MapReduce
Jeffrey Dean and Sanjay Ghemawat

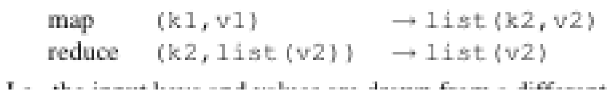
Background context

- BIG DATA!!
    - Large-scale services generate huge volumes of data: logs, crawls, user databases, web site content, etc.
    - Very useful to be able to crunch through the data (mining / processing)
- But, doing so efficiently requires a large-scale, **parallel** data processing system
    - How long does it take to go through 1 TB sequentially?
        - 3 hours at 80 MB/s
        - What if you have to do multiple scans or phases?
    - Parallel processing introduces a host of nasty distribution issues
        - Communication and routing
            - which nodes should be involved?
            - what transport protocol should be used?
            - how do you manage threads / events / connections?
            - remote execution of your processing code?
        - Fault tolerance and fault detection
            - more broadly, group membership
        - Load balancing / partitioning of data
            - heterogeneity of nodes
            - skew in data
            - network topology and bisection bandwidth issues
    - Also requires you to think about parallelization strategy
        - How do you split work? Algorithmic issues.

Goal of work

- To solve these distribution issues once in a reusable library
    - To shield the programmer from having to resolve them each time they write a data analysis program
- To obtain adequate throughput and scalability out of the system
- To provide the programmer with a useful conceptual framework for designing their parallel algorithm

Map/reduce



```
map     (k1,v1)          → list(k2,v2)
reduce  (k2,list(v2))    → list(v2)
```

- Two phases, conceptually, with hidden intermediate shuffle phase
- Map
    - For each key/value pair in the input file, the map() function is invoked
        - set of input records is split into M different map splits
        - the splits are processed in parallel on different worker invocations
        - each worker serially iterates through its split
        - each map() produces a set of intermediate key/value pairs as output
            - into a local file on the map worker, bucketed by reduce partitioning function R; i.e., one bucket per reduce task
- Shuffle
    - for each reduce bucket, a reduce worker is identified for it. the worker slurps all of the buckets from the map worker local files
    - reduce worker then aggregates by key and sorts within the key, to get the (k2, list(v2)) form.
- Reduce
    - For each key in the intermediate output, the reduce() function is invoked
        - the invocation accepts the key and a sorted list of values
        - reduce() does some kind of merge or filtering, and emits a list of values to an output file

Examples

- sort
    - map(linenum, line) { emit(sortkey(line), line) }
    - reduce(sortkey, list(line)) { emit(list(line)) }

- inverted index
    - map(docid, doctext) { foreach word(doctext) { emit(word, docid) }
    - reduce(word, list(docid)) { emit(word, sort(list(docid))); }

- grep
    - map(linenum, line) { if (grep(line, pattern) emit(linenum, line) }
    - reduce(linenum, list(line) { emit(list(line)) }

- join – much harder ("A Comparison of Join Algorithms for Log Processing in MapReduce")
    - **standard is "repartition join", aka partitioned sort-merge in database parlance**
        - imagine you have a log table L and some reference table R that contains user information
            - you want to do an equijoin, L (equijoin where L.k = R.k) with R

                - |L| >> |R|
            - implement in single MR job
                - input is both L and R; each map task works on a split of L or R
                - map task:
                    - tags record with its originating table, and outputs:
                        - (join key, tagged record) as key/val
                - shuffle phase will partition across join key, aggregate join key partitions, and sort
                - reduce task:
                    - for each join key:
                        - separate and buffer input records into two sets, according to the tag
                        - do a cross-product between the two sets
            - problems
                - both L and R end up having to be sorted by the shuffle phase
                - both L and R end up having to be sent over the network during the shuffle phase
                - reduce task: might not be able to keep entire input in memory, for skewed inputs
        - **can do broadcast join**
            - if R is tiny, can replicate R on each map worker, and builds an in-memory hash table of R in each worker. then, hash-join against L in the map phase
        - **several others as well**
            - which is best? depends on a bunch of things – rest of MR job, data selectivity and skew, etc.

Semantic issues

- In what order are the input values consumed?
    - Order determined by a split configuration value in user program
    - Right way to think about it is as unordered consumption – order shouldn't matter for the correctness of your map/reduce program
        - but, order could affect performance of the shuffle phase in the middle
- In what order are the intermediate values consumed?
    - This matters more: you might want the final output to be grouped across reduce keys in some additional way, so need to control the partitioning of the intermediate file across reduce tasks
        - Partition function specifiable by mapreduce program (with a default that does hash partitioning)
- Should map or reduce have side-effects?
    - Its up to you, but….
    - Implementation does not guarantee exactly once semantics – it provides at least once (except for deterministic bugs)

        - Implies side-effects must be idempotent for deterministic output
        - Failure implies side-effects should be atomic for correct output
    - Should map and reduce functions be deterministic?
        - Its up to you, but…
        - For same reason, non-determinism confuses semantics
            - If deterministic, worker restart and atomic rename to commit output guarantees the execution is equivalent to some sequential execution
            - If non-deterministic, get the blend of sequential executions (e.g., timestamps or counters go zooey)

What mapreduce is built on

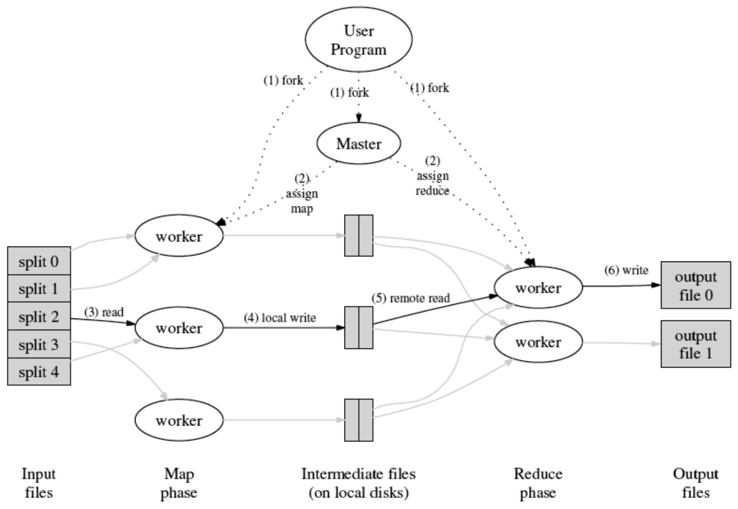


Figure 1: Execution overview

- Underlying GFS – why?
    - Single namespace for input and output files; simplifies initial data distribution
    - Why not use GFS for the intermediate files??
        - Probably because they are transient and do not need to be globally readable
        - More efficient to produce locally and special-purpose the RPC slurp to the reduce tasks

- Cluster management software
    - Failures
    - Group membership
    - Load balancing and scheduling

Tweaks / improvements

- Combiner: basically a reduce run locally on the output of a map

        - I/O types: marshall/unmarshall libraries for parsing files
        - Skip bad records: bad input deterministically crashing user function
        - Status info: HTTP server with stats on progress, stdin/stdout of tasks, etc.
        - Counters: propagated to master, included in HTTP server output

<u>Comparison to databases</u>

Huge source of controversy in DB community

- misconceptions in sys community about the degree of contribution of MR
- parallel databases have much more advanced data processing support that leads to much, much more efficient processing
- parallel databases support a much richer semantic model (arbitrary SQL, including joins)

Paper by Stonebraker, DeWitt, and others: "A Comparison of Approaches to Large-Scale Data Analysis"

- parallel databases support a schema; MR does not
    - the structure of the data in MR must be built into map() and reduce() programs. hinders data sharing across programs, programmers.
    - if sharing is done, implicit agreement to a data model – might as well codify in schema
    - lack of schema means lack of automated mechanism to enforce integrity constraints / typing
- parallel databases support indexes; MR does not
    - selection is accelerated massively with an index, including range queries (or other indexable subset operators)
    - no index at all in MR framework; if a programmer wants one, they have to implement the index themselves, and also enhance the data fetching mechanism in MR to use it. means it's hard to share indexes!
- parallel databases support full relational programming model (SQL); MR forces programmers to a more assembly-like model
    - e.g., have to implement your own join strategy if you want it in MR; SQL just naturally supports
    - difference between stating what you want (relational query), and how to do it (SQL)
- query optimization
    - parallel databases use existence of schema, indices, and declared query, as well as knowledge about how data is partitioned and the locality of it across the cluster, to support full query optimization: they calculate the most efficient query plan
    - MR forces programmers to do this, if they do it at all
        - e.g., selectivity calculation try to push the least selective (i.e., most pruning) selection operator early, to minimize how much data has to flow between query operators

            - cannot do this in MR, and as a result, massive intermediate data sets end up materializing locally in files and being shipped / sorted in the shuffle phase
        - e.g2, when the reduce phase of MR starts, all disks of map workers will be hammered by all of this pull of materialized local files.
            - databases use push to avoid this, and avoid materializing files to disk where possible

Where does MR win?

- loading data into system: easier to bulk load files into GFS than to pump records into DB, and current DBs are bad about parallelizing the load phase
- fault tolerance: unit of work (map or reduce) can fail and be restarted in MR. In DB, has fault tolerance, but unit of restart is the entire query, in part because intermediate results are not materialized on disk.
    - implication? as you scale up to larger and larger clusters, more likely that a query will need to be restarted because of a failure during the execution.
    - stonebraker paper: efficiency gains of DB means that you execute on much smaller clusters in practice, so this is a non-issue except for a very small number of players in the world
- approachability: more people are familiar with coding C++ than writing SQL

Where do neither win?

- low-latency, incremental calculations (instead of high-throughput, batch-oriented programs)
    - google abandoned MR for its indexing operation, in favor of "percolator", a system that can efficiently and transactionally update existing index as new pages are crawled.
        - before percolator, had to recalculate entire index in a giant batch, so updates were delayed for weeks to accumulate new data in batch

Evaluation

What questions are you curious about?

- What is their performance bottleneck for typical map/reduces? And what performance optimizations does this imply are possible?
- How well utilized are the physical machines during execution?
- What is the scaling bottleneck in the system?
- What are the effects of different scheduling policies on throughput?
- What happens if you schedule multiple mapreduces concurrently?

What questions did they answer?

- How fast does it run on two programs? (grep, sort)
- Fast. I think.

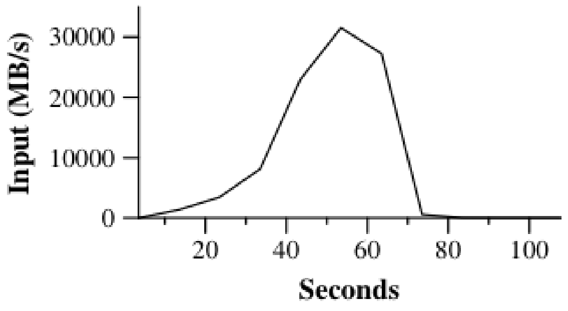


Figure 2: Data transfer rate over time

- GREP

Things to note from this graph
- exponential ramp up in rate
  - why?
  - hard to tell. artifact of their scheduler. my guess is that they took inspiration from slowstart to do "congestion-based" growth.
    - is there equivalent of packet loss and backoff?
    - yes – two tasks co-scheduled on same node?
- peak input rate 30,000 MB/s
  - over 1764 nodes
  - $\rightarrow$ 17 MB/s per node. pretty good seq throughput!
  - why do they get this?
  - GFS property.
- heavy tail to the right
  - waiting for stragglers
  - very careful scheduling needed to avoid here
  - but they care about throughput, not latency, so who cares