Motivation: Large Scale Data Processing

- Want to process lots of data (> 1 TB)
- Want to parallelize across hundreds/thousands of commodity computers
  - New definition of cluster computing: large numbers of low-end processors working in parallel to solve a computing problem.
  - Parallel DB: a small number of high-end servers.
- Want to make this easy
MapReduce

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

Programming Model

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

- Input key-value pairs: records from the data source, e.g., lines out of files (filename, line), rows of a database, etc.
- 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)
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));
Click Stream Analysis: Page Frequencies

Clicks(time, url, referral_url, user_id, geo_info...)

map(String tuple_id, String tuple):
   EmitIntermediate(url, "1");

reduce(String url, Iterator list_tuples):
   int result = 0;
   for each t in list_tuples:
      result += ParseInt(t);
   Emit(AsString(result));

Select count(*)
From Clicks
Group By url;

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.
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
Optimizations

- “Combiner” functions can run on the 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?

Locality

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


Some Comments on MapReduce

- Strengths
  - Simplicity
  - Infrastructure support:
    - massive parallelism, fault tolerance, with proven success
    - Finer grained tolerance, less than a transaction
  - Cost benefits
  - Complex map functions for parse, transform; complex analytics for data mining, clustering, etc.
  - Storage system independent; heterogeneous storage systems
- Limitations:
  - Performance for structured data analysis
  - No indexes; large, repeated scans