MapReduce

- Goal: efficient parallelization of various tasks across 1000's of machines without the user having to worry about the details such as:
  - How to parallelize
  - How to distribute the data
  - How to handle failures
- Basic Idea:
  - If you force programs to be written using two primitives (*map* and *reduce*), parallelism can be gotten for free
    - Replace: map-reduce with SQL, parallelism with speed/ease-of-use
  - More programs than you might think can be written this way

MapReduce: Applications

- From Nice Overview by Curt Monash
- Three major classes:
  - Text tokenization, indexing, and search
  - Creation of other kinds of data structures (e.g., graphs)
  - Data mining and machine learning
- See this blog post for a long list of applications
- Or See Hadoop List
- For Machine Learning algorithms, see MAHOUT

Mapreduce

- Users needs to write two key functions:
  - Map: generate a set of (key, value) pairs
  - Reduce: group the pairs by *key's* and combine them (GROUP BY)
- Borrowed from Lisp

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

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

Mapreduce: Execution Overview

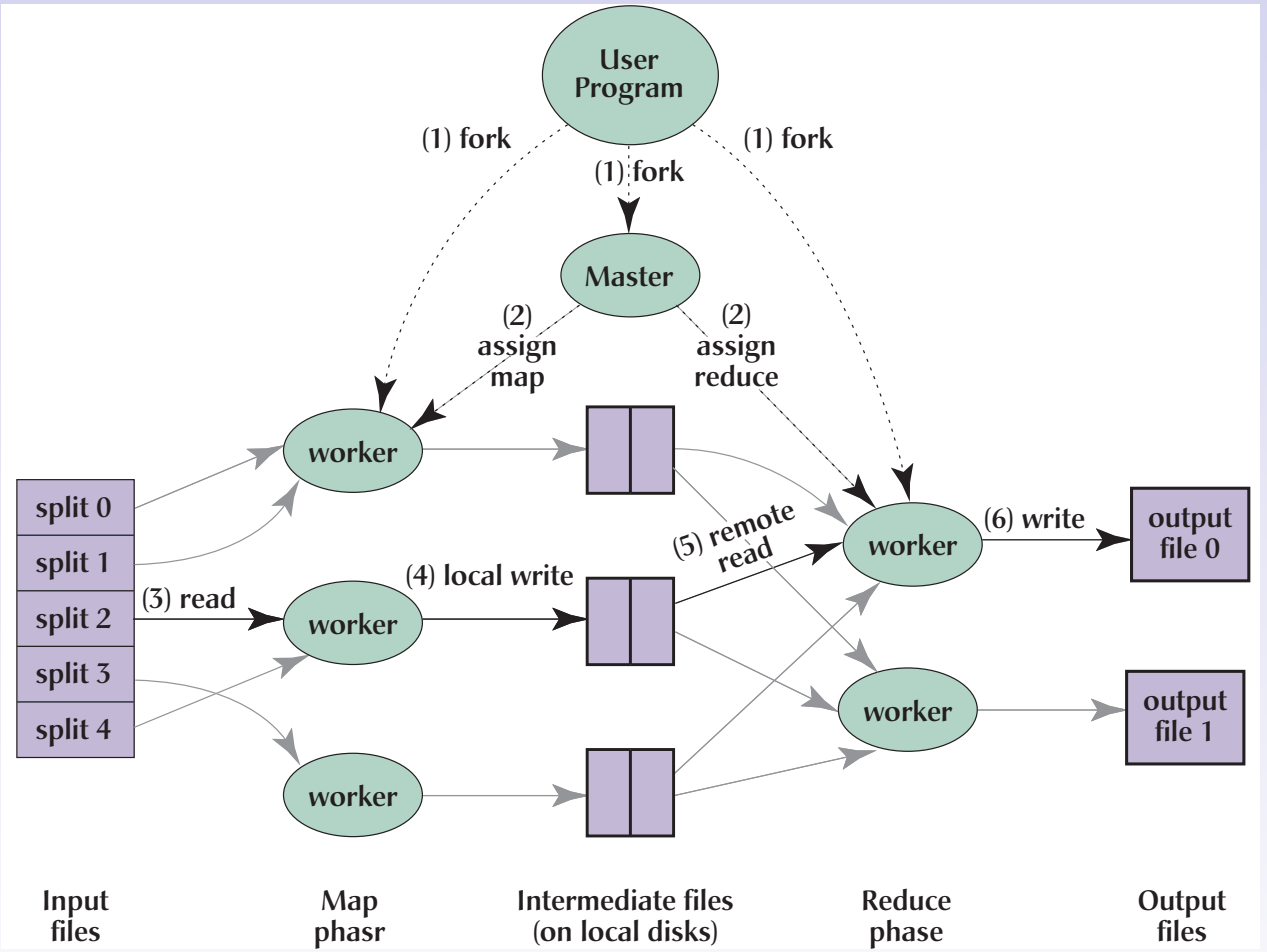

Fig. 1. Execution overview.

Mapreduce: Implementation

- A master for each tasks, assigns tasks to workers
- Data transfers using the file system (by passing file-names)
- Master pings the workers to make sure they are alive
  - If not, reassign the task to some other worker
- Work is divided into a large number of small chunks
  - Similar ideas used in parallel database for handling data skew
- Atomic commits using the file system

Mapreduce: Implementation

- Google File System
  - A distributed, fault-tolerant file system
  - Data divided into blocks of 64MB
  - Each block stored on several machines (typically 3)
- Mapreduce uses the location information to assign work

Mapreduce: Implementation

- Google File System
  - A distributed, fault-tolerant file system
  - Data divided into blocks of 64MB
  - Each block stored on several machines (typically 3)
- Mapreduce uses the location information to assign work
- Many other optimizations
  - Backup tasks to handle “straggler”
  - Control over partitioning functions
  - Ability to skip “bad” records

Mapreduce

- Has been used within Google for:
  - Large-scale machine learning problems
  - Clustering problems for Google News etc..
  - Generating summary reports
  - Large-scale graph computations
- Also replaced the original tools for large-scale indexing
  - ie., generating the inverted indexes etc.
  - runs as a sequence of 5 to 10 Mapreduce operations

Mapreduce: Thoughts

- Hadoop
  - Open-source implementation of Mapreduce
    - Has support for both the distributed file system and Mapreduce
- Cloud Computing
  - Somewhat vague term, but quite related

Mapreduce + Databases: Thoughts

- Abstract ideas have been known before
  - See Mapreduce: A Major Step Backwards; DeWitt and Stonebraker
  - Can be implemented using user-defined aggregates in PostgreSQL quite easily
  - Top-down, declarative design
    - The user specifies what is to be done, not how many machines to use etc...

Mapreduce + Databases: Thoughts

- Abstract ideas have been known before
  - See Mapreduce: A Major Step Backwards; DeWitt and Stonebraker
  - Can be implemented using user-defined aggregates in PostgreSQL quite easily
  - Top-down, declarative design
    - The user specifies what is to be done, not how many machines to use etc...
- The strength comes from simplicity and ease of use
  - No database system can come close to the performance of Mapreduce infrastructure
  - RDBMSs can't scale to that degree, are not as fault-tolerant etc...
    - Again: this is mainly because of ACID
    - Databases were designed to support it
    - Most of the Google tasks don't worry about that

Mapreduce + Databases: Thoughts

- Mapreduce is very good at what it was designed for
  - But may not be ideal for more complex tasks
    - E.g. no notion of "Query Optimization" (in particular, operator order optimization)
    - The sequence of Mapreduce tasks makes it procedural within a single machine
  - Joins are tricky to do
    - Mapreduce assumes a single input

Mapreduce + Databases: Thoughts

- Mapreduce is very good at what it was designed for
  - But may not be ideal for more complex tasks
    - E.g. no notion of "Query Optimization" (in particular, operator order optimization)
    - The sequence of Mapreduce tasks makes it procedural within a single machine
  - Joins are tricky to do
    - Mapreduce assumes a single input
- Trying to force use of Mapreduce may not be the best option
- However, much work in recent years on extending the functionality
  - See Pig project at Yahoo, Map-reduce-merge etc..

Outline

Overview

- Something that fits in between SQL and MapReduce
  - To make it easy for programmers to write procedural, non-SQL code
- Open source, on top of Hadoop
- No transactions – read-only analysis queries
- Supports nested data model (i.e., not in 1NF)
  - Allows sets/maps as fields
  - Interestingly: need this for GROUP operator
- UDFs written in Java

Example

EXAMPLE 1. *Suppose we have a table* `urls: (url, category, pagerank)`. *The following is a simple SQL query that finds, for each sufficiently large category, the average pagerank of high-pagerank urls in that category.*

```
SELECT category, AVG(pagerank)
FROM urls WHERE pagerank > 0.2
GROUP BY category HAVING COUNT(*) > 10^6
```

An equivalent Pig Latin program is the following. (Pig Latin is described in detail in Section 3; a detailed understanding of the language is not required to follow this example.)

```
good_urls = FILTER urls BY pagerank > 0.2;
groups = GROUP good_urls BY category;
big_groups = FILTER groups BY COUNT(good_urls)>10^6;
output = FOREACH big_groups GENERATE
            category, AVG(good_urls.pagerank);
```