Distributed Computing at Web Scale

Serge Abiteboul   Ioana Manolescu   Philippe Rigaux
Marie-Christine Rousset   Pierre Senellart

Web Data Management and Distribution
http://webdam.inria.fr/textbook

November 10, 2011

Outline

1. MapReduce
   - Introduction
   - The MapReduce Computing Model
   - MapReduce Optimization
   - Application: PageRank
   - MapReduce in Hadoop
2. Toward Easier Programming Interfaces: Pig
3. Conclusions

Outline

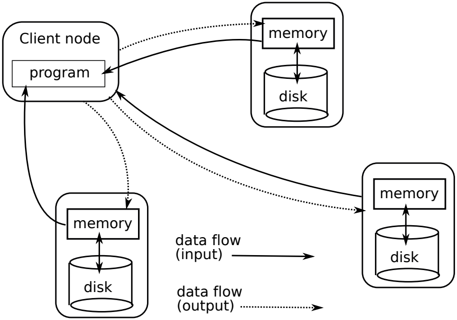
1. MapReduce
   - Introduction
   - The MapReduce Computing Model
   - MapReduce Optimization
   - Application: PageRank
   - MapReduce in Hadoop
2. Toward Easier Programming Interfaces: Pig
3. Conclusions

Data analysis at a large scale

- Very large data collections (TB to PB) stored on distributed filesystems:
  - Query logs
  - Search engine indexes
  - Sensor data
- Need efficient ways for analyzing, reformatting, processing them
- In particular, we want:
  - Parallelization of computation (benefiting of the processing power of all nodes in a cluster)
  - Resilience to failure

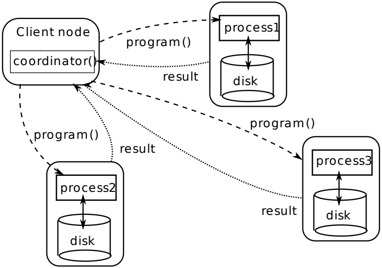
Centralized computing with distributed data storage

Run the program at client node, get data from the distributed system.



Downsides: important data flows, no use of the cluster computing resources.

Pushing the program near the data



- MapReduce: A programming model (inspired by standard functional programming operators) to facilitate the development and execution of distributed tasks.
- Published by Google Labs in 2004 at OSDI [DG04]. Widely used since then, open-source implementation in Hadoop.

MapReduce in Brief

- The programmer defines the program logic as two functions:

  Map transforms the input into key-value pairs to process

  Reduce aggregates the list of values for each key

- The MapReduce environment takes in charge distribution aspects
- A complex program can be decomposed as a succession of Map and Reduce tasks
- Higher-level languages (Pig, Hive, etc.) help with writing distributed applications

Outline

1. MapReduce
   - Introduction
   - The MapReduce Computing Model
   - MapReduce Optimization
   - Application: PageRank
   - MapReduce in Hadoop
2. Toward Easier Programming Interfaces: Pig
3. Conclusions

Three operations on key-value pairs

1. User-defined: *map* : $(K, V) \rightarrow \text{list}(K', V')$

```
function map(uri, document)
  foreach distinct term in document
    output (term, count(term, document))
```

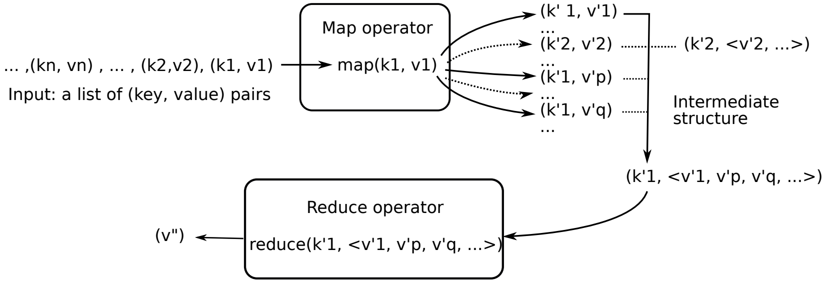
2. Fixed behavior: *shuffle* : $\text{list}(K', V') \rightarrow \text{list}(K', \text{list}(V'))$ regroups all intermediate pairs on the key

3. User-defined: *reduce* : $(K', \text{list}(V')) \rightarrow \text{list}(K'', V'')$

```
function reduce(term, counts)
  output (term, sum(counts))
```

Job workflow in MapReduce

Important: each pair, at each phase, is processed independently from the other pairs.



Network and distribution are transparently managed by the MapReduce environment.

Example: term count in MapReduce (input)

URL	Document
u_1	the jaguar is a new world mammal of the felidae family.
u_2	for jaguar, atari was keen to use a 68k family device.
u_3	mac os x jaguar is available at a price of us $199 for apple's new "family pack".
u_4	one such ruling family to incorporate the jaguar into their name is jaguar paw.
u_5	it is a big cat.

Example: term count in MapReduce

term	count
jaguar	1
mammal	1
family	1
jaguar	1
available	1
jaguar	1
family	1
family	1
jaguar	2
...	

output

shuffle input

map

Example: term count in MapReduce

term	count
jaguar	1
mammal	1
family	1
jaguar	1
available	1
jaguar	1
family	1
family	1
jaguar	2
...	

output
shuffle input

map

term	count
jaguar	1,1,1,2
mammal	1
family	1,1,1
available	1
...	

shuffle output
reduce input

Example: term count in MapReduce

term	count
jaguar	1
mammal	1
family	1
jaguar	1
available	1
jaguar	1
family	1
family	1
jaguar	2
...	

output
shuffle input

map

term	count
jaguar	1,1,1,2
mammal	1
family	1,1,1
available	1
...	

shuffle output
reduce input

term	count
jaguar	5
mammal	1
family	3
available	1
...	

output

final

Example: simplification of the *map*

```
function map(uri, document)
  foreach distinct term in document
    output (term, count(term, document))
```

can actually be further simplified:

```
function map(uri, document)
  foreach term in document
    output (term, 1)
```

since all counts are aggregated.

Might be less efficient though (we may need a combiner, see further)

A MapReduce cluster

Nodes inside a MapReduce cluster are decomposed as follows:

- A jobtracker acts as a master node; MapReduce jobs are submitted to it
- Several tasktrackers run the computation itself, i.e., *map* and *reduce* tasks
- A given tasktracker may run several tasks in parallel
- Tasktrackers usually also act as data nodes of a distributed filesystem (e.g., GFS, HDFS)

+ a client node where the application is launched.

Processing a MapReduce job

A MapReduce job takes care of the distribution, synchronization and failure handling. Specifically:

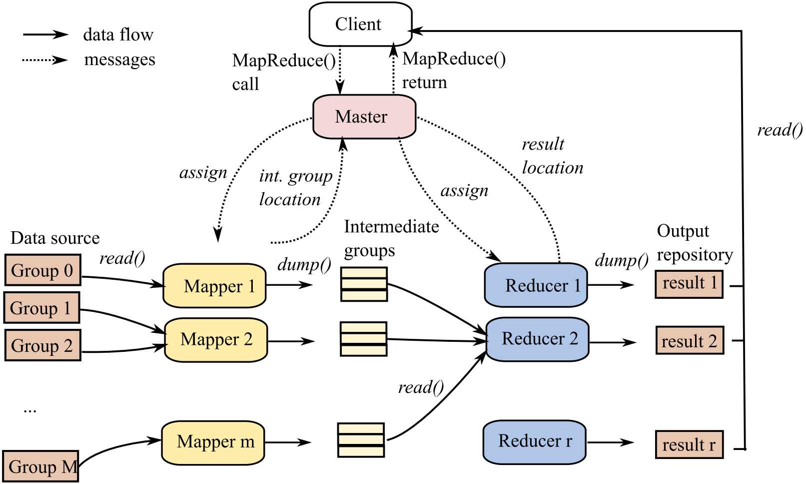
- the input is split into M groups; each group is assigned to a mapper (assignment is based on the data locality principle)
- each mapper processes a group and stores the intermediate pairs locally
- grouped instances are assigned to reducers thanks to a hash function
- (*shuffle*) intermediate pairs are sorted on their key by the reducer
- one obtains grouped instances, submitted to the *reduce* function

Remark: the data locality does no longer hold for the *reduce* phase, since it reads from the mappers.

Assignment to reducer and mappers

- Each mapper task processes a fixed amount of data (split), usually set to the distributed filesystem block size (e.g., 64MB)
- The number of mapper nodes is function of the number of mapper tasks and the number of available nodes in the cluster: each mapper nodes can process (in parallel and sequentially) several mapper tasks
- Assignment to mapper tries optimizing data locality: the mapper node in charge of a split is, if possible, one that stores a replica of this split (or if not possible, a node of the same rack)
- The number of reducer tasks is set by the user
- Assignment to reducers is done through a hashing of the key, usually uniformly at random; no data locality possible

Distributed execution of a MapReduce job.



Processing the term count example

Let the input consists of documents, say, one million 100-terms documents of approximately 1 KB each.

The split operation distributes these documents in groups of 64 MBs: each group consist of 64,000 documents. Therefore $M = \lceil 1{,}000{,}000/64{,}000 \rceil \approx 16{,}000$ groups.

If there are 1,000 mapper node, each node processes on average 16 splits.

If there are 1,000 reducers, each reducer r_i processes all key-value pairs for terms t such that $hash(t) = i$ ($1 \leq i \leq 1,000$)

Processing the term count example (2)

Assume that *hash('call')* = *hash('mine')* = *hash('blog')* = *i* = 100. We focus on three Mappers m_p, m_q and m_r:

1. G_i^p *=(<. . . , ('mine', 1), . . . , ('call',1), . . . , ('mine',1), . . . , ('blog', 1) . . . >*
2. G_i^q *=(< . . . , ('call',1), . . . , ('blog',1), . . . >*
3. G_i^r *=(<. . . , ('blog', 1), . . . , ('mine',1), . . . , ('blog',1), . . . >*

r_i reads G_i^p, G_i^p and G_i^p from the three Mappers, sorts their unioned content, and groups the pairs with a common key:

. . . , ('blog', <1, 1, 1, 1>), . . . , ('call', <1, 1>), . . . , ('mine', <1, 1, 1>)

Our *reduce* function is then applied by r_i to each element of this list. The output is *('blog', 4), ('call', 2) and ('mine', 3)*

Failure management

In case of failure, because the tasks are distributed over hundreds or thousands of machines, the chances that a problems occurs somewhere are much larger; starting the job from the beginning is not a valid option.

The Master periodically checks the availability and reachability of the tasktrackers (heartbeats) and whether *map* or *reduce* jobs make any progress

1. if a reducer fails, its task is reassigned to another tasktracker; this usually require restarting mapper tasks as well (to produce intermediate groups)
2. if a mapper fails, its task is reassigned to another tasktracker
3. if the jobtracker fails, the whole job should be re-initiated

Joins in MapReduce

Two datasets, *A* and *B* that we need to join for a MapReduce task

- If one of the dataset is small, it can be sent over fully to each tasktracker and exploited inside the *map* (and possibly *reduce*) functions
- Otherwise, each dataset should be grouped according to the join key, and the result of the join can be computing in the *reduce* function

Not very convenient to express in MapReduce. Much easier using Pig.

Using MapReduce for solving a problem

- Prefer:
  - Simple *map* and *reduce* functions
  - Mapper tasks processing large data chunks (at least the size of distributed filesystem blocks)
- A given application may have:
  - A chain of *map* functions (input processing, filtering, extraction. . .)
  - A sequence of several *map-reduce* jobs
  - No *reduce* task when everything can be expressed in the *map* (zero reducers, or the identity reducer function)
- Not the right tool for everything(see further)

Outline

1 MapReduce
- Introduction
- The MapReduce Computing Model
- MapReduce Optimization
- Application: PageRank
- MapReduce in Hadoop

2 Toward Easier Programming Interfaces: Pig

3 Conclusions

Combiners

- A mapper task can produce a large number of pairs with the same key
- They need to be sent over the network to the reducer: costly
- It is often possible to combine these pairs into a single key-value pair

Example

(jaguar,1), (jaguar, 1), (jaguar, 1), (jaguar, 2)$\rightarrow$(jaguar, 5)

- *combiner* : $\text{list}(V') \rightarrow V'$ function executed (possibly several times) to combine the values for a given key, on a mapper node
- No guarantee that the *combiner* is called
- Easy case: the combiner is the same as the *reduce* function. Possible when the aggregate function α computed by *reduce* is distributive: $\alpha(k_1, \alpha(k_2, k_3)) = \alpha(k_1, k_2, k_3)$

Compression

- Data transfers over the network:
  - From datanodes to mapper nodes (usually reduced using data locality)
  - From mappers to reducers
  - From reducers to datanodes to store the final output
- Each of these can benefit from data compression
- Tradeoff between volume of data transfer and (de)compression time
- Usually, compressing *map* outputs using a fast compressor increases efficiency

Optimizing the *shuffle* operation

- Sorting of pairs on each reducer, to compute the groups: costly operation
- Sorting much more efficient in memory than on disk
- Increasing the amount of memory available for *shuffle* operations can greatly increase the performance
- ... at the downside of less memory available for *map* and *reduce* tasks (but usually not much needed)

Speculative execution

- The MapReduce jobtracker tries detecting tasks that take longer than usual (e.g., because of hardware problems)
- When detected, such a task is speculatively executed on another tasktracker, without killing the existing task
- Eventually, when one of the attempts succeeds, the other one is killed

Outline

1. MapReduce
   - Introduction
   - The MapReduce Computing Model
   - MapReduce Optimization
   - Application: PageRank
   - MapReduce in Hadoop

2. Toward Easier Programming Interfaces: Pig

3. Conclusions

PageRank computation

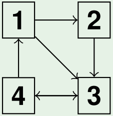
PageRank: importance score for nodes in a graph, used for ranking query results of Web search engines. Fixpoint computation, as follows:

1. Compute G. Make sure lines sum to 1.
2. Let u be the uniform vector of sum 1, $v = u$.
3. Repeat N times:
   - Set $v := (1 - d)G^T v + du$ (say, $d = \frac{1}{4}$).

Exercise

Express PageRank computation as a MapReduce problem. Main program? *map* and *reduce* functions? *combiner* function?

Illustrate on this graph.



Outline

1. MapReduce
   - Introduction
   - The MapReduce Computing Model
   - MapReduce Optimization
   - Application: PageRank
   - MapReduce in Hadoop
2. Toward Easier Programming Interfaces: Pig
3. Conclusions

Hadoop

- Open-source software, Java-based, managed by the Apache foundation, for large-scale distributed storage and computing
- Originally developed for Apache Nutch (open-source Web search engine), a part of Apache Lucene (text indexing platform)
- Open-source implementation of GFS and Google's MapReduce
- Yahoo!: a main contributor of the development of Hadoop
- Hadoop components:
  - Hadoop filesystem (HDFS)
  - MapReduce
  - Pig (data exploration), Hive (data warehousing): higher-level languages for describing MapReduce applications
  - HBase: column-oriented distributed DBMS
  - ZooKeeper: coordination service for distributed applications

Hadoop programming interfaces

- Different APIs to write Hadoop programs:
  - A rich Java API (main way to write Hadoop programs)
  - A Streaming API that can be used to write *map* and *reduce* functions in any programming language (using standard inputs and outputs)
  - A C++ API (Hadoop Pipes)
  - With a higher-language level (e.g., Pig, Hive)
- Advanced features only available in the Java API
- Two different Java APIs depending on the Hadoop version; presenting the "old" one

Java *map* for the term count example

```
public class TermCountMapper extends MapReduceBase
  implements Mapper<Text,Text,Text,IntWritable> {
  public void map(
    Text uri, Text document,
    OutputCollector<Text, IntWritable> output,
    Reporter reporter)
  {
    Pattern p=Pattern.compile("[\\p{L}]+");
    Matcher m=p.matcher(document);
    while(matcher.find()) {
      String term=matcher.group().
      output.collect(new Text(term),new IntWritable(1));
    }
  }
}
```

Java *reduce* for the term count example

```
public class TermCountReducer extends MapReduceBase
  implements Reducer<Text,IntWritable,Text,IntWritable> {
  public void reduce(
    Text term, Iterator<IntWritable> counts,
    OutputCollector<Text, IntWritable> output,
    Reporter reporter)
  {
    int sum=0;
    while(counts.hasNext()) {
      sum+=values.next().get();
    }
    output.collect(term, new IntWritable(sum));
  }
}
```

Java driver for the term count example

```
public class TermCount {
  public static void main(String args[]) throws IOException {
    JobConf conf = new JobConf(TermCount.class);
    FileInputFormat.addInputPath(conf, new Path(args[0]));
    FileOutputFormat.addOutputPath(conf, new Path(args[1]));

    // In a real application, we would have a custom input
    // format to fetch URI-document pairs
    conf.setInputFormat(KeyValueTextInputFormat.class);

    conf.setMapperClass(TermCountMapper.class);
    conf.setCombinerClass(TermCountReducer.class);
    conf.setReducerClass(TermCountReducer.class);

    conf.setOutputKeyClass(Text.class);
    conf.setOutputValueClass(IntWritable.class);

    JobClient.runJob(conf);
  }
}
```

Testing and executing a Hadoop job

- Required environment:
  - JDK on client
  - JRE on all Hadoop nodes
  - Hadoop distribution (HDFS + MapReduce) on client and all Hadoop nodes
  - SSH servers on each tasktracker, SSH client on jobtracker (used to control the execution of tasktrackers)
  - An IDE (e.g., Eclipse + plugin) on client
- Three different execution modes:

  local One mapper, one reducer, run locally from the same JVM as the client

  pseudo-distributed mappers and reducers are launched on a single machine, but communicate over the network

  distributed over a cluster for real runs

Debugging MapReduce

- Easiest: debugging in local mode
- Web interface with status information about the job
- Standard output and error channels saved on each node, accessible through the Web interface
- Counters can be used to track side information across a MapReduce job (e.g., number of invalid input records)
- Remote debugging possible but complicated to set up (impossible to know in advance where a *map* or *reduce* task will be executed)
- IsolationRunner allows to run in isolation part of the MapReduce job

Task JVM reuse

- By default, each *map* and *reduce* task (of a given split) is run in a separate JVM
- When there is a lot of initialization to be done, or when splits are small, might be useful to reuse JVMs for subsequent tasks
- Of course, only works for tasks run on the same node

Hadoop in the cloud

- Possibly to set up one's own Hadoop cluster
- But often easier to use clusters in the cloud that support MapReduce:
  - Amazon EC2
  - Cloudera
  - etc.
- Not always easy to know the cluster's configuration (in terms of racks, etc.) when on the cloud, which hurts data locality in MapReduce

Outline

1. MapReduce

2. Toward Easier Programming Interfaces: Pig
   - Basics
   - Pig operators
   - From Pig to MapReduce

3. Conclusions

Outline

1 MapReduce

2 Toward Easier Programming Interfaces: Pig
- Basics
- Pig operators
- From Pig to MapReduce

3 Conclusions

Pig Latin

Motivation: define high-level languages that use MapReduce as an underlying data processor.

A Pig Latin statement is an operator that takes a relation as input and produces another relation as output.

Pig Latin statements are generally organized in the following manner:

1. A **LOAD** statement reads data from the file system as a *relation* (list of tuples).
2. A series of "transformation" statements process the data.
3. A **STORE** statement writes output to the file system; or, a **DUMP** statement displays output to the screen.

Statements are executed as composition of MapReduce jobs.

Using Pig

- Part of Hadoop, downloadable from the Hadoop Web site
- Interactive interface (Grunt) and batch mode
- Two execution modes:

  local data is read from disk, operations are directly executed, no MapReduce

  MapReduce on top of a MapReduce cluster (pipeline of MapReduce jobs)

Example input data

A flat file, tab-separated, extracted from DBLP.

```
2005  VLDB J.        Model-based approximate querying in sensor net
1997  VLDB J.        Dictionary-Based Order-Preserving String Compr
2003  SIGMOD Record Time management for new faculty.
2001  VLDB J.        E-Services - Guest editorial.
2003  SIGMOD Record Exposing undergraduate students to database sy
1998  VLDB J.        Integrating Reliable Memory in Databases.
1996  VLDB J.        Query Processing and Optimization in Oracle Rc
1996  VLDB J.        A Complete Temporal Relational Algebra.
1994  SIGMOD Record Data Modelling in the Large.
2002  SIGMOD Record Data Mining: Concepts and Techniques - Book Re
```

Computing average number of publications per year

```
-- Load records from the file
articles = load 'journal.txt'
  as (year: chararray, journal:chararray,
      title: chararray) ;

sr_articles = filter articles
  by journal=='SIGMOD Record';

year_groups = group sr_articles by year;

avg_nb = foreach year_groups
  generate group, count(sr_articles.title);

dump avg_nb;
```

The data model

The model allows nesting of bags and tuples. Example: the `year_group` temporary bag.

```
group: 1990
sr_articles:
 {
  (1990, SIGMOD Record, SQL For Networks of Relations.),
  (1990, SIGMOD Record, New Hope on Data Models and Types.
 }
```

Unlimited nesting, but no references, no constraint of any kind (for parallelization purposes).

Flexible representation

Pig allows the representation of heterogeneous data, in the spirit of semi-structured dat models (e.g., XML).

The following is a bag with heterogeneous tuples.

```
{
  (2005, {'SIGMOD Record', 'VLDB J.'},
             {'article1', article2'})
  (2003, 'SIGMOD Record', {'article1', article2'},
             {'author1', 'author2'})
}
```

Outline

1. MapReduce
2. Toward Easier Programming Interfaces: Pig
   - Basics
   - Pig operators
   - From Pig to MapReduce
3. Conclusions

Main Pig operators

Operator	Description
foreach	Apply one or several expression(s) to each of the input tuples
filter	Filter the input tuples with some criteria
distinct	Remove duplicates from an input
join	Join of two inputs
group	Regrouping of data
cogroup	Associate two related groups from distinct inputs
cross	Cross product of two inputs
order	Order an input
limit	Keep only a fixed number of elements
union	Union of two inputs (note: no need to agree on a same schema, as in SQL)
split	Split a relation based on a condition

Example dataset

A simple flat file with tab-separated fields.

```
1995  Foundations of Databases Abiteboul
1995  Foundations of Databases Hull
1995  Foundations of Databases Vianu
2010  Web Data Management      Abiteboul
2010  Web Data Management      Manolescu
2010  Web Data Management      Rigaux
2010  Web Data Management      Rousset
2010  Web Data Management      Senellart
```

NB: Pig accepts inputs from user-defined function, written in Java – allows to extract data from any source.

The group operator

The “program”:

```
books = load 'webdam-books.txt'
    as (year: int, title: chararray, author: chararray) ;
group_auth = group books by title;
authors = foreach group_auth
  generate group, books.author;
dump authors;
```

and the result:

```
(Foundations of Databases,
   {(Abiteboul),(Hull),(Vianu)})
(Web Data Management,
   {(Abiteboul),(Manolescu),(Rigaux),(Rousset),(Senellart)}
```

Unnesting with flatten

Flatten serves to unnest a nested field.

```
-- Take the 'authors' bag and flatten the nested set
flattened = foreach authors
  generate group, flatten(author);
```

Applied to the previous `authors` bags, one obtains:

```
(Foundations of Databases,Abiteboul)
(Foundations of Databases,Hull)
(Foundations of Databases,Vianu)
(Web Data Management,Abiteboul)
(Web Data Management,Manolescu)
(Web Data Management,Rigaux)
(Web Data Management,Rousset)
(Web Data Management,Senellart)
```

The cogroup operator

Allows to gather two data sources in nested fields

Example: a file with publishers:

```
Fundations of Databases Addison-Wesley              USA
Fundations of Databases Vuibert                     France
Web Data Management     Cambridge University Press  USA
```

The program:

```
publishers = load 'webdam-publishers.txt'
    as (title: chararray, publisher: chararray) ;
cogrouped = cogroup flattened by group,
                    publishers by title;
```

The result

For each grouped field value, two nested sets, coming from both sources.

```
(Foundations of Databases,
  { (Foundations of Databases,Abiteboul),
    (Foundations of Databases,Hull),
    (Foundations of Databases,Vianu)
  },
  {(Foundations of Databases,Addison-Wesley),
   (Foundations of Databases,Vuibert)
  }
)
```

A kind of join? Yes, at least a preliminary step.

Joins

Same as before, but produces a flat output (cross product of the inner nested bags). The nested model is usually more elegant and easier to deal with.

```
-- Take the 'flattened' bag, join with 'publishers'
joined = join flattened by group, publishers by title;
```

```
(Foundations of Databases,Abiteboul,
      Fundations of Databases,Addison-Wesley)
(Foundations of Databases,Abiteboul,
      Fundations of Databases,Vuibert)
(Foundations of Databases,Hull,
      Fundations of Databases,Addison-Wesley)
(Foundations of Databases,Hull,
     ...
```

Outline

1 MapReduce

2 Toward Easier Programming Interfaces: Pig
- Basics
- Pig operators
- From Pig to MapReduce

3 Conclusions

Plans

- A Pig program describes a logical data flow
- This is implemented with a physical plan, in terms of grouping or nesting operations
- This is in turn (for MapReduce execution) implemented using a sequence of *map* and *reduce* steps

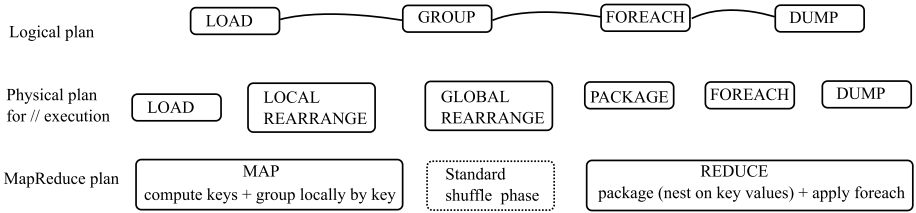
Physical operators

Local Rearrange group tuples with the same key, on a local machine

Global Rearrange group tuples with the same key, globally on a cluster

Package construct a nested tuple from tuples that have been grouped

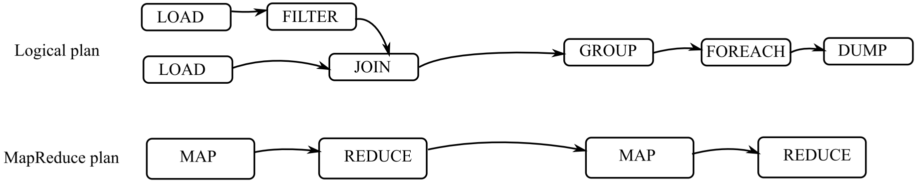
Translation of a simple Pig program



A more complex join-group program

```
-- Load books, but keep only books from Victor Vianu
books = load 'webdam-books.txt'
    as (year: int, title: chararray, author: chararray) ;
vianu = filter books by author == 'Vianu';

publishers = load 'webdam-publishers.txt'
    as (title: chararray, publisher: chararray) ;

-- Join on the book title
joined = join vianu by title, publishers by title;

-- Now, group on the author name
grouped = group joined by vianu::author;

-- Finally count the publishers
-- (nb: we should remove duplicates!)
count = foreach grouped
  generate group, COUNT(joined.publisher);
```

Translation of a join-group program



Outline

1. MapReduce
2. Toward Easier Programming Interfaces: Pig
3. Conclusions

MapReduce limitations (1/2)

- High latency. Launching a MapReduce job has a high overhead, and *reduce* functions are only called after all *map* functions succeed, not suitable for applications needing a quick result.
- Batch processing only. MapReduce excels at processing a large collection, not at retrieving individual items from a collection.
- Write-once, read-many mode. No real possibility of updating a dataset using MapReduce, it should be regenerated from scratch
- No transactions. No concurrency control at all, completely unsuitable for transactional applications [PPR^{+}09].

MapReduce limitations (2/2)

- Relatively low-level. Ongoing efforts for more high-level languages: Scope [CJL+08], Pig [ORS+08, GNC+09], Hive [TSJ+09], Cascading `http://www.cascading.org/`
- No structure. Implies lack of indexing, difficult to optimize, etc. [DS87]
- Hard to tune. Number of reducers? Compression? Memory available at each node? etc.

Hybrid systems

- Best of both worlds?
  - DBMS are good at transactions, point queries, structured data
  - MapReduce is good at scalability, batch processing, key-value data
- HadoopDB [ABPA+09]: standard relational DBMS at each node of a cluster, MapReduce allows communication between nodes
- Possible to use DBMS inputs natively in Hadoop, but no control about data locality

Job Scheduling

- Multiple jobs concurrently submitted to the MapReduce jobtracker
- Fair scheduling required:
  - each submitted job should have some share of the cluster
  - prioritization of jobs
  - long-standing jobs should not block quick jobs
  - fairness with respect to users
- Standard Hadoop scheduler: priority queue
- Hadoop Fair Scheduler: ensures cluster resources are shared among users. Preemption (= killing running tasks) possible in case the sharing becomes unbalnaced.

What you should remember on distributed computing

MapReduce is a simple model for batch processing of very large collections.
⇒ good for data analytics; not good for point queries (high latency).

The systems brings robustness against failure of a component and transparent distribution and scalability.
⇒ more expressive languages required (Pig)

Resources

- Original description of the MapReduce framework [DG04]
- Hadoop distribution and documentation available at `http://hadoop.apache.org/`
- Documentation for Pig is available at `http://wiki.apache.org/pig/`
- Excellent textbook on Hadoop [Whi09]
- Online MapReduce exercises with validation `http://cloudcomputing.ruc.edu.cn/login/login.jsp`

References I

Azza Abouzeid, Kamil Bajda-Pawlikowski, Daniel J. Abadi, Alexander Rasin, and Avi Silberschatz.
HadoopDB: An Architectural Hybrid of MAPREDUCE and DBMS Technologies for Analytical Workloads.
Proceedings of the VLDB Endowment (PVLDB), 2(1):922–933, 2009.

Sergey Brin and Lawrence Page.
The anatomy of a large-scale hypertextual Web search engine.
Computer Networks, 30(1–7):107–117, April 1998.

Ronnie Chaiken, Bob Jenkins, Per-Åke Larson, Bill Ramsey, Darren Shakib, Simon Weaver, and Jingren Zhou.
SCOPE: easy and efficient parallel processing of massive data sets.
Proc. Intl. Conf. on Very Large Databases (VLDB), 1(2):1265–1276, 2008.

References II

- Jeffrey Dean and Sanjay Ghemawat.
  MAPREDUCE: Simplified Data Processing on Large Clusters.
  In *Intl. Symp. on Operating System Design and Implementation (OSDI)*, pages 137–150, 2004.

- D. DeWitt and M. Stonebraker.
  MAPREDUCE, a major Step Backward.
  DatabaseColumn blog, 1987.
  http://databasecolumn.vertica.com/database-innovation/mapreduce-a-major-step-backwards/.

- Alan Gates, Olga Natkovich, Shubham Chopra, Pradeep Kamath, Shravan Narayanam, Christopher Olston, Benjamin Reed, Santhosh Srinivasan, and Utkarsh Srivastava.
  Building a HighLevel Dataflow System on top of MAPREDUCE: The PIG Experience.
  Proceedings of the VLDB Endowment (PVLDB), 2(2):1414–1425, 2009.

References III

- Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins.
  Pig latin: a not-so-foreign language for data processing.
  In *Proc. ACM Intl. Conf. on the Management of Data (SIGMOD)*, pages 1099–1110, 2008.
- Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel J. Abadi, David J. DeWitt, Samuel Madden, and Michael Stonebraker.
  A comparison of approaches to large-scale data analysis.
  In *Proc. ACM Intl. Conf. on the Management of Data (SIGMOD)*, pages 165–178, 2009.
- Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy.
  Hive - A Warehousing Solution Over a Map-Reduce Framework.
  Proceedings of the VLDB Endowment (PVLDB), 2(2):1626–1629, 2009.

References IV

Tom White.
Hadoop: The Definitive Guide.
O'Reilly, Sebastopol, CA, USA, 2009.