



MapReduce & GFS

This presentation contains contents © University of Washington,
licensed under the Creative Commons Attribution 3.0 License.



Outline

- MapReduce
- PageRank
- GFS
- Hadoop Nuts'n'bolts



Motivations for MapReduce

- Data processing: > 1 TB
- Massively parallel (hundreds or thousands of CPUs)
- Must be easy to use



How MapReduce is Structured

- Functional programming meets distributed computing
- A batch data processing system
- Factors out many reliability concerns from application logic

MapReduce Provides:

- Automatic parallelization & distribution
- Fault-tolerance
- Status and monitoring tools
- A clean abstraction for programmers

Programming Model

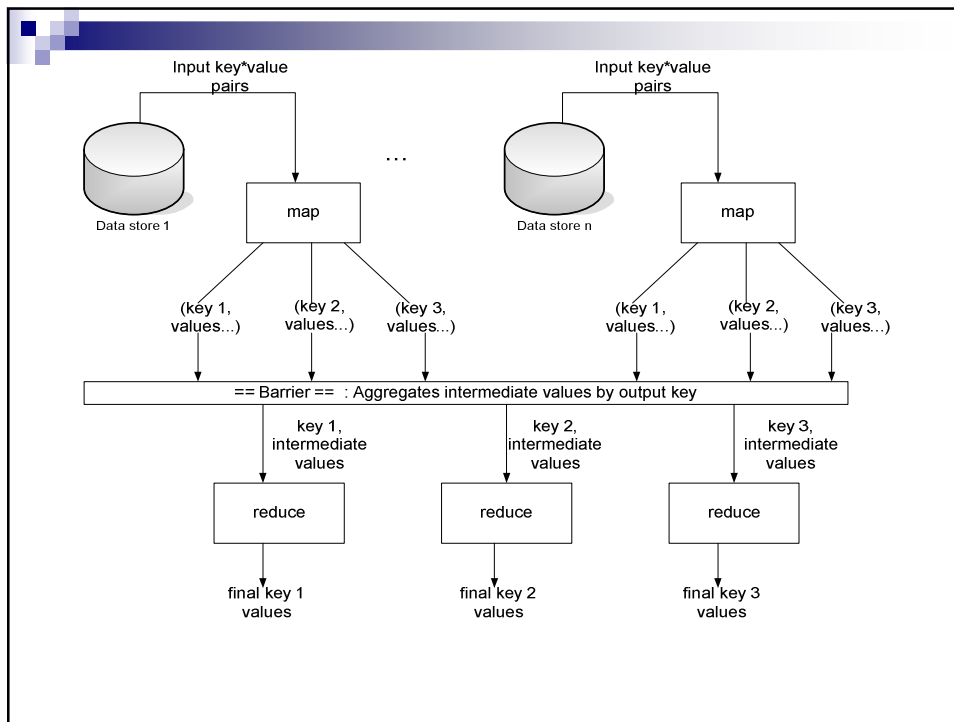
- Borrows from functional programming
- Users implement interface of two functions:
 - `map (in_key, in_value) -> (out_key, intermediate_value) list`
 - `reduce (out_key, intermediate_value list) -> out_value list`

map

- Records from the data source (lines out of files, rows of a database, etc) are fed into the map function as key*value pairs: e.g., (filename, line).
- map() produces one or more *intermediate* values along with an output key from the input.

reduce

- After the map phase is over, all the intermediate values for a given output key are combined together into a list
- reduce() combines those intermediate values into one or more *final values* for that same output key
- (in practice, usually only one final value per key)



Parallelism

- map() functions run in parallel, creating different intermediate values from different input data sets
- reduce() functions also run in parallel, each working on a different output key
- All values are processed *independently*
- Bottleneck: reduce phase can't start until map phase is completely finished.

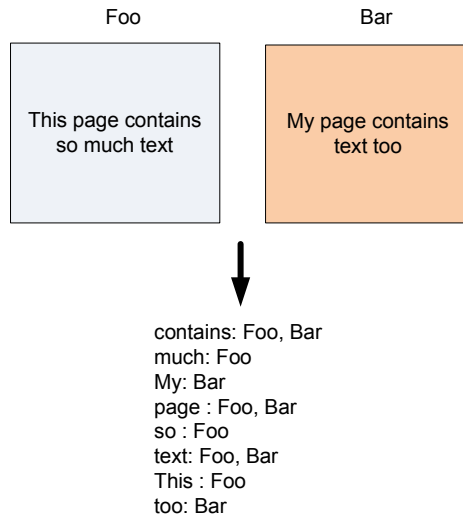
Example: Count word occurrences

```
map(String input_key, String input_value):  
    // input_key: document name  
    // input_value: document contents  
    for each word w in input_value:  
        EmitIntermediate(w, "1");  
  
reduce(String output_key, Iterator  
    intermediate_values):  
    // output_key: a word  
    // output_values: a list of counts  
    int result = 0;  
    for each v in intermediate_values:  
        result += ParseInt(v);  
    Emit(AsString(result));
```

Example vs. Actual Source Code

- Example is written in pseudo-code
- Actual implementation is in C++, using a MapReduce library
- Bindings for Python and Java exist via interfaces
- True code is somewhat more involved (defines how the input key/values are divided up and accessed, etc.)

Inverted Index

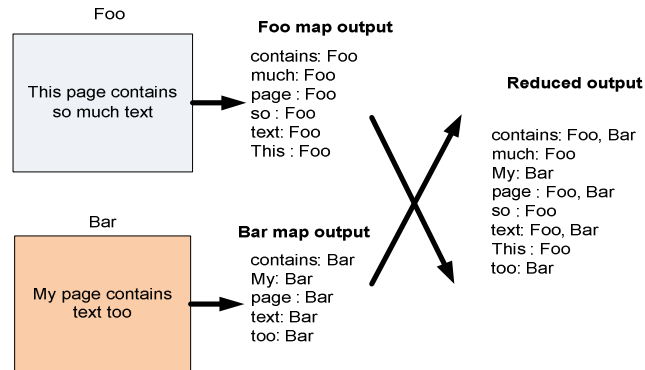


Inverted Index: MapReduce

- Mapper:
 - Key: page name
 - Value: page text

foreach word w **in** pageText:
emitIntermediate(w, pageName);
done
- Reducer:
 - Key: word
 - Values: all page names for word
 - ... Just the identity function

Inverted Index: Data flow



Locality

- Master program divvies up tasks based on location of data: tries to have map() tasks on same machine as physical file data, or at least same rack
- map() task inputs are divided into 64 MB blocks: same size as Google File System chunks

Fault Tolerance

- Master detects worker failures
 - Re-executes completed & in-progress map() tasks
 - Re-executes in-progress reduce() tasks
- Master notices particular input key/values cause crashes in map(), and skips those values on re-execution.
 - Effect: Can work around bugs in third-party libraries!

Optimizations

- No reduce can start until map is complete:
 - A single slow disk controller can rate-limit the whole process
- Master redundantly executes “slow-moving” map tasks; uses results of first copy to finish

Why is it safe to redundantly execute map tasks? Wouldn't this mess up the total computation?

Optimizations

- “Combiner” functions can run on same machine as a mapper
- Causes a mini-reduce phase to occur before the real reduce phase, to save bandwidth

Under what conditions is it sound to use a combiner?

The Example Again

```
map(String input_key, String input_value):
    // input_key: document name
    // input_value: document contents
    for each word w in input_value:
        EmitIntermediate(w, "1");

reduce(String output_key, Iterator
intermediate_values):
    // output_key: a word
    // output_values: a list of counts
    int result = 0;
    for each v in intermediate_values:
        result += ParseInt(v);
    Emit(AsString(result));
```

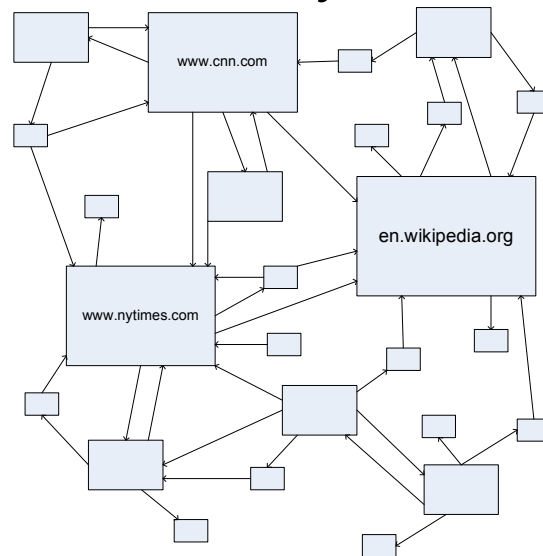
MapReduce Conclusions

- MapReduce has proven to be a useful abstraction
- Greatly simplifies large-scale computations at Google
- Functional programming paradigm can be applied to large-scale applications
- Fun to use: focus on problem, let library deal w/ messy details

PageRank: Random Walks Over The Web

- If a user starts at a random web page and surfs by clicking links and randomly entering new URLs, what is the probability that s/he will arrive at a given page?
- The *PageRank* of a page captures this notion
 - More “popular” or “worthwhile” pages get a higher rank

PageRank: Visually



PageRank: Formula

Given page A, and pages T_1 through T_n linking to A, PageRank is defined as:

$$PR(A) = (1-d) + d \left(\frac{PR(T_1)}{C(T_1)} + \dots + \frac{PR(T_n)}{C(T_n)} \right)$$

$C(P)$ is the cardinality (out-degree) of page P
d is the damping (“random URL”) factor

PageRank: Intuition

- Calculation is iterative: PR_{i+1} is based on PR_i
- Each page distributes its PR_i to all pages it links to. Linkees add up their awarded rank fragments to find their PR_{i+1}
- d is a tunable parameter (usually = 0.85) encapsulating the “random jump factor”

$$PR(A) = (1-d) + d (PR(T_1)/C(T_1) + \dots + PR(T_n)/C(T_n))$$

PageRank: First Implementation

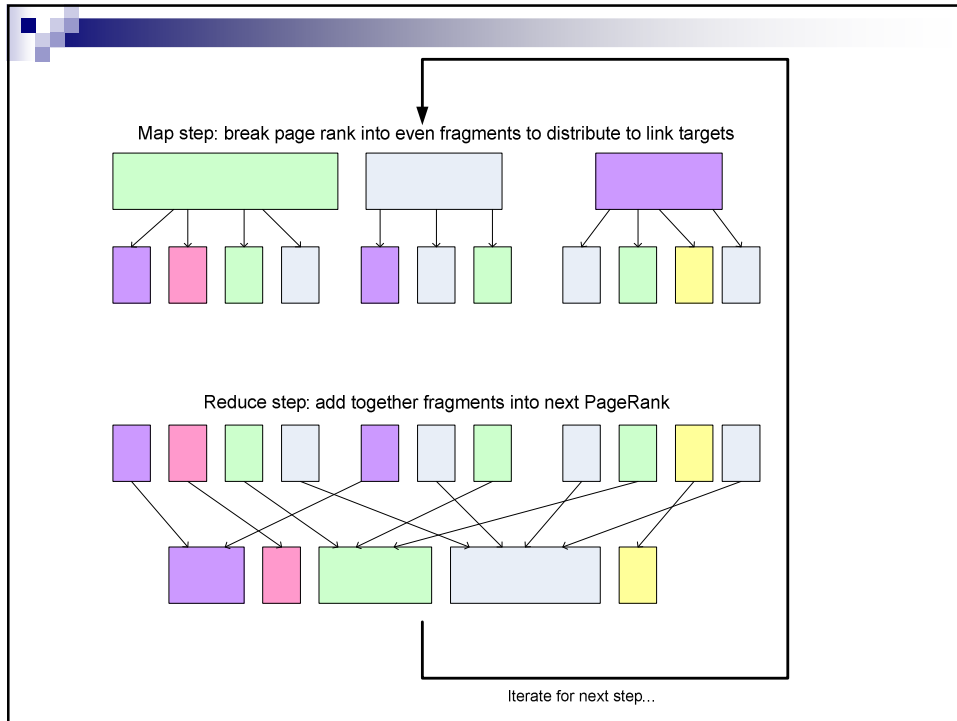
- Create two tables 'current' and 'next' holding the PageRank for each page. Seed 'current' with initial PR values
- Iterate over all pages in the graph, distributing PR from 'current' into 'next' of linkees
- `current := next; next := fresh_table();`
- Go back to iteration step or end if converged

Distribution of the Algorithm

- Key insights allowing parallelization:
 - The 'next' table depends on 'current', but not on any other rows of 'next'
 - Individual rows of the adjacency matrix can be processed in parallel
 - Sparse matrix rows are relatively small

Distribution of the Algorithm

- Consequences of insights:
 - We can *map* each row of 'current' to a list of PageRank “fragments” to assign to linkees
 - These fragments can be *reduced* into a single PageRank value for a page by summing
 - Graph representation can be even more compact; since each element is simply 0 or 1, only transmit column numbers where it's 1



Phase 1: Parse HTML

- Map task takes (URL, page content) pairs and maps them to (URL, (PR_{init} , list-of-urls))
 - PR_{init} is the “seed” PageRank for URL
 - list-of-urls contains all pages pointed to by URL
- Reduce task is just the identity function

Phase 2: PageRank Distribution

- Map task takes (URL, (cur_rank, url_list))
 - For each u in url_list, emit (u , $\text{cur_rank}/|\text{url_list}|$)
 - Emit (URL, url_list) to carry the points-to list along through iterations

$$\text{PR}(A) = (1-d) + d (\text{PR}(T_1)/C(T_1) + \dots + \text{PR}(T_n)/C(T_n))$$

Phase 2: PageRank Distribution

- Reduce task gets (URL, url_list) and many (URL, val) values
 - Sum vals and fix up with d
 - Emit (URL, (new_rank, url_list))

$$\text{PR}(A) = (1-d) + d (\text{PR}(T_1)/C(T_1) + \dots + \text{PR}(T_n)/C(T_n))$$

Finishing up...

- A non-parallelizable component determines whether convergence has been achieved (Fixed number of iterations? Comparison of key values?)
- If so, write out the PageRank lists - done!
- Otherwise, feed output of Phase 2 into another Phase 2 iteration

PageRank Conclusions

- MapReduce isn't the greatest at iterated computation, but still helps run the "heavy lifting"
- Key element in parallelization is independent PageRank computations in a given step
- Parallelization requires thinking about minimum data partitions to transmit (e.g., compact representations of graph rows)
 - Even the implementation shown today doesn't actually scale to the whole Internet; but it works for intermediate-sized graphs



GFS

Some slides designed by Alex Moschuk, University of Washington
Redistributed under the Creative Commons Attribution 3.0 license



Distributed Filesystems

- Support access to files on remote servers
- Must support concurrency
- Can support replication and local caching
- Different implementations sit in different places on complexity/feature scale

NFS: Tradeoffs

- NFS Volume managed by single server
 - Higher load on central server
 - Simplifies coherency protocols
- Full POSIX system means it “drops in” very easily, but isn’t “great” for any specific need

GFS: Motivation

- Google needed a good distributed file system
 - Redundant storage of massive amounts of data on cheap and unreliable computers
 - ... What does “good” entail?
- Why not use an existing file system?
 - Google’s problems are different from anyone else’s
 - Different workload and design priorities
 - Particularly, bigger data sets than seen before
 - GFS is designed for Google apps and workloads
 - Google apps are designed for GFS

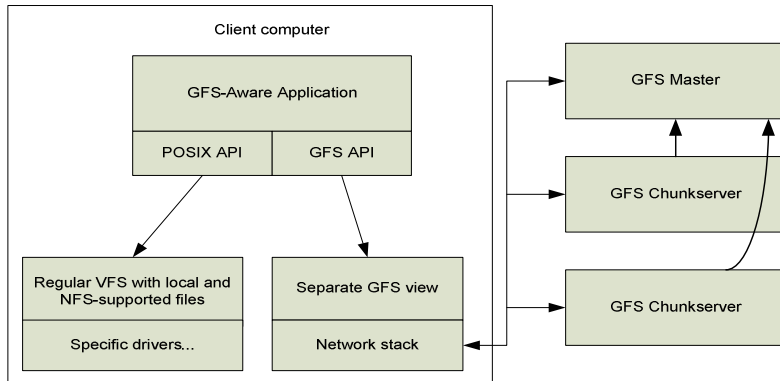
Assumptions

- High component failure rates
 - Inexpensive commodity components fail all the time
- “Modest” number of HUGE files
 - Just a few million
 - Each is 100MB or larger; multi-GB files typical
- Files are write-once, mostly appended to
 - Perhaps concurrently
- Large streaming reads
- High sustained throughput favored over low latency

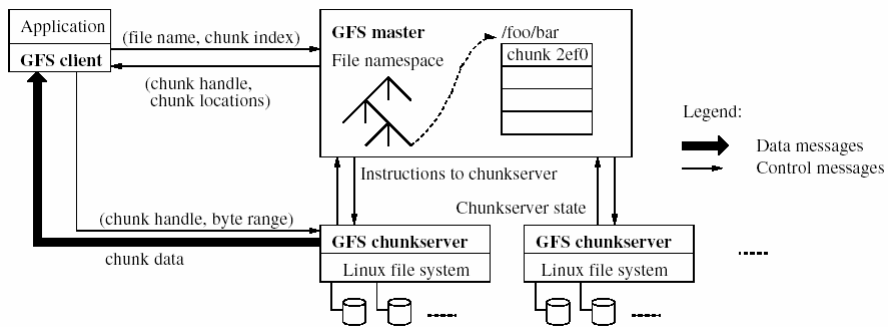
GFS Design Decisions

- Files stored as chunks
 - Fixed size (64MB)
- Reliability through replication
 - Each chunk replicated across 3+ *chunkservers*
- Single master to coordinate access, keep metadata
 - Simple centralized management
- No data caching
 - Little benefit due to large data sets, streaming reads
- Familiar interface, but customize the API
 - Simplify the problem; focus on Google apps
 - Add *snapshot* and *record append* operations

GFS Client Block Diagram



GFS Architecture



Single master

- From distributed systems we know this is a:
 - Single point of failure
 - Scalability bottleneck
- GFS solutions:
 - Shadow masters
 - Minimize master involvement
 - never move data through it, use only for metadata
 - and cache metadata at clients
 - large chunk size
 - master delegates authority to primary replicas in data mutations (chunk leases)
- Simple, and good enough!

Metadata (1/2)

- Global metadata is stored on the master
 - File and chunk namespaces
 - Mapping from files to chunks
 - Locations of each chunk's replicas
- All in memory (64 bytes / chunk)
 - Fast
 - Easily accessible

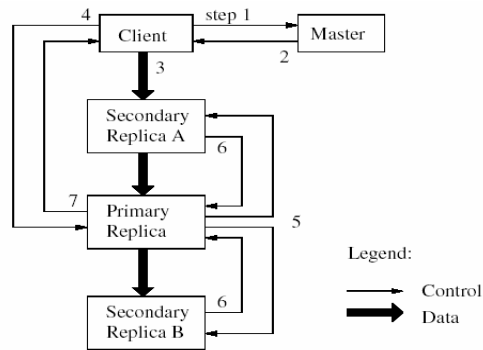
Metadata (2/2)

- Master has an *operation log* for persistent logging of critical metadata updates
 - persistent on local disk
 - replicated
 - checkpoints for faster recovery

Mutations

- Mutation = write or append
 - must be done for all replicas
- Goal: minimize master involvement
- Lease mechanism:
 - master picks one replica as primary; gives it a “lease” for mutations
 - primary defines a serial order of mutations
 - all replicas follow this order
- Data flow decoupled from control flow

Mutations



Atomic record append

- Client specifies data
- GFS appends it to the file atomically at least once
 - GFS picks the offset
 - works for concurrent writers
- Used heavily by Google apps
 - e.g., for files that serve as multiple-producer/single-consumer queues

Relaxed consistency model (1/2)

- “Consistent” = all replicas have the same value
- “Defined” = replica reflects the mutation, consistent
- Some properties:
 - concurrent writes leave region consistent, but possibly undefined
 - failed writes leave the region inconsistent
- Some work has moved into the applications:
 - e.g., self-validating, self-identifying records

Relaxed consistency model (2/2)

- Simple, efficient
 - Google apps can live with it
 - what about other apps?
- Namespace updates atomic and serializable

Master's responsibilities (1/2)

- Metadata storage
- Namespace management/locking
- Periodic communication with chunkservers
 - give instructions, collect state, track cluster health
- Chunk creation, re-replication, rebalancing
 - balance space utilization and access speed
 - spread replicas across racks to reduce correlated failures
 - re-replicate data if redundancy falls below threshold
 - rebalance data to smooth out storage and request load

Master's responsibilities (2/2)

- Garbage Collection
 - simpler, more reliable than traditional file delete
 - master logs the deletion, renames the file to a hidden name
 - lazily garbage collects hidden files
- Stale replica deletion
 - detect "stale" replicas using chunk version numbers

Fault Tolerance

- High availability

- fast recovery
 - master and chunkservers restartable in a few seconds
- chunk replication
 - default: 3 replicas.
- shadow masters

- Data integrity

- checksum every 64KB block in each chunk

Performance

Cluster	A	B
Chunkservers	342	227
Available disk space	72 TB	180 TB
Used disk space	55 TB	155 TB
Number of Files	735 k	737 k
Number of Dead files	22 k	232 k
Number of Chunks	992 k	1550 k
Metadata at chunkservers	13 GB	21 GB
Metadata at master	48 MB	60 MB

Cluster	A	B
Read rate (last minute)	583 MB/s	380 MB/s
Read rate (last hour)	562 MB/s	384 MB/s
Read rate (since restart)	589 MB/s	49 MB/s
Write rate (last minute)	1 MB/s	101 MB/s
Write rate (last hour)	2 MB/s	117 MB/s
Write rate (since restart)	25 MB/s	13 MB/s
Master ops (last minute)	325 Ops/s	533 Ops/s
Master ops (last hour)	381 Ops/s	518 Ops/s
Master ops (since restart)	202 Ops/s	347 Ops/s

Deployment in Google

- 50+ GFS clusters
- Each with thousands of storage nodes
- Managing petabytes of data
- GFS is under BigTable, etc.

Conclusion

- GFS demonstrates how to support large-scale processing workloads on commodity hardware
 - design to tolerate frequent component failures
 - optimize for huge files that are mostly appended and read
 - feel free to relax and extend FS interface as required
 - go for simple solutions (e.g., single master)
- GFS has met Google's storage needs... win!



Working With Hadoop



Some MapReduce Terminology

- *Job* – A “full program” - an execution of a Mapper and Reducer across a data set
- *Task* – An execution of a Mapper or a Reducer on a slice of data
 - a.k.a. Task-In-Progress (TIP)
- *Task Attempt* – A particular instance of an attempt to execute a task on a machine

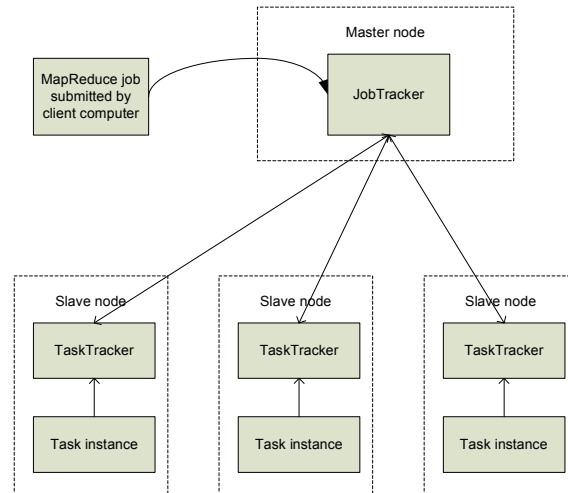
Terminology Example

- Running “Word Count” across 20 files is one *job*
- 20 files to be mapped imply 20 *map tasks* + some number of *reduce tasks*
- At least 20 *map task attempts* will be performed... more if a machine crashes, etc.

Task Attempts

- A particular task will be attempted at least once, possibly more times if it crashes
 - If the same input causes crashes over and over, that input will eventually be abandoned
- Multiple attempts at one task may occur in parallel with speculative execution turned on
 - Task ID from *TaskInProgress* is not a unique identifier; don't use it that way

MapReduce: High Level



Job Distribution

- MapReduce programs are contained in a Java “jar” file + an XML file containing serialized program configuration options
- Running a MapReduce job places these files into the HDFS and notifies TaskTrackers where to retrieve the relevant program code
- ... Where’s the data distribution?

Data Distribution

- Implicit in design of MapReduce!
 - All mappers are equivalent; so map whatever data is local to a particular node in HDFS
- If lots of data does happen to pile up on the same node, nearby nodes will map instead
 - Data transfer is handled implicitly by HDFS

Configuring With JobConf

- MR Programs have many configurable options
- *JobConf* objects hold (key, value) components mapping String → 'a'
 - e.g., "mapred.map.tasks" → 20
 - JobConf is serialized and distributed before running the job
- Objects implementing *JobConfigurable* can retrieve elements from a JobConf

Job Launch Process: Client

- Client program creates a *JobConf*
 - Identify classes implementing *Mapper* and *Reducer* interfaces
 - `JobConf.setMapperClass()`, `setReducerClass()`
 - Specify inputs, outputs
 - `JobConf.setInputPath()`, `setOutputPath()`
 - Optionally, other options too:
 - `JobConf.setNumReduceTasks()`,
`JobConf.setOutputFormat()...`

Job Launch Process: *JobClient*

- Pass *JobConf* to `JobClient.runJob()` or `submitJob()`
 - `runJob()` blocks, `submitJob()` does not
- *JobClient*:
 - Determines proper division of input into *InputSplits*
 - Sends job data to master *JobTracker* server

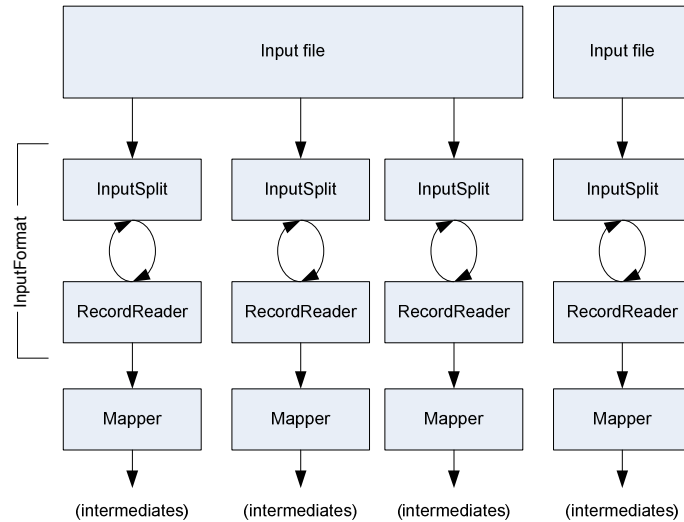
Job Launch Process: *TaskTracker*

- *TaskTrackers* running on slave nodes periodically query *JobTracker* for work
- Retrieve job-specific jar and config
- Launch task in separate instance of Java
 - `main()` is provided by Hadoop

Creating the *Mapper*

- You provide the instance of *Mapper*
 - Should extend *MapReduceBase*
- One instance of your *Mapper* is initialized by the *MapTaskRunner* for a *TaskInProgress*
 - Exists in separate process from all other instances of *Mapper* – no data sharing!

Getting Data To The Mapper



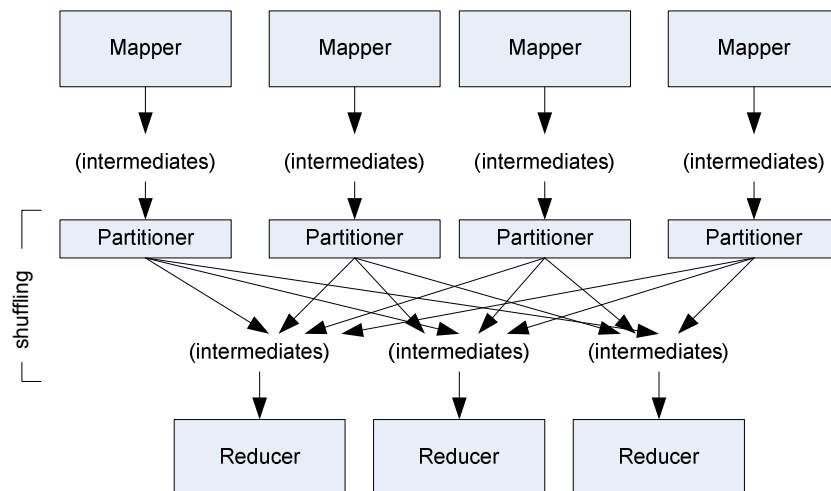
Mapper

- void map(WritableComparable key,
Writable value,
OutputCollector output,
Reporter reporter)

What is Writable?

- Hadoop defines its own “box” classes for strings (*Text*), integers (*IntWritable*), etc.
- All values are instances of *Writable*
- All keys are instances of *WritableComparable*

Partition And Shuffle



Partitioner

- `int getPartition(key, val, numPartitions)`
 - Outputs the partition number for a given key
 - One partition == values sent to one Reduce task
- *HashPartitioner* used by default
 - Uses `key.hashCode()` to return partition num
- *JobConf* sets *Partitioner* implementation

Reduction

- `reduce(WritableComparable key,
Iterator values,
OutputCollector output,
Reporter reporter)`
- Keys & values sent to one partition all go to the same reduce task
- Calls are sorted by key – “earlier” keys are reduced and output before “later” keys

