Can we make MapReduce easier?

Apache Pig

• Why?
  – Make it easy to use MapReduce via scripting instead of Java
  – Make it easy to use multiple MapReduce stages
  – Built-in common operations for join, group, filter, etc.

• How to use?
  – Use Grunt – the pig shell
  – Submit a script directly to pig
  – Use the PigServer Java class
  – PigPen – Eclipse plugin
  – Pig compiles to several Hadoop MapReduce jobs

Pig: Loading Data

Load/store relations in the following formats:
  • PigStorage: field-delimited text
  • BinStorage: binary files
  • BinaryStorage: single-field tuples with a value of bytearray
  • TextLoader: plain-text
  • PigDump: stores using toString() on tuples, one per line

Example

```
log = LOAD 'test.log' AS (user, timestamp, query);
gropl = GROUP log BY user;
undl = FOREACH gropl GENERATE group, COUNT(log);
filtrl = FILTER undl BY cnt > 50;
srtl = ORDER filtrl BY cnt;
STORE srtl INTO 'output';
```

• Each statement defines a new dataset
  – Datasets can be given aliases to be used later
• FOREACH iterates over the members of a "bag"
  – Input is gropl, list of log entries grouped by user
  – Output is group, COUNT(log): list of (user, count)
• FILTER applies conditional filtering
• ORDER applies sorting
See pig.apache.org for full documentation

MapReduce isn’t always the answer
- MapReduce works well for certain problems
  - Framework provides
    - Automatic parallelization
    - Automatic job distribution
- For others:
  - May require many iterations
  - Data locality usually not preserved between Map and Reduce
  - Lots of communication between map and reduce workers

Bulk Synchronous Parallel (BSP)
Computing model for parallel computation
- Series of supersteps
  1. Concurrent computation
  2. Communication
  3. Barrier synchronization

Series of supersteps
- Processes (workers) are randomly assigned to processors
- Each process uses only local data
- Each computation is asynchronous of other concurrent computation
- Computation time may vary

- Messaging is restricted to the end of a computation superstep
- Each worker sends a message to 0 or more workers
- These messages are inputs for the next superstep
### Bulk Synchronous Parallel (BSP)

- **Series of supersteps**
  1. Concurrent computation
  2. Communication
  3. Barrier synchronization

  - The next superstep does not begin until all messages have been received.
  - Barriers ensure no deadlock; no circular dependency can be created.
  - Provide an opportunity to checkpoint results for fault tolerance.
  - If failure, restart computation from last superstep.

### BSP Implementation: Apache Hama

- **Hama**: BSP framework on top of HDFS
  - Provides automatic parallelization & distribution
  - Uses Hadoop RPC
  - Data is serialized with Google Protocol Buffers
  - Zookeeper for coordination (Apache version of Google’s Chubby)
  - Handles notifications for Barrier Sync

- **Good for applications with data locality**
  - Matrices and graphs
  - Algorithms that require a lot of iterations

### Hama programming (high-level)

- **Pre-processing**
  - Define the number of peers for the job
  - Split initial inputs for each of the peers to run their supersteps
  - Framework assigns a unique ID to each worker (peer)

- **Superstep**: the worker function is a superstep
  - `getCurrentMessage()` — input messages from previous superstep
  - Compute — your code
  - `send(peer, msg)` — send messages to a peer
  - `sync()` — synchronize with other peers (barrier)

- **File I/O**
  - Key/value model used by Hadoop MapReduce & HBase
  - `readNext(key, value)`
  - `write(key, value)`

### For more information

- Architecture, examples, API
- Take a look at:
  - Apache Hama project page
  - http://hama.apache.org
  - Hama BSP tutorial
  - Apache Hama Programming document

### Graph computing

- **Graphs are common in computing**
  - Social links
    - Friends
    - Academic citations
    - Music
    - Movies
  - Web pages
  - Network connectivity
  - Roads
  - Disease outbreaks
Processing graphs on a large scale is hard

- Computation with graphs
  - Poor locality of memory access
  - Little work per vertex
- Distribution across machines
  - Communication complexity
  - Failure concerns
- Solutions
  - Application-specific, custom solutions
  - MapReduce or databases
  - Single-computer libraries: limits scale
  - Parallel libraries: do not address fault tolerance

But require many iterations (and a lot of data movement)

- Single-computer libraries: limits scale
- Parallel libraries: do not address fault tolerance
- BSP: close but too general

Pregel: a vertex-centric BSP

Input: directed graph
- A vertex is an object
  - Each vertex uniquely identified with a name
  - Each vertex has a modifiable value
  - Directed edges: links to other objects
  - Associated with source vertex
  - Each edge has a modifiable value
  - Each edge has a target vertex identifier

Pregel: computation

Computation: series of supersteps
- Same user-defined function runs on each vertex
- Receives messages sent from the previous superstep
- May modify the state of the vertex or its outgoing edges
- Sends messages that will be received in the next superstep
  - Typically to outgoing edges
  - But can be sent to any known vertex
- May modify the graph topology
- Each superstep ends with a barrier (synchronization point)

Pregel: termination

Pregel terminates when every vertex votes to halt
- Initially, every vertex is in an active state
  - Active vertices compute during a superstep
  - Each vertex may choose to deactivate itself by voting to halt
    - The vertex has no more work to do
    - Will not be executed by Pregel
    - UNLESS the vertex receives a message
    - Then it is reactivated
    - Will stay active until it votes to halt again
  - Algorithm terminates when all vertices are inactive and there are no messages in transit

Pregel: output

- Output is the set of values output by the vertices
  - Often a directed graph
    - May be non-isomorphic to original since edges & vertices can be added or deleted
  - Or may be summary data

Examples of graph computations

- Shortest path to a node
  - Each iteration, a node sends the shortest distance received to all neighbors
- Cluster identification
  - Each iteration: get info about clusters from neighbors
    - Add myself
    - Pass useful clusters to neighbors (e.g., within a certain depth or size)
    - May combine related vertices
  - Output is a smaller set of disconnected vertices representing clusters of interest
- Graph mining
  - Traverse a graph and accumulate global statistics
- Page rank
  - Each iteration: update web page ranks based on messages from incoming links.
Simple example: find the maximum value

- Each vertex contains a value
- In the first superstep:
  - A vertex sends its value to its neighbors
- In each successive superstep:
  - If a vertex learned of a larger value from its incoming messages, it sends it to its neighbors
  - Otherwise, it votes to halt
- Eventually, all vertices get the largest value
- When no vertices change in a superstep, the algorithm terminates

Semi-pseudocode:

```java
class MaxValueVertex
    : public Vertex<int, void, int>
{
    void Compute(MessageIterator *msgs) {
        int maxv = GetValue();
        for (; !msgs->Done(); msgs->Next())
            maxv = max(msgs.Value(), maxv);
        if (maxv > GetValue()) || (step == 0)) {
            *MutableValue() = maxv;
            OutEdgeIterator out = GetOutEdgeIterator();
            for (; !out.Done(); out.Next())
                sendMessageTo(out.Target(), maxv);
        } else
            VoteToHalt();
    }
};
```

Locality

- Vertices and edges remain on the machine that does the computation
- To run the same algorithm in MapReduce
  - Requires chaining multiple MapReduce operations
  - Entire graph state must be passed from Map to Reduce
    ... and again as input to the next Map
**Pregel API: Basic operations**

- A user subclasses a Vertex class
- Methods
  - `Compute(MessageIterator*)`: Executed per active vertex in each superstep
  - `MessageIterator`: Identifies incoming messages from previous supersteps
  - `GetValue()`: Get the current value of the vertex
  - `MutableValue()`: Set the value of the vertex
  - `GetOutEdgeIterator()`: Get a list of outgoing edges
  - `Target()`: Identify target vertex on an edge
  - `GetValue()`: Get the value of the edge
  - `MutableValue()`: Set the value of the edge
  - `SendMessageTo()`: Send a message to a vertex

**Combines**

- Each message has an overhead — let's reduce # of messages
- Many vertices are processed per worker (multi-threaded)
- Pregel can combine messages targeted to one vertex into one message
- Combiners are application specific
  - Programmer subclasses a Combiner class and overrides Combine() method
  - No guarantee on which messages may be combined

**Aggregators**

- Handle global data
- A vertex can provide a value to an aggregator during a superstep
  - Aggregator combines received values to one value
  - Value is available to all vertices in the next superstep
- User subclasses an Aggregator class
- Examples
  - Keep track of total edges in a graph
  - Generate histograms of graph statistics
  - Global flags: execute until some global condition is satisfied
  - Election: find the minimum or maximum vertex

**Topology modification**

- Examples
  - If we're computing a spanning tree: remove unneeded edges
  - If we're clustering: combine vertices into one vertex
  - Add/remove edges/vertices
  - Modifications visible in the next superstep

**Execution environment**

- Many copies of the program are started on a cluster of machines
  - One copy becomes the master
    - Will not be assigned a portion of the graph
    - Responsible for coordination
  - Cluster's name server is `chubby`
    - Master registers itself with the name service
    - Workers contact the name service to find the master
Partition assignment

- **Master**
  - Determines # partitions in graph
  - One or more partitions assigned to each worker
  - Partition = set of vertices
  - Default: for $N$ partitions
    \[ \text{hash(vertex ID)} \mod N \Rightarrow \text{worker} \]
  - May deviate: e.g., place vertices representing the same web site in one partition
  - More than 1 partition per worker: improves load balancing

- **Worker**
  - Responsible for its section(s) of the graph
  - Each worker knows the vertex assignments of other workers

Input assignment

- **Master** assigns parts of the input to each worker
  - Data usually sits in GFS or Bigtable
- **Input** = set of records
  - Record = vertex data and edges
  - Assignment based on file boundaries

- **Worker**
  - Reads input
  - If it belongs to any of the vertices it manages, messages sent locally
  - Else worker sends messages to remote workers
  - After data is loaded, all vertices are active

Computation

- **Master** tells each worker to perform a superstep
- **Worker**:
  - Iterates through vertices (one thread per partition)
  - Calls `Compute()` method for each active vertex
  - Delivers messages from the previous superstep
  - Outgoing messages
    - Sent asynchronously
    - Delivered before the end of the superstep
  - When done
    - Worker tells master how many vertices will be active in the next superstep
  - Computation done when no more active vertices in the cluster
    - Master may instruct workers to save their portion of the graph

Handling failure

- **Checkpointing**
  - Controlled by master … every $N$ supersteps
  - Master asks a worker to checkpoint at the start of a superstep
  - Save state of partitions to persistent storage
    - Vertex values
    - Edge values
    - Incoming messages
    - Master is responsible for saving aggregator values

- **Failure detection**: master sends `ping` messages to workers
  - If worker does not receive a ping within a time period
    - Worker terminates
  - If the master does not hear from a worker
    - Master marks worker as failed

- **Restart**: when failure is detected
  - Master reassigns partitions to the current set of workers
  - All workers reload partition state from most recent checkpoint

Pregel outside of Google

**Apache Giraph**
- Initially created at Yahoo
- Used at LinkedIn & Facebook to analyze the social graphs of users
- Facebook is the main contributor to Giraph
- Runs under Hadoop MapReduce framework
- Runs as a Map-only job
- Adds fault-tolerance to the master by using `Zookeeper` for coordination
- Uses Java instead of C++

**Chubby**

Conclusion

- **Vertex-centric approach to BSP**
  - Computation = set of supersteps
    - Computer(s) called on each vertex per superstep
    - Communication between supersteps: barrier synchronization
  - Hides distribution from the programmer
    - Framework creates lots of workers
    - Distributes partitions among workers
    - Distributes input
    - Handles message sending, receipt, and synchronization
    - A programmer just has to think from the viewpoint of a vertex

- **Checkpoint-based fault tolerance**
Spark: Generalizing MapReduce

Apache Spark Goals

- Generalize MapReduce
  - Similar shard-and-gather approach to MapReduce
  - Create multi-step pipelines based on directed acyclic graphs (DAGs) of data flows
- Create a general functional programming model
  - Transformation and action
  - In Map-Reduce, transformation = map, action = reduce
  - Support operations beyond map and reduce
- Add fast data sharing
  - In-memory caching
  - Different computation phases can use the same data if needed
- And generic data storage interfaces
  - Storage agnostic: use HDFS, Cassandra database, whatever
  - Resilient Distributed Data (RDD) sets
    - An RDD is a chunk of data that gets processed – a large collection of stuff

Spark Design: RDDs

RDD: Resilient Distributed Datasets
- Table that can be shared across many servers
- Holds any type of data
- Immutable: you can process the RDD to create a new RDD but not modify the original

Two operations on RDDs
1. Transformations: transformation function takes RDD as input & creates a new RDD
   - Examples: map, filter, groupByKey, reduceByKey, aggregateByKey, ...
2. Actions: evaluates an RDD and creates a value:
   - Examples: reduce, collect, count, first, take, countByKey, ...

- Shared variables
  - Broadcast Variables: define read-only data that will be cached on each system
  - Accumulators: used for counters (e.g., in MapReduce) or sums
    - Only the driver program can read the value of the accumulator.

High-level view

- Job = bunch of transformations & actions on RDDs
- Cluster manager breaks the job into tasks
- Sends tasks to worker nodes where the data lives
Spark Transformations

<table>
<thead>
<tr>
<th>Transformation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>map(func)</td>
<td>Pass each element through a function func</td>
</tr>
<tr>
<td>filter(func)</td>
<td>Select elements of the source on which func returns true</td>
</tr>
<tr>
<td>flatmap(func)</td>
<td>Each input item can be mapped to 0 or more output items</td>
</tr>
<tr>
<td>sample(withReplacement, fraction, seed)</td>
<td>Sample a fraction fraction of the data, with or without replacement, using a given random number generator seed</td>
</tr>
<tr>
<td>union(otherDataset)</td>
<td>Union of the elements in the source data set and otherDataset</td>
</tr>
<tr>
<td>distinct(numtasks)</td>
<td>The distinct elements of the source dataset</td>
</tr>
</tbody>
</table>

Properties of RDDs

- **Immutable**
  - You cannot change it – only create new RDDs
  - The framework will eventually collect unused RDDs
- **Typed (table)**
  - Contain some parsable data structure – e.g., key-value set
- **Created from**
  - Either original source data or computed from one or more other RDDs
- **Partitioned**
  - Parts of an RDD may go to different servers
  - Function can be defined for computing each split
  - Default partitioning function = hash(key) mod server_count
- **Ordered (optional)**
  - Elements in an RDD can be sorted

Operations on RDDs

- **Two types of operations on RDDs**
  - **Transformations**: create new RDDs
    - Lazy: computed when needed, not immediately
    - Transformed RDD is computed when an action is run on it
    - Work backwards:
      - What RDDs do you need to apply to get an action?
      - What RDDs do you need to apply to get the input to this RDD?
    - RDD can be persisted into memory or disk storage
  - **Actions**: create result values
    - Finalizing operations
      - Reduce, count, grab samples, write to file

Data & RDDs

- **Data organized into RDDs**
  - One RDD may be partitioned across lots of computers
- **How are RDDs created?**
  1. Create from any file stored in HDFS or other storage supported in Hadoop (Amazon S3, HDFS, HBase, Cassandra, etc.)
  2. Created externally (e.g., event stream, text files, database)
  3. Example:
     - Query a database & make query the results an RDD
     - Any Hadoop InputFormat, such as a list of files or a directory
  2. Streaming sources (via Spark Streaming)
    - Fault-tolerant stream with a sliding window
  3. Output of a Spark transformation function
    - Example, filter out data, select key-value pairs

Spark Transformations

<table>
<thead>
<tr>
<th>Transformation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>groupByKey(numtasks)</td>
<td>When called on a dataset of (K, V) pairs, returns a dataset of (K, seq(V))</td>
</tr>
<tr>
<td>reduceByKey(func, numtasks)</td>
<td>Aggregate the values for each key using the given reduce function</td>
</tr>
<tr>
<td>sortByKey(ascending), numtasks</td>
<td>Sort keys in ascending or descending order</td>
</tr>
<tr>
<td>join(otherDataset, numtasks)</td>
<td>Combines two datasets, (K, V) and (K, W) into K, (V, W))</td>
</tr>
<tr>
<td>cogroup(otherDataset, numtasks)</td>
<td>Given (K, V) and (K, W), returns (K, Seq(V), Seq(W))</td>
</tr>
<tr>
<td>cartesian(otherDataset)</td>
<td>For two datasets of types T and U, returns a dataset of (T, U) pairs</td>
</tr>
</tbody>
</table>
Spark Actions

<table>
<thead>
<tr>
<th>Action</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>reduce(func)</code></td>
<td>Aggregate elements of the dataset using <code>func</code>.</td>
</tr>
<tr>
<td><code>collect()</code></td>
<td>Return all elements of the dataset as an array.</td>
</tr>
<tr>
<td><code>count()</code></td>
<td>Return the number of elements in the dataset.</td>
</tr>
<tr>
<td><code>first()</code></td>
<td>Return the first element of the dataset.</td>
</tr>
<tr>
<td><code>take(n)</code></td>
<td>Return an array with the first <code>n</code> elements of the dataset.</td>
</tr>
<tr>
<td><code>takeSample()</code></td>
<td>Return an array with a random sample of <code>num</code> elements of the dataset.</td>
</tr>
<tr>
<td><code>saveAsTextFile(path)</code></td>
<td>Write dataset elements as a text file</td>
</tr>
<tr>
<td><code>saveAsSequenceFile(path)</code></td>
<td>Write dataset elements as a Hadoop SequenceFile</td>
</tr>
<tr>
<td><code>countByKey()</code></td>
<td>For (K, V) RDDs, return a map of (K, Int) pairs with the count of each key</td>
</tr>
<tr>
<td><code>foreach(func)</code></td>
<td>Run <code>func</code> on each element of the dataset</td>
</tr>
</tbody>
</table>

Data Storage

- Spark does not care how source data is stored
  - RDD connector determines that
  - E.g., read RDDs from tables in a Cassandra DB, write new RDDs to HBase tables
- RDD Fault tolerance
  - RDDs track the sequence of transformations used to create them
  - Enables recomputing of lost data
  - Go back to the previous RDD and apply the transforms again

Example: processing logs

- Transform (creates new RDDs)
  - Extract error message from a log
  - Parse out the source of error
- Actions: count mysql & php errors

```scala
// base RDD
val lines = sc.textFile("hdfs://...")
// transformed RDDs
val errors = lines.filter(_.startsWith("ERROR"))
val messages = errors.map(_.split("\t").map(r => r(1)))
messages.cache()

// action 1
messages.filter(_.contains("mysql")).count()
// action 2
messages.filter(_.contains("php")).count()
```

Spark Ecosystem

- **Spark Streaming**: process real-time streaming data
  - Micro-batch style of processing
  - Uses DStream: series of RDDs
- **Spark SQL**: access Spark data over JDBC API
  - Use SQL-like queries on Spark data
- **Spark Mlib**: machine learning library
  - Utilities for classification, regression, clustering, filtering, ...
- **Spark GraphX**: graph computation
  - Adds Pregel API to Spark
  - Extends RDD by introducing a directed multi-graph with properties attached to each vertex & edge.
  - Set of operators to create subgraphs, join vertices, aggregate messages, ...

Spark Streaming

- Map-Reduce & Pregel expect static data
- Spark Streaming enables processing live data streams
  - Same programming operations
  - Input data is chunked into batches
  - Programmer specifies time interval
Spark Streaming: DStreams

- **Discretized Stream = DStream**
  - Continuous stream of data (from source or a transformation)
  - Appears as a continuous series of RDDs, each for a time interval
  
  - Each operation on a DStream translates to operations on the RDDs
  
  - Join operations allow combining multiple streams

Spark Summary

- **Supports streaming**
  - Handle continuous data streams via Spark Streaming

- **Fast**
  - Often up to 10x faster on disk and 100x faster in memory than MapReduce
  - General execution graph model
    - No need to have "useless" phases just to fit into the model
  - In-memory storage for RDDs

- **Fault tolerant: RDDs can be regenerated**
  - You know what the input data set was, what transformations were applied to it, and what output it creates