Map-Reduce

Mark Greenstreet & Ian M. Mitchell

CPSC 418 – February 27, 2017

Portrait of a Data Centre

- Sketch of a big data center:
  - Tens of thousands of machines, each with its own disk(s).
  - Distributed file system—what is that?
  - Commodity networks and routers.
    - Each machine has a network interface (e.g. 10Gb ethernet)
    - Cross-section bandwidth is **way** smaller than the number of machines times the per-machine bandwidth.
  - Scale is so big that there **will be failures**: chips, cores, memory, disks, network interfaces, switches, ...
    - If the average lifetime of a machine is five years, then 10,000 machine data center will have a failure every four hours.
    - Even without failure, maintenance will take machines offline.
- Need to analyse the huge data sets stored on these machines.

Problem Domain: Large-Scale Data Analysis

- Data analysis requires:
  - Fetching the relevant records.
  - Performing analysis of related records.
  - Summarizing the results.
- Example: word frequency in documents
- Example: core curriculum
  - How do 200-level courses impact success in 400-level courses?
  - Look at all transcripts.
  - Analyze relationships for *(2XX, 4YY)* pairs.
- Google noted that each such problem was getting its own custom code.
  - All of that code development is expensive.
  - Often required similar rewrites when underlying system changed.

The MapReduce Pattern

Slight generalization of description from [Dean & Ghemawat 2008].

- All data is represented as collections of *(Key, Value)* pairs.
- **map**
  - For each *(Key1, Value1)* pair of the input, user code produces a collection of *(Key2, Value2)* pairs for the output.
- **shuffle**
  - All *(Key2, Value2)* pairs with the same *Key2* are combined into a *(Key2, [list of Value2])* result and sorted by *Key2*.
- **reduce**
  - For each *(Key2, [list of Value2])*, user code produces a *(Key2, Value3)* result (where *Value3* might be a list itself).

Apache Hadoop is an open-source implementation of this basic framework.

MapReduce: Word-Count

Example from [Dean & Ghemawat 2008] revised.

- Input data:
  - *Key1* is the document name.
  - *Value1* is document text.
- map:
  - *Key2* is a word.
  - *Value2* is the count of times it appears in the document.
- shuffle:
  - Collect counts from all documents for each word *(Word, [list of Counts])*.
- reduce:
  - *Value3* is the sum of all counts; in other words, the total number of times the word appears in all documents.

MapReduce: Curriculum

- Input data *(Key1, Value1)*:
  - *Key1* is the student number for the transcript
  - *Value1* is a list of *(CourseNumber, Grade)* pairs.
- map: For each 200-level course and for each 400-level course in the transcript:
  - *Key2* is the pair *(Course200, Course400)*.
  - *Value2* is the pair *(Grade200, Grade400)*.
- shuffle:
  - Collect matching course pairs from all students *((Course200, Course400), [(Grade200, Grade400), (Grade200, Grade400), . . .])*.
- reduce:
  - *Value3* is (for example) the sample Pearson correlation coefficient r for the data set *[(Grade200, Grade400), (Grade200, Grade400), . . .]*.
  - More complex analyses could be performed.

Wait a Minute Now…

But didn't we already study "reduce"?

- The course library's `wtree:reduce()` in Erlang had `leaf()`, `combine()` and `root()`.
- MapReduce at Google has *(Key1, Value1)*, *(Key2, Value2)*, and shuffle?

These patterns have similar names but seek to solve different problems.

- **Reduce** is a generic name for a functional programming pattern which takes a collection of data and produces some kind of summary information.

Many flavours of Reduce

- In Erlang, `wtree:reduce()` is designed to spread the computation of a reduction across many workers.
  - Implementation maximizes parallelism for a single reduce operation.
  - Collection and combination of data occurs in `combine()` functions.
  - Note that the `leaf()` function can perform a map operation before the reduction, so no loss of generality.
- In MapReduce, many independent reductions (one for each *Key2*) are spread across many workers, but each reduction is performed by a single worker.
  - Implementation emphasizes fault tolerance and disk-based data storage.
  - Collection (but not combination) of data occurs in shuffle step.
  - If reduction is too big for a single worker, user must change the intermediate *(Key2, Value2)* representation and/or break the problem into multiple MapReduce steps.

Programming Model

The user creates a **MapReduce specification object** which provides:

- The *map* and *reduce* functions.
- The names of the input and output files.
- Optionally other tuning parameters; for example:
  - Number of map and reduce workers to use.
  - Custom function to combine intermediate results within a map worker to reduce size of intermediate data.
  - A custom hashing function to help with shuffle step.

The user then invokes the `MapReduce` function.

Execution (Part 1)

- The `MapReduce` function spawns M map worker, R reduce workers, and one master.
- Each map worker:
  - Is assigned fragment(s) of the input file by the master – these fragments are called **splits**.
  - Reads a *(Key1, Value1)* record from an assigned split.
  - Runs user `map` code on that record.
  - Writes the *(Key2, Value2)* result to a temporary file.
  - Repeats until all records in the split are completed.
  - Repeats until all assigned splits are completed.
  - Notifies the master when it is done.
- Result is a bunch of temporary files holding *(Key2, Value2)* pairs.

Execution (Part 2)

- Start from temporary files holding *(Key2, Value2)* pairs.
- Shuffle:
  - Each reduce worker is assigned *Key2* value(s).
  - Corresponding *Value2* lists are taken from map worker's temporary output files and written to reduce worker's temporary input files.
  - Reduce worker receives *(Key2, [list of Value2])* pairs sorted by *Key2*.
- Each reduce worker:
  - Reads a *(Key2, [list of Value2])* record from a temporary file.
  - Runs user `reduce` code on that record.
  - Writes the *(Key2, Value3)* result to a file.
  - Notifies the master when it is done.
- When all the reduce computations are complete, the master sends a message to wake up the user process, and the `MapReduce` function returns.

Do the MapReduce Shuffle

How do the intermediate results get from the map workers to the reduce workers? Simple version described in [Dean & Ghemawat, 2008]:

- Map workers know the number of reduce workers R.
- Each *(Key2, Value2)* is written to a different file according to *hash(Key2) mod R*.
- The master tells the reduce worker which file to read from each map worker.

Later versions of MapReduce utilized more complex or even user-specified hashing; for example to:

- Better balance size of reduce problems.
- Reduce network traffic and/or simplify sorting during shuffle step.
- Cluster certain *Key2* tuples onto the same reduce workers.

Fault Tolerance

Bad things happen: Failed disks, partitioned networks, power shortages, . . .

- Key Idea:
  - The `map` and `reduce` operations are based on functional programming ideas: referential transparency and no side-effects.
  - If a worker crashes, it is as if it never existed.
  - The master can restart the task on another machine.
- The master periodically pings the tasks, and restarts dead ones.
  - Map tasks produce only temporary files, so if a completed map task fails before informing the master then it must be re-executed.
  - Reduce tasks produce files in the distributed file system (redundant and fault-tolerant), so no need to re-execute.

Semantics

- If the *map* and *reduce* functions are deterministic, then the result of `MapReduce` is the same as a sequential execution.
  - **This is really cool!**
  - There **is** a sequential implementation of `MapReduce`:
    - Read all of the *(Key1, Value1)* pairs from the input file.
    - Write all of the *(Key2, [list of Value2])* tuples to an intermediate file.
    - Sort the intermediate file by the *Key2* values.
    - Perform the reduce operation for each *Key2* value and write the results to the output file.
- If the *map* and *reduce* functions are not deterministic, then
  - It's a bit more complicated, but it's still reasonable.
  - If the reduce tasks are non-deterministic, then the result for each *Key2* is the result from some sequential implementation.
  - The paper doesn't talk about non-determinism for *map*, but it is probably similar.

Work Stealing

- Sometimes a worker is slow – **stragglers**.
- If the `MapReduce` task is near completion, the master assigns straggler tasks to idle processors.
- These are called **backup tasks**.
- Either the original or the backup process can complete the task.
- In practice, this **work stealing** by backup tasks:
  - Only adds a few percent to the total compute resources used.
  - Can result in substantial performance improvements: The paper reported a 44% slow-down when the sort benchmark was run without backup tasks.

Performance Issues

- The master attempts to schedule map tasks on the processor whose local disk holds the split being processed, or are nearby (by the network connections).
- Shuffle moves data from many map tasks to many reduce tasks.
  - Easily saturates the cross-section bandwidth of the network.
- For good performance, the map tasks should be filters that output much less data than they read.
  - Often not true of the "natural" intermediate representation (such as curriculum problem above).
  - Fewer distinct *Key2* values means less parallelism in reduce tasks.
- Can often reduce intermediate data size by partial reduction in the map workers.
  - May change the semantics of MapReduce (but not if reduce is associative and commutative).

MapReduce is Designed for BIG Data

- Communication between standard linux machines with generic networks takes milliseconds.
- Reading large disk files takes seconds.
- The task needs to be big enough to justify these overheads:
  - Equivalent to increasing λ by a few orders of magnitude.
  - MapReduce makes sense if the task is disk-limited and harnassing a few thousand disks provides the necessary disk bandwidth.
    - Think of it as “disk parallelism” instead of “CPU parallelism”.
    - Note: big-data companies like Amazon, Facebook and Google are moving to using FLASH memory and DRAM instead of disks, exactly because of these I/O bottlenecks.
  - Or if you have a really huge data set the compute time may dominate all of these overheads.

Results (Part 1)

Achieves impressive performance on massive data sets 2008–2013(?)

- Report in [Dean & Ghemawat, 2008]: Good performance on $\sim$ 2000 machines: `grep` and `sort` work through 10^{10} 100-byte records (1TB) in minutes.
- Google estimates $\sim$ 20PB / day in total MapReduce processing in January 2008.
- Google research blog reports sorting 10^{13} 100-byte records (1PB) on $\sim$ 4000 machines (and $\sim$ $48,000$ disks) in six hours in November 2008, then 33 minutes for 1 PB or 6 hours for 10 PB on 8000 machines in September 2011.
- Open source implementation in Hadoop widely available as a cloud service.
- Many example algorithms documented; for example, search for "map reduce" on http://scholar.google.ca.

Results (Part 2)

Big data processors are now moving away from MapReduce.

- Framework emphasizes batch processing of data residing in distributed file system, which limits flexibility and efficiency.
- ”We don’t really use MapReduce anymore” [Urs H olzle, senior vice president of technical infrastructure at Google speaking at Google I/O conference in 2014]
- Machine learning project Apache Mahout is shifting away from MapReduce algorithms to alternatives such as Apache Spark.
  - Worth noting: Spark also uses a functional programming model with referential transparency.

Summary

- MapReduce is a parallel programming pattern
  - Data are represented by collections of *(Key, Value)* pairs.
  - User provides **map** to take input *(Key1, Value1)* pairs to intermediate *(Key2, Value2)* pairs.
  - Shuffle step collects intermediate data into *(Key2, [list of Value2])* pairs and sorts it by *Key2*.
  - User provides **reduce** to take sorted *(Key2, [list of Value2])* pairs into *(Key2, Value3)* pairs.
- Details of the parallel implementation are handled by the MapReduce API:
  - Creating workers processes, delivering input and output files, shuffling intermediate data between map and reduce workers.
  - Handling failures and slow nodes.
- Performance is often bandwidth limited.
  - Locality matters: perform *map* on the machine with the data.
  - If *map* is an effective filter (bytes out $\ll$ bytes in), then we can reduce the impact of network congestion.
  - In practice, user must choose *(Key2, Value2)* representation wisely to trade-off shuffle effort for reduce parallelism.

Review: Properties of MapReduce

- How does MapReduce distribute work between map tasks?
- How does MapReduce distribute work between reduce tasks?
- How does MapReduce handle machine, network or other failures?
- How does MapReduce handle slow (i.e. straggler) machines?
- What are the requirements for the type-signatures of the *map* and *reduce* functions in a map-reduce?

Review: Example of a MapReduce Problem

I want to fly from Vancouver to Timbuktu. There are no direct flights, so I want to find the fastest route with one stop. How could I do this using map reduce?

- Input data is a table of airline flights of the form:

  ```
  (DepartCity, DepartTime, ArriveCity, ArriveTime)
  ```

  - Hint: use the intermediate city as *Key2*.
  - For simplicity, assume that all times are GMT (no need for time-zone conversion).
  - How does *map* filter out irrelevant flights?