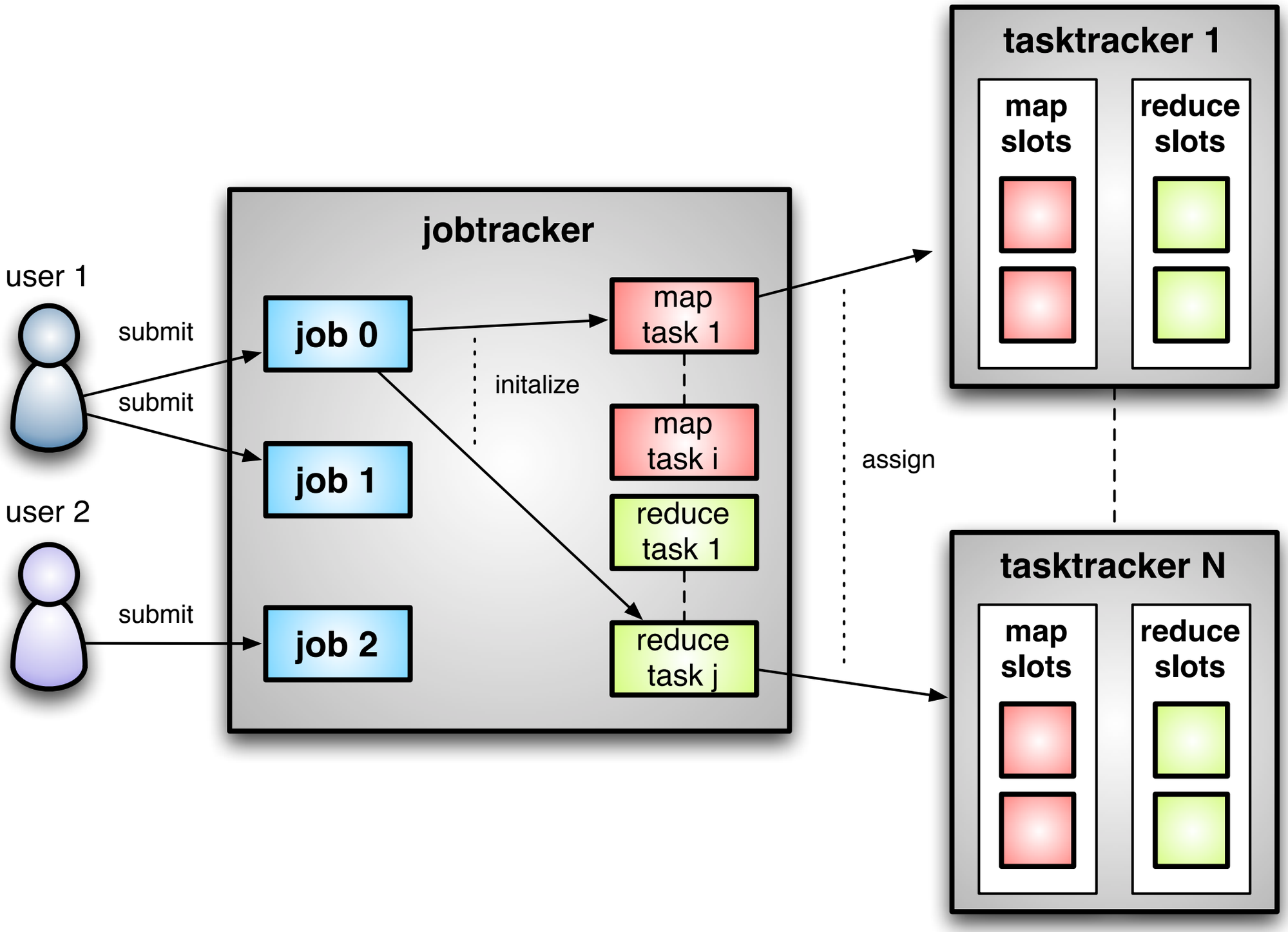
Runtime

- **Handles scheduling**
  - Assigns workers to map and reduce tasks
- **Handles "data distribution"**
  - Moves processes to data
- **Handles synchronization**
  - Gathers, sorts, and shuffles intermediate data
- **Handles errors and faults**
  - Detects worker failures and restarts
- **Everything happens on top of a distributed FS**

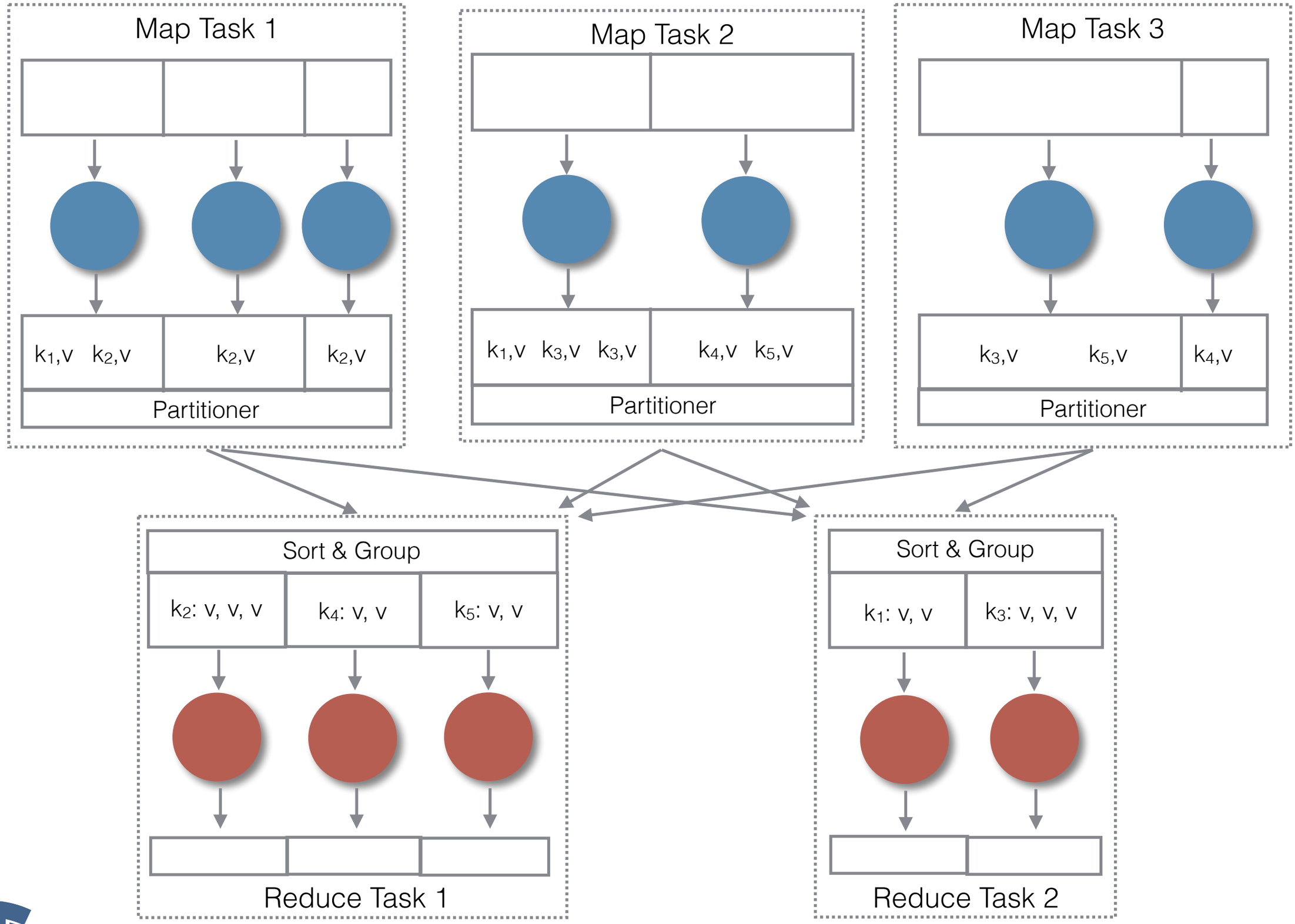
Terminology

- **Job**
- **Task**
- **Slot**
- **JobTracker**
  - Accepts Map/Reduce jobs submitted by users
  - Assigns Map and Reduce tasks to Task Trackers
  - Monitors task and Task Tracker status, re-executes tasks upon failure
- **TaskTracker**
  - Run Map and Reduce tasks upon instruction from the Job Tracker
  - Manage storage and transmission of intermediate output
- **Splits**
  - Data locality optimization

Runtime


user 1
submit
submit
user 2
submit
jobtracker
job 0
job 1
job 2
initalize
map task 1
map task i
reduce task 1
reduce task j
assign
tasktracker 1
map slots
reduce slots
tasktracker N
map slots
reduce slots

Diagram


Map Task 1
k1,v k2,v
k2,v
k2,v
Partitioner
Map Task 2
k1,v k3,v k3,v
k4,v k5,v
Partitioner
Map Task 3
k3,v
k5,v
k4,v
Partitioner
Sort & Group
k2: v, v, v
k4: v, v
k5: v, v
Reduce Task 1
Sort & Group
k1: v, v
k3: v, v, v
Reduce Task 2

Scheduling

- **One master, many workers**
  - Input data split into M map tasks (typically 64 MB in size)
  - Reduce phase partitioned into R reduce tasks (hash(k) mod R)
  - Tasks are assigned to workers dynamically
  - Often: M=200,000; R=4000; workers=2000
- **Master assigns each map task to a free worker**
  - Considers locality of data to worker when assigning a task
  - Worker reads task input (often from local disk)
  - Worker produces R local files containing intermediate k/v pairs
- **Master assigns each reduce task to a free worker**
  - Worker reads intermediate k/v pairs from map workers
  - Worker sorts & applies user's reduce operation to produce the output

Parallelism

- **Map functions run in parallel, create intermediate values from each input data set**
  - The programmer must specify a proper input split (chunk) between mappers to enable parallelism
- **Reduce functions also run in parallel, each will work on different output keys**
  - Number of reducers is a key parameter which determines map-reduce performance

Speculative Execution

- **Problem: Stragglers (i.e., slow workers) significantly lengthen the completion time**
  - Other jobs may be consuming resources on machine
  - Bad disks with soft (i.e., correctable) errors transfer data very slowly
  - Other weird things: processor caches disabled at machine init
- **Solution: Close to completion, spawn backup copies of the remaining in-progress tasks.**
  - Whichever one finishes first, “wins”
- **Additional cost: a few percent more resource usage**
- **Example: A sort program without backup = 44% longer.**

The right tool for the right job

	MPI	MapReduce	DBMS/SQL
What it is	A general parallel programming model	A programming paradigm and its associated execution system	A system to store, manipulate and serve data
Programming Model	Message passing between nodes	Restricted to map and reduce operations	Declarative on data query Stored procedures
Data Organization	No assumptions	Shared files	Organized data structures
Data to be manipulated	Any	key-value pairs	Tables with rich attributes
Execution Model	Nodes are independent	Map/Shuffle/Reduce Checkpointing/Backup Physical data locality	Transaction Query optimization Materialized view
Usability	Steep learning curve difficult to debug	Simple concept Could be hard to optimize	Declarative interface Could be hard to debug at runtime
Key Selling Point	Flexible to accommodate various applications	Process large amount of data with commodity hardware	Interactive querying Maintain a consistent view across client

taken from: http://research.google.com/

HADOOP