Data Analytics – Other approaches

Giovanni Simonini

DBGroup
Università di Modena e Reggio Emilia
Dipartimento di Ingegneria "Enzo Ferrari"

Outline

- Abstractions on top of MapReduce
  - Pig
  - Hive
- MapReduce and Machine learning
  - Mahout
- MapReduce and graphs
  - Giraph

Pig and Hive

ABSTRACTIONS ON TOP OF MAPREDUCE

Abstractions on top of MapReduce

High level languages: easier to program MR jobs

Input Data

```
urls:(url, category, pagerank)
```

PIG

```
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);
```

HIVE

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

Finds, for each sufficiently large category, the average pagerank of high-pagerank urls in that category.

Not only easier languages!

PIG and HIVE

- Allow to express programs through declarative languages that is transformed in a series of MapReduce Job.

- Bring Relational Algebra on top of MapReduce
  - higher layer of abstraction that allows a series of optimization of the generates MapReduce jobs

What is Pig

- Pig provides an engine for executing data flow in parallel on Hadoop
  - Pig Latin: language for expressing the data flows
    - Operators for many traditional data operations (join, sort, filter)
    - Ability to develop UDF (User Defined Function) for reading, processing, and writing data
- Pig Latin use cases tend to fall into three separate categories:
  - traditional extract transform load (ETL) data pipelines
  - research on raw data
  - iterative processing

A Pig Latin program describes a data flow

- How do we go from Pig Latin to MapReduce?
  - Pig Compiler
    - Complex execution environment that interacts with Hadoop MapReduce
    - Pig Latin operators are translated into MapReduce code
- Pig Optimizer
  - Pig Latin data flows undergo an (automatic) optimization phase
  - These optimizations are borrowed from the RDBMS community (thanks to Relational Algebra)

In Pig Latin

```
Users = load 'users' as (name, age);
Fltrd = filter Users by
        age >= 18 and age <= 25;
Pages = load 'pages' as (user, url);
Jnd = join Fltrd by name, Pages by user;
Grpd = group Jnd by url;
Smmd = foreach Grpd generate group,
       COUNT(Jnd) as clicks;
Srtd = order Smmd by clicks desc;
Top5 = limit Srtd 5;
store Top5 into 'top5sites';
```

9 lines of code, 15 minutes to write

Writing Pig Program vs "Native" MapReduce (2)

In MapReduce

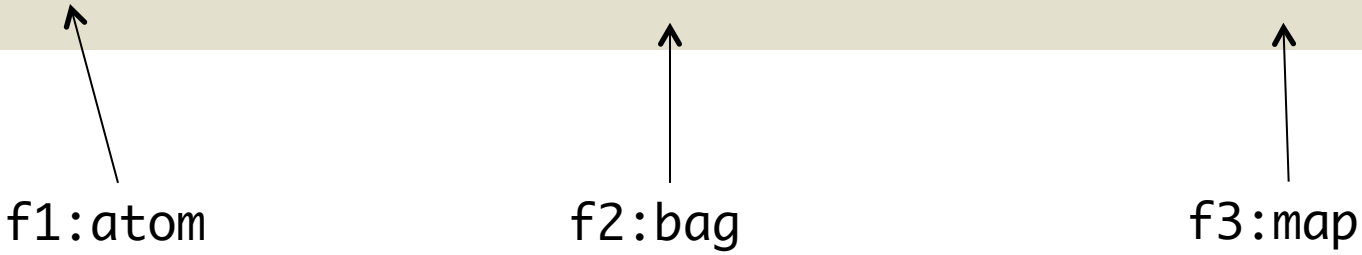
```
import java.io.IOException;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.io.Writable;
import org.apache.hadoop.io.WritableComparable;
import org.apache.hadoop.mapred.FileInputFormat;
import org.apache.hadoop.mapred.FileOutputFormat;
import org.apache.hadoop.mapred.JobConf;
import org.apache.hadoop.mapred.KeyValueTextInputFormat;
import org.apache.hadoop.mapred.Mapper;
import org.apache.hadoop.mapred.MapReduceBase;
import org.apache.hadoop.mapred.OutputCollector;
import org.apache.hadoop.mapred.RecordReader;
import org.apache.hadoop.mapred.Reducer;
import org.apache.hadoop.mapred.Reporter;
import org.apache.hadoop.mapred.SequenceFileInputFormat;
import org.apache.hadoop.mapred.SequenceFileOutputFormat;
import org.apache.hadoop.mapred.TextInputFormat;
import org.apache.hadoop.mapred.jobcontrol.Job;
import org.apache.hadoop.mapred.jobcontrol.JobControl;
import org.apache.hadoop.mapred.lib.IdentityMapper;

public class MRExample {
    public static class LoadPages extends MapReduceBase
        implements Mapper<LongWritable, Text, Text, Text> {

        public void map(LongWritable k, Text val,
                OutputCollector<Text, Text> oc,
                Reporter reporter) throws IOException {
            // Pull the key out
            String line = val.toString();
            int firstComma = line.indexOf(',');
            String key = line.substring(0, firstComma);
            String value = line.substring(firstComma + 1);
            Text outKey = new Text(key);
            // Prepend an index to the value so we know which file
            // it came from.
            Text outVal = new Text("1" + value);
            oc.collect(outKey, outVal);
        }
    }
    public static class LoadAndFilterUsers extends MapReduceBase
        implements Mapper<LongWritable, Text, Text, Text> {

        public void map(LongWritable k, Text val,
                OutputCollector<Text, Text> oc,
                Reporter reporter) throws IOException {
            // Pull the key out
            String line = val.toString();
            int firstComma = line.indexOf(',');
            String value = line.substring(firstComma + 1);
            int age = Integer.parseInt(value);
            if (age < 18 || age > 25) return;
            String key = line.substring(0, firstComma);
            Text outKey = new Text(key);
            // Prepend an index to the value so we know which file
            // it came from.
            Text outVal = new Text("2" + value);
            oc.collect(outKey, outVal);
        }
    }
    public static class Join extends MapReduceBase
        implements Reducer<Text, Text, Text, Text> {

        public void reduce(Text key,
                Iterator<Text> iter,
                OutputCollector<Text, Text> oc,
                Reporter reporter) throws IOException {
            // For each value, figure out which file it's from and
store it
            // accordingly.
            List<String> first = new ArrayList<String>();
            List<String> second = new ArrayList<String>();

            while (iter.hasNext()) {
                Text t = iter.next();
                String value = t.toString();
                if (value.charAt(0) == '1')
first.add(value.substring(1));
                else second.add(value.substring(1));
                reporter.setStatus("OK");
            }

            // Do the cross product and collect the values
            for (String s1 : first) {
                for (String s2 : second) {
                    String outval = key + "," + s1 + "," + s2;
                    oc.collect(null, new Text(outval));
                    reporter.setStatus("OK");
                }
            }
        }
    }
    public static class LoadJoined extends MapReduceBase
        implements Mapper<Text, Text, Text, LongWritable> {

        public void map(
                Text k,
                Text val,
                OutputCollector<Text, LongWritable> oc,
                Reporter reporter) throws IOException {
            // Find the url
            String line = val.toString();
            int firstComma = line.indexOf(',');
            int secondComma = line.indexOf(',', firstComma);
            String key = line.substring(firstComma, secondComma);
            // drop the rest of the record, I don't need it anymore,
            // just pass a 1 for the combiner/reducer to sum instead.
            Text outKey = new Text(key);
            oc.collect(outKey, new LongWritable(1L));
        }
    }
    public static class ReduceUrls extends MapReduceBase
        implements Reducer<Text, LongWritable, WritableComparable,
Writable> {

        public void reduce(
                Text key,
                Iterator<LongWritable> iter,
                OutputCollector<WritableComparable, Writable> oc,
                Reporter reporter) throws IOException {
            // Add up all the values we see

            long sum = 0;
            while (iter.hasNext()) {
                sum += iter.next().get();
                reporter.setStatus("OK");
            }

            oc.collect(key, new LongWritable(sum));
        }
    }
    public static class LoadClicks extends MapReduceBase
        implements Mapper<WritableComparable, Writable, LongWritable,
Text> {

        public void map(
                WritableComparable key,
                Writable val,
                OutputCollector<LongWritable, Text> oc,
                Reporter reporter) throws IOException {
            oc.collect((LongWritable)val, (Text)key);
        }
    }
    public static class LimitClicks extends MapReduceBase
        implements Reducer<LongWritable, Text, LongWritable, Text> {

        int count = 0;
        public void reduce(
                LongWritable key,
                Iterator<Text> iter,
                OutputCollector<LongWritable, Text> oc,
                Reporter reporter) throws IOException {

            // Only output the first 100 records
            while (count < 100 && iter.hasNext()) {
                oc.collect(key, iter.next());
                count++;
            }
        }
    }
    public static void main(String[] args) throws IOException {
        JobConf lp = new JobConf(MRExample.class);
        lp.setJobName("Load Pages");
        lp.setInputFormat(TextInputFormat.class);
        lp.setOutputKeyClass(Text.class);
        lp.setOutputValueClass(Text.class);
        lp.setMapperClass(LoadPages.class);
        FileInputFormat.addInputPath(lp, new
Path("/user/gates/pages"));
        FileOutputFormat.setOutputPath(lp,
                new Path("/user/gates/tmp/indexed_pages"));
        lp.setNumReduceTasks(0);
        Job loadPages = new Job(lp);

        JobConf lfu = new JobConf(MRExample.class);
        lfu.setJobName("Load and Filter Users");
        lfu.setInputFormat(TextInputFormat.class);
        lfu.setOutputKeyClass(Text.class);
        lfu.setOutputValueClass(Text.class);
        lfu.setMapperClass(LoadAndFilterUsers.class);
        FileInputFormat.addInputPath(lfu, new
Path("/user/gates/users"));
        FileOutputFormat.setOutputPath(lfu,
                new Path("/user/gates/tmp/filtered_users"));
        lfu.setNumReduceTasks(0);
        Job loadUsers = new Job(lfu);

        JobConf join = new JobConf(MRExample.class);
        join.setJobName("Join Users and Pages");
        join.setInputFormat(KeyValueTextInputFormat.class);
        join.setOutputKeyClass(Text.class);
        join.setOutputValueClass(Text.class);
        join.setMapperClass(IdentityMapper.class);
        join.setReducerClass(Join.class);
        FileInputFormat.addInputPath(join, new
Path("/user/gates/tmp/indexed_pages"));
        FileInputFormat.addInputPath(join, new
Path("/user/gates/tmp/filtered_users"));
        FileOutputFormat.setOutputPath(join, new
Path("/user/gates/tmp/joined"));
        join.setNumReduceTasks(50);
        Job joinJob = new Job(join);
        joinJob.addDependingJob(loadPages);
        joinJob.addDependingJob(loadUsers);

        JobConf group = new JobConf(MRExample.class);
        group.setJobName("Group URLs");
        group.setInputFormat(KeyValueTextInputFormat.class);
        group.setOutputKeyClass(Text.class);
        group.setOutputValueClass(LongWritable.class);
        group.setOutputFormat(SequenceFileOutputFormat.class);
        group.setMapperClass(LoadJoined.class);
        group.setCombinerClass(ReduceUrls.class);
        group.setReducerClass(ReduceUrls.class);
        FileInputFormat.addInputPath(group, new
Path("/user/gates/tmp/joined"));
        FileOutputFormat.setOutputPath(group, new
Path("/user/gates/tmp/grouped"));
        group.setNumReduceTasks(50);
        Job groupJob = new Job(group);
        groupJob.addDependingJob(joinJob);

        JobConf top100 = new JobConf(MRExample.class);
        top100.setJobName("Top 100 sites");
        top100.setInputFormat(SequenceFileInputFormat.class);
        top100.setOutputKeyClass(LongWritable.class);
        top100.setOutputValueClass(Text.class);
        top100.setOutputFormat(SequenceFileOutputFormat.class);
        top100.setMapperClass(LoadClicks.class);
        top100.setCombinerClass(LimitClicks.class);
        top100.setReducerClass(LimitClicks.class);
        FileInputFormat.addInputPath(top100, new
Path("/user/gates/tmp/grouped"));
        FileOutputFormat.setOutputPath(top100, new
Path("/user/gates/top100sitesforusers18to25"));
        top100.setNumReduceTasks(1);
        Job limit = new Job(top100);
        limit.addDependingJob(groupJob);

        JobControl jc = new JobControl("Find top 100 sites for users
18 to 25");
        jc.addJob(loadPages);
        jc.addJob(loadUsers);
        jc.addJob(joinJob);
        jc.addJob(groupJob);
        jc.addJob(limit);
        jc.run();
    }
}
```

170 lines of code, 4 hours to write

Pig's Data Model

- Atom
  - Int, float, string …
- Complex Types
  - Tuple
    - An ordered set of fields, (19,2)
    - Each field of any type
  - Bag
    - A collection of tuples, {(19,2), (18,1)}
    - not necessary the same type
    - duplicate allowed
  - Map
    - A set of key value pairs, [open#apache]

```
( a ,{(19,2),(18,1),(19,2)},[open#apache] )
```



expression	result
$0	a
f2	bag{(19,2),(18,1),(19,2)}
f2.$0	bag{(19),(18),(19)}
f3#'open'	'apache'
sum(f2.$1)	2 + 1 + 2

Schemas

- Schemas enable you to assign names to and declare types for fields
  - Schemas are optional
- Type declarations result in better parse-time error checking and more efficient code execution
- You can define a schema that includes the field name and field type
- Definition of a schema that includes the field name only
  - you can refer to that field using the name or by positional notation
  - the field type defaults to bytearray
- Undefined schema
  - the field is un-named and the field type defaults to bytearray
  - you can only refer to the field using positional notation

Load

- Input is assumed to be a bag (sequence of tuples)
- Assumes that every dataset is a sequence of tuples
- Specify a parsing function with "USING"
  - you can define your own function
- Specify a schema with "AS"

```
A = LOAD 'myfile.txt' USING PigStorage('\t') AS (a1,a2,a3);
```

Pig Operators

- FILTER
  - Getting rid of data
  - Arbitrary Boolean conditions
  - Regular expressions allowed

- GROUP
  - The result is a relation that includes one tuple per group. This tuple contains two fields:
    - The first field is named "group" and is the same type as the group key.
    - The second field takes the name of the original relation and is type bag.

- COGROUP
  - similar to GROUP, but with multiple relations re involved

- FOREACH
  - Takes a set of expressions and applies them to every record in the pipeline

Pig operators: JOIN vs COGRUP

- JOIN is a two-step process
  - Create groups with shared keys
  - Produce joined tuples

- COGROUP only performs the first step
  - You might do different processing on the groups

Pig Operators

Relational Operators
COGROUP
CROSS
DISTINCT
FILTER
FOREACH
GROUP
JOIN (inner/outer)
LIMIT
LOAD
ORDER
SAMPLE
SPLIT
STORE
STREAM
UNION

UDF Statements
DEFINE
REGISTER

Diagnostic Operators
DESCRIBE
DUMP
EXPLAIN
ILLUSTRATE

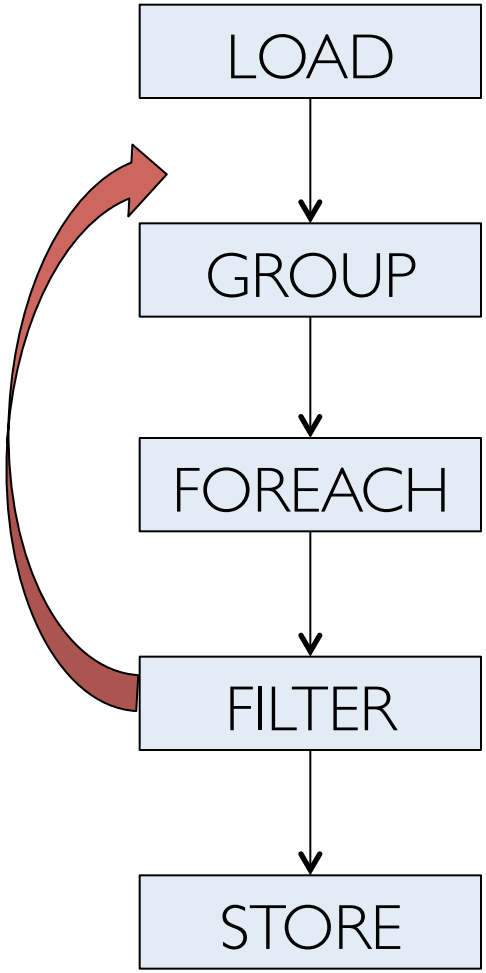
Eval Functions
AVG
CONCAT
COUNT
COUNT_STAR
DIFF
IsEmpty
MAX
MIN
SIZE
SUM
TOKENIZE

Load/Store Functions Handling
Compression
BinStorage
PigStorage
PigDump
TextLoader

Arithmetic Operators Arithmetic Operators
Comparison Operators
Null Operators
Boolean Operators
Dereference Operators
Sign Operators
Flatten Operator
Cast Operators

Example

```
A = LOAD 'traffic.dat' AS (ip, time, url);
B = GROUP A BY ip;
C = FOREACH B GENERATE group AS ip,
        COUNT(A);
D = FILTER C BY ip IS '192.168.0.1'
        OR ip IS '192.168.0.0';
STORE D INTO 'local_traffic.dat';
```



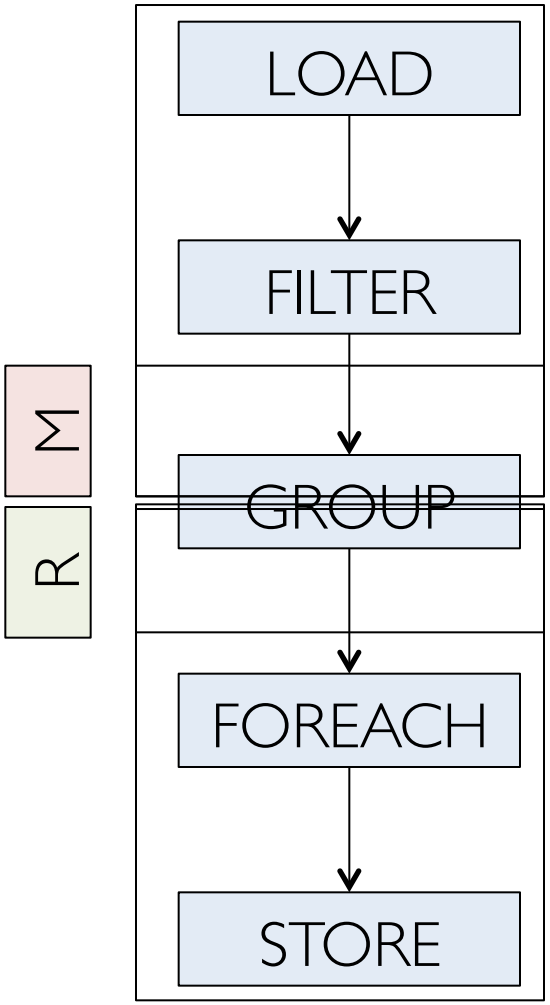
Lazy Evaluation:
no work is done until the store

Example: from DAG to MapReduce

1. (CO)GROUP requires both Map and Reduce phase:
   - create a MR job for each (CO)GROUP
2. Adds other operator where possible

Certain operator requires their own MR job

(e.g. ORDER)



Efficiency through Algebraic Function

Conceptually speaking, our (CO)GROUP command places tuples belonging to the same group into one or more nested bags.

- In many cases, the system can avoid actually materializing these bags, which is especially important when the bags are larger than one machine's main memory
- One common case is where the user applies a algebraic aggregation function over the result of a (CO)GROUP operation

An algebraic function is one that can be structured as a tree of sub-functions, with each leaf sub-function operating over a subset of the input data. (remember monoids?)

- If nodes in this tree achieve data reduction, then the system can keep the amount of data materialized in any single location small.
- Examples: COUNT, SUM, MIN, MAX, AVERAGE, VARIANCE, although some useful functions are not algebraic, e.g., MEDIAN
- Pig provides a special API for algebraic UDF

Join – advanced techniques

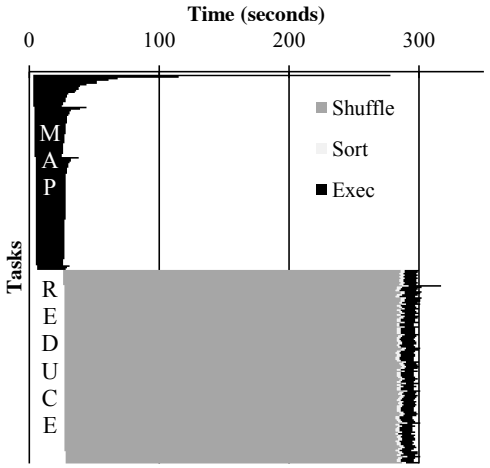
MEMORY-BACKED JOIN:

- If one of the two dataset can fit in memory, it is possible to store in memory a copy of the dataset for each mapper
- Reduce phase only to aggregate the data

```
C = JOIN big BY b1, tiny BY t1, mini BY m1 USING 'replicated';
```

SKEW JOIN:

- Struggler tasks: a small fraction of reducers (even only one) are doing the majority of the work
- load imbalances will swamp any of the parallelism gains
  - e.g.: most of the keys have few hundreds of tuples, while only one joining key correspond to millions of tuples
- Solution:
  - computes a histogram of the key space and uses this data to allocate reducers for a given key
  - splits the left input on the join predicate and streaming the right input

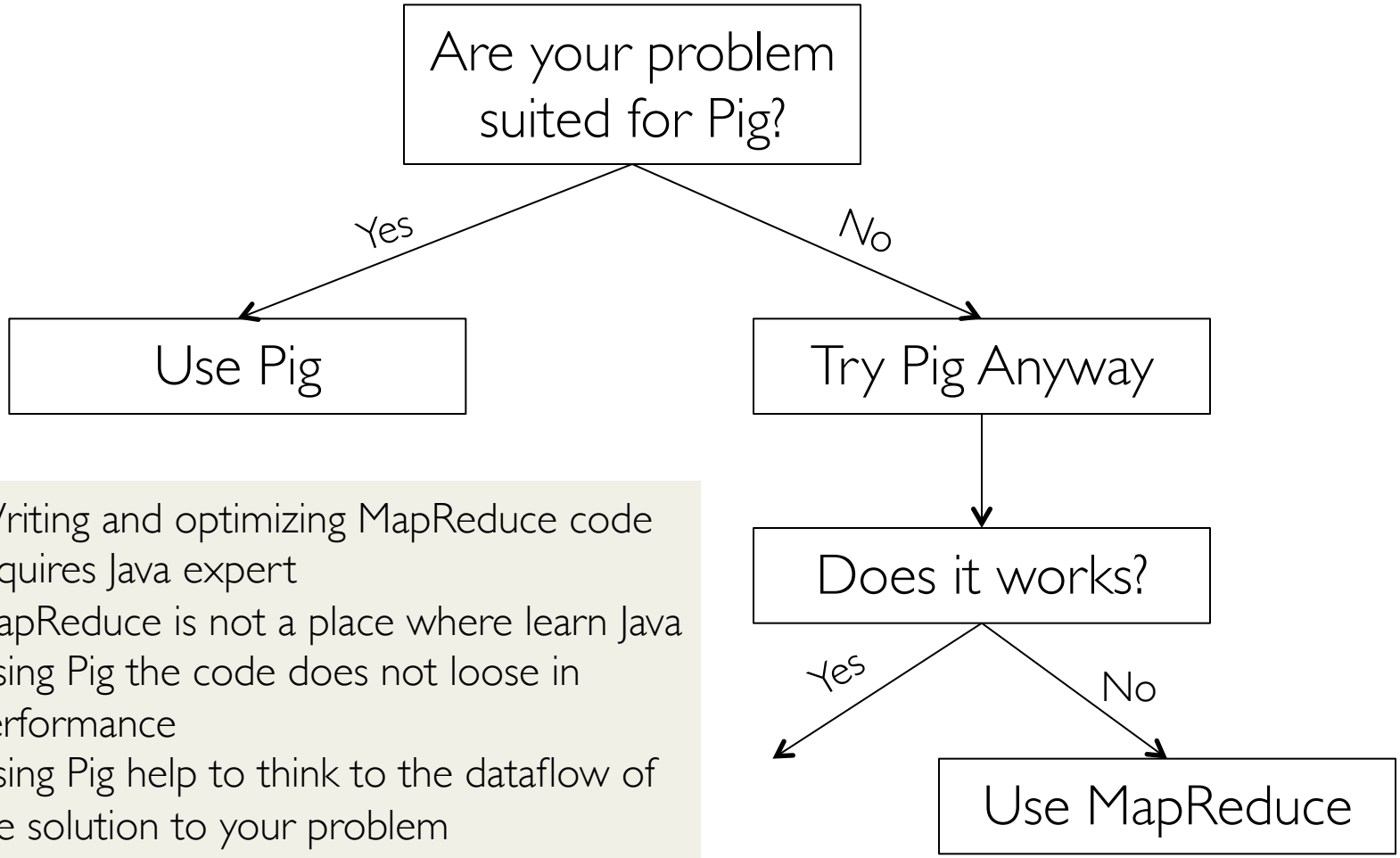



```
C = JOIN big BY b1, massive BY m1 USING 'skewed';
```

Pig vs. MapReduce

Pig	MapReduce
Pig Latin provides standard data-processing operations, such as join, filter, group by, order by, union, …	MapReduce provides the group by operation directly, the order by operation indirectly through the grouping. Filter and projection can be implemented in the map phase
Pig provides some complex implementations of standard data operations. • For example the data sent to the reducers is often skewed. • Pig has join and order by operators that will handle this case and (in some cases) rebalance the reducers.	In MapReduce, the data processing inside the map and reduce phases is opaque to the system. This means that MapReduce has no opportunity to optimize or check the user' code
Pig, can analyze a Pig Latin script and understand the data flow that the user is describing. • It can do early error checking and optimizations	MapReduce does not have a type system. This gives users the flexibility to use their own data types and serialization frameworks. • this limits the system's ability to check users' code for errors both before and during runtime.

Choosing Between Pig and MapReduce



- Writing and optimizing MapReduce code requires Java expert
- MapReduce is not a place where learn Java
- Using Pig the code does not loose in performance
- Using Pig help to think to the dataflow of the solution to your problem
  - Pig has an interactive shell…

- Grunt is Pig's interactive shell
  - Local Mode
    To run the scripts in local mode, no Hadoop or HDFS installation is required
  - Mapreduce Mode
    To run the scripts in mapreduce mode, you need access to a Hadoop cluster and HDFS installation

```
pig {-x local}
```

Hive

- Provide a SQL-like language (*HiveQL*)
- Under the covers, generates MapReduce jobs that run on Hadoop (like Pig)
- Enabling Hive requires almost no extra work by the system administrator

Hive Data Model

- Requires table definition
  - typed columns (int, float, string, boolean…)
  - allows array, struct, map…
- Hive Metastore
  - a database containing table definition and other metadata
  - Default: stored locally on the client machine in a Derby database (embedded RDBMS)
  - if needed: shared Metastore (usually MySQL).
    - but the system administrator should create it

Hive Limitations

- Not all 'standard' SQL is supported
  - Subqueries are only supported in the FROM clause
  - No correlated subqueries

- No support for UPDATE or DELETE

- All inserts overwrite the existing data. Accordingly, Hive has an explicit syntax:
  - `INSERT OVERWRITE TABLE t1`

  The only option is to append row to the table:
  - `INSERT INTO TABLE t1`

Pig vs. Hive

Pig	Hive
PIG LATIN	HiveQL (~SQL)

Schema defined dynamically while importing data
 	
`text = LOAD 'PATH_TO_FILE' AS (freq:INT, word:CHARARRAY);`
	Schema defined before importing data, e.g.:
	
	`CREATE TABLE text (freq INT, word STRING)`
	`ROW FORMAT DELIMITED`
	`FIELD TERMINATED BY '\t'`
	`STORE AS TEXTFILE;`
	`---`
	`LOAD DATA INPATH 'PATH_TO_FILE'`
	`INTO TABLE text`

Mahout

MAPREDUCE AND MACHINE LEARNING

Mahout

In theory, Mahout is a project open to implementations of all kinds of machine learning techniques
In practice, it's a project that focuses on three key areas of machine learning at the moment. These are recommender engines (collaborative filtering), clustering, and classification

Recommendation
- For a given set of input, make a recommendation
- Rank the best out of many possibilities

Clustering
- Finding similar groups (based on a definition of similarity)
- Algorithms do not require training
- Stopping condition: iterate until close enough

Classification
- identifying to which of a set of (predefined)categories a new observation belongs
- Algorithms do require training

Mahout News

25 April 2014 - Goodbye MapReduce

The Mahout community decided to move its codebase onto modern data processing systems that offer a richer programming model and more efficient execution than Hadoop MapReduce. **Mahout will therefore reject new MapReduce algorithm implementations from now on**. We will however keep our widely used MapReduce algorithms in the codebase and maintain them.

We are building our future implementations on top of a DSL for linear algebraic operations which has been developed over the last months. Programs written in this DSL are automatically optimized and executed in parallel on Apache Spark.

Furthermore, there is an experimental contribution undergoing which aims to integrate the h20 platform into Mahout.

Scala & Spark Bindings for Mahout:

- Scala DSL and algebraic optimizer
  - The main idea is that a scientist writing algebraic expressions cannot care less of distributed operation plans and works entirely on the logical level just like he or she would do with R.
  - Another idea is decoupling logical expression from distributed back-end. As more back-ends are added, this implies "write once, run everywhere".

Mahout Algorithm (1)

	Single Machine	MapReduce	Spark
Collaborative Filtering *with CLI Drivers*			
User-Based Collaborative Filtering	x		x
Item-Based Collaborative Filtering	x	x	x
Matrix Factorization with ALS	x	x	
Matrix Factorization with ALS on Implicit Feedback	x	x	
Weighted Matrix Factorization, SVD++	x		
Classification *with CLI Drivers*			
Logistic Regression - trained via SGD	x		
Naive Bayes / Complementary Naive Bayes		x	*in development*
Random Forest		x	
Hidden Markov Models	x		
Multilayer Perceptron	x		
Clustering *with CLI Drivers*			
Canopy Clustering	*deprecated*	*deprecated*	
k-Means Clustering	x	x	
Fuzzy k-Means	x	x	
Streaming k-Means	x	x	
Spectral Clustering		x	

http://mahout.apache.org/users/basics/algorithms.html

Mahout Algorithm (2)

	Single Machine	MapReduce	Spark
Dimensionality Reduction *with CLI Drivers - note: most scala-based dimensionality reduction algorithms are available through the* Mahout Math-Scala Core Library for all engines			
Singular Value Decomposition	x	x	
Lanczos Algorithm	x	x	
Stochastic SVD	x	x	
PCA (via Stochastic SVD)	x	x	
QR Decomposition	x	x	
Topic Models			
Latent Dirichlet Allocation	x	x	
Miscellaneous			
RowSimilarityJob		x	x
ConcatMatrices		x	
Collocations		x	
Sparse TF-IDF Vectors from Text		x	
XML Parsing		x	
Email Archive Parsing		x	
Lucene Integration		x	
Evolutionary Processes	x		

http://mahout.apache.org/users/basics/algorithms.html

Mahout Algorithm (3)

	Single Machine	MapReduce	Spark
Collaborative Filtering *with CLI Drivers*			
User-Based Collaborative Filtering	x		x
Item-Based Collaborative Filtering	x	x	x
Matrix Factorization with ALS	x	x	
Matrix Factorization with ALS on Implicit Feedback	x	x	
Weighted Matrix Factorization, SVD++	x		
Classification *with CLI Drivers*			
Logistic Regression - trained via SGD	x		
Naive Bayes / Complementary Naive Bayes		x	*in development*
Random Forest		x	
Hidden Markov Models	x		
Multilayer Perceptron	x		
Clustering *with CLI Drivers*			
Canopy Clustering	*deprecated*	*deprecated*	
k-Means Clustering	x	x	
Fuzzy k-Means	x	x	
Streaming k-Means	x	x	
Spectral Clustering		x	

http://mahout.apache.org/users/basics/algorithms.html

Giraph

MAPREDUCE AND GRAPHS

Girpah Motivations

- Representing graphs in MapReduce is complex (and "unnatural"):
  e.g: <key: vertex_id ; value:{weight,…, [list of neighbor] }>

- Computation is not efficient:
  - Each vertex depends on its neighbors, recursively
  - Recursive problems are nicely solved iteratively
  - In MapReduce iterations means chains of MR jobs
    - must store graph state in each stage, too much communication between stages

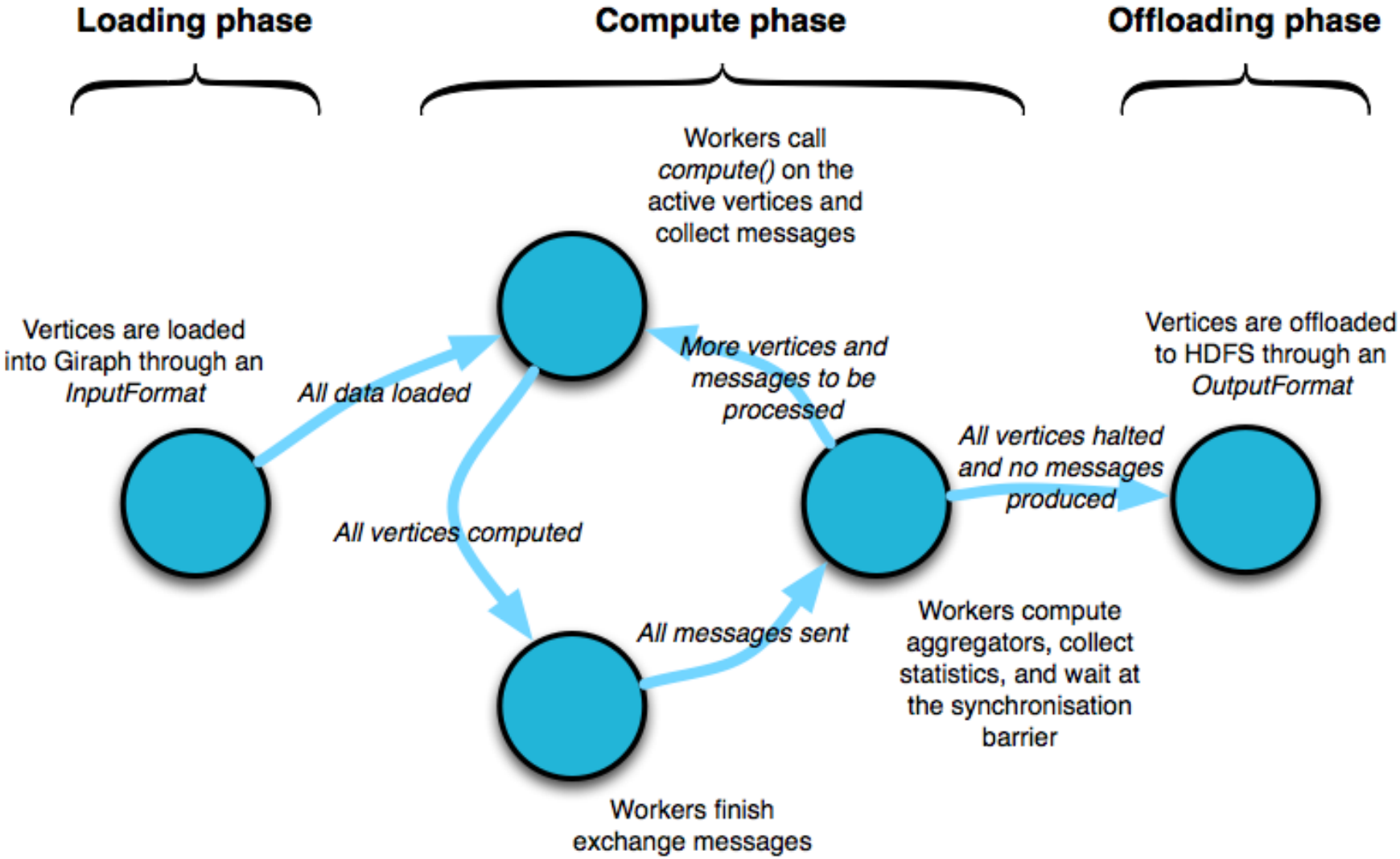
Solution

- Pregel (Google 2010)
- Giraph (Apache open-source equivalent)

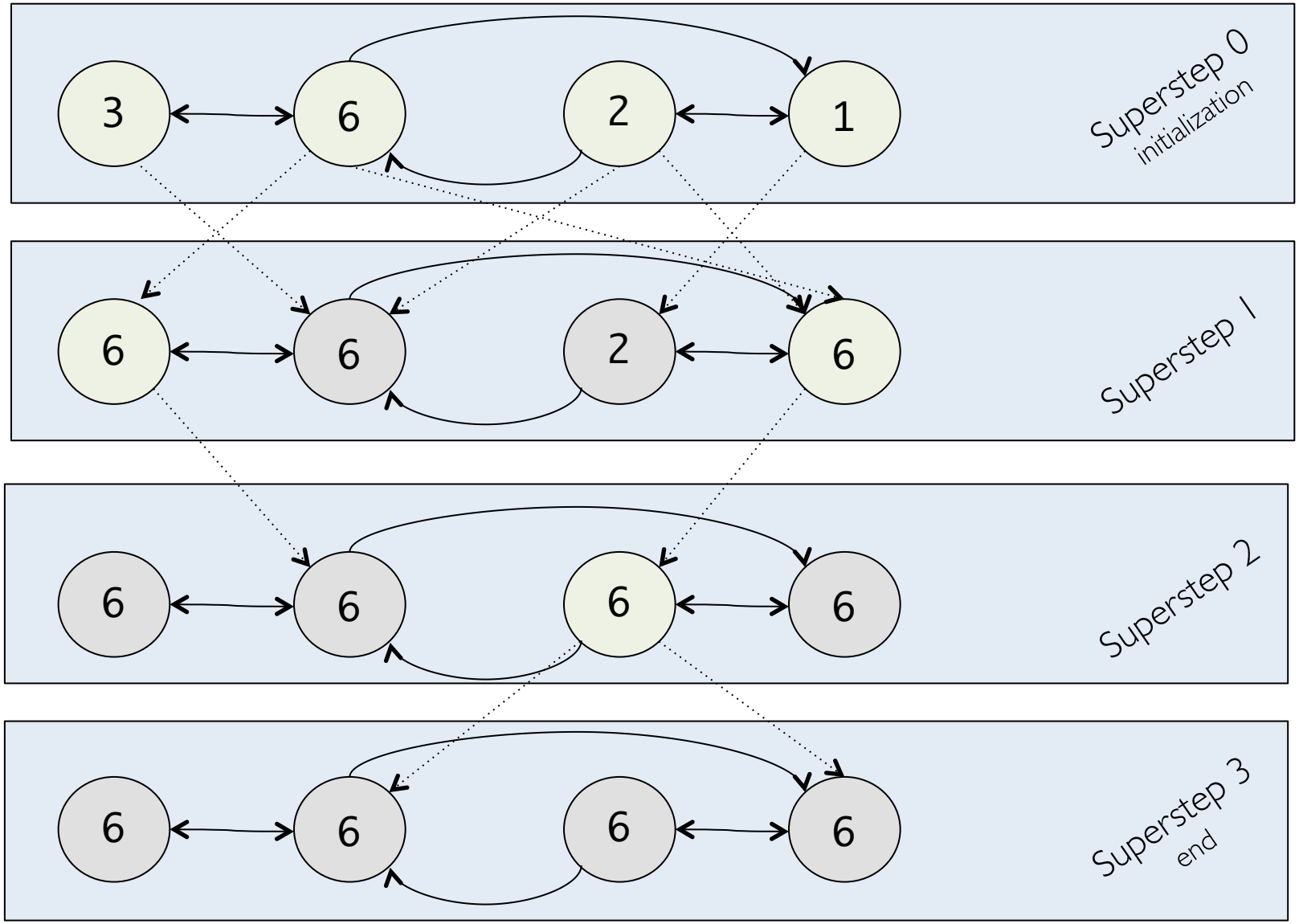
Solution: BSP (Bulk Synchronous Parallel)

- Computations consist of a sequence of iterations, called supersteps.
- During a superstep the framework invokes a user-defined function for each vertex, conceptually in parallel.
- The function specifies behavior at a single vertex V and a single superstep S.
- It can read messages sent to V in superstep S − 1, send messages to other vertices that will be received at superstep S + 1, and modify the state of V and its outgoing edges.
- Messages are typically sent along outgoing edges, but a message may be sent to any vertex whose identifier is known.
- The synchronicity of this model makes it easier to reason about program semantics when implementing algorithms, and ensures that Giraph programs are inherently free of dead-locks and data races (common in asynchronous systems).

Giraph Job


Loading phase
Compute phase
Offloading phase
Workers call compute() on the active vertices and collect messages
Vertices are loaded into Giraph through an InputFormat
All data loaded
More vertices and messages to be processed
Vertices are offloaded to HDFS through an OutputFormat
All vertices halted and no messages produced
All vertices computed
All messages sent
Workers compute aggregators, collect statistics, and wait at the synchronisation barrier
Workers finish exchange messages

Example: finding maximum value



Data Locality

- The assignment of vertices to worker machines is the main place where distribution is not transparent in Giraph.
- Some applications work well with the default assignment, but some benefit from defining custom assignment functions to better exploit locality inherent in the graph.
  - custom partitioning function (similar to the *customPartitioner* in Hadoop)
- E.g., a typical heuristic employed for the Web graph is to co-locate vertices representing pages of the same site

Example: shortest path

```
public void compute(Iterable<DoubleWritable> messages) {
    double minDist = Double.MAX_VALUE;
    for (DoubleWritable message : messages) {
      minDist = Math.min(minDist, message.get());
    }
    if (minDist < getValue().get()) {
      setValue(new DoubleWritable(minDist));
      for (Edge<LongWritable, FloatWritable> edge : getEdges()) {
        double distance = minDist + edge.getValue().get();
        sendMessage(edge.getTargetVertexId(), new DoubleWritable(distance));
      }
    }
    voteToHalt();
}
```

Machine Learning library for Giraph

- Collaborative Filtering
  - Alternating Least Squares (ALS)
  - Bayesian Personalized Ranking (BPR) –beta-
  - Collaborative Less-is-More Filtering (CLiMF) –beta-
  - Singular Value Decomposition (SVD++)
  - Stochastic Gradient Descent (SGD)
- Graph Analytics
  - Graph partitioning
  - Similarity
  - SybilRank
- Clustering
  - Kmeans

http://grafos.ml/#Okapi

References

- Olston, Christopher, et al. "Pig latin: a not-so-foreign language for data processing." Proceedings of the 2008 ACM SIGMOD international conference on Management of data. ACM, 2008.
- Thusoo, Ashish, et al. "Hive: a warehousing solution over a map-reduce framework." Proceedings of the VLDB Endowment 2.2 (2009): 1626-1629.
- Malewicz, Grzegorz, et al. "Pregel: a system for large-scale graph processing." Proceedings of the 2010 ACM SIGMOD International Conference on Management of data. ACM, 2010.
- Hadoop, Module 2: The Hadoop Distributed File System, Yahoo! http://developer.yahoo.com/hadoop/tutorial/module2.html
- Pig Tutorial, http://pig.apache.org/docs/r0.7.0/index.html
- Alan Gates: Programming Pig. O'Reilly Media, Inc. 2011 http://ofps.oreilly.com/titles/9781449302641/index.html
- Introduction to Pig, Cloudera 2009 http://blog.cloudera.com/wp-content/uploads/2010/01/IntroToPig.pdf

References

- http://pig.apache.org/
- https://hive.apache.org/
- http://giraph.apache.org/
- Lam, Chuck. Hadoop in action. Manning Publications Co., 2010.
- Rajaraman, Anand, and Jeffrey David Ullman. Mining of massive datasets. Cambridge University Press, 2011.
- http://hadoop.apache.org/
- https://www.coursera.org/course/datasci
- https://www.coursera.org/course/mmds
- https://www.coursera.org/course/bigdata