Problem

Writing parallel code is hard for several reasons:

- parallelize computation
- distribute data
- handle failure
- load balancing
- fault tolerance

Solution

Uses a library that handle (and hides) all these details and makes the programmer life easier
Outline

- MapReduce
- Dryad
- PigLatin
MapReduce

Input Data → Map() → Map() → Map() → Reduce() → Reduce() → Output Data
Grep - MapReduce

input
file1.txt
file2.txt
file3.txt

Splitting
file1.txt
file2.txt
file3.txt

Mapping
file1.txt, 13
file1.txt, 150
file2.txt, 8
file3.txt, 100

Reducing
file1.txt, 13
file1.txt, 150
file2.txt, 8
file3.txt, 100

Final Result
file1.txt, 13
file1.txt, 150
file2.txt, 8
file3.txt, 100
Benefits for User

- Programmer writes two functions
  - map
  - reduce
- Doesn’t have to worry about distributed computing
  - faults are handled by the system
  - distributing the work
Benefits for System

- Run on commodity hardware
  - fault tolerant
    - unresponsive worker
    - master failure
  - backup tasks
Performance - Grep

- Scanned through $10^{10}$ 100-byte records
- 1764 workers were assigned
- Entire computation took 150 seconds including 60 sec of startup overhead
Dryad vs. MapReduce

- Generalization of MapReduce workflow.
- Gives programmer fine-grained control over communication graph
- Steeper learning curve to using API
System Overview
Fault Tolerance

- Job manager informed if a vertex execution fails
- If the process crashes the daemon notifies the job manager.
- If the daemon fails the job manager will get a heartbeat timeout.
PigLatin

Tries to improve the flexibility of Map-Reduce and increase code reusability by using:

● high level declarative querying (similar to SQL)
● low level procedural programming
Example: SQL

SQL:
SELECT category, AVG(pagerank)
FROM urls WHERE pagerank > 0.2
GROUP BY category HAVING COUNT(*) > 10^6
Example: Pig Latin

Pig Latin:

```pig
urls = LOAD 'urls_log.txt' USING myLoad()
    AS (urls, pagerank, category)
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_group GENERATE category,
    AVG(good_urls.pagerank)
```
Example: Pig Latin

Pig Latin:
urls = LOAD 'urls_log.txt' USING myLoad()
    AS (urls, pagerank, category)
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_group GENERATE category,
    AVG(good_urls.pagerank)

LOAD: specifies input data files, how to deserialize and convert into Pig Latin
Example: Pig Latin

Pig Latin:

urls = LOAD 'urls_log.txt' USING myLoad()
    AS (urls, pagerank, category)
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_group GENERATE category,
    AVG(good_urls.pagerank)
Example: Pig Latin

Pig Latin:

urls = LOAD ‘urls_log.txt’ USING myLoad() AS (urls, pagerank, category)
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_group GENERATE category, AVG(good_urls.pagerank)

FILTER: retains only part of the data and discards the rest
Example: Pig Latin

Pig Latin:

urls = LOAD 'urls_log.txt' USING myLoad()
    AS (urls, pagerank, category)

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_group GENERATE category,
    AVG(good_urls.pagerank)
Example: Pig Latin

Pig Latin:

urls = LOAD ‘urls_log.txt’ USING myLoad()
    AS (urls, pagerank, category)
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_group GENERATE category,
        AVG(good_urls.pagerank)

(co)GROUP: groups together tuples from more data sets
Example: Pig Latin

Pig Latin:

urls = LOAD ‘urls_log.txt’ USING myLoad()
  AS (urls, pagerank, category)
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_group GENERATE category,
  AVG(good_urls.pagerank)
Example: Pig Latin

**Pig Latin:**

```
urls = LOAD 'urls_log.txt' USING myLoad() AS (urls, pagerank, category)
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_group GENERATE category,
         AVG(good_urls.pagerank)
```

**FOREACH:** applies some processing to each tuple in the data set
Commands

Every command only performs one transformation on the data.

The programmer can write finer-grained optimizations.
Data Model

- **Atom**: simple atomic values (i.e., 20, ‘alice’)
- **Tuple**: a sequence of fields of any data type
- **Bag**: a collection of tuples with duplicates and not with the same schema
  (i.e., {('alice', 'lakers'), ('alice', ('iPod', 'apple')), ('alice', 'lakers')})
- **Map**: a collection of data items associated with a key
  (i.e., ['fanOf'->{('lakers'),('iPod')} ‘age’ -> 20])
Implementation (1)

Implemented using Hadoop, by compiling Pig Latin into map-reduce jobs.

- The Pig Latin interpreter parses the input files and bags to verify the command is valid
- A logical plan (~ relational algebra) for every bag defined
- Execution is carried out only when STORE is invoked
- Each COGROUP command is converted into a map-reduce job.
- The map function initially assigns key to tuples based on BY clauses.
- FILTER and FOREACH commands from the LOAD to the first COGROUP are pushed into the map of $C_1$.
- Subsequent commands ($C_i$) are pushed in the reduce functions of their corresponding COGROUP.
Grep - PigLatin

messages = LOAD 'messages'
warns = FILTER messages BY $0 MATCHES '.*WARN+.*'
STORE warns INTO 'warnings'
PigLatin vs MapReduce and Dryad

- No quantitative results
- PigLatin is much more focused on usability
  - Allows for User Defined Functions
  - It come together with a debugging environment
Questions?
References


Graph

- Dryad library is used to create a graph vertex.
- New edges are created by applying either pointwise or complete bipartite composition operation to two existing graphs.
- Users can also define new composition operations
- Graphs can also be merged
Job

- vertices are created according to partitioned input data.
- outputs are concatenated to produce a single named distributed file.
- Each vertex is placed into a stage to simplify job management.
Job Execution

- job manager tracks state and history of vertices
- job is terminated if job manager fails
- job manager performs greedy scheduling