HAMAKE: A Dataflow Approach to Data Processing in Hadoop

Vadim Zaliva
Codeminders
Email: lord@crocodile.org

Vladimir Orlov
Codeminders
Email: vorl@codeminders.com

***Abstract*—Most non-trivial data processing scenarios using Hadoop typically involve launching more than one MapReduce job. Usually, such processing is data-driven with the data funneled through a sequence of jobs. The processing model could be expressed in terms of dataflow programming, represented as a directed graph with datasets as vertices. Using *fuzzy timestamps* as a way to detect which dataset needs to be updated, we can calculate a sequence in which Hadoop jobs should be launched to bring all datasets up to date. Incremental data processing and parallel job execution fit well into this approach.**

These ideas inspired the creation of the hamake utility. We attempted to emphasize data allowing the developer to formulate the problem as a data flow, in contrast to the workflow approach commonly used. Hamake language uses just two data flow operators: *fold* and *foreach*, providing a clear processing model similar to MapReduce, but on a dataset level.

I. Motivation and Background

Hadoop[1] is a popular open-source implementation of MapReduce, a data processing model introduced by Google[2].

Hadoop is typically used to process large amounts of data through a series of relatively simple operations. Usually Hadoop jobs are I/O-bound [3], [4], and execution of even trivial operations on a large dataset could take significant system resources. This makes incremental processing especially important. Our initial inspiration was the Unix *make* utility. While applying some of the ideas implemented by *make* to Hadoop, we took the opportunity to generalize the processing model in terms of dataflow programming.

Hamake was developed in late 2008 to address the problem of incremental processing of large data sets in a collaborative filtering project.

We've striven to create an easy to use utility that developers can start using right away without complex installation or extensive learning curve.

Hamake is open source and is distributed under Apache License v2.0. The project is hosted at Google Code at the following URL: http://code.google.com/p/hamake/.

II. Processing Model

Hamake operates on *files* residing on a local or distributed file system accessible from the Hadoop job. Each file has a timestamp reflecting the date and time of its last modification. A file system directory or folder is also a file with its own timestamp. A *Data Transformation Rule (DTR)* defines an operation which takes files as input and produces other files as output.

If file *A* is listed as input of a DTR, and file *B* is listed as output of the same DTR, it is said that *"B depends on A."* **Hamake** uses file time stamps for dependency up-to-date checks. DTR output is said to be *up to date* if the minimum time stamp on all outputs is greater than or equal to the maximum timestamp on all inputs. For the sake of convenience, a user could arrange groups of files and folders into a *fileset* which could later be referenced as the DTR's input or output.

Hamake uses *fuzzy timestamps*[1] which can be compared, allowing for a slight margin of error. The "fuzziness" is controlled by a tolerance of σ. Timestamp a is considered to be older than timestamp b if $(b - a) > \sigma$. Setting $\sigma = 0$ gives us a non-fuzzy, strict timestamp comparison.

Hamake attempts to ensure that all outputs from a DTR are up to date[2] To do so, it builds a *dependency graph* with DTRs as edges and individual files or filesets as vertices. Below, we show that this graph is guaranteed to be a *Directed Acyclic Graph* (DAG).

After building a *dependency graph*, a graph reduction algorithm (shown in Figure 1) is executed. Step 1 uses Kahn's algorithm[5] of topological ordering. In step 6, when the completed DTR is removed from the dependency graph, all edges pointing to it from other DTRs are also removed.

The algorithm allows for parallelism. If more than one DTR without input dependencies is found during step 1, the subsequent steps 2-6 can be executed in parallel for each discovered DTR.

It should be noted that if DTR exectuion has failed, **hamake** can and will continue to process other DTRs which do not depend directly or indirectly on the results of this DTR. This permits the user to fix problems later and re-run **hamake**, without the need to re-process all data.

Cyclic dependencies must be avoided, because a dataflow containing such dependencies is not guaranteed to terminate. Implicit checks are performed during the reading of DAG definitions and the building of the dependency graph. If a cycle is detected, it is reported as an error. Thus the dependency

[1] The current stable version of **hamake** uses exact (non-fuzzy) timestamps.

[2] Because **hamake** has no way to update them, it does not attempt to ensure that files are up to date, unless they are listed as one of a DTR's outputs.

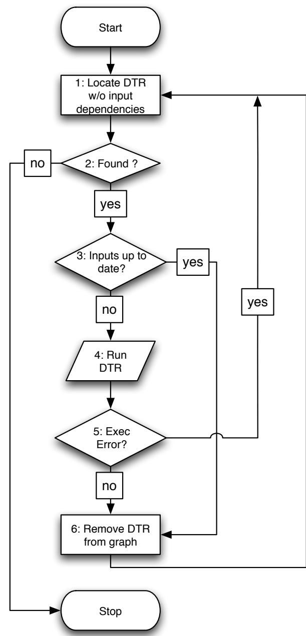

Fig. 1. **hamake** dependency graph reduction algorithm

graph used by **hamake** is assured to be a *directed acyclic graph*.

However, **hamake** supports a limited scenario of iterative processing with a feature called *generations*. Each input or output file can be marked with a *generation* attribute. Any two files referencing the same path in the file system while having different generations are represented as two distinct vertices in the dependency graph. This permits resolution of cyclic dependencies within the context of a single **hamake** execution.

One useful consequence of **hamake** dataflow being a DAG is that for each vertex we can calculate the list of vertices it depends on directly and indirectly using simple *transitive closure*. This allows us to easily estimate the part of a dataflow graph being affected by updating one or more files, which could be especially useful for datasets where the cost of re-calculation is potentially high due to data size or computational complexity.

Hamake is driven by dataflow description, expressed in a simple XML-based language. The full syntax is described in [6]. The two main elements, *fold* and *foreach*, correspond to two types of DTRs. Each element has input, output, and processing instructions. The execution of processing instructions brings the DTR output up to date.

Fold implies a many-to-one dependency between input and output. In other words, the output depends on the entirety of the input, and if any of the inputs have been changed, the outputs need to be updated. *Foreach* implies a one-to-one dependency where for each file in an input set there is a corresponding file in an output set, each updated independently.

Hamake dataflow language has declarative semantics making it easy to implement various dataflow analysis and optimization algorithms in the future. Examples of such algorithms include: merging dataflows, further execution parallelization, and analysis and estimation of dataflow complexity.

III. Parallel Execution

While determining the sequence and launching of Hadoop jobs required to bring all datasets up-to-date, **hamake** attempts to perform all required computations in the shortest possible time. To achieve this, **hamake** aims for maximal cluster utilization, running as many Hadoop jobs in parallel as cluster capacity permits.

There are three main factors that drive job scheduling logic: file timestamps, dependencies, and cluster computational capacity. On the highest level, DTR dependencies determine the sequence of jobs to be launched.

In the case of *fold* DTR, a single Hadoop job, PIG script or shell command, may be launched, and hence there is no opportunity for parallel execution. In the example shown in Figure 2, since fileset *B* depends on all files in fileset *A*, a single job associated with *fold* DTR will be executed.

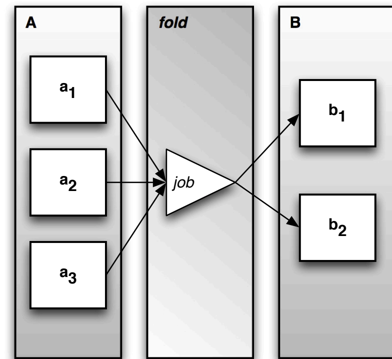

Fig. 2. Decomposition of *fold* DTR

A *foreach* DTR works by mapping individual files in fileset *A* to files in fileset *B*. Assuming that fileset *A* consists of 3 files: a_1, a_2, a_3, the dependency graph could be represented as shown in Figure 3. In this case, we have an opportunity to execute the three jobs in parallel.

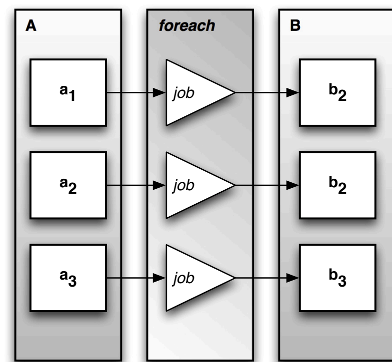

Fig. 3. Decomposition of *foreach* DTR

The Hadoop cluster capacity is defined in terms of the number of *map slots* and *reduce slots*. When a DTR launches a Hadoop job, either directly as defined by *mapreduce* processing instruction or via PIG script, a single job will spawn one or more *mapper* or *reducer* tasks, each taking one respective slot. The number of mappers and reducers launched depends on many factors, such as the size of the HDFS block, Hadoop cluster settings, and individual job settings. In general, **hamake** has neither visibility of nor control over most of these factors, so it does not currently deal with individual tasks. Thus **hamake** parallel execution logic is controlled by a command line option specifying how many jobs it may run in parallel.

IV. EXAMPLE

In a large, online library, the **hamake** utility can be used to automate searches for duplicates within a corpus of millions of digital text documents. Documents with slight differences due to OCR errors, typos, differences in formatting, or added material such as a foreword or publishers note can be found and reported.

To illustrate **hamake** usage, consider the simple approach of using the *Canopy clustering algorithm*[7] and a *vector space model*[8] based on word frequencies. The implementation could be split into a series of steps, each implemented as *MapReduce job*:

- *ExtractText* — Extract a plain text from native document format (e.g. PDF).
- *Tokenize* — Split plain text into tokens which roughly correspond to words. Deal with hyphens, compound words, accents, and diacritics, as well as case-folding, stemming, or lemmatization, resulting in a list of normalized tokens.
- *FilterStopwords* — Filter out *stopwords*, like *a*, *the*, and *are*.
- *CalculateTF* — Calculate a feature vector of term frequencies for each document.
- *FindSimilar* — Run *Canopy clustering algorithm* to group similar documents into clusters using *cosine distance* as a fast approximate distance metric.
- *OutputResult* — Output document names, which are found in clusters with more than one element.

Each of the six MapReduce jobs produces an output file which depends on its input. For each document, these jobs must be invoked sequentially, as the output of one task is used as input of the next. Additionally, there is a configuration file containing a list of stop words, and some task outputs depend on this file content. These dependencies could be represented by a DAG, as shown in Figure 4, with vertices representing documents and jobs assigned to edges. The XML file describing this dataflow in **hamake** language is shown as Listing 1.

Listing 1. *hamakefile*, describing process for detecting duplicate documents

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <project name="FindSimilarBooks">
3
4   <property name="lib" value="/lib/" />
5
6   <fileset id="input" path="/doc" mask="*.pdf" />
7   <file id="output" path="/result.txt" />
8
9   <foreach name="ExtractText">
10    <input>
11      <include idref="input" />
12    </input>
13    <output>
14      <file id="plainText" path="/txt/${foreach:filename}" />
15    </output>
16    <mapreduce jar="${lib}/hadoopJobs.job" main="com.example.TextExtractor">
17      <parameter>
18        <literal value="${foreach:path}" />
19      </parameter>
20      <parameter>
21        <reference idref="plainText" />
22      </parameter>
23    </mapreduce>
24  </foreach>
25
26  <foreach name="Tokenize">
27    <input>
28      <file id="plainText" path="/txt" />
29    </input>
30    <output>
31      <file id="tokens" path="/tokens/${foreach:filename}" />
32    </output>
33    <mapreduce jar="${lib}/hadoopJobs.job" main="com.example.Tokenizer">
34      ...
35    </mapreduce>
36  </foreach>
37
38  <foreach name="FilterStopWords">
39    <input>
40      <file id="stopWords" path="/stopwords.txt" />
41      <file id="tokens" path="/tokens" />
42    </input>
43    <output>
44      <file id="terms" path="/terms/${foreach:filename}" />
45    </output>
46    <mapreduce jar="${lib}/hadoopJobs.job" main="com.example.Tokenizer">
47      ...
48    </mapreduce>
49  </foreach>
50
51  <foreach name="CalculateTF">
52    <input>
53      <file id="terms" path="/terms" />
54    </input>
55    <output>
56      <file id="TFVector" path="/TF" />
57    </output>
58    <mapreduce jar="${dist}/hadoopJobs.job" main="com.example.CalculateTF">
59      ...
60    </mapreduce>
61  </foreach>
62
63  <fold name="FindSimilar">
64    <input>
65      <file id="TFVector" path="/TF" />
66    </input>
67    <output>
68      <include idref="clustersList" path="/clusters"/>
69    </output>
70    <mapreduce jar="${lib}/hadoopJobs.job" main="com.example.Canopy">
71      ...
72    </mapreduce>
73  </fold>
74
75  <fold name="OutputResult">
76    <input>
77      <file id="clustersList" path="/clusters" />
78    </input>
79    <output>
80      <include idref="output" />
81    </output>
82    <mapreduce jar="${lib}/hadoopJobs.job" main="com.example.OutputSimilarBooks">
83      ...
84    </mapreduce>
85  </fold>
86 </project>
```

The first DTR (lines 10-25) converts a document from a native format such as PDF to plain text. The input of the DTR is a reference to the */doc* folder, and the output is the */txt* folder. The *foreach* DTR establishes one-to-one dependencies between files with identical names in these two folders. The Hadoop job which performs the actual text extraction is defined using the *mapreduce* element. It will be invoked by **hamake** for each unsatisfied dependency. The job takes two parameters, defined with *parameter* elements - a path to an original document as the input and a path to a file where the plain text version will be written. The remaining five DTRs are defined in a similar manner.

Hamake, when launched with this XML dataflow definition, will execute a graph reduction algorithm, as shown in Figure 1,

and will find the first DTR to process. In our example, this is *ExtractPlainText*. First, **Hamake** will launch the corresponding Hadoop job and immediately following, execute DTRs which depend on the output of this DTR, and so on until all output files are up to date. As a result of this data flow, a file named *results.txt* with a list of similar documents will be generated.

This data flow could be used for incremental processing.

When new documents are added, **hamake** will refrain from running the following DTRs: *ExtractText*, *Tokenize*, *FilterStopWords*, and *CalculateTF* for previously processed documents. However, it will run those DTRs for newly added documents and then, re-run *FindSimilar* and *OutputResults*.

If the list of stop words has been changed, **hamake** will re-run only *FilterStopWords*, *CalculateTF*, *FindSimilar*, and *OutputResults*.

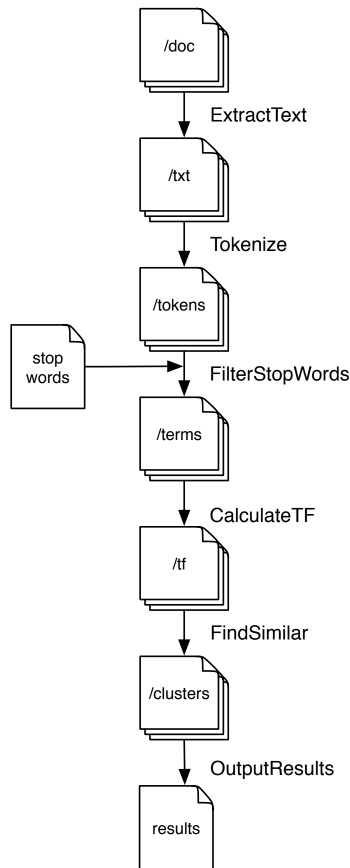

Fig. 4. Directed acyclic graph of a data flow for duplicate document detection

V. Related Work

Several workflow engines exist for Hadoop, such as Oozie, Azkaban, and Cascading. Although all of these products could be used to solve similar problems, they differ significantly in design, philosophy, target user profile, and usage scenarios limiting the usefulness of a simple, feature-wise comparison.

The most significant difference between these engines and **hamake** lies in the *workflow* vs. *dataflow* approach. All of them use the former, explicitly specifying dependencies between jobs. **Hamake**, in contrast, uses dependencies between datasets to derive workflow. Both approaches have their advantages, but for some problems, the dataflow representation as used by **hamake** is more natural.

VI. Future Directions

One possible **hamake** improvement may be better integration with Hadoop schedulers. For example, if *Capacity Scheduler* or *Fair Scheduler* is used, it would be useful for **hamake** to take information about scheduler *pools* or *queues* capacity into account in its job scheduling algorithm.

More granular control over parallelism could be achieved if the **hamake** internal dependency graph for *foreach* DTR contained individual files rather than just filesets. For example, consider a dataflow consisting of three filesets *A*, *B*, *C*, and two *foreach* DTR's: D_1, mapping *A* to *B*, and D_2, mapping *B* to *C*. File-level dependencies would allow some jobs to run from D_2 without waiting for all jobs in D_1 to complete.

Another potential area of future extension is the **hamake** dependency mechanism. The current implementation uses a fairly simple timestamp comparison to check whether dependency is satisfied. This could be generalized, allowing the user to specify custom dependency check predicates, implemented either as plugins, scripts (in some embedded scripting languages), or external programs. This would allow for decisions based not only on file meta data, such as the timestamp, but also on its contents.

Several **hamake** users have requested support for iterative computations with a termination condition. Possible use-cases include fixed-point computations and clustering or iterative regression algorithms. Presently, to embed this kind of algorithm into the **hamake** dataflow, it requires the use of the *generations* feature combined with external automation, which invokes **hamake** repeatedly until a certain exit condition is satisfied. **Hamake** users could certainly benefit from native support for this kind of dataflow.

References

[1] A. Bialecki, M. Cafarella, D. Cutting, and O. O'Malley. Hadoop: a framework for running applications on large clusters built of commodity hardware. 2005.

[2] J. Dean and S. Ghemawat. Map Reduce: Simplified data processing on large clusters. *Communications of the ACM-Association for Computing Machinery-CACM*, 51(1):107–114, 2008.

[3] Kevin Weil. Hadoop at twitter. http://engineering.twitter.com/2010/04/hadoop-at-twitter.html.

[4] Mukesh Gangadhar. Benchmarking and optimizing hadoop. http://www.slideshare.net/ydn/hadoop-summit-2010-benchmarking-and-optimizing-hadoop.

[5] AB Kahn. Topological sorting of large networks. *Communications of the ACM*, 5(11):558–562, 1962.

[6] Vladimir Orlov and Alexander Bondar. Hamake syntax reference. http://code.google.com/p/hamake/wiki/ HamakeFileSyntaxReference.

[7] Andrew McCallum, Kamal Nigam, and Lyle H. Ungar. Efficient Clustering of High-Dimensional Data Sets with Application to Reference Matching. *KDD '00*, 2000.

[8] C.D. Manning, P. Raghavan, and H. Schütze. *Introduction to information retrieval*. Cambridge University Press, 2008.