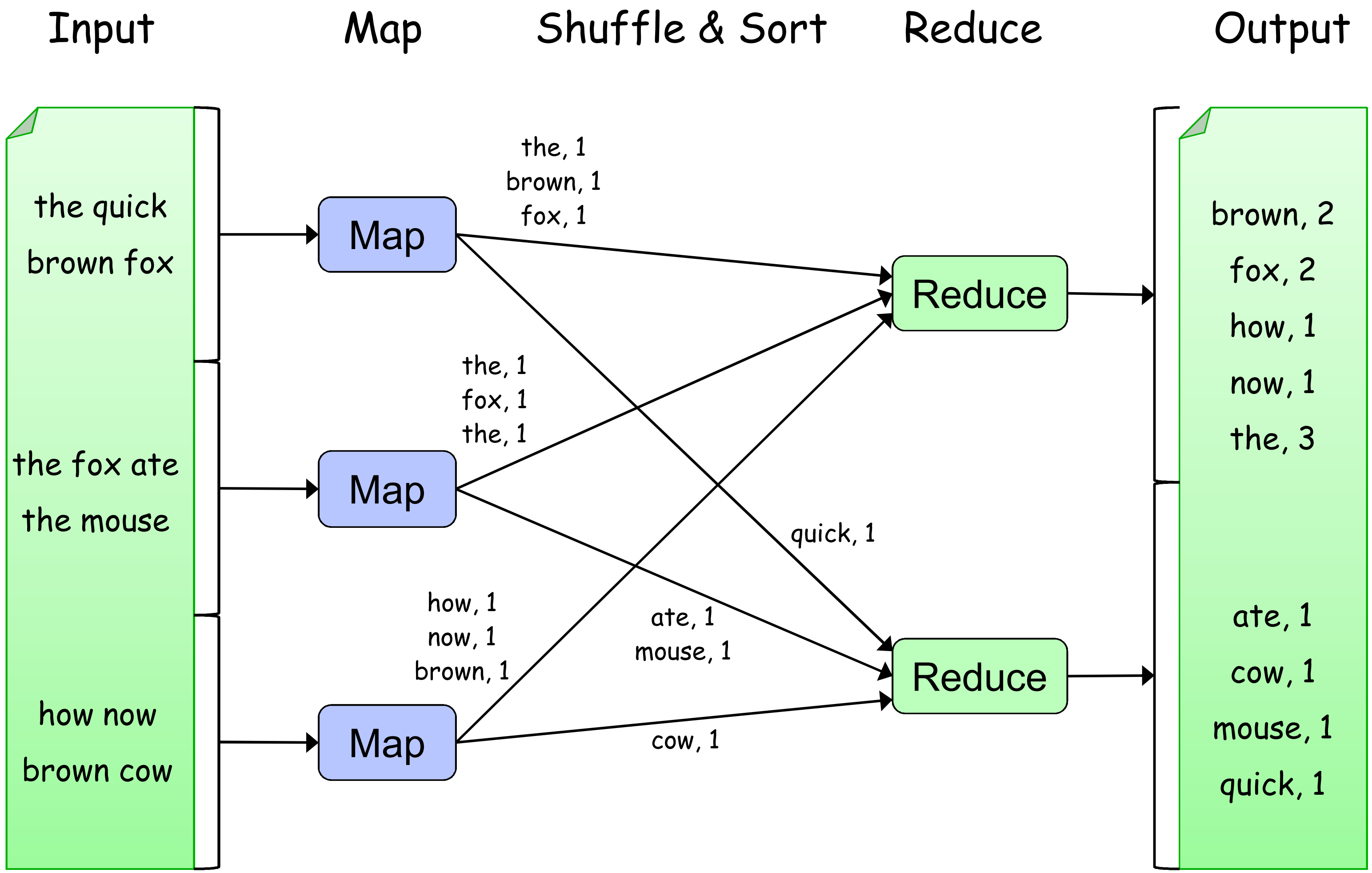
- Reduce
  - $(K_{inter}, list(V_{inter})) \rightarrow list\ (K_{out}, V_{out})$

Word Count 例子

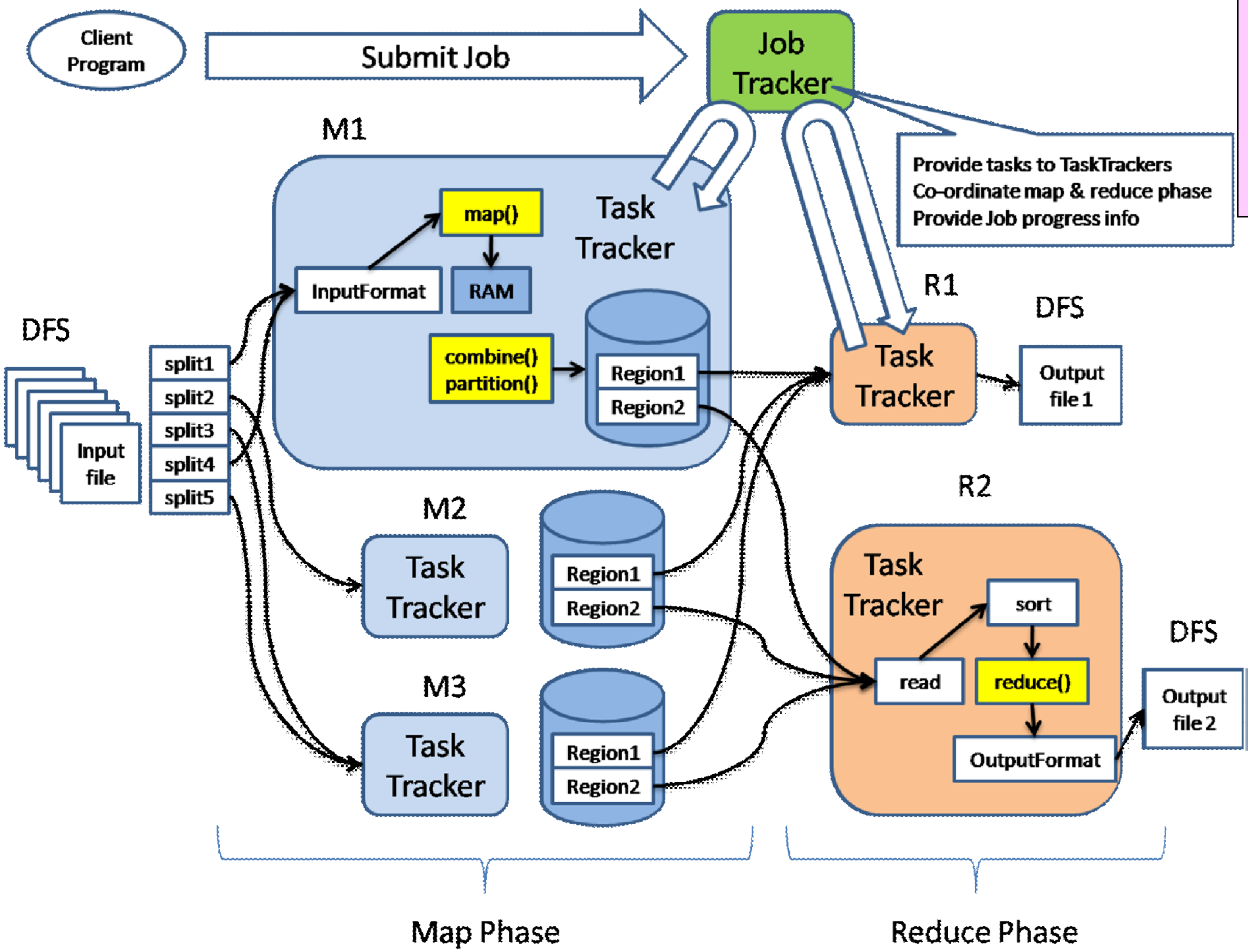
```
def mapper(line):
    foreach word in line.split():
        output(word, 1)


def reducer(key, values): {the,
{1, 1, 1}}
    output(key, sum(values))
```

Word Count执行表示



MapReduce内部使如何工作的？




```
def mapper(line):
    foreach word in line.split():
        output(word, 1)
def reducer(key, values):
    output(key, sum(values))
```

- 问题
  - 假设输入block为128MB，map任务输出为166 MB，超过默认io.sort.mb = 100 MB，怎么办？
  - 假设我们处理10TB数据，其中0.01%的单词为the，则有1GB的the被送到reduce，远超过Reduce JVM heap (256 MB)，怎么办？

在生产环境用Hadoop处理大数据

MapReduce编程简单 V.S. 在生产系统中高效运行的程序

- MapReduce编程人员必须理解程序如何在集群中被执行
  - OOM、并行度、作业数、负载均衡
- Hadoop管理员必须理解程序如何在集群中执行
  - 正确/优化的Hadoop参数配置

错误的**M/R**程序设计例子

- 过小粒度的输入切片（比如小文件1MB）→ 过大的任务启动、结束、调度开销

- Map/Reduce中的缓存（内存消耗和记录个数成正比）→ 内存溢出

- Reduce key过少（甚至只有1个）导致的负载均衡 （例如，求max, min, mean, top K, etc) → 并发度过低

Hadoop集群调优

- 企业用Hadoop的缺口
  - 私有部署：App开发/admin
  - Cloud (EMR): 分析师

	Tuned vs. Default
Running time	Often 10x
System resource usage	Often 10x
Failures	May avoid OOM, out of disk, job time out, etc.

典型MapReduce参数

- mapreduce.job.maps
- mapreduce.job.reduces
- mapreduce.task.io.sort.mb
- mapreduce.task.io.sort.factor
- ~~io.sort.record.percent~~
- mapreduce.map.sort.spill.percent
- mapreduce.reduce.shuffle.input.buffer.percent
- mapreduce.reduce.input.buffer.percent
- mapreduce.job.reduce.slowstart.completedmaps
- mapreduce.reduce.shuffle.parallelcopies
- mapreduce.map.output.compress
- mapred.map.child.java.opts
- mapred.reduce.child.java.opts

MR调优示例

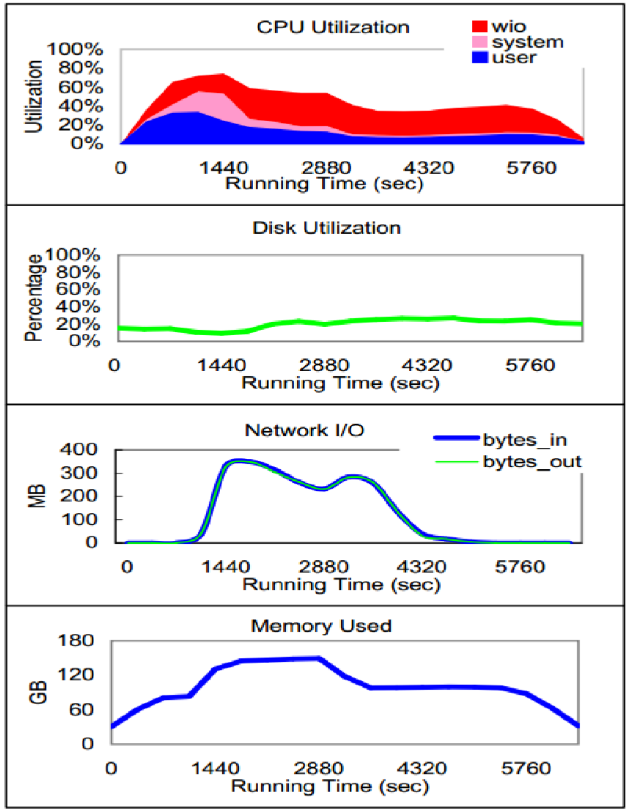
JobName	ID	Clu-ster	Input (GB)	Hadoop -X(sec)	MRTuner (sec)	Speed -up
Terasort	TS-1	$\mathcal{A}$	10	469	278	1.7
Terasort	TS-2	$\mathcal{A}$	50	2109	1122	1.87
Terasort	TS-3	$\mathcal{B}$	200	767	295	2.60
Terasort	TS-4	$\mathcal{B}$	1000	6274	2192	2.86
N-Gram	NG-1	$\mathcal{A}$	0.18	4364	192	22.7
N-Gram	NG-2	$\mathcal{A}$	0.7	N/A	661	∞
N-Gram	NG-3	$\mathcal{A}$	1.4	N/A	1064	∞
N-Gram	NG-4	$\mathcal{B}$	1.4	1100	249	4.41
N-Gram	NG-5	$\mathcal{B}$	2.8	1292	452	2.86
N-Gram	NG-6	$\mathcal{B}$	5.6	1630	930	1.75
PR(Trans.)	PR-1	$\mathcal{A}$	3.23	962	446	2.2
PR(Deg.)	PR-2	$\mathcal{A}$	Inter	49	41	1.2
PR(Iter.)	PR-3	$\mathcal{A}$	Inter	933	639	1.5
PR(Trans.)	PR-4	$\mathcal{B}$	3.23	148	65	2.28
PR(Deg.)	PR-5	$\mathcal{B}$	Inter	24	22	1.09
PR(Iter.)	PR-6	$\mathcal{B}$	Inter	190	82	2.32

map输入 >> map输出

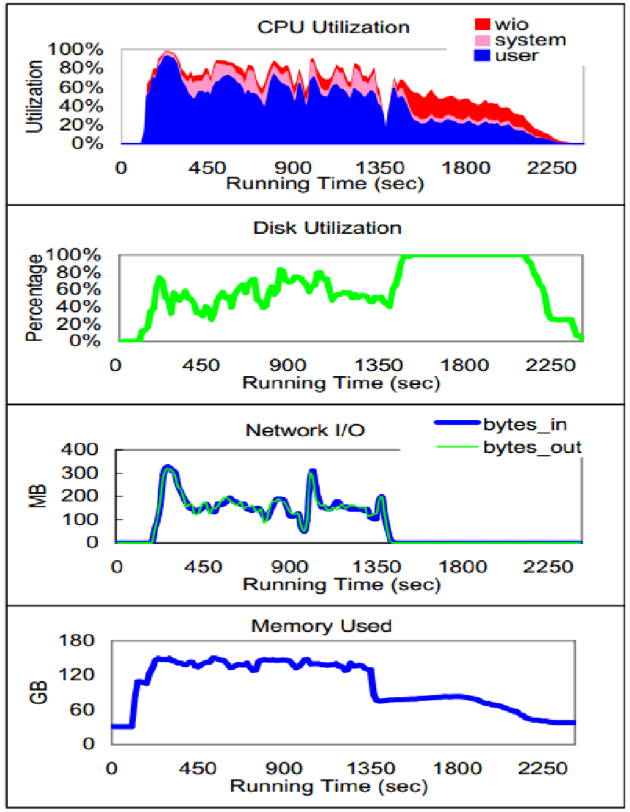
Out of disk

TeraSort 1TB

Cluster-wide Resource Usage from Ganglia



Hadoop-X



MRTuner

MRTuner achieves

✓**2.86x speedup**

✓Much better CPU utilization

✓Less context switch overhead

✓Much better memory utilization

✓Less network overhead

Why

✓Task slots optimization based on system resources of each node (21.2%)

✓No disk spills before Map task complete (estimated map output size and average record length) (26.99%)

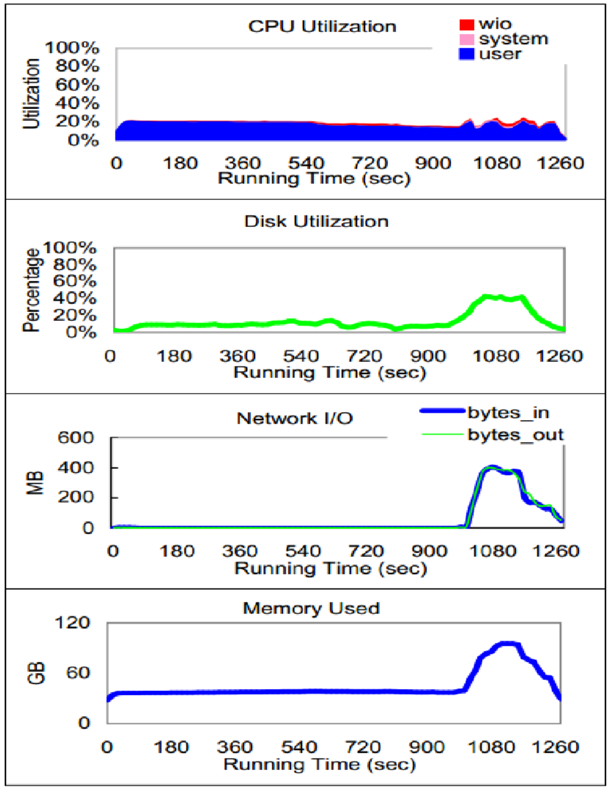
✓More reduce side buffer (estimated reduce memory usage) (6.27%)

✓Fully pipeline the Map phase and Reduce shuffle phase while avoid copier thread contention (estimated the throughput of map output and shuffle) (11.25%)

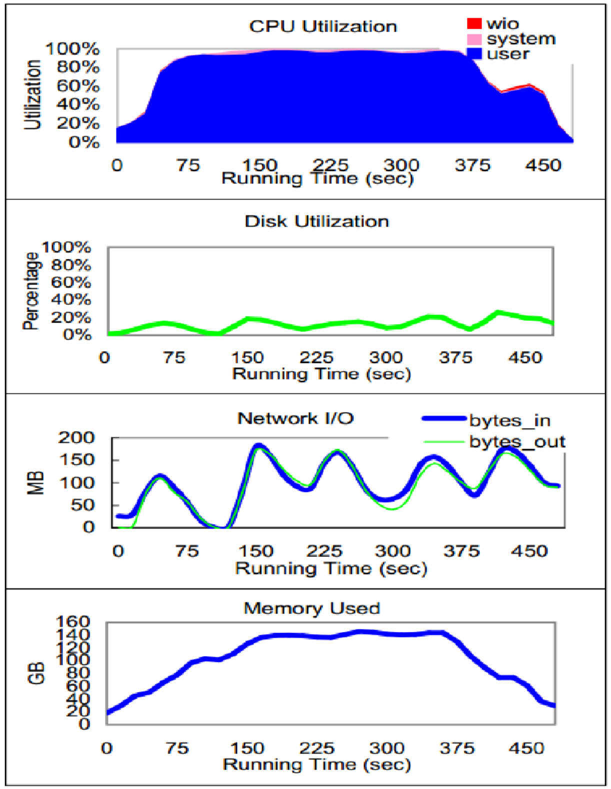
✓Compression of intermediate results (50.52%)

N-gram for text classification

Cluster-wide Resource Usage from Ganglia



Hadoop-X



MRTuner

MRTuner achieves

✓**2.8x speedup**

✓Much better CPU utilization

✓Much better memory utilization

Why

✓90% disk spills are reduced before Map task completes! (estimated map output size and average record length)

✓Splitting input based on Map Output/Input ratio instead of splitting input based on block size (Significantly improved parallelization)

✓Avoid copier thread contention by estimating the throughput of map output and shuffle

✓Task slots optimization based on system resources of each node

议程表

- Hadoop MapReduce简介
- Spark简介
- Hadoop vs. Spark 性能剖析

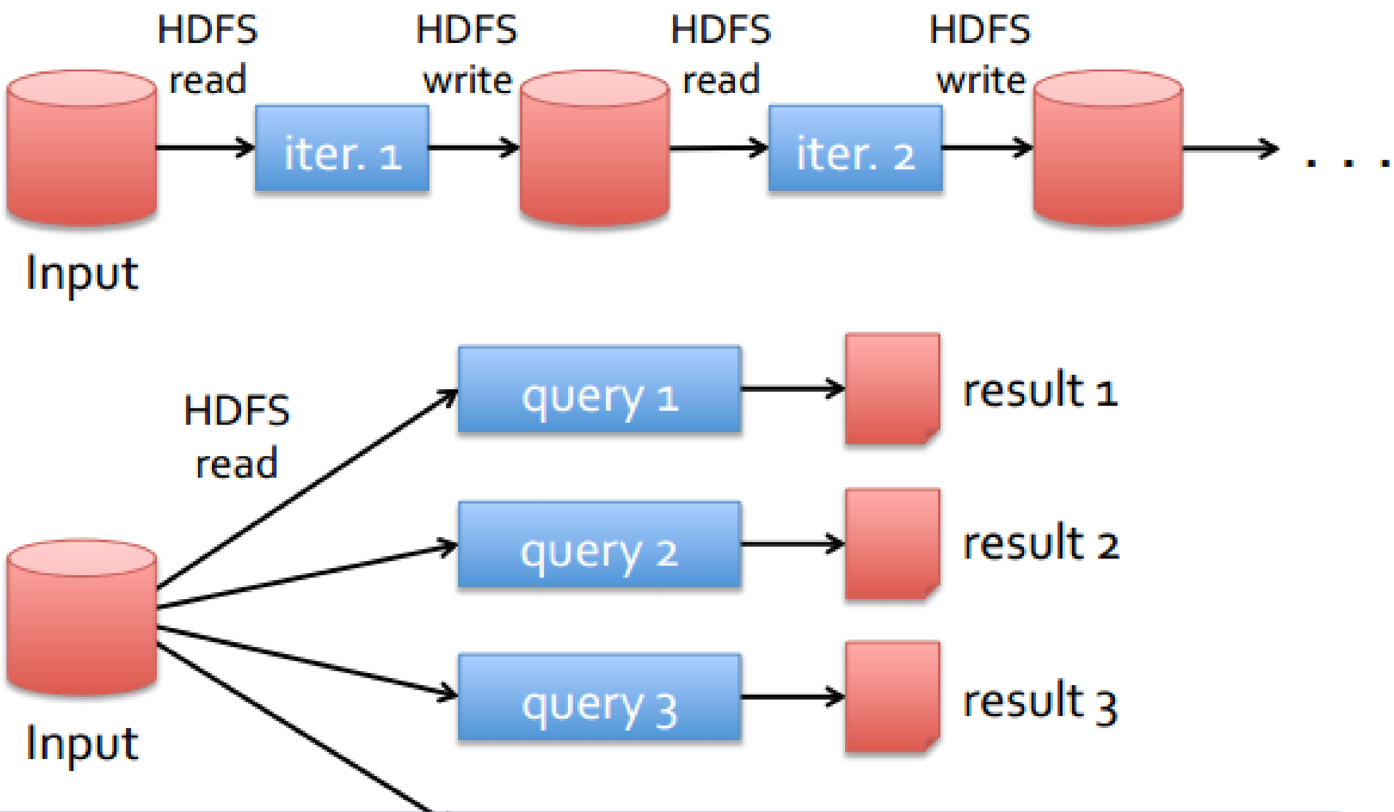
Spark动机

- MapReduce大大简化了集群计算的开发部署成本

- 缺点：过于频繁的物化（每个job都需要写HDFS）
  - 机器学习迭代算法：每轮job都需要扫描输入
  - 图分析算法：每轮job都需要扫描输入并且物化中间结果

- 开销在哪儿
  - 序列化/反序列化(CPU)、网络I/O、磁盘I/O
  - HDFS/OS caching不能避免主要开销

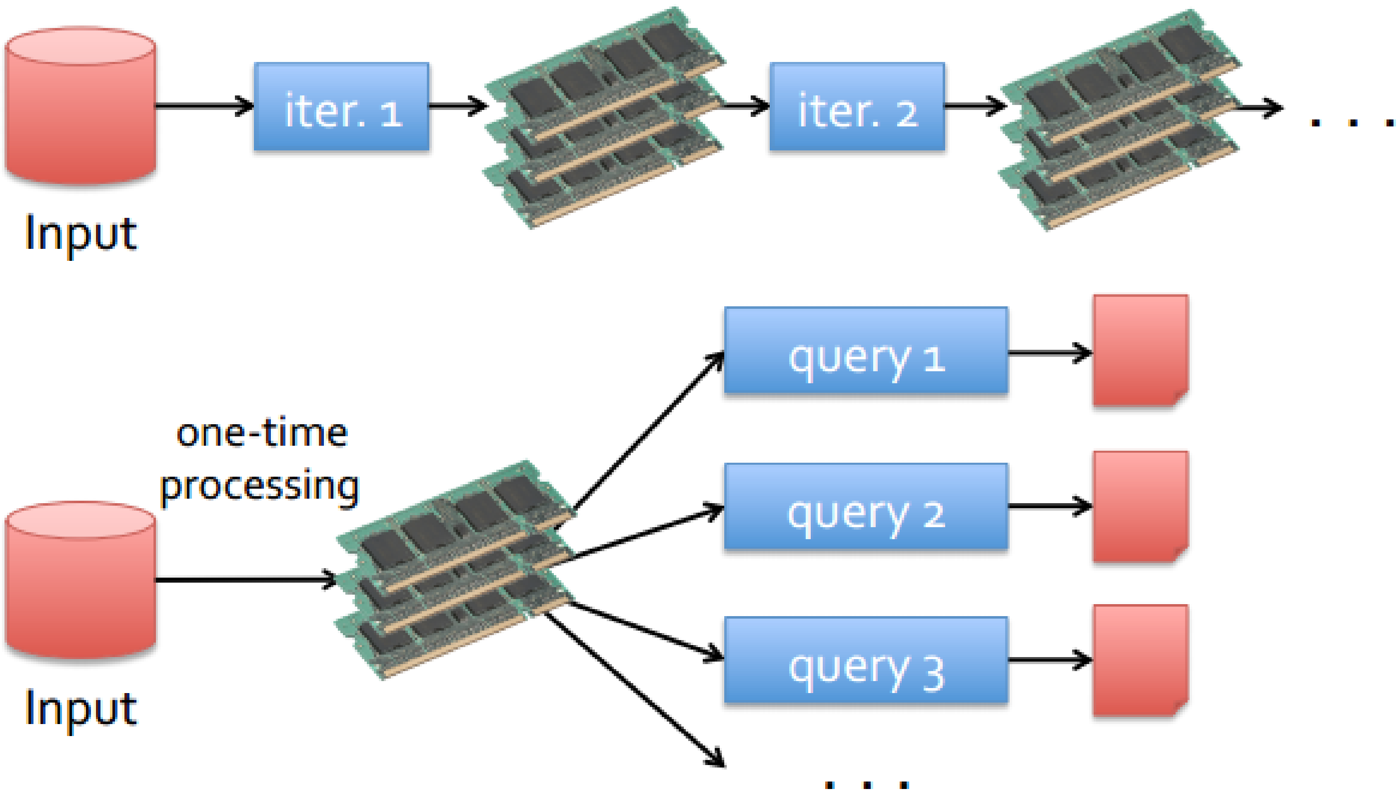
MapReduce应用模式



Slow due to replication and disk I/O,
but necessary for fault tolerance

Spark目标

- 在job之间共享数据
  - 尽量将共享数据存在内存



Spark核心挑战：容错

- 检查点
  - 显式
  - Shuffle write output

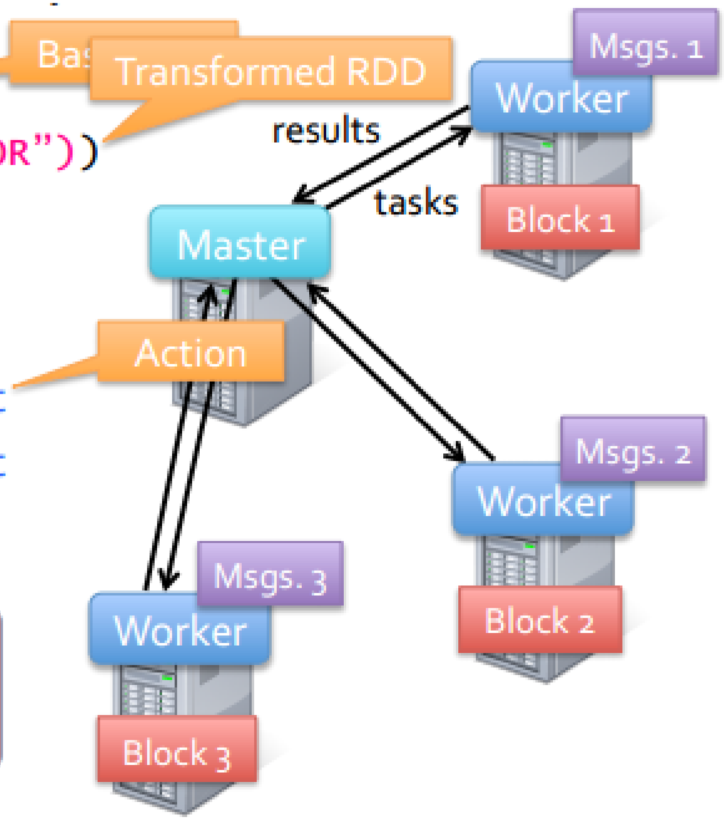
- 重算
  - Tradeoff：重算代价 vs. 每轮job在HDFS物化代价

Spark示例： 日志分析

- 将错误数据load到内存（RRD），然后迭代地搜索各种模式

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(_.startsWith("ERROR"))
messages = errors.map(_.split('\t')(2))
messages.persist()

messages.filter(_.contains("foo")).count
messages.filter(_.contains("bar")).count
```

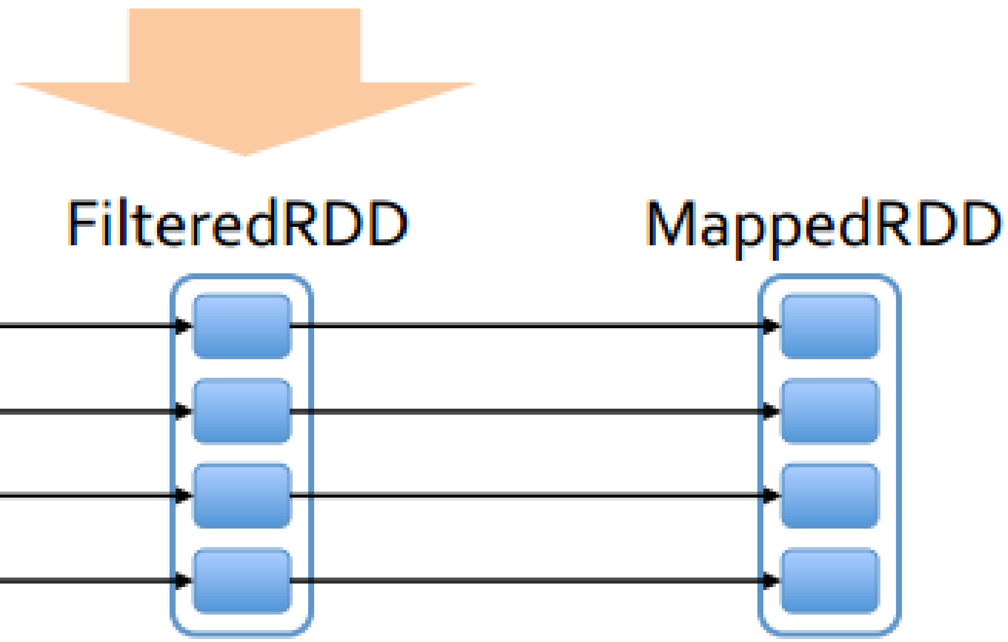


Result: scaled to 1 TB data in 5-7 sec (vs 170 sec for on-disk data)

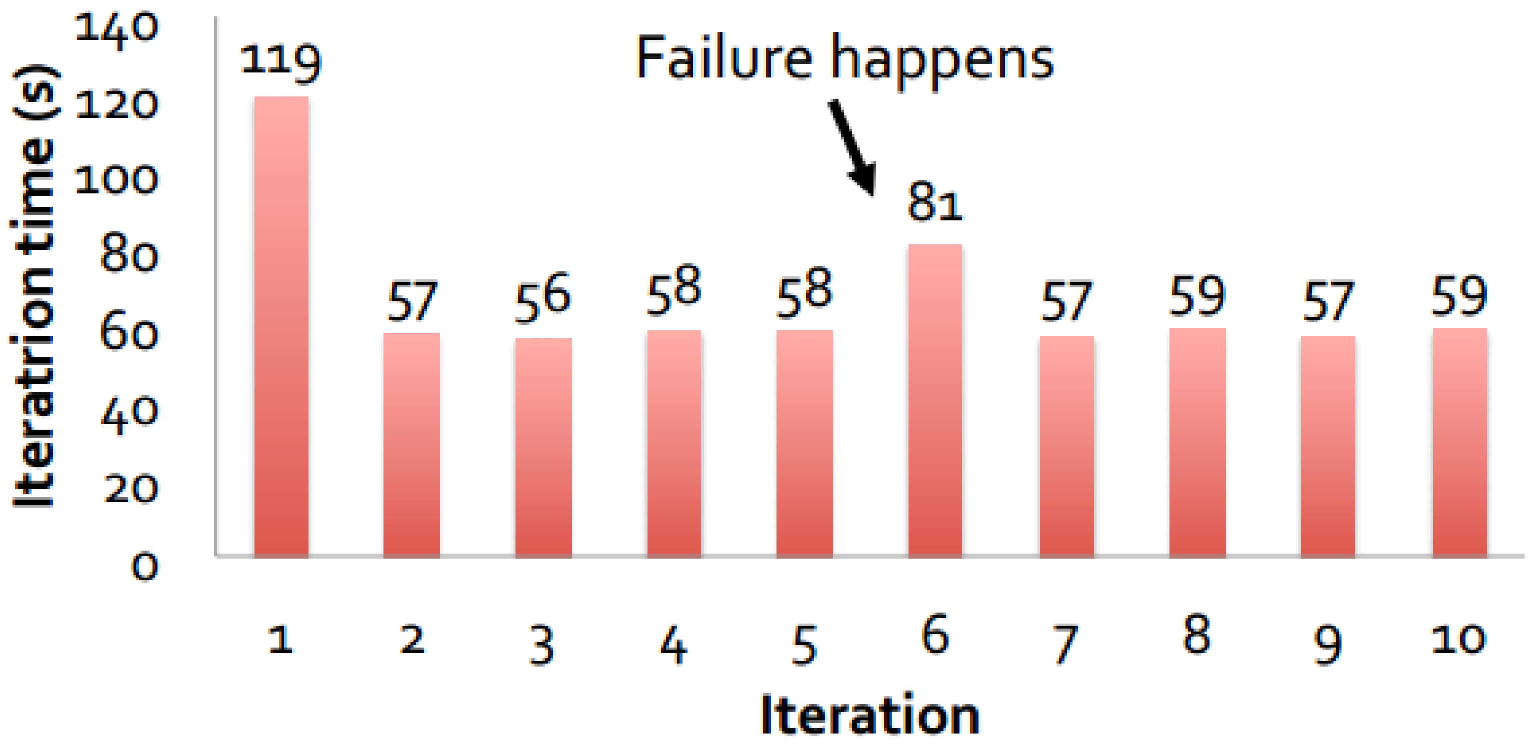
容错

- RDD跟踪生成其的lineage，从而以最小代价重算

E.g.:
```
messages = textFile(...).filter(_.contains("error"))
                        .map(_.split('\t')(2))
```



容错结果

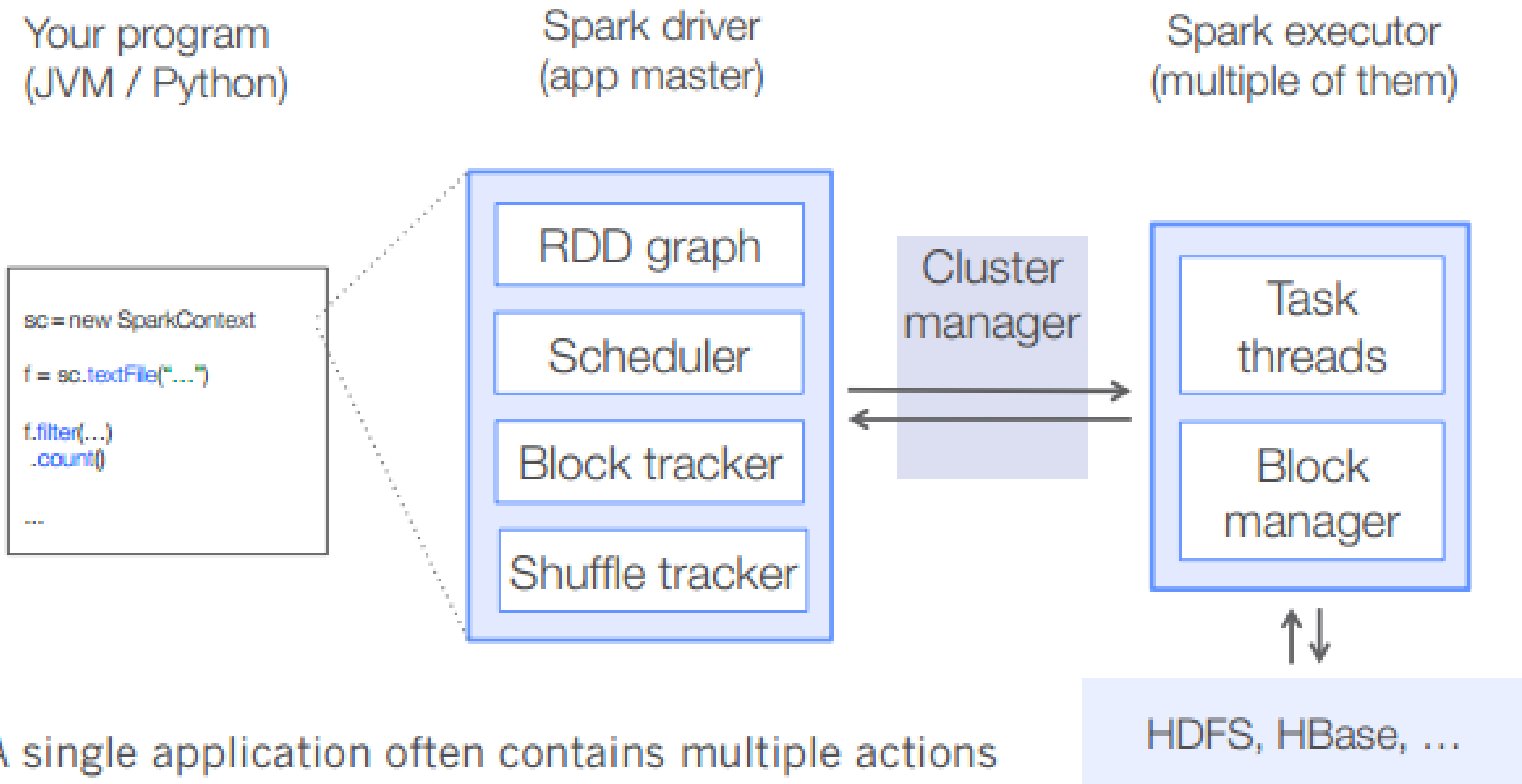


Failure happens
119
57
56
58
58
81
57
59
57
59
Iteratrion time (s)
140
120
100
80
60
40
20
0
1
2
3
4
5
6
7
8
9
10
Iteration

Spark算子

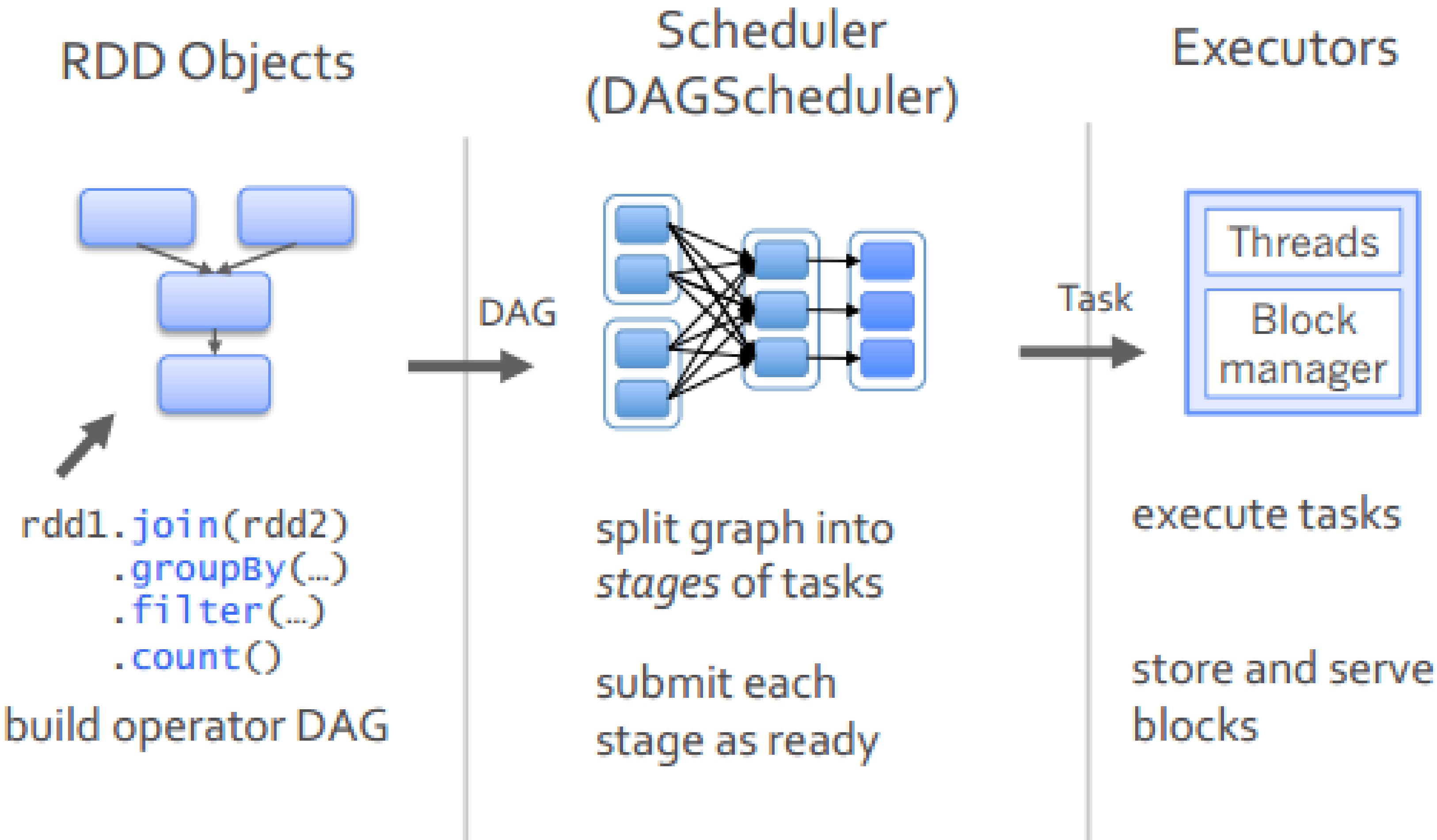
Transformations (define a new RDD)	map filter sample groupByKey reduceByKey sortByKey	flatMap union join cogroup cross mapValues
Actions (return a result to driver program)	collect reduce count save lookupKey	

Spark执行逻辑



A single application often contains multiple actions

Spark作业调度流程

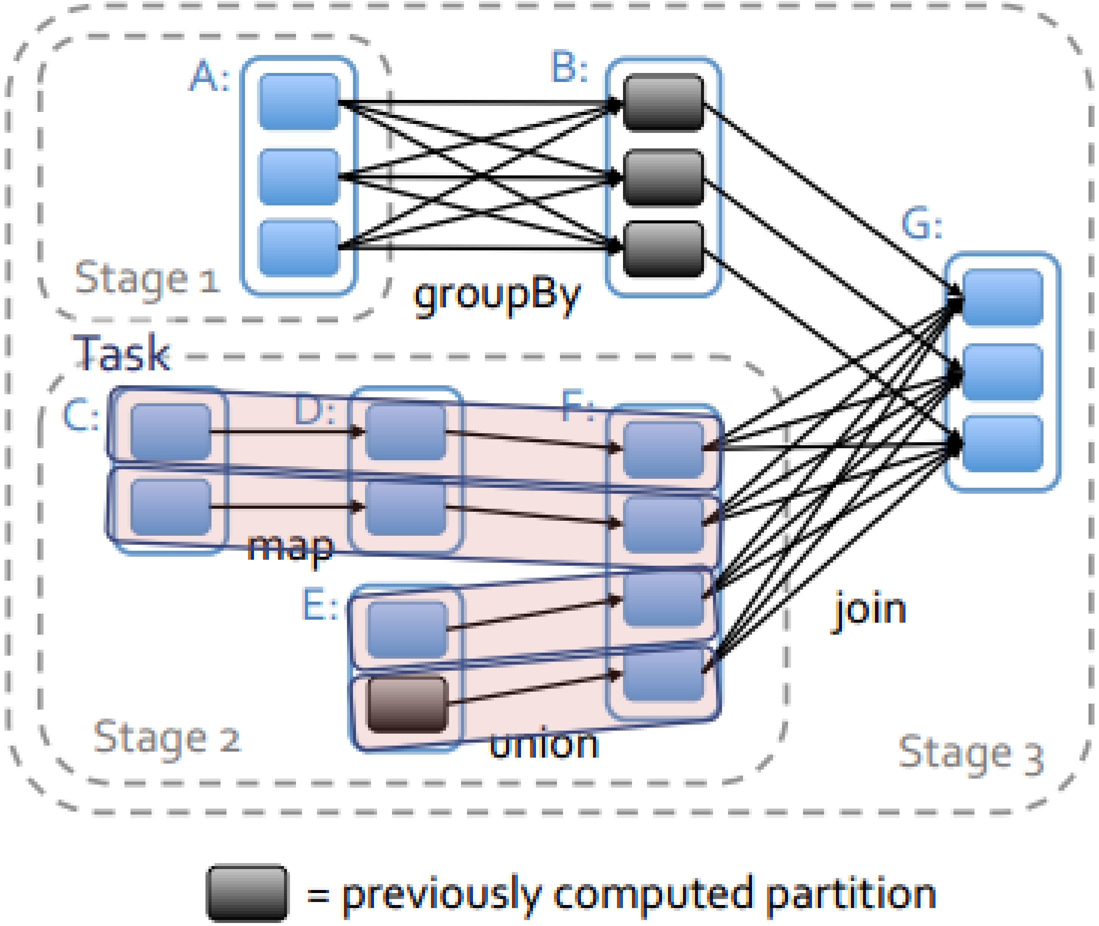


Spark调度优化

Pipelines operations within a stage

Picks join algorithms based on partitioning (minimize shuffles)

Reuses previously cached data



GC in Spark

Look at the “GC Time” column in the web UI

Tasks

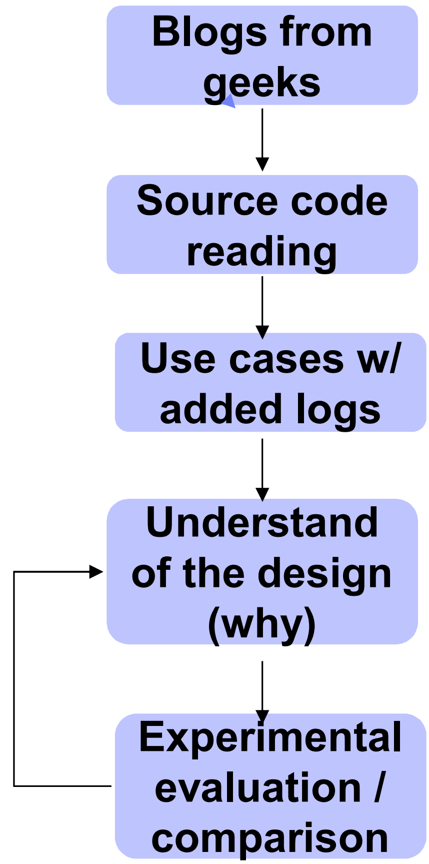
Task Index	Task ID	Status	Locality Level	Executor	Launch Time	Duratio	GC Time	R T
0	96	SUCCESS	PROCESS_LOCAL	ip-172-31-2-222.us-west-2.compute.internal	2014/07/02 08:05:53	7 s	58 ms	

To discover whether GC is the problem:

1. Set spark.executor.extraJavaOptions to include: “-XX:-PrintGCDetails -XX:+PrintGCTimeStamps”
2. Look at spark/work/app…/[n]/stdout on executors
3. Short GC times are OK. Long ones are bad.

理解Spark内核

- 重要组件
  - Execution model
  - Physical task plan transformation
  - Shuffle
  - Memory management
  - Serialization



记录级别日志插入

```
14/12/04 17:30:46 INFO MappedRDD: ***** 1.0 compute() MappedRDD[3] at  on element in
14/12/04 17:30:46 INFO MappedRDD: ***** 1.0 compute() MappedRDD[3] at  on element North
14/12/04 17:30:46 INFO MappedRDD: ***** 1.0 compute() MappedRDD[3] at  on element Ame
14/12/04 17:30:46 INFO HadoopRDD: ***** 1.0 read HDFS, key=178, value=rica is rooted in English traditions dating from the Protestant Reformation. It also has aspects of
14/12/04 17:30:46 INFO MappedRDD: ***** 1.0 compute() MappedRDD[1] at  on element (178,rica is rooted in English traditions dating from the Protestant Reformation. It al
14/12/04 17:30:46 INFO MappedRDD: ***** 1.0 compute() MappedRDD[3] at  on element rica
14/12/04 17:30:46 INFO MappedRDD: ***** 1.0 compute() MappedRDD[3] at  on element is
14/12/04 17:30:46 INFO MappedRDD: ***** 1.0 compute() MappedRDD[3] at  on element rooted
14/12/04 17:30:46 INFO MappedRDD: ***** 1.0 compute() MappedRDD[3] at  on element in
14/12/04 17:30:46 INFO MappedRDD: ***** 1.0 compute() MappedRDD[3] at  on element English
14/12/04 17:30:46 INFO MappedRDD: ***** 1.0 compute() MappedRDD[3] at  on element traditions
14/12/04 17:30:46 INFO HashShuffleWriter: ***** 1.1 Aggregation Finished
14/12/04 17:30:46 INFO MappedRDD: ***** 1.0 compute() MappedRDD[3] at  on element dating
14/12/04 17:30:46 INFO MappedRDD: ***** 1.0 compute() MappedRDD[3] at  on element from
14/12/04 17:30:46 INFO HashShuffleWriter: ***** 1.1 Writing output object: (Thanksgiving,1)
14/12/04 17:30:46 INFO MappedRDD: ***** 1.0 compute() MappedRDD[3] at  on element the
14/12/04 17:30:46 INFO MappedRDD: ***** 1.0 compute() MappedRDD[3] at  on element Protestant
14/12/04 17:30:46 INFO MappedRDD: ***** 1.0 compute() MappedRDD[3] at  on element Reformation.
14/12/04 17:30:46 INFO MappedRDD: ***** 1.0 compute() MappedRDD[3] at  on element It
14/12/04 17:30:46 INFO MappedRDD: ***** 1.0 compute() MappedRDD[3] at  on element also
14/12/04 17:30:46 INFO MappedRDD: ***** 1.0 compute() MappedRDD[3] at  on element has
14/12/04 17:30:46 INFO MappedRDD: ***** 1.0 compute() MappedRDD[3] at  on element aspects
14/12/04 17:30:46 INFO MappedRDD: ***** 1.0 compute() MappedRDD[3] at  on element of
14/12/04 17:30:46 INFO MappedRDD: ***** 1.0 compute() MappedRDD[3] at  on element a
14/12/04 17:30:46 INFO MappedRDD: ***** 1.0 compute() MappedRDD[3] at  on element harvest
14/12/04 17:30:46 INFO MappedRDD: ***** 1.0 compute() MappedRDD[3] at  on element festival,
14/12/04 17:30:46 INFO MappedRDD: ***** 1.0 compute() MappedRDD[3] at  on element even
14/12/04 17:30:46 INFO MappedRDD: ***** 1.0 compute() MappedRDD[3] at  on element though
14/12/04 17:30:46 INFO MappedRDD: ***** 1.0 compute() MappedRDD[3] at  on element the
14/12/04 17:30:46 INFO MappedRDD: ***** 1.0 compute() MappedRDD[3] at  on element harvest
14/12/04 17:30:46 INFO MappedRDD: ***** 1.0 compute() MappedRDD[3] at  on element in
14/12/04 17:30:46 INFO MappedRDD: ***** 1.0 compute() MappedRDD[3] at  on element New
14/12/04 17:30:46 INFO HashShuffleWriter: ***** 1.1 Writing output object: (is,1)
14/12/04 17:30:46 INFO MappedRDD: ***** 1.0 compute() MappedRDD[3] at  on element England
14/12/04 17:30:46 INFO HashShuffleWriter: ***** 1.1 Writing output object: (on,1)
14/12/04 17:30:46 INFO HashShuffleWriter: ***** 1.1 Writing output object: (celebrated.,1)
14/12/04 17:30:46 INFO HashShuffleWriter: ***** 1.1 Writing output object: (,1)
14/12/04 17:30:46 INFO HashShuffleWriter: ***** 1.1 Writing output object: (which,1)
14/12/04 17:30:46 INFO HashShuffleWriter: ***** 1.1 Writing output object: (late-November,1)
14/12/04 17:30:46 INFO HashShuffleWriter: ***** 1.1 Writing output object: (holiday,1)
14/12/04 17:30:46 INFO HashShuffleWriter: ***** 1.1 Writing output object: (date,1)
```

ormal text file   length : 19779   lines : 214   Ln : 1   Col : 1   Sel : 0 | 0   Dos\Windows   UTF-8 w/o BOM   IN

Pipeline: compute() of RDD
Blocked: aggregation in HashMap

Shuffle write and read

- Shuffle write
  - In the end of a *ShuffleMapTask*, it will use the write() of the configured *ShuffleWriter* to write the output of the last RDD of the task to disk.

- Shuffle read
  - To generate the *ShuffledRDD* as the first RDD of a task, it will use the read() of the configured *ShuffleReader* to fetch the outputs of the parent stages, and aggregate them as “reduce” input.

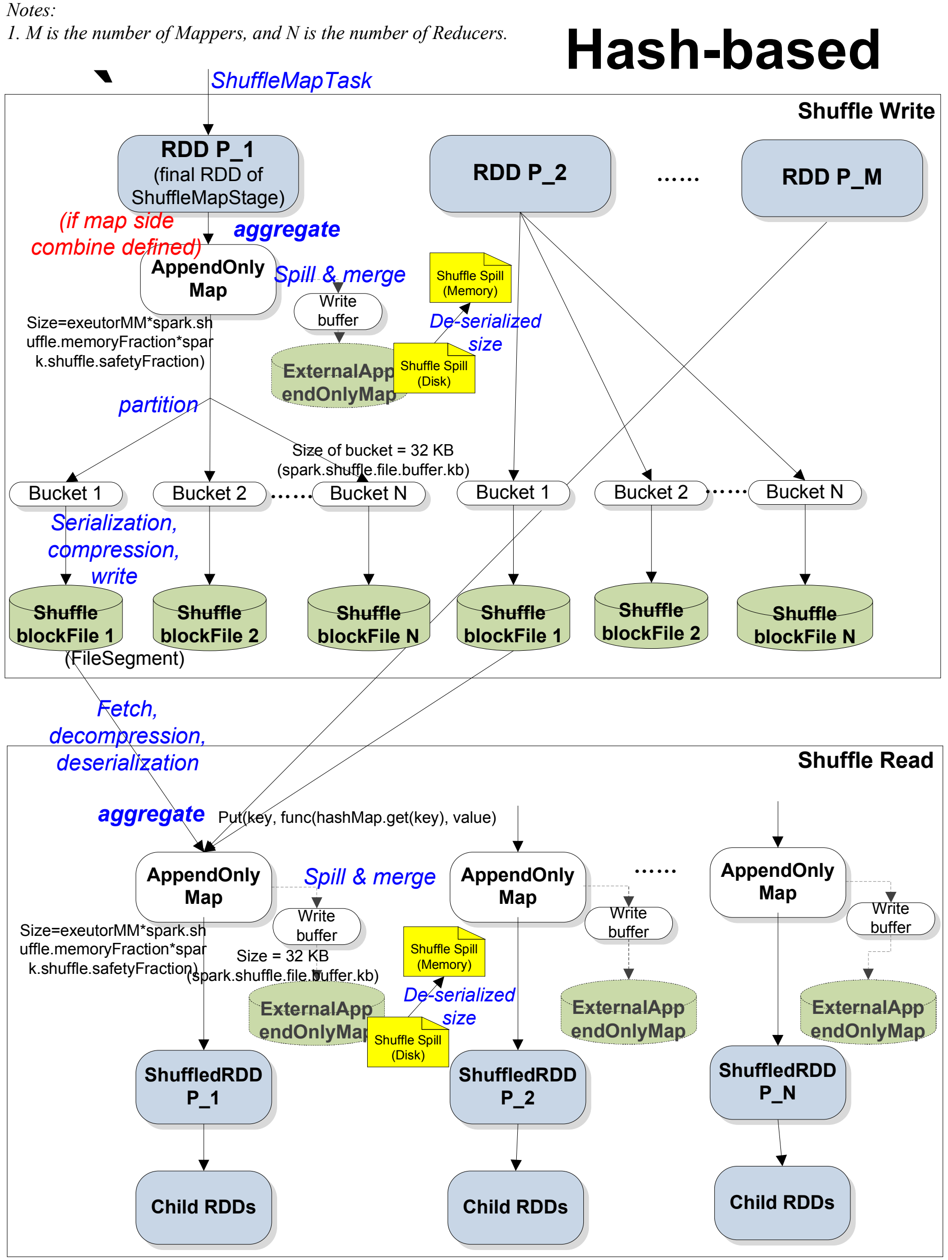
Map side combiner

- In Spark, there is no need for users to define the map side combiner function, the APIs like reduceByKey will automatic combine the value using the AppendOnlyMap
  - For HashShuffleWriter, it use the AppendOnlyMap based Aggregator, and can spill to disk as needed.
  - For SortShuffleWriter, it use the AppendOnlyMap directly, and will use the sort-merge to handle the case that map output is larger than the buffer.

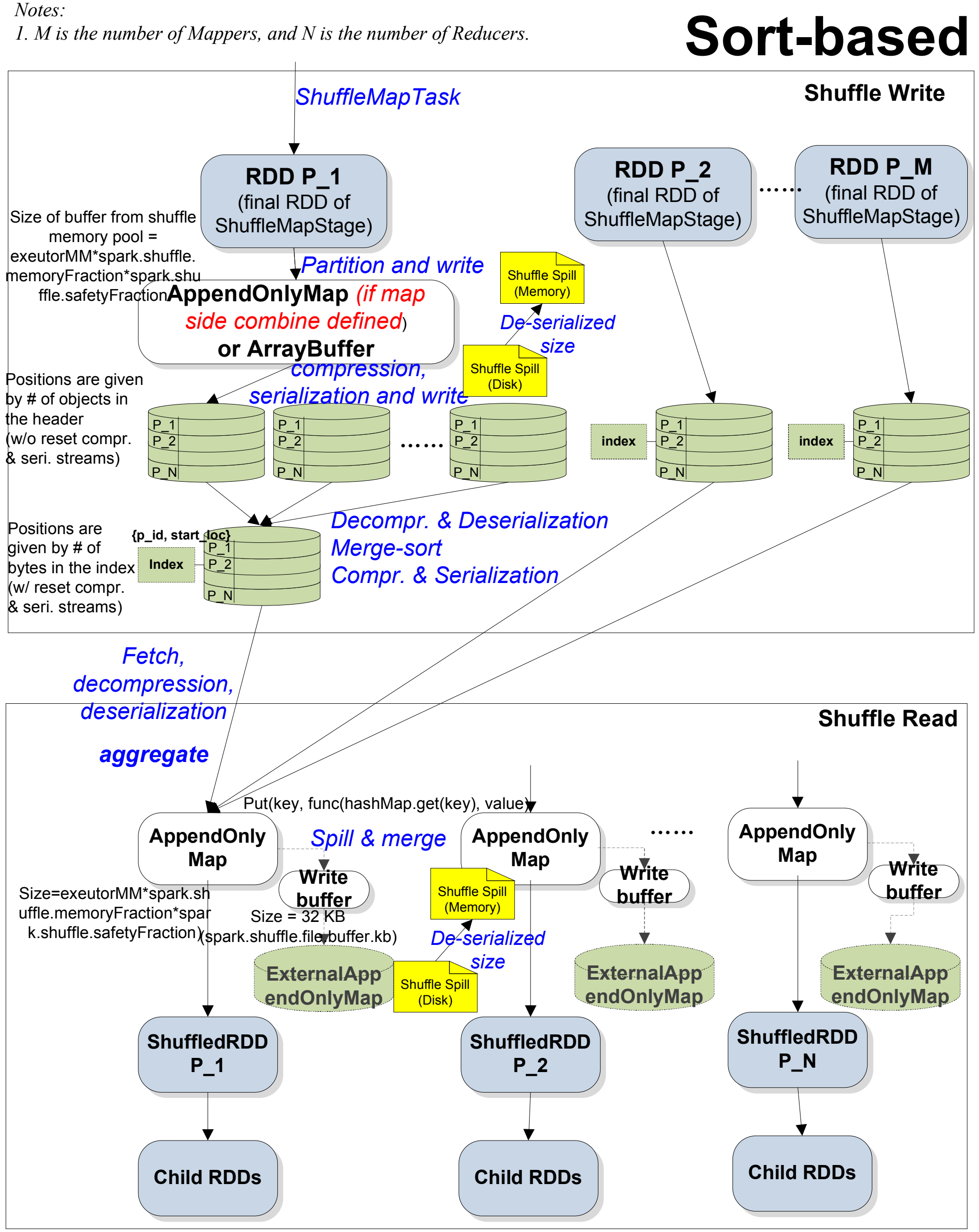
Shuffle Spill (Memory) and (Disk)

- **What are the metrics**
  - Both metrics are counted when the records in AppendOnlyMap will be spilled to disk.
- **Where is the spills**
  - For Sort-based shuffle writer
    - If there is Map side combiner, may spill in AppendOnlyMap.
    - If there is no Map side combiner, may spill in Array
  - For Hash-based shuffle writer
    - If there is Map side combiner, may spill in AppendOnlyMap
    - If there is no Map side combiner, then there is no spill
  - For shuffle reader
    - May spill in AppendOnlyMap during aggregation

Notes:
1. M is the number of Mappers, and N is the number of Reducers.



Notes:
1. M is the number of Mappers, and N is the number of Reducers.



Hash-based 和 Sort-based 区别

- Scalability issue of Hash based shuffling
  - Too many intermediate files (M*R/nodes for each node) → File Consolidation (cores*R)
  - Buffer for the buckets (cores*R*32KB) → Sort-based Shuffle

- Hash-based vs. sort-based
  - Both use the HashMap based aggregator for map side combine
  - The main drawback of hash based shuffle is the scalability issue (at least cores*R buckets and files even with consolidated files)
  - If there is no Map side buffer, the hash-based shuffler writer requires no HashMap buffer to keep map output, and it will flush records to disk directly.

MapReduce和Spark编程接口区别

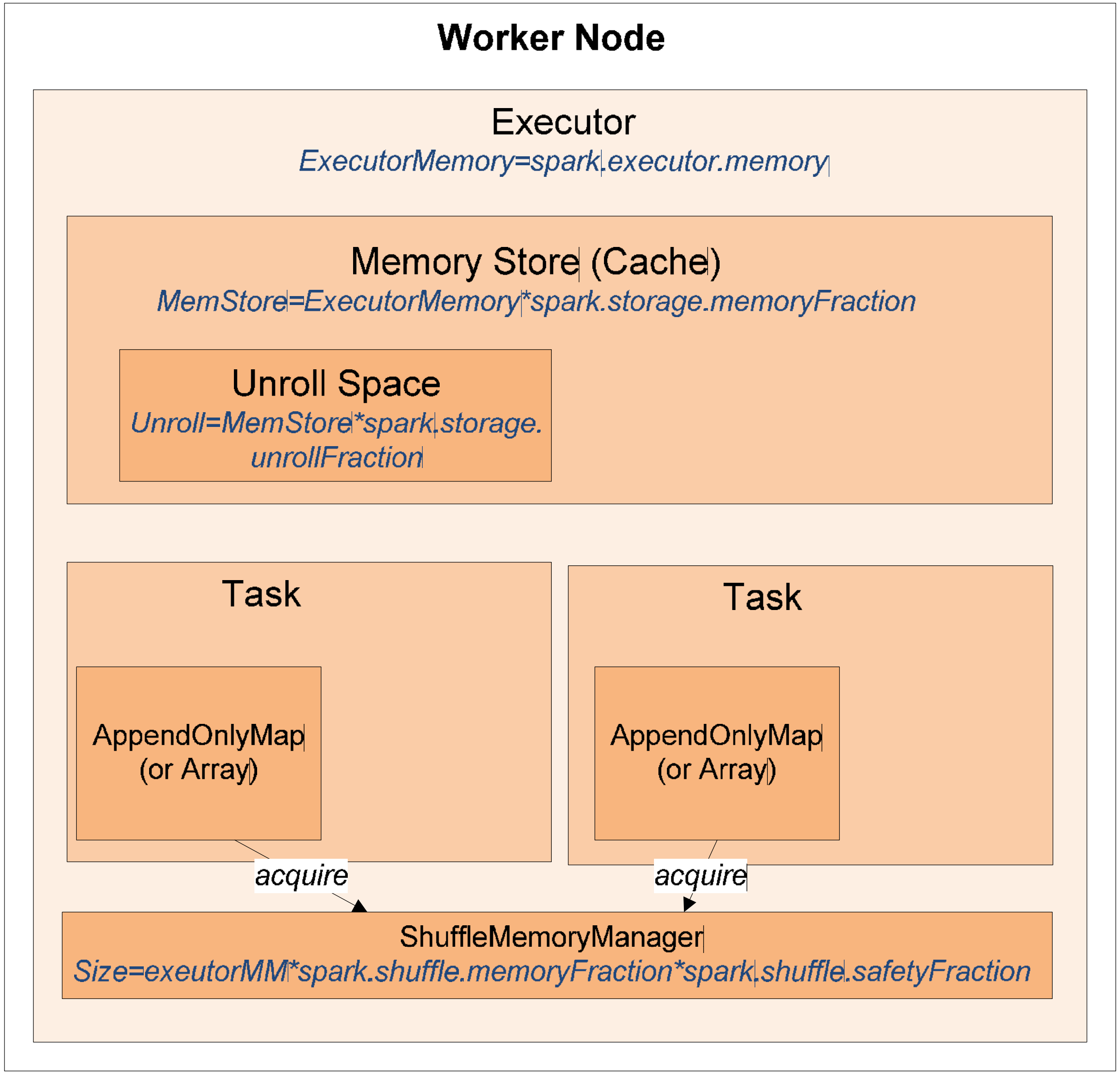
- 本质是基于排序和基于Hash的聚合框架的区别

Put(key, func(HashMap.get(key), value))

```
// MapReduce
reduce(K key, Iterable<V> values) {
    result = process(key, values)
    return result
}
```

```
// Spark
reduce(K key, Iterable<V> values) {
    result = null
    for (V value : values)
        result  = func(result, value)
    return result
}
```

Spark 内存分配



Worker Node
Executor
ExecutorMemory=spark.executor.memory
Memory Store (Cache)
MemStore=ExecutorMemory*spark.storage.memoryFraction
Unroll Space
Unroll=MemStore*spark.storage.unrollFraction
Task
AppendOnlyMap (or Array)
Task
AppendOnlyMap (or Array)
acquire
acquire
ShuffleMemoryManager
Size=exeutorMM*spark.shuffle.memoryFraction*spark.shuffle.safetyFraction

议程表

- Hadoop MapReduce简介

- Spark简介

- Hadoop vs. Spark 性能比较

比较的关键组件

- Shuffle: Exchange intermediate data between two computational stages
  - Affects scalability of a framework
- Execution Model: How user defined functions are translated into a physical execution plan
  - Affects resource utilization for parallel task execution
- Caching: Reuse of intermediate data across multiple stages
  - Speeds up iterative algorithms at the cost of additional space in memory or on disk

集群调优

Spark Configuration

- Clean the OS cache before each run
- Increase the memory capacity of worker/executor to 32 GB (i.e. 1 GB per task)
- WORKER_INSTANCES = 8 (i.e. 8 * cores tasks in parallel)
- Use snappy compression for map output
- spark.shuffle.file.buffer.kb = 32
- spark.shuffle.consolidateFiles = true
- Reducers = 500
- Output dfs replica = 1

MR Configuration

- Clean the OS cache before each run
- Change maximum assigned tasks per heartbeat to 64 so that all parallel tasks can be started in one heartbeat
- Other tuned parameters
  - mapreduce.map.output.compress=true \
  - mapreduce.map.output.compress.codec=org.apache.hadoop.io.compress.SnappyCodec \
  - mapreduce.task.io.sort.mb=500 \
  - mapred.reduce.tasks=120 \
  - mapreduce.job.reduce.slowstart.completedmaps=0.25 \
  - mapreduce.reduce.shuffle.parallelcopies=2 \
  - mapreduce.reduce.shuffle.input.buffer.percent=0.5 \
  - mapreduce.reduce.java.opts=-Xmx8192m \
  - mapreduce.reduce.memory.mb=8192 \
  - mapreduce.map.memory.mb=8192 \

Word Count

Overall Results: Word Count

Platform	Spark	MR	Spark	MR	Spark	MR
Input size (GB)	1	1	40	40	200	200
Number of map tasks	9	9	360	360	1800	1800
Number of reduce tasks	8	8	120	120	120	120
Job time (Sec)	26	64	71	180	237	630
Median time of map tasks (Sec)	5	34	12	40	12	40
Median time of reduce tasks (Sec)	2	4	8	15	32	50
Map Output on disk (GB)	0.02	0.015	0.9	0.7	4.1	3.5

- Map
  - Spark is 3x faster than MapReduce
- Reduce
  - Similar
- Why?

Sort

Overall Results: Sort

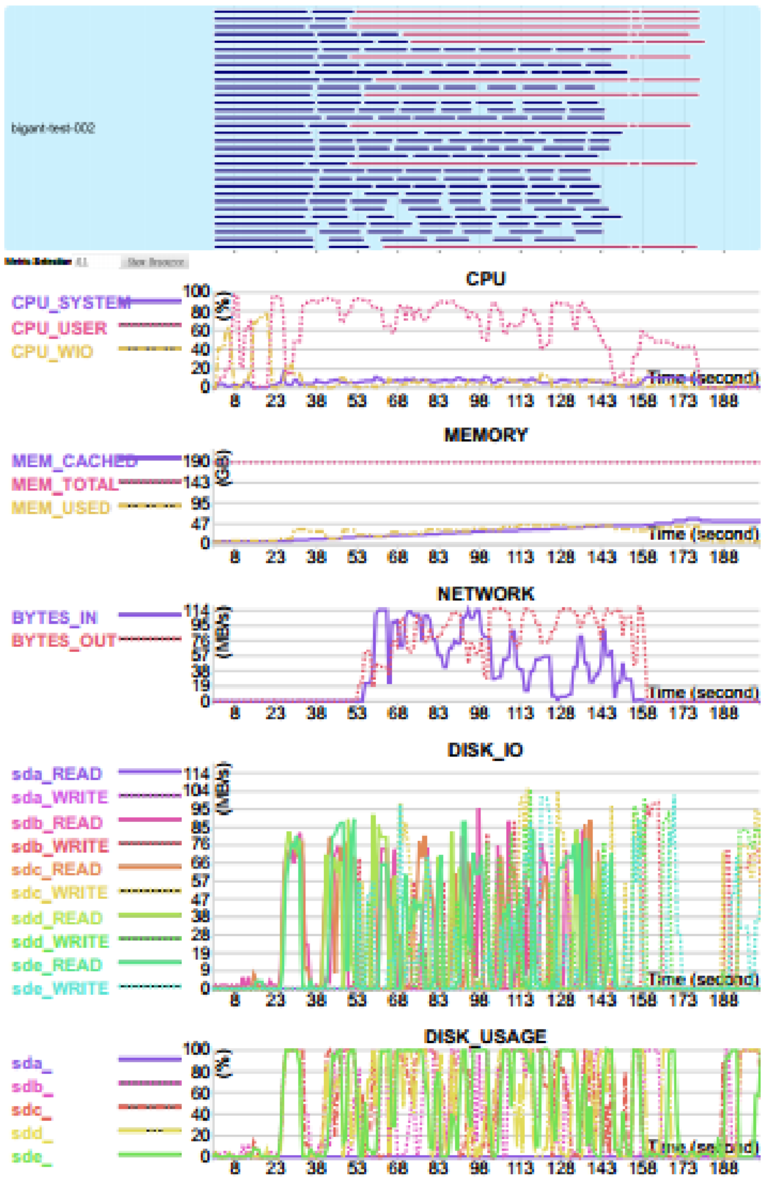
Platform	Spark	MR	Spark	MR	Spark	MR
Input size (GB)	1	1	100	100	500	500
Job time	22sec	35sec	286sec	214sec	48min	24min
Number of map tasks	9	9	745	745	4000	4000
Number of reduce tasks	8	8	248	60	2000	60
Sampling stage time	5sec	1sec	78sec	1sec	5.1min	1sec
Map stage time	6sec	11sec	60sec	150sec	25min	13.9min
Reduce stage time	7sec	24sec	148sec	45sec	16min	9.2min
Map output on disk (GB)	0.48	0.44	44.9	41.3	252.1	227.2

- Spark is 1.3x and 2x faster than MR for 100 GB and 500 GB Sort

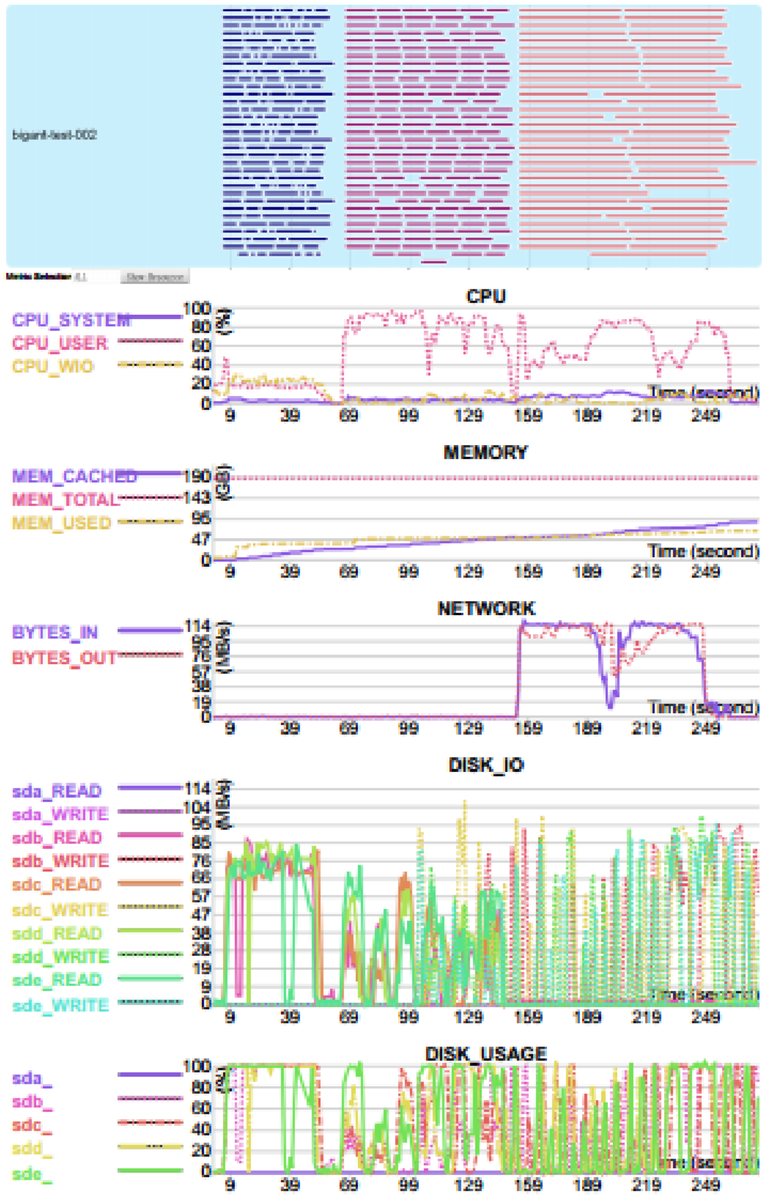
- Why?

Sort执行计划

- Overlap
- Scalability issues of Spark
  - Number of opened files
  - Page swapping algorithm
  - GC overhead



(a) MapReduce



(b) Spark

Kmeans

Overall Results: K-means

Platform	Spark	MR	Spark	MR	Spark	MR
Input size (million records)	1	1	10	10	192	192
Map stage time 1st (Sec)	11	19	15	31	92	140
Reduce stage time 1st (Sec)	1	1	1	1	1	1
Map stage time SubSeq. (Sec)	3	19	3	31	26	140
Reduce stage time SubSeq. (Sec)	1	1	1	1	1	1
Shuffle data (KB) (GB)	7	415	17	419	171	4316

- Caching raw file vs. objects
- Impact of caching: CPU or disk?
- OS buffer caches
- DFS replicas for OS buffer caches

The Impact of Storage Levels

Storage Levels	Caches Size	First Iterations	Subsequent Iterations
NONE	-	1.5 min	1.4 min
DISK_ONLY	36.1 GB	1.5 min	29 sec
DISK_ONLY_2	36.1 GB	2.1 min	28 sec
MEMORY_ONLY	42.9 GB	1.5 min	26 sec
MEMORY_ONLY_2	42.9 GB	2.1 min	26 sec
MEMORY_ONLY_SER	36.1 GB	1.7 min	29 sec
MEMORY_ONLY_SER_2	36.1 GB	2.1 min	31 sec
MEMORY_AND_DISK	42.9 GB	1.5 min	26 sec
MEMORY_AND_DISK_2	42.9 GB	1.8 min	27 sec
MEMORY_AND_DISK_SER	36.1 GB	1.5 min	29 sec
MEMORY_AND_DISK_SER_2	36.1 GB	2.0 min	29 sec
OFF_HEAP (Tachyon)	36.1 GB	1.7 min	30 sec

almost no difference

PageRank

Overall Results: PageRank

Platform	Spark-Naive	Spark-GraphX	MR	Spark-Naive	Spark-GraphX	MR
Input (million edges)	17.6	17.6	17.6	1470	1470	1470
Pre-processing	28 sec	35 sec	93 sec	7.4 min	3.8 min	8.0 min
1st Iter.	5 sec	5.3 sec	43 sec	5.0 min	1.3 min	9.3 min
Subsequent Iter.	1 sec	2.5 sec	43 sec	2.8 min	0.8 min	9.3 min
Shuffle data	65 MB	55 MB	141MB	7.6 GB	8.8 GB	21.5GB

- **Data pipelining in Spark: avoid materializing graph data structures on HDFS across iterations**
  - –serialization/de-serialization, disk I/O and network I/O
- **Power-law of social networks**
  - –Serialization/de-serialization overhead

The Impact of Storage Levels for PageRank

Storage Levels	Algorithm	Caches (GB)	First Iteration (min)	Subsequent Iteration (min)
NONE	Naive	-	6.5	5.3
MEMORY_ONLY	Naive	77.1	5.0	2.8
DISK_ONLY	Naive	15.5	33	2.8
MEMORY_ONLY_SER	Naive	15.5	31	3.1
OFF_HEAP (Tachyon)	Naive	15.5	36	3.1
NONE	GraphX	-	6.5	-
MEMORY_ONLY	GraphX	62.2	2.8	0.8
DISK_ONLY	GraphX	39.6	1.6	1.3
MEMORY_ONLY_SER	GraphX	39.6	1.6	1.4

6x slower!

结论

- MR和Spark处理大数据都需要理解物理执行过程和调优

- 不同类型作业M/R和Spark的性能表现差异较大。

- Spark基于线程的模型虽然减少了上下文切换和任务启动的开销，但是对GC带来较大挑战。