Distributed Systems
CS 545

Badri Nath
Rutgers University
badri@cs.rutgers.edu

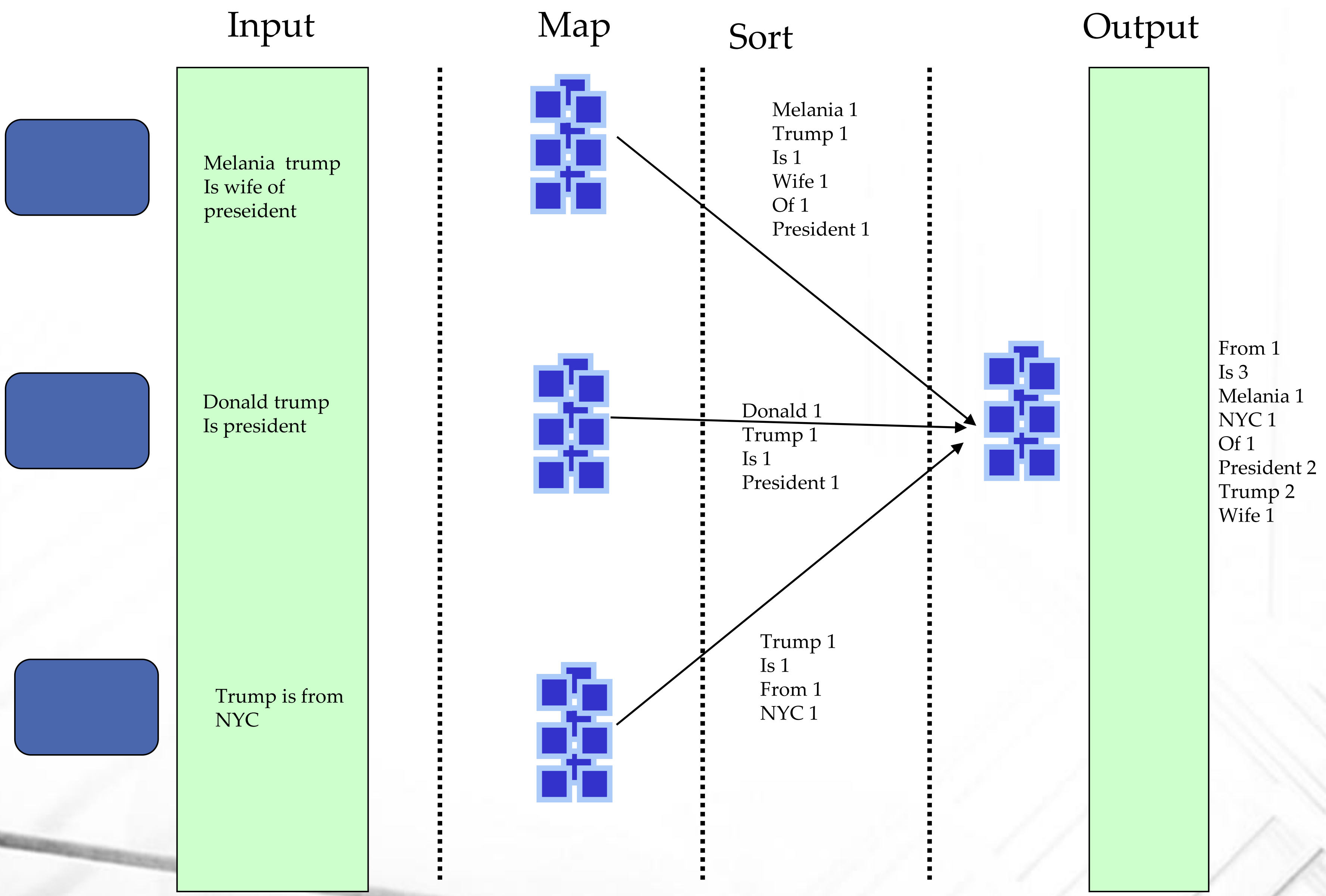
Distributed Programming Models
Map reduce, Spark

1. Map reduce SOSP 2004
2. Map reduce, CACM , 2010
3. RDD, NSDI , 2012

Map reduce

- Programming paradigm for large scale distributed computing
- You are already familiar with python functions for map, reduce, filter
- All these functions operate over iterators
- [y]<-map(f,[x]), r<-reduce(f,[x]) ,[y]<-filter(f,[x])
- What if you want to do these operations on lists of size in the M, G, or even P
- Run it in a cluster (shared disk)
- Run it on PACKS (partition the data)

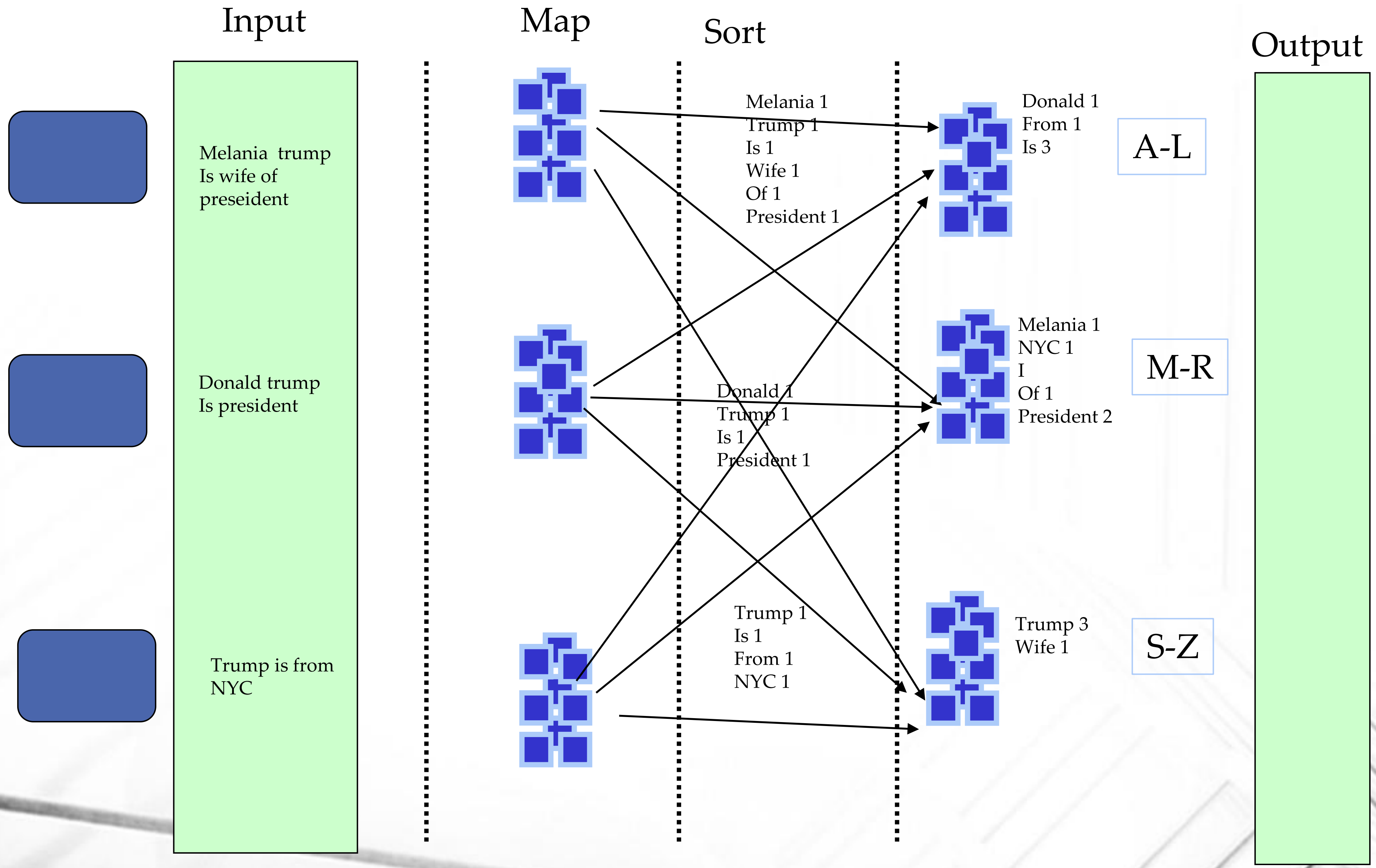
Term frequency: count the occurrence of each word



Problem with this approach

- Result is on one machine
- What if result is huge; frequency count of first name of all facebook users!!
- Vertical scale server – cost - growth
- Network bandwidth
- Single point of failure
- Storage of map results? Disk or memory

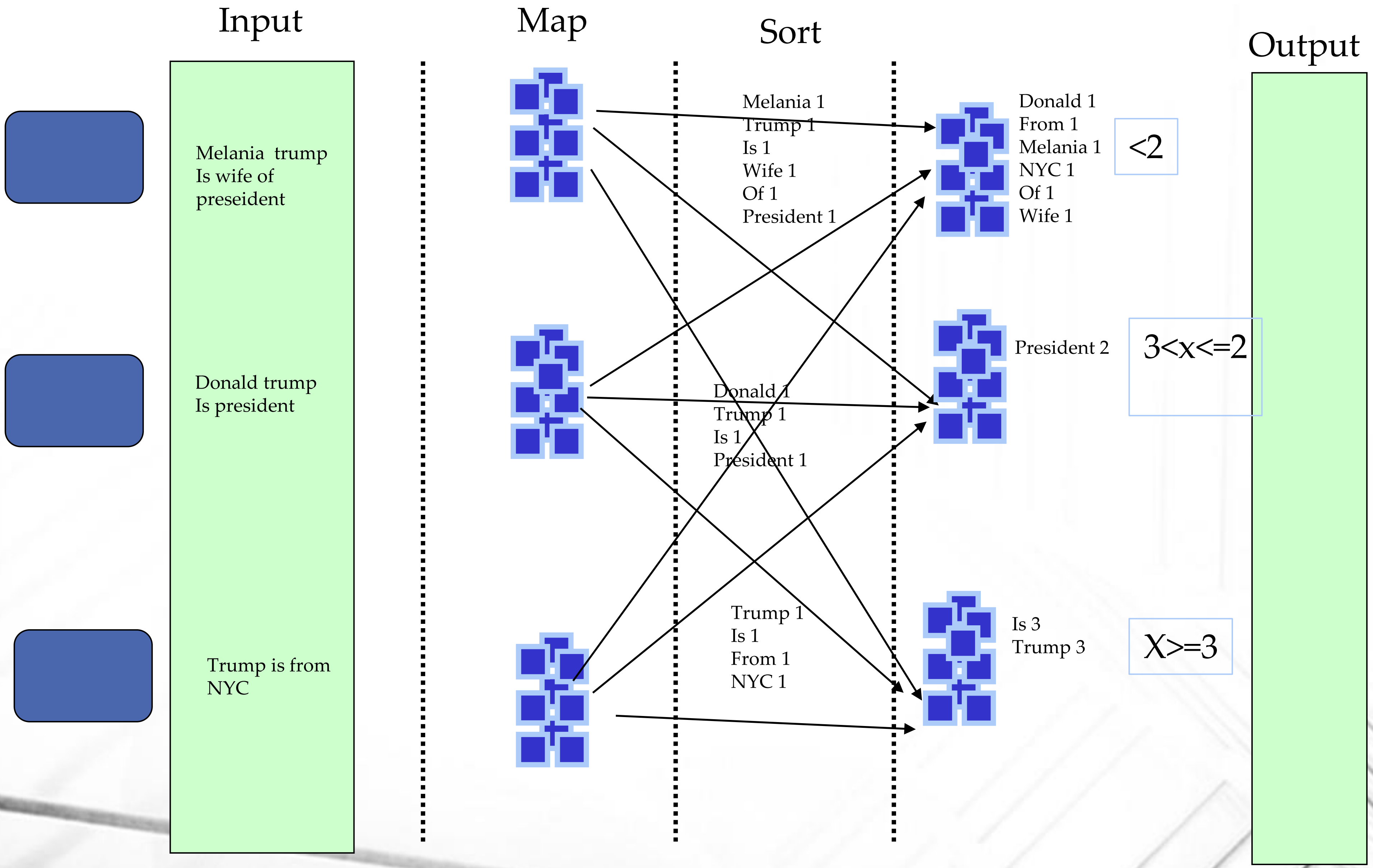
Term frequency: partition the result as well



Map, reduce functions

- This basic idea of partitioning data and programs to work on large datasets over a distributed cluster is MR
  - – MR paper by Jeff Dean and Sanjay Ghemawat 2004 SOSP- citation today 26532
- Map (k1,v1) → list (k2, v2)
  - E.g., k1 document names, v1 content of documents
  - E.g., k2 words, v2 count of words in doc
- Reduce (k2, list(v2)) → k2, sum(list(v2))
  - E.g., sum is word count for each word
- Map(Ki, Vi) ➔ [(ka,va), (kb,vb),(ka,vx)….]
- Reduce(ka,[va,vx])➔ [(kx,vj),….]

Term frequency sorted based on count



Basis of Map reduce

- Model taken from Functional programming such as list
  - Map (square ‘ (1, 2, 3, 4)) → (1, 4, 9, 16)
  - Reduce (sum ‘ (1, 4, 9, 16)) → (30)
- Distributed Grep
  - `cat inp.dat  | grep | sort | uniq -c | cat > out.dat`
  - `Input|            map| sort | reduce  | output`

Processing lots of data

- O(B) web pages; each O(K) bytes to O(M) bytes gives you O(T) to O(P) bytes of data
- O(B) FB pages; … X data per page O(P) or even O(E) bytes of data
- Disk Bandwidth per computer O(100 MB/sec)
- Processing time $O(10^6)$ secs for O(T) data
- Reduce it to $O(10^3)$ with 1 K processors
- Need high parallelism to process web data
- Web Processing: Process, transform, store

Programming model

- Computation takes input of key, value pairs and <u>transforms</u> into output of key value pairs
- Divided into two
  - map function that produces from the input an intermediate set of key value pairs
  - Reduce function that takes the intermediate key value pairs and merges the values for a given key
- Inherent parallelism as map operates on partition of input and no dependency between map processes

MR Execution Overview

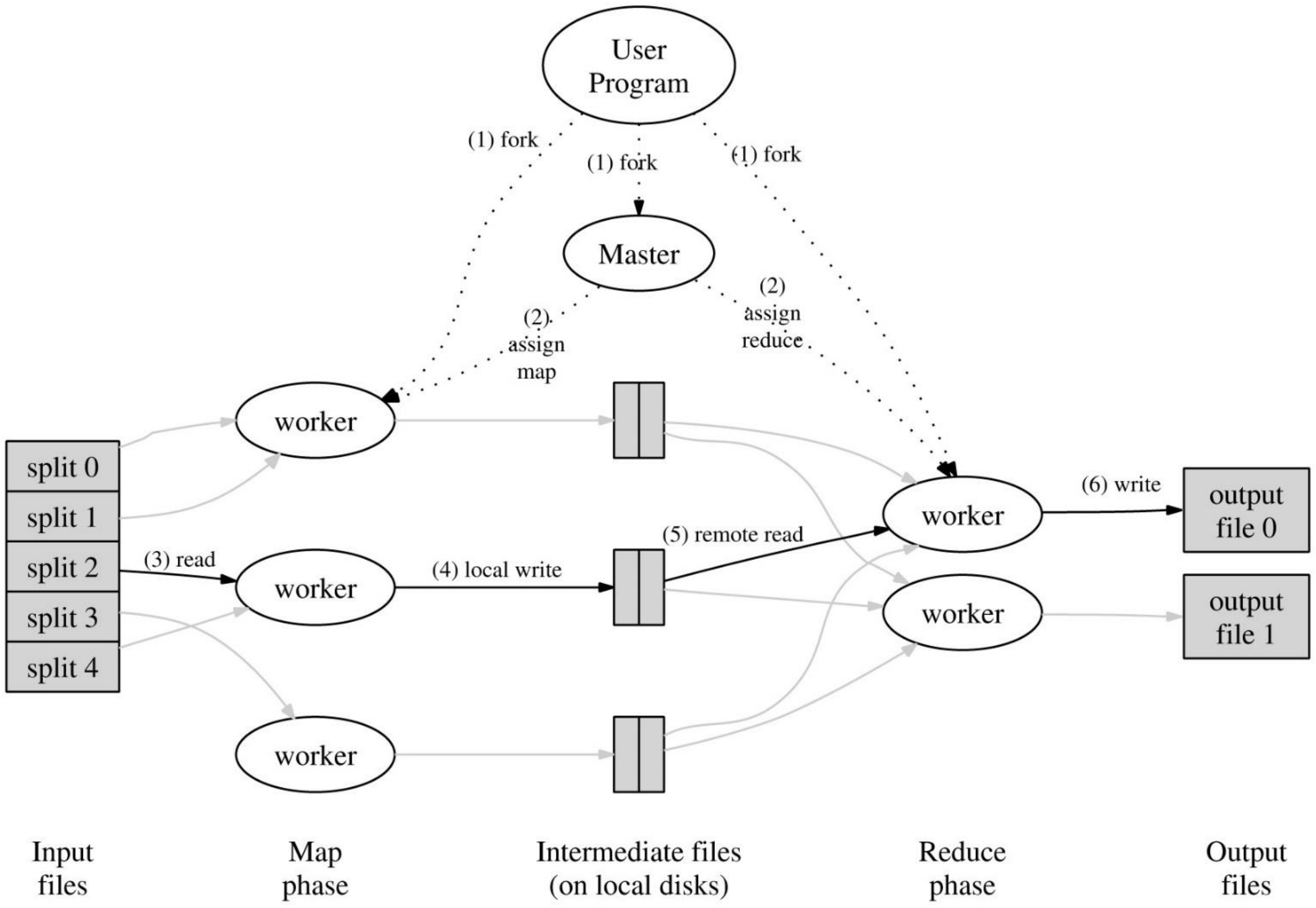
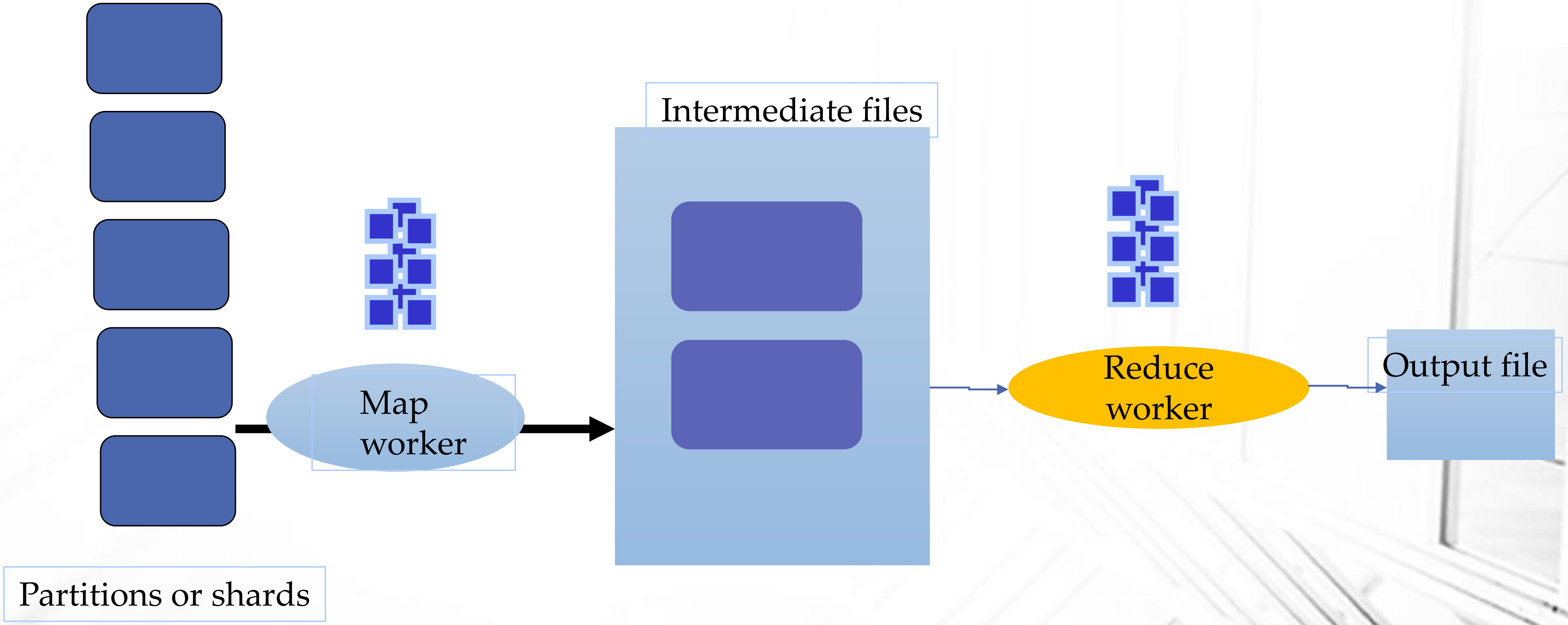


Figure 1: Execution overview

MR execution steps

- Partition the input file <1…N>
- Master thread creates as many worker threads as needed
- Workers is assigned a map task that takes data from partition (i)
- Map outputs to buffer (intermediate values)
- Master notifies the reduce worker, it reads the the output produced by the mapper
- Reduce worker iterates over sorted data and applies the reduce function
- The output of the reduce function is the final result

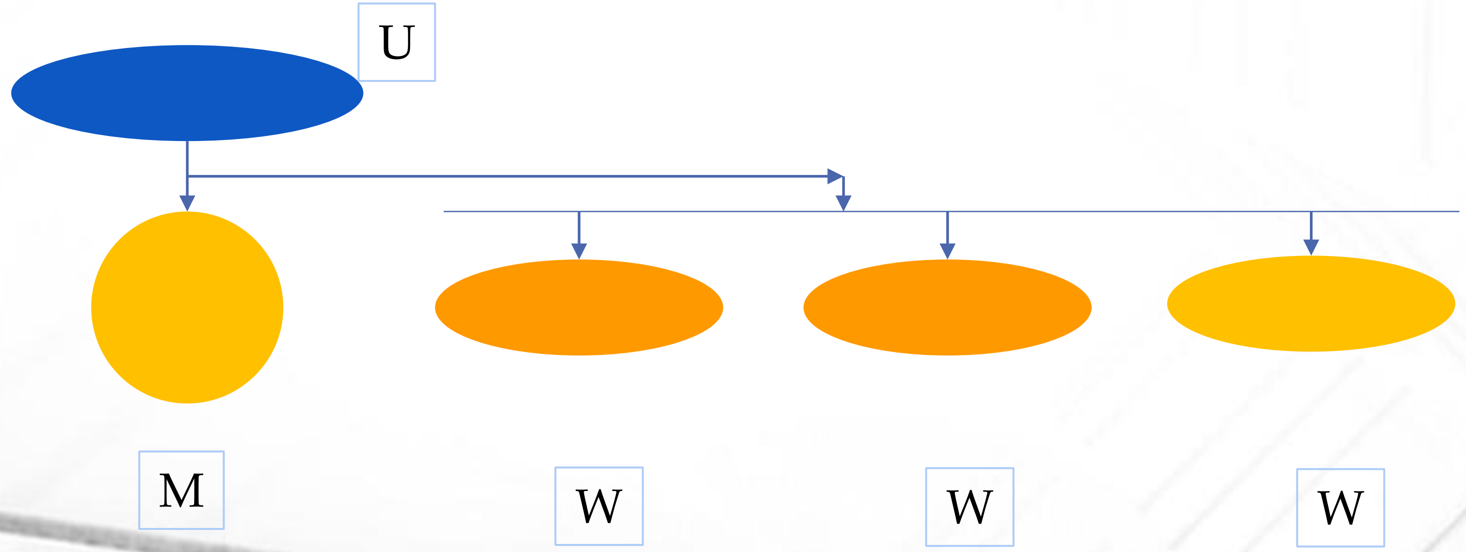
Basic steps



Intermediate files
Map
worker
Reduce
worker
Output file
Partitions or shards

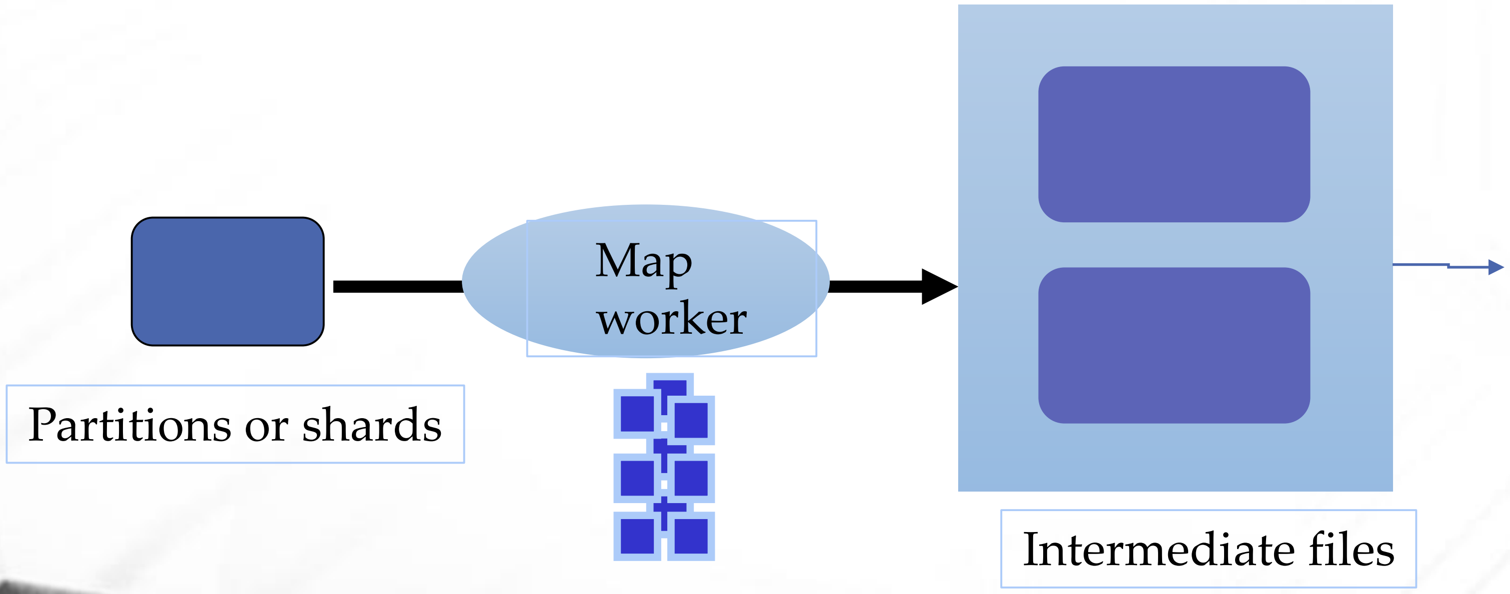
Create map, reduce processes

- User program spawns, master and worker processes
- Master is the scheduler and coordinator
- Worker threads are either map or reducer
- Map works on input files, reduce workers work on intermediate files



Mapper process flow

- Read contents from its partition
- Passes key, value pair to user provided map function
- Output of function is intermediate key value pairs
- Intermediate key value pairs written onto buffer and then onto disk

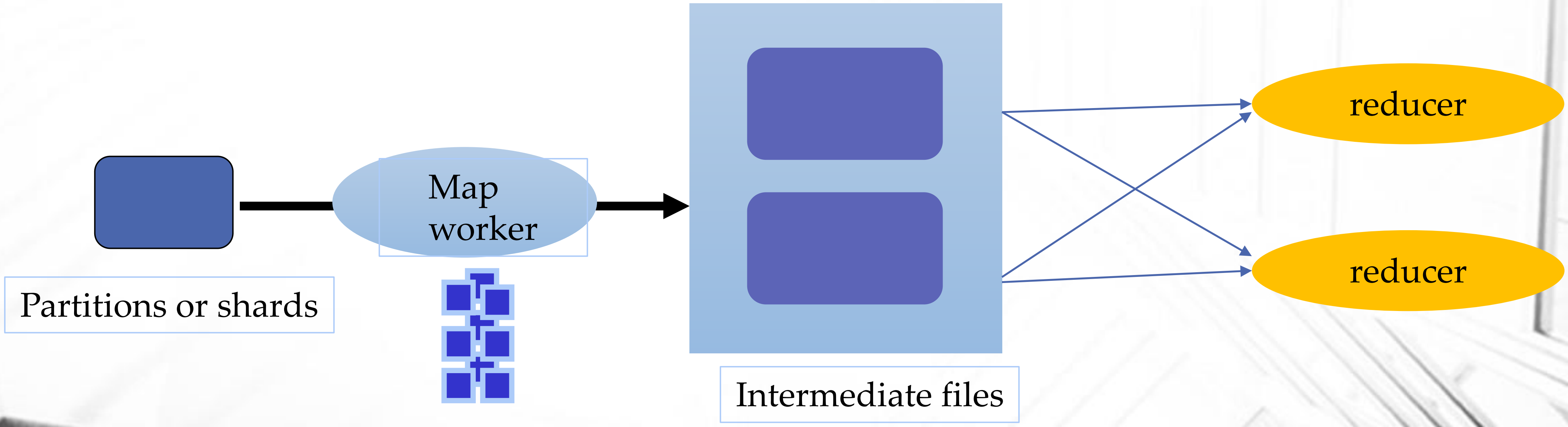


reducer process flow

Master informs reduce threads location of Intermediate files for its partition

Reducer reads sorted by intermediate keys

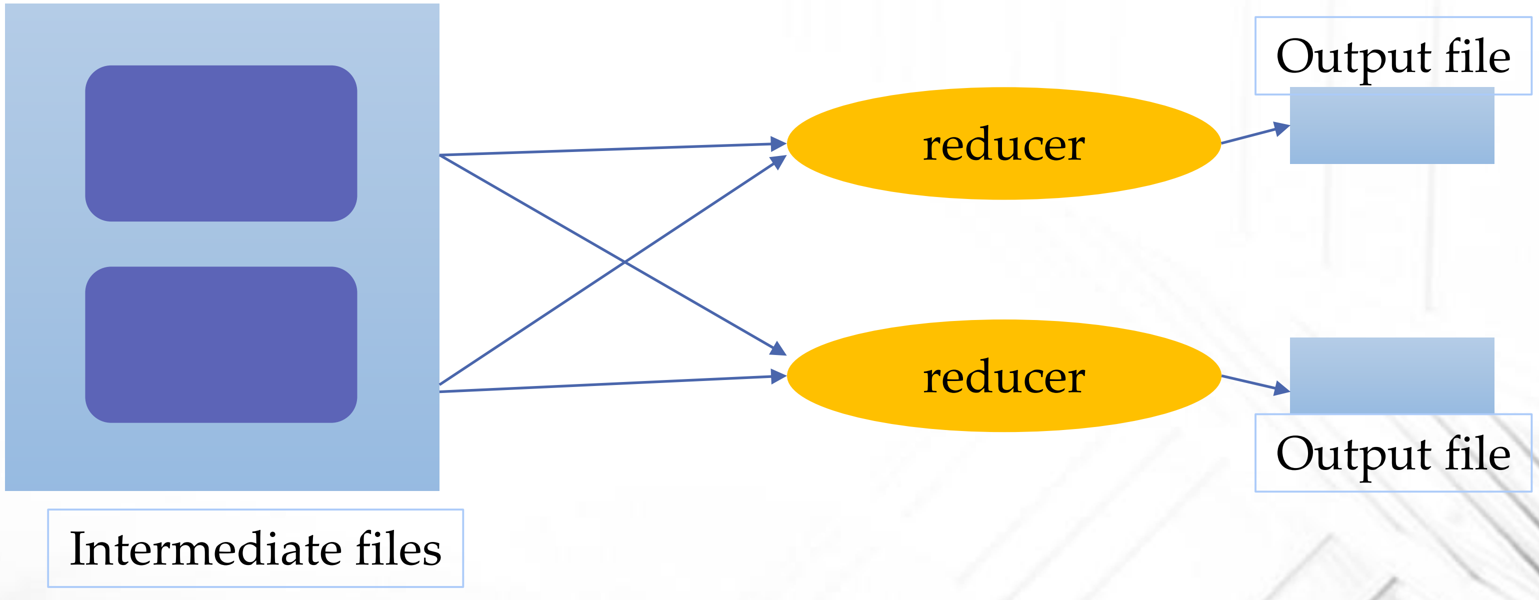
Keys are grouped together



reducer final step

User provided reduce function is applied to key value tuples

Result is written onto output file



MR steps summary

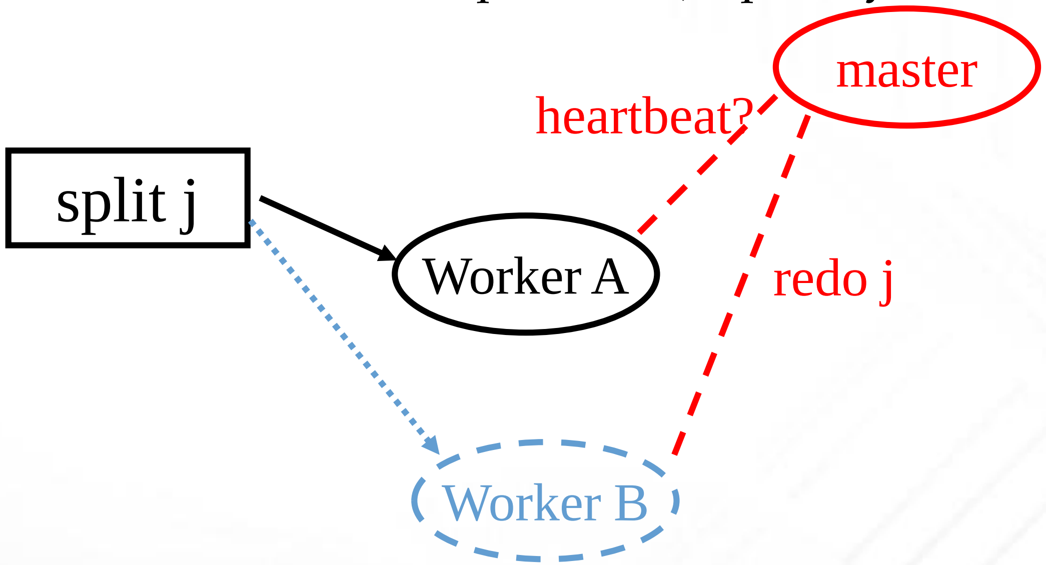
- Break data into K partitions
- Span number of worker threads (map)
- Run map tasks on key, value iterator
- Produce intermediate files that are output of map functions
- Intermediate files contain <key, value> lists
- Reduce worker "pulls" data from Intermediate files
- Reduce computes aggregate function on specific keys
- Final: After all the map, reducer have executed, master wakes up the user program
- Output of MR available in output files

Implementation Issues

- Failures
- Locality
- Partition and Combine function
- Backup tasks
- Ordering of results

Failures

- A parallel, distributed computation
- Running on thousands of machines
- Need to deal with failures
- Master detects worker failures, and has work re-done by another worker.
- Master failure : restart the computation (hopefully not make a habit of it)



MR failure: mapper fails, reexecute

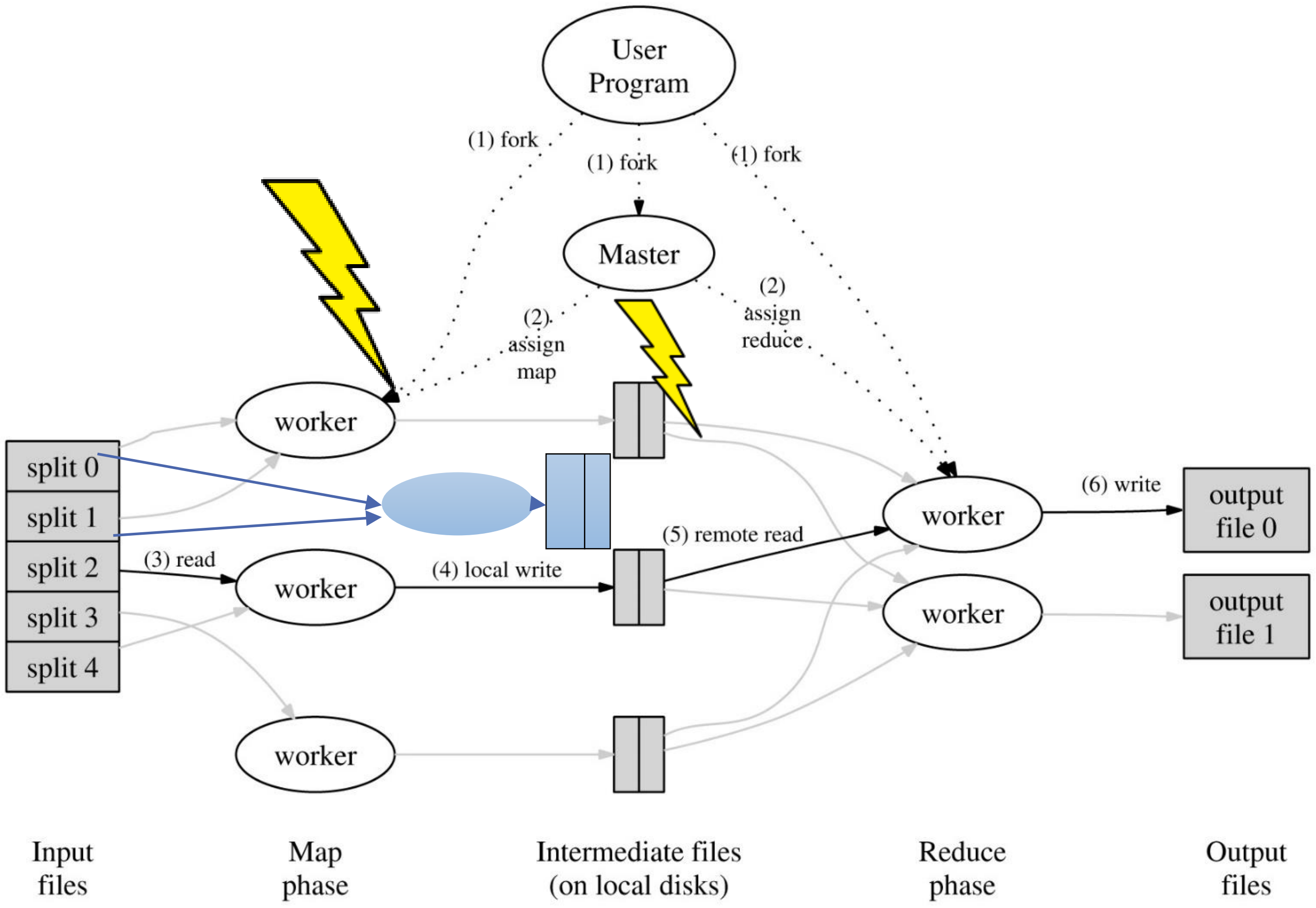


Figure 1: Execution overview

MR failure: mapper fails, reexecute, but periodically checkpoint

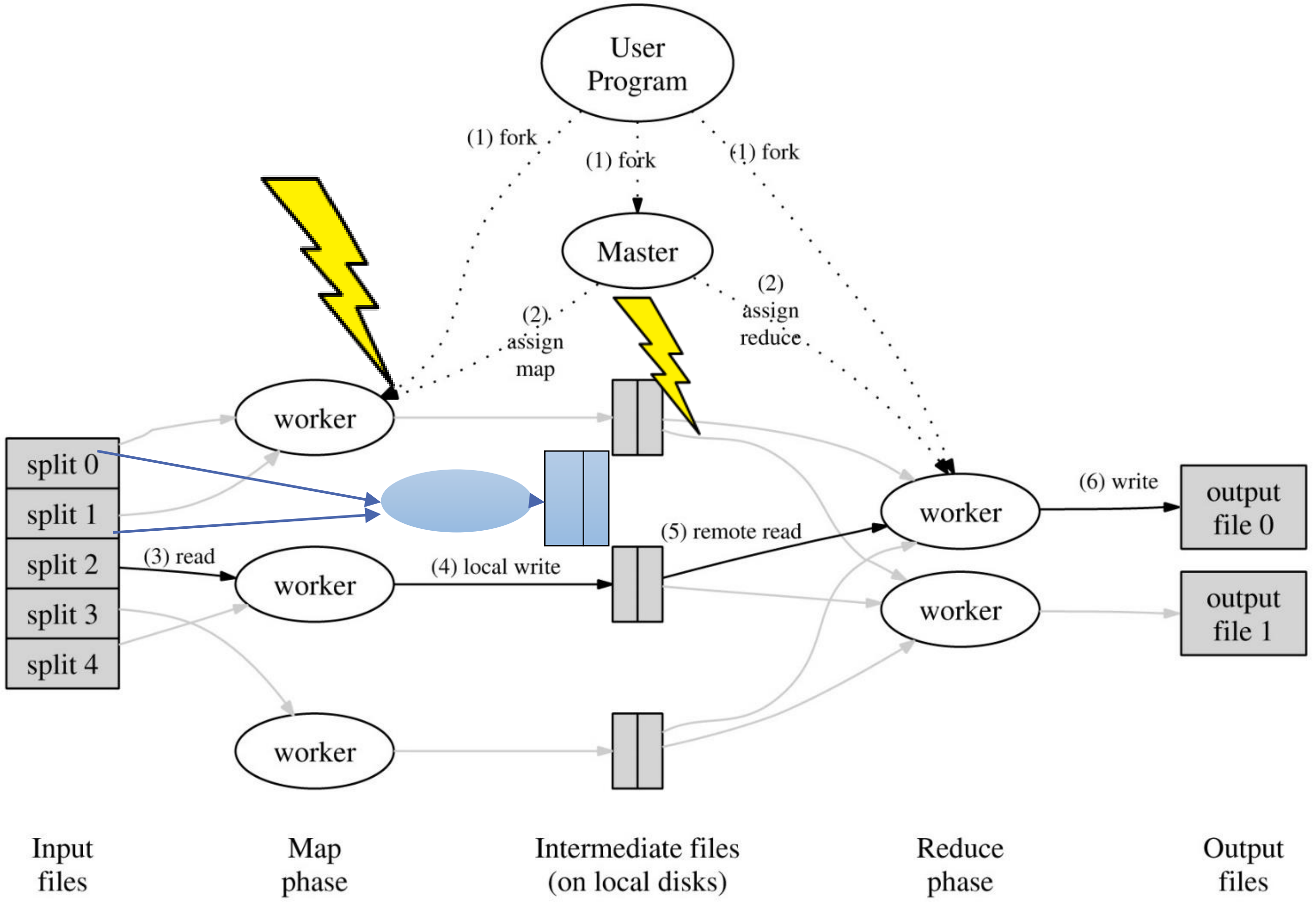


Figure 1: Execution overview

Master failure: reexecute from log

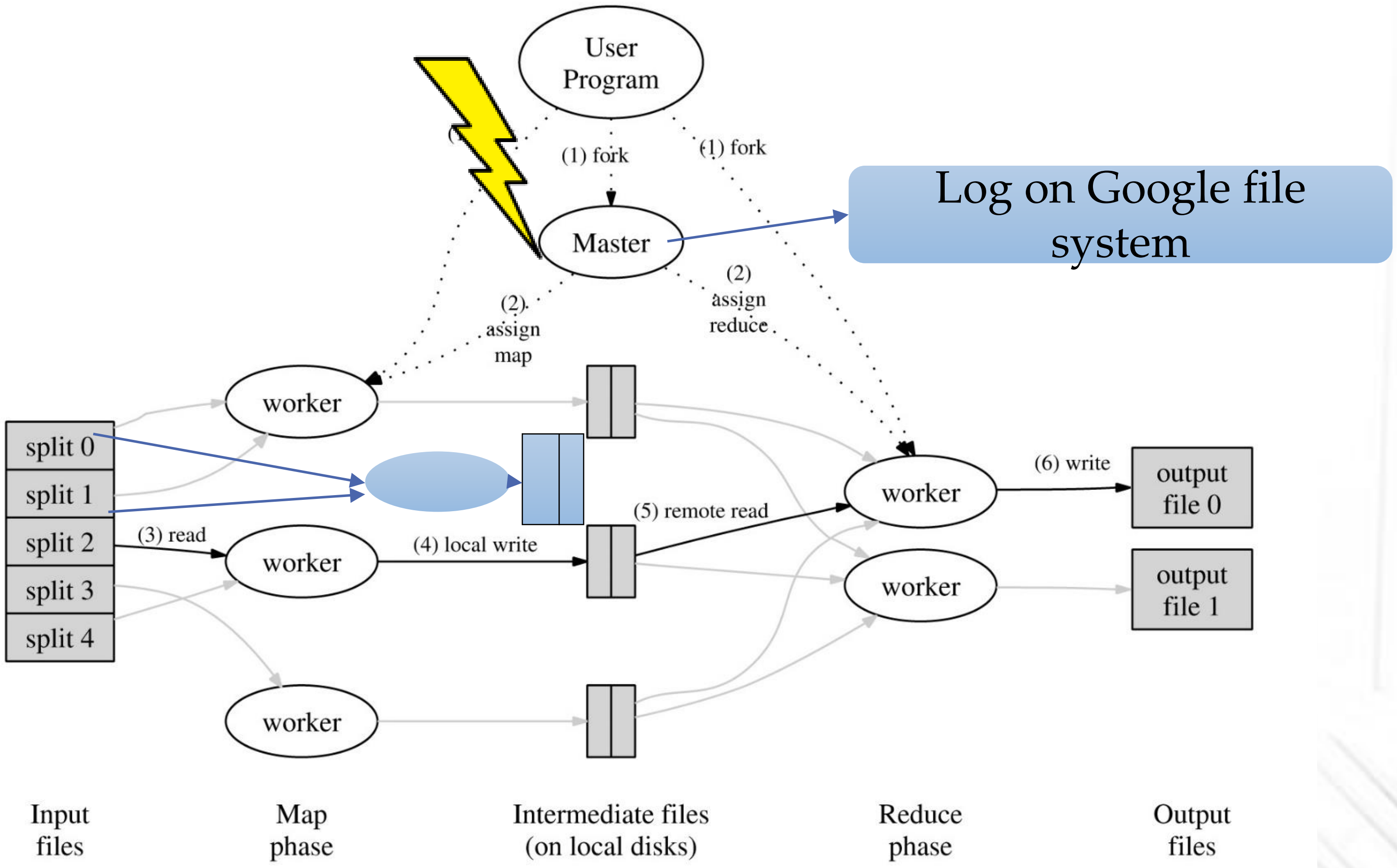


Figure 1: Execution overview

Failure Semantics

- Distributed computation should produce the same result as non-faulting sequential execution of single program
- Atomic commit of map and reduce tasks
- Written to files and communicated to master
- if multiple copies of the same reduce task are executed, atomic rename will be used so that only 1 out file is created despite redundancy

Slow mappers

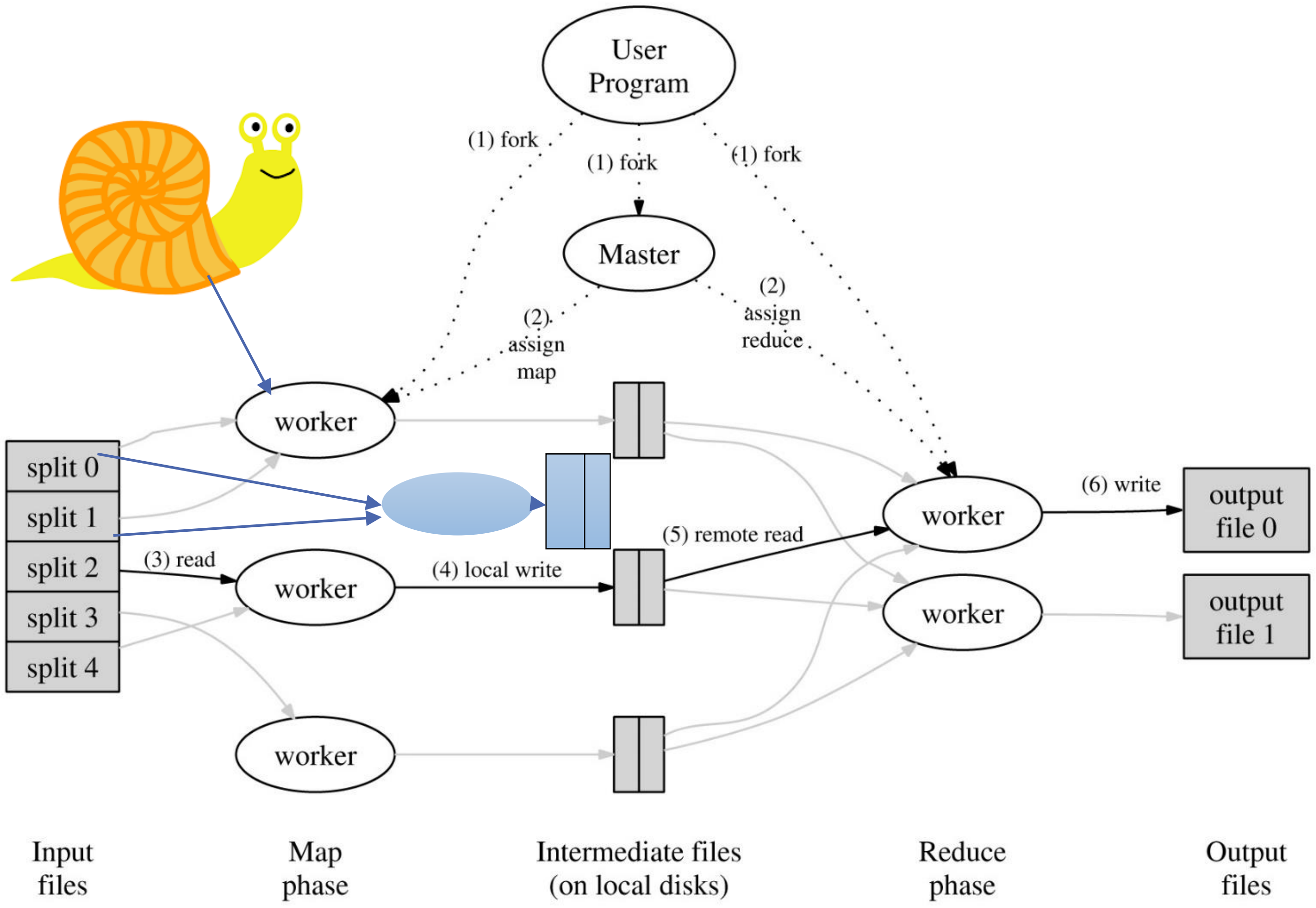


Figure 1: Execution overview

Backup Tasks

- Reduce task cannot start until map is complete
- Straggler is a machine that takes unusually long (e.g., bad disk) to finish its work.
- A straggler can delay final completion.
- When task is close to finishing, master schedules backup executions for remaining *in-progress* tasks.
- Must be able to eliminate duplicate results

Locality

- Master program: assigns task threads based on location of data: tries to have map() tasks on same machine as physical file data, or at least same rack/switch
- map() task inputs are divided into 64 MB blocks: same size as Google File System chunks
- Working set mapped to underlying GFS

Task Granularity

- M + R >> Number of machines (for load balancing)
- Not too many Rs, final result need to be combined
- Master needs to keep a mapping of O(M*R)
- M = K (Number of machines) K is 100
- R = F (number of machines) F = 2.5

Natural-Join Operation

Natural Join : rows in R union S, where the values of the attributes in R∩S are same

- Notation: r ⋈ s

Example:

$R = (A, B, C, D)$

$S = (E, B, D)$

- Result schema = (A, B, C, D, E)
- $r \bowtie s$ is defined as:

$$\prod_{r.A,\ r.B,\ r.C,\ r.D,\ s.E} (\sigma_{r.B = s.B \wedge r.D = s.D} (r \times s))$$

Natural Join Operation – Example

- Relations r, s:

A	*B*	*C*	*D*
A	*1*	*A*	a
B	*2*	*C*	a
C	*4*	*B*	b
A	*1*	*C*	a
E	*2*	*B*	b

r

B	*D*	*E*
1	a	*A*
3	a	*B*
1	a	*C*
2	b	*D*
3	b	*E*

s

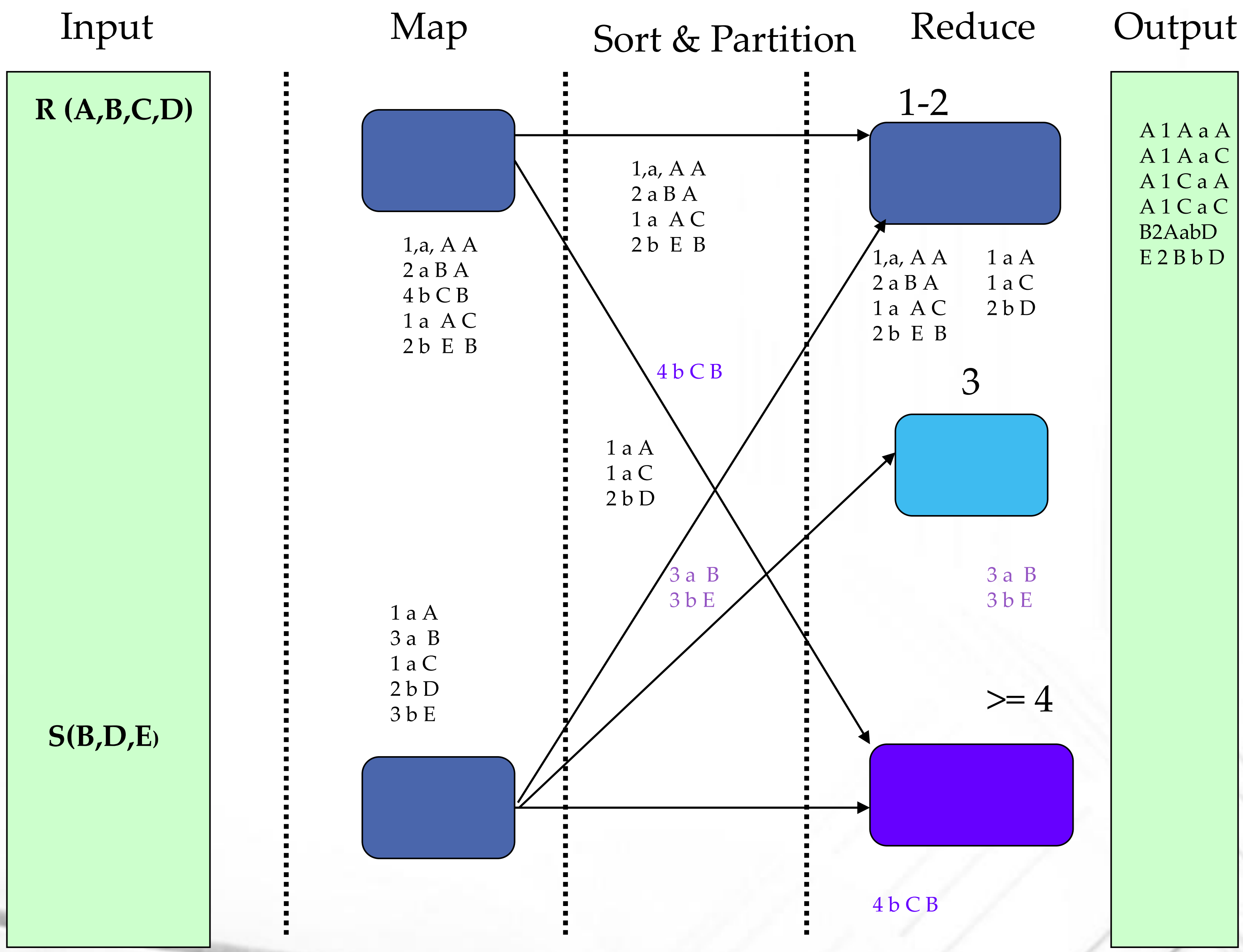
$r \bowtie s$

A	*B*	*C*	*D*	*E*
A	*1*	*A*	a	*A*
A	*1*	*A*	a	*C*
A	*1*	*C*	a	*A*
A	*1*	*C*	a	*C*
E	*2*	*B*	b	*D*

Converting into map reduce

- Convert common attributes into a key, rest of attributes of the relation is value
- Do that for R and S
- Send tuples in R, and S with the same key value to reducer
- Partition key values to distribute load among reducers
- Partition R and S among mappers

Map reduce example



Join as Map-Reduce

- Each reducer matches all pairs with same key values
- Reducer outputs (A, B, C, D, E) once all mappers are done and no more tuples
- Parallelism can be controlled by assigning horizontal fragments to mappers
- Parallelism can be controlled by adjusting range value of keys in reducers

Input/output

- Each line as key, value (grep)
- Key is the line # and value is content of line
- Sequence of key value pairs, ordered by keys
- Output file of reducer can also be stored in key order
- Any new file type should be transformed so that it is suitable for range partitioning to map tasks

Performance (this is from Circa 2004)

- Cluster Configuration
- 1800 machines
- Each machine
  - 2GHz Xeons,
  - 4GB RAM, 2 160GB disk, Gb Ethernet.
  - Two level tree switched network with 100-200 Gbps aggregate at root

Performance

- Grep experiment: look for a pattern
- M=15000, R = 1
- 10 G, 100 byte records
- 1 minute startup, 150 seconds total!
- Startup cost includes code propagation, opening files in GFS, getting GFS metadata for locality optimization.
- Completes 80 seconds after startup

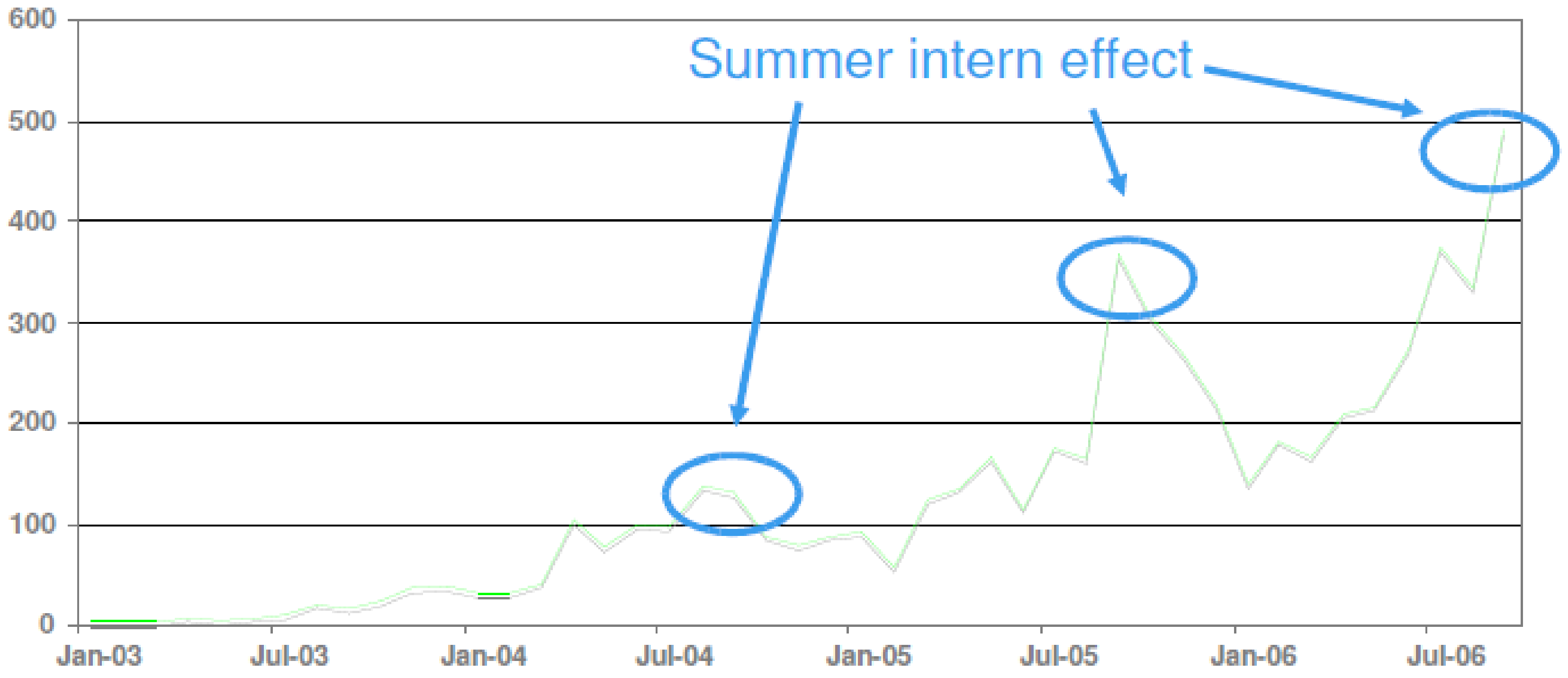
Performance

- Sort
- Input is 10 G, 100 byte records
- M = 15000, R = 4000
- Completes in 891 secs
- Terasort benchmark 1057!!

MapReduce Conclusions

- MapReduce has proven to be a useful abstraction for large scale data processing using clusters
- Greatly simplifies large-scale computations at Google
- Shown that functional programming paradigm can be applied to large-scale applications
- Lots of problems in processing of web data can be cast as map-reduce
- Now database people have joined the party
    - Map-reduce-merge (sigmod 2007)

New MapReduce Programs Per Month

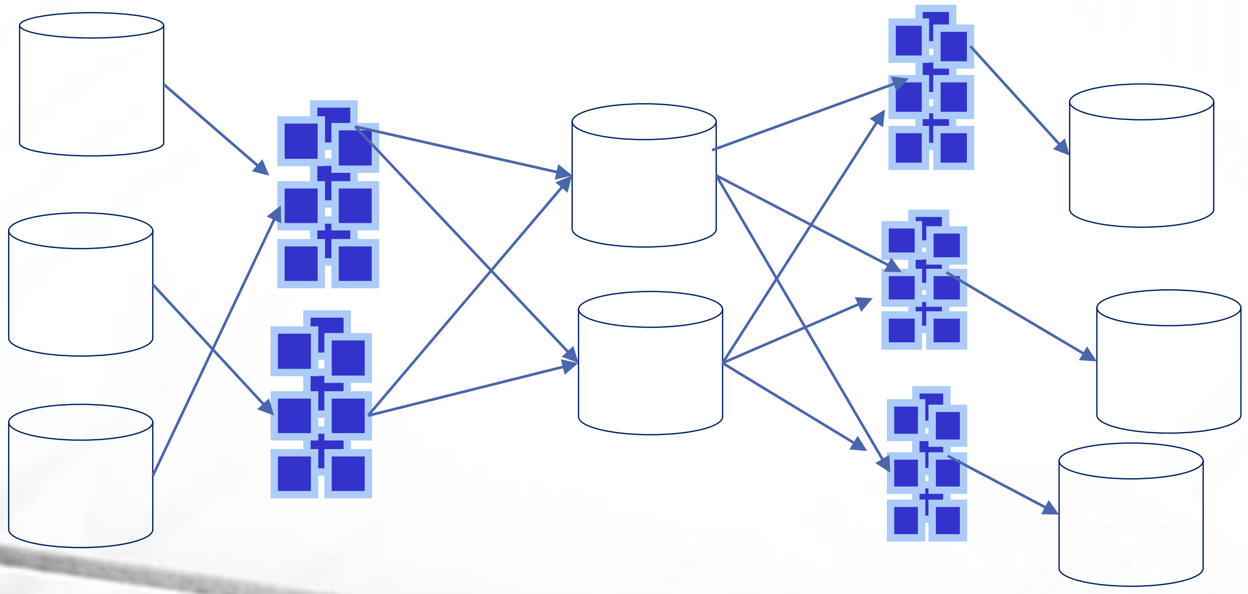


Resilient Distributed Datasets
A Fault-Tolerant Abstraction for In-Memory Cluster Computing

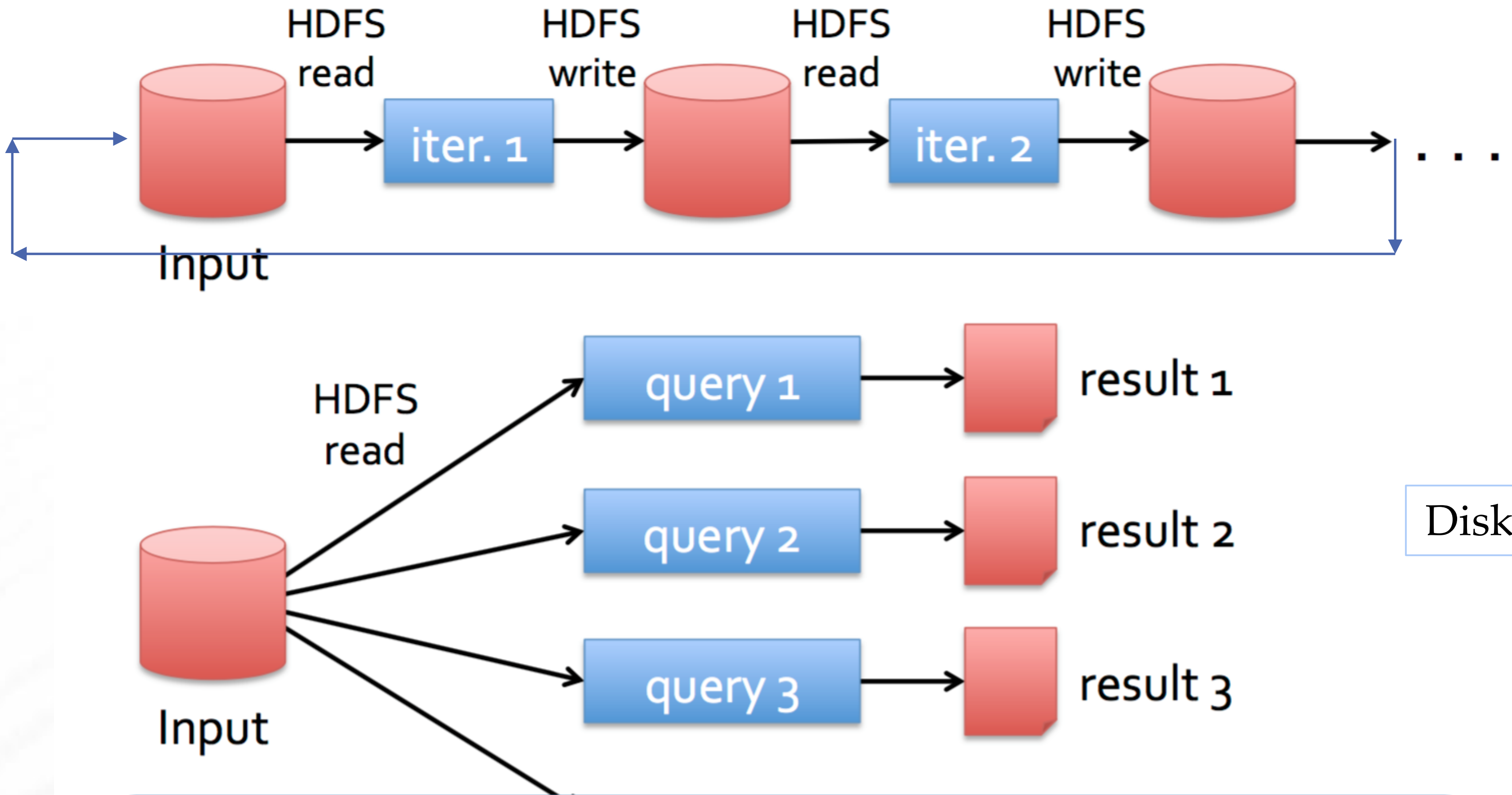
Matei Zaharia, et.al, NSDI 2012 Best Paper Award
Slides from NSDI presentation

Motivation

- In MR, mapper writes to disk (intermediate files)
- If the program has iterative MR tasks then lots of read/write from disk (M, R, M, R,…)
- What if interactive queries need to be supported from a map task?



Motivation



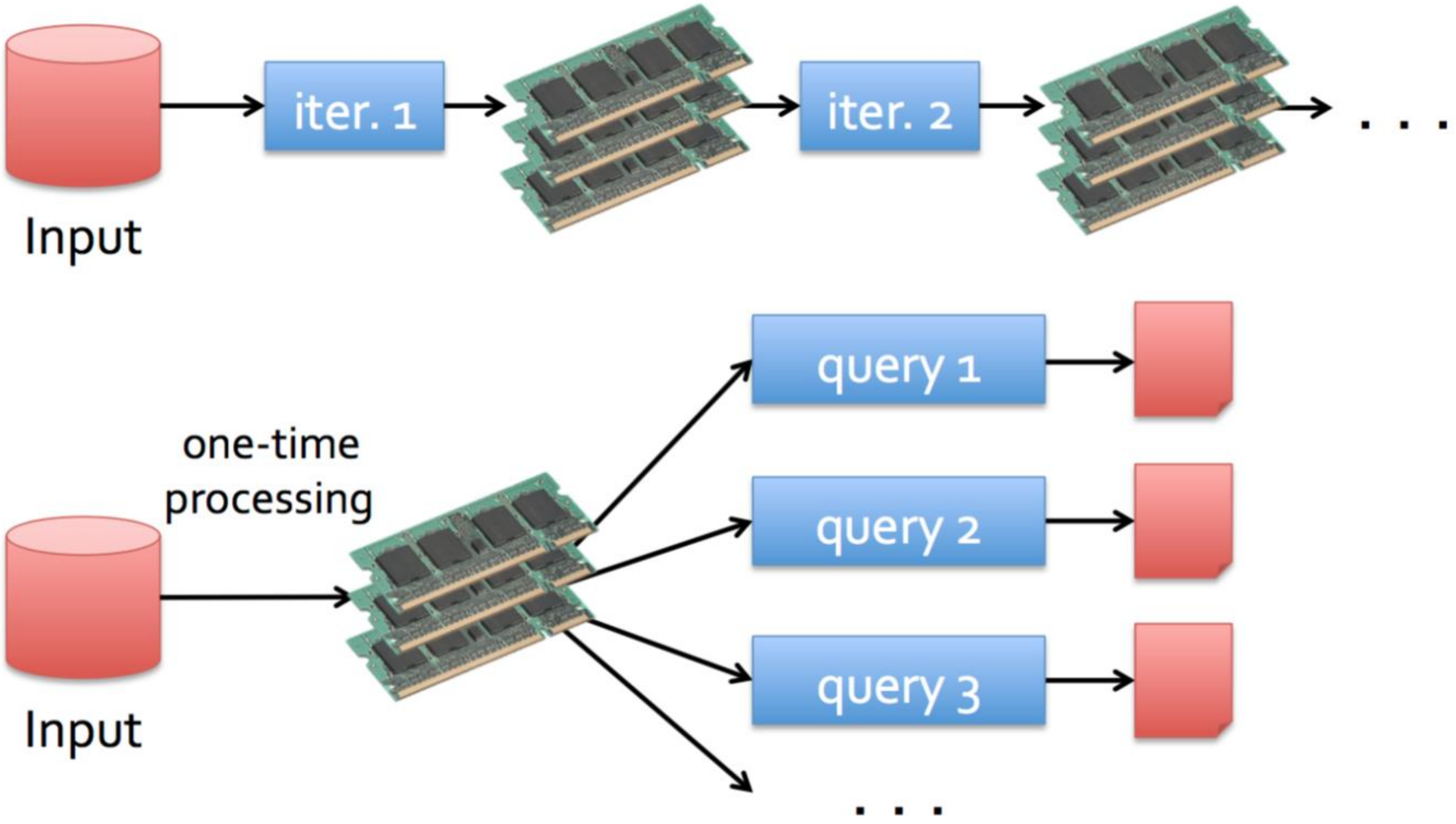
Disk I/O is very slow

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

Basic Idea

- Cost of memory is dropping
- 7$/ GB or 7 cents /MB
- Why not use memory instead of writing to disk
- Especially for iterative tasks (think ML problems)
- Queries can read from distributed memory as opposed to disks

Goal: In-Memory Data Sharing



Challenges and approach

- What happens when there is failure ?
  - Data in memory is lost
- May be periodic checkpoint data
  - But data sets are huge; defeats the purpose of using main memory
- Instead support a immutable datasets
  - Resilient Distributed Datasets (RDDs)
- Define well defined operations on these RDDs
  - Log only operations (lineage)
- On a failure, using the recovery log, rebuild RDDs
- This is done automatically
- Rest is details!!!!

What about operation logs?

- Define only coarse operations
- Apply transformations, take on RDD can create another RDD
- RDDs does not have to have values
- Just a way to construct them, (a series coarse operations)
- Users can choose a strategy for storage of RDDS
- Users can chose a strategy for partitioning (which machines store what portion of RDD based on key)

Coarse Grained: Resilient Distributed Datasets (RDDs)

- Restricted form of distributed shared memory
  - Immutable, partitioned collections of records
  - Can only be built through *coarse-grained* deterministic transformations (map, filter, join, …)
- Efficient fault recovery using *lineage*
  - Log one operation to apply to many elements
  - Recompute lost partitions on failure
  - Minimal cost if nothing fails

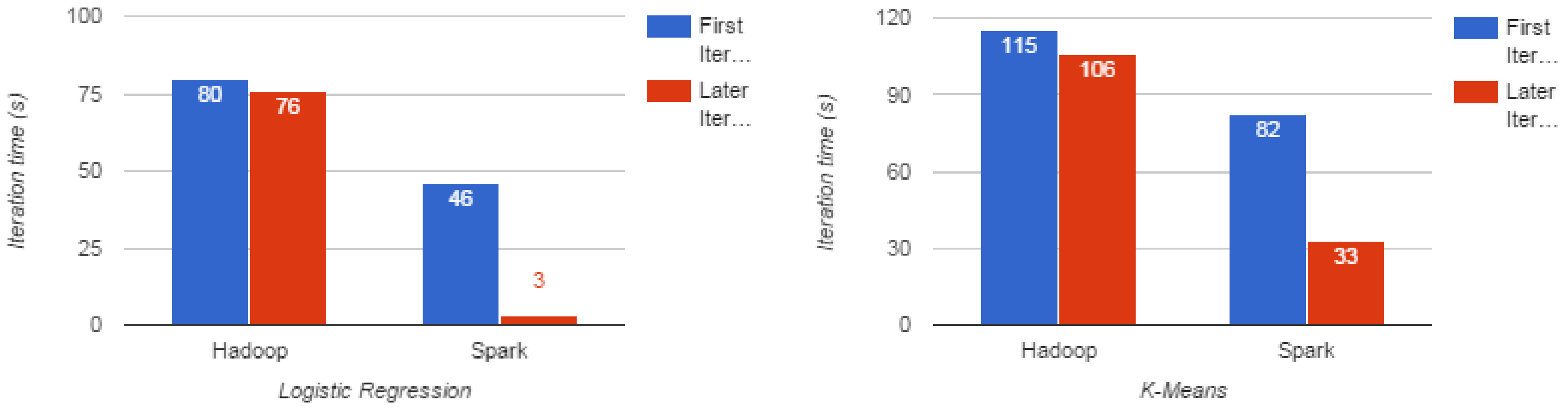
RDD Abstraction

- Restricted form of distributed shared memory
  - immutable, partitioned collection of records
  - can only be built through coarse-grained deterministic transformations (map, filter, join…)

- Efficient fault-tolerance using lineage
  - Log coarse-grained operations instead of fine-grained data updates
  - An RDD has enough information about how it’s derived from other dataset
  - Recompute lost partitions on failure

Evaluation

10 iterations on 100GB data using 25-100 machines



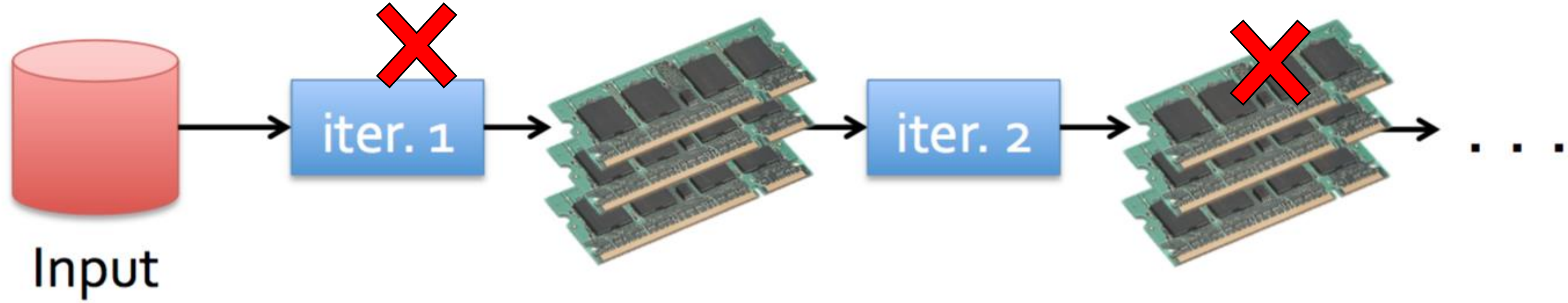
Operations on RDD

- RDDs are immutable
- Two kinds of operations
  - Transformations
    - One RDD to another RDD
  - Actions
    - Returns a value to the application or export data to storage
- RDD can be split into partitions
- Assigned to worker threads and machines

Using RDD

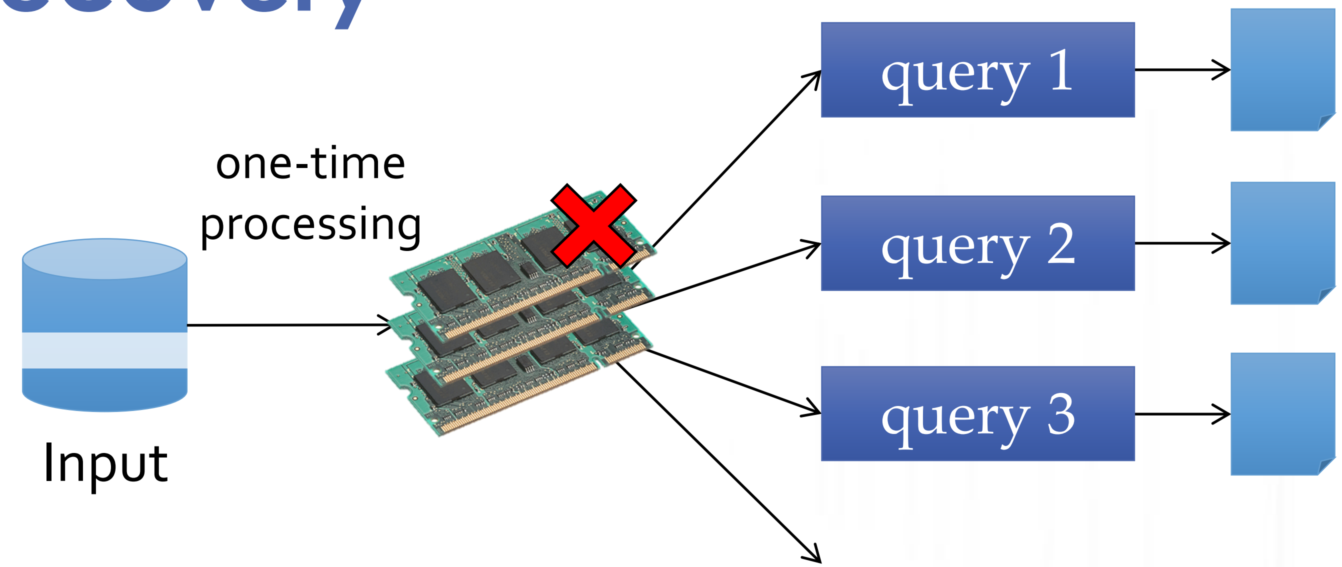
- Create RDD
- Specify number of partitions
    - RDD Myrdd=(filename, 8) .. 8 partitions
- Apply transformations to a RDD (map, filter, join, reduce)
- Remember operations (lineage)
- Apply actions like count collect, save
- Transformations are lazily evaluated only when an action is called for !!!
- Pyspark API   has lots of information on how to program using RDDS

Fault-tolerance



- M, F
- If crash occurs during iter2 , means output of iter2 is lost then just reconstruct using F
- If output of iter 1 is lost, then re execute M, F
- Possible from lineage defined and stored

RDD Recovery

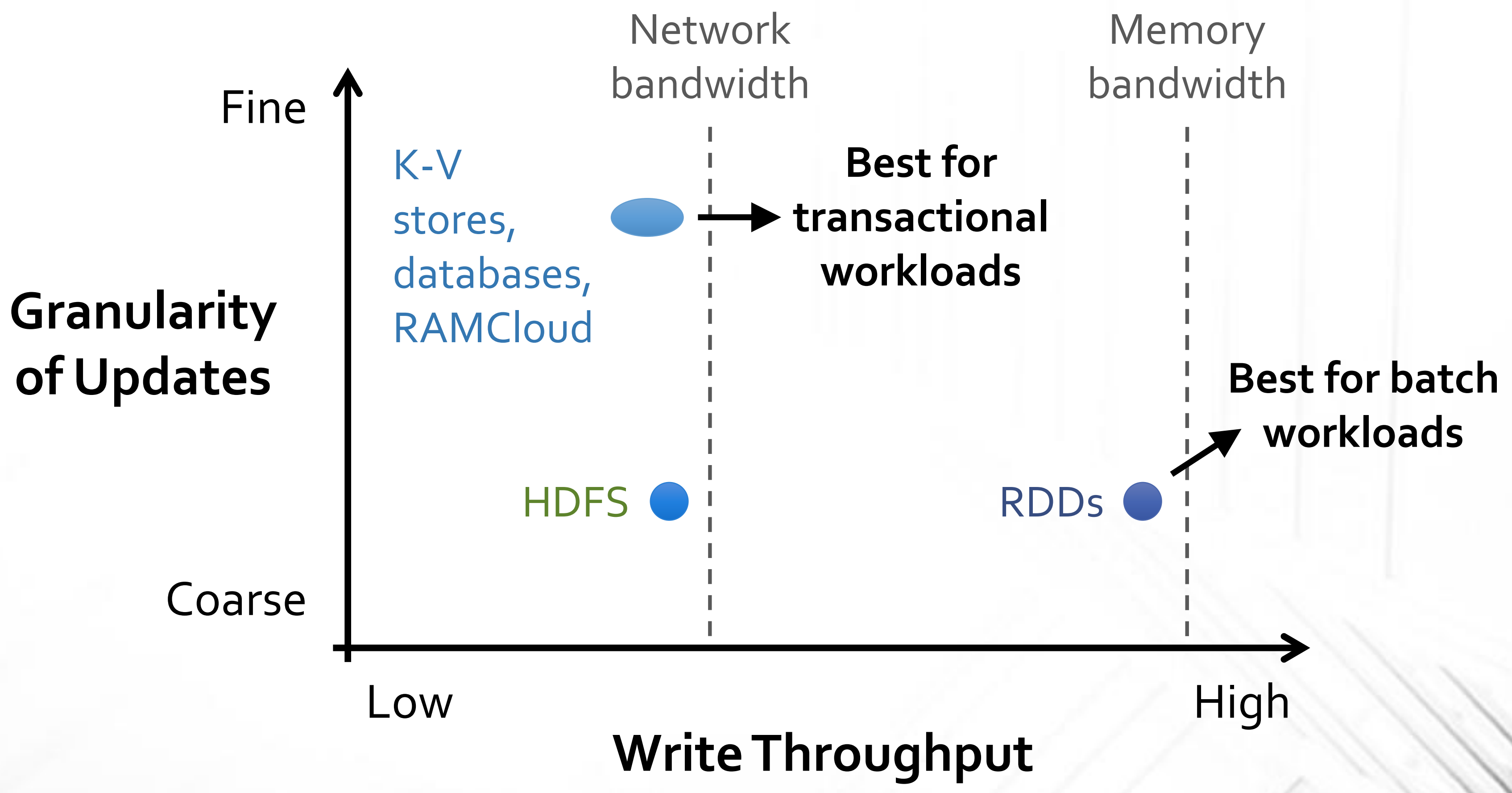


- Lose a piece of the data set
- One machine crashes in the cluster
- RDD track lineage at the level of partition of datasets
- Recompute only this part of RDD from lineage

Generality of RDDs

- Despite their restrictions, RDDs can express surprisingly many parallel algorithms
  - These naturally *apply the same operation to many items*
- Unify many current programming models
  - *Data flow models:* MapReduce, Dryad, SQL, …
  - *Specialized models* for iterative apps: BSP (Pregel), iterative MapReduce (Haloop), bulk incremental, …
- Support *new apps* that these models don't

Tradeoff Space



Spark

- Implements Resilient Distributed Datasets (RDDs)

- Operations on RDDs
    - Transformations: defines new dataset based on previous ones
    - Actions: starts a job to execute on cluster

- Well-designed interface to represent RDDs
    - Makes it very easy to implement transformations
    - Most Spark transformation implementation < 20 LoC

Operation	Meaning
partitions()	Return a list of Partition objects
preferredLocations(*p*)	List nodes where partition *p* can be accessed faster due to data locality
dependencies()	Return a list of dependencies
iterator(*p*, *parentIters*)	Compute the elements of partition *p* given iterators for its parent partitions
partitioner()	Return metadata specifying whether the RDD is hash/range partitioned

Table 3: Interface used to represent RDDs in Spark.

Spark Programming Interface

- DryadLINQ-like API in the Scala language
- Usable interactively from Scala interpreter
- Provides:
  - Resilient distributed datasets (RDDs)
  - Operations on RDDs: *transformations* (build new RDDs), *actions* (compute and output results)
  - Control of each RDD's *partitioning* (layout across nodes) and *persistence* (storage in RAM, on disk, etc)

Spark Operations

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	

Conclusion

RDDs offer a simple yet efficient programming model for a broad range of distributed applications

RDDs provides outstanding performance and efficient fault-tolerance
PySpark, available