Cloud Objects: Programming the Cloud with Object-Oriented Map/Reduce

Julian Friedman
Dept. of Computer Science
University of York
York, UK
julz@cs.york.ac.uk

IBM Cloud Labs
IBM United Kingdom Ltd.
Hursley Park, UK
julz.friedman@uk.ibm.com

Manuel Oriol
Dept. of Computer Science
University of York
York, UK
manuel@cs.york.ac.uk

ABB Corporate Research
Industrial Software Systems
Baden-Dättwil, Switzerland
manuel.oriol@ch.abb.com

***Abstract*—Cloud Objects parallelizes Object-Oriented programs written in Java using Map/Reduce by adding simple, declarative annotations to the code. The system automatically persists objects to a partitioned filesystem and efficiently executes methods across the partitioned object data. Using Cloud Objects, data-intensive programs can be written in a simple, readable, object-oriented manner, while maintaining the performance and scalability of the Map/Reduce platform. Cloud Objects shows that it is possible to combine the benefits of an object-oriented model and the power of Map/Reduce.**

***Keywords*-Map/Reduce, Hadoop, JPA, Cloud Computing**

I. Introduction

Data-parallel frameworks such as Map/Reduce [1] have become increasingly popular. The developers of Internet applications have turned to these technologies to harness the power of large numbers of distributed machines to address the challenges of processing huge amounts of data.

Map/Reduce was designed for a specific domain — data-dependent batch computations — it was not designed to be a general approach to designing whole applications. Programmers must fit their code into the structure of a Map/Reduce algorithm. Programming a Map/Reduce application involves splitting the algorithm into separate Mappers, Reducers, InputFormats and Drivers (to name a few) and encourages a tight coupling of these components.

Splitting the application logic between many tightly coupled classes compromises many of the advantages of object-oriented design, such as composability, modularity and encapsulation. The lack of proper object orientation makes it difficult to evolve and compose Map/Reduce programs in to larger systems, to maintain Map/Reduce programs, and to quickly develop new applications using Map/Reduce.

This paper describes *Cloud Objects*, a new system which allows Map/Reduce applications to be written in an object-oriented style. *Cloud Objects* uses simple declarative annotations to describe how object data should be persisted to the distributed filesystem. As well as automatically generating the code to persist the objects to a partitioned datastore (in a manner similar to existing systems such as DataNucleus [2]), the system generates the Map/Reduce code to run methods across the partitioned object data. A program can be structured and composed in an object-oriented style and deployed to existing Map/Reduce clusters.

The paper is structured as follows. The following section, Section II, introduces the programming model with a simple example. Section III describes the programming model in detail. Section IV describes our prototype implementation. Section V discusses *Cloud Objects* in the context of related work. SectionVI concludes the paper.

II. An Introductory Example

Cloud Objects' persistence annotations are based on JPA (Java Persistence Annotations, JSR 317 [3] and 220 [4]). This allows maximum compatibility with existing code and minimizes the need for developers to learn a new syntax. Persisting an object to a distributed store is as simple as using standard JPA annotations. Once persisted, methods can be run in parallel across an object's partitioned data.

In Listing 1, we illustrate the programming model with a simple example. The Wiki class contains a map of page names to WikiPage objects which the framework will persist to the distributed filesystem. Assuming the map is large, the data will be partitioned across many machines.

While persistence annotations alone allow persisting and querying object data in the distributed file system, they do not allow efficient processing of the object data. The scalability and efficiency of the Map/Reduce model is based on the ability to distribute code to data. Map/Reduce is a computation framework as well as a data storage and querying framework. While a simple approach based on JPA alone would suffice to persist the object data, it would not be able to efficiently run object methods with data-locality; it would gain the benefits of the distributed file system to persist and retrieve the object data, but not the advantages of the Map/Reduce model to execute fault-tolerant, resilient methods across the data.

Cloud Objects adds the ability to add @Multicast methods to a class which can be distributed automatically, with data-locality, by the framework. A multicast method across the 'pages' member variable is shown in Listing 1. The multicast method is automatically run across each shard of the partitioned 'pages' variable using Map/Reduce, and the results are combined to a single array using a standard UNION reduction.

 ISBN: 978-1-61208-216-5

```
@Entity public class Wiki {
  @OneToMany @KeyField("url") private Map<String, WikiPage> pages;

  public Collection<String> search(String phrase) { this.search(pages, phrase); }

  @Multicast(reduce=UNION)
  protected Collection<String> search(Map<String, WikiPage> pages, String phrase) {
     Collection<String> matches = new ArrayList<String>();
     for(Map.Entry<String, WikiPage> page : pages) {
       if(page.getContents().indexOf(phrase) > -1) { matches.add(url); }
     }
     return matches;
  }
}
```

Listing 1: A Distributed Wiki

The results of the search(..) method are themselves written to the distributed filesystem and can be processed with data-locality by another @Multicast method. This allows seamless, efficient composition of multicast methods in to full applications.

III. Programming Model

The aim of *Cloud Objects* is to allow the business logic of an application to be expressed in a simple, object-oriented style while efficiently running methods across partitioned object data using Map/Reduce.

Cloud Objects can be split into annotations which relate to persisting and retrieving objects from the datastore (which are based on JPA), and annotations related to running methods across the partitioned data. As the persistence-related annotations are based directly on a subset of the JPA standard, in this section, we focus only on the additional annotations we have introduced to run methods across the persisted object data.

A. Multicast Methods

Multicast methods interact with partitioned instance data by running appropriate Map/Reduce jobs over their input data (which is stored in the distributed filesystem).

The programmer uses an EntityManager instance to retrieve a Cloud Object from the datastore. When the EntityManager retrieves an object, it creates proxy collections to wrap distributed member data. These proxy collections are initialized with the location of their data in the distributed file system, and contain methods to configure a Map/Reduce job to run against their contents.

On the client machine, the EntityManager replaces any methods of a returned instance which have been annotated with the @Multicast annotation using byte-code rewriting.

1) Inputs to Multicast Methods: For simplicity, multicast methods only allow one partitioned input collection to be passed as an argument. The programmer is free to run methods over multiple partitioned inputs by using a multicast method to create a collection containing a cross product or join of two other lists. This resulting list will be stored on the distributed filesystem and can be passed as an input to another multicast method. Alternatively one of the lists (usually the smaller) can be passed as class data and retrieved from the distributed filesystem on demand.

To maximize encapsulation, the programmer is encouraged to provide an external client method (not annotated with @Multicast) which calls a protected or private @Multicast method with the needed arguments. This is shown in the 1-arg and 2-arg versions of the search (..) method in Listing 1.

2) Outputs from Multicast Methods: Safely running a Multicast method across a number of machines requires a number of constraints on multicast methods. If the method replicas were allowed to write directly to the member variables of the class the individual jobs would no longer be independent. Instead, multicast methods may only write to member variables in the following (safe) ways:

a) Shared: The framework sends instance variables marked with the @Shared annotation to every node using Hadoop's distributed cache. On worker machines, any updates made to @Shared variables other than Counters and Joinables (see next) are ignored and may throw exceptions. Updates made to member variables that are not annotated with @Shared are limited to a particular object on a particular node. This can be useful for caches and other data structures which do not need to be maintained across machines.

b) Counters: Counters allow methods to safely update member variables which increase monotonically. Counters are implemented using the underlying Hadoop framework's Counter functionality. Hadoop's Counters are global, which breaks encapsulation. Counters in *Cloud Objects* are automatically given generated, private IDs based on the object class and the unique identity of the object instance.

c) Joinables: Joinables allow for a more general method of updating an instance variable from multiple methods. Joinables are inspired by the Concurrent Revisions programming model [5]. Joinables may be declared either by deriving from the Joinable marker interface or by the addition of

 ISBN: 978-1-61208-216-5

a specific @JoinedWith(..) annotation. Classes which inherit from the Joinable marker interface are expected to have either a static join(..) method or to be themselves annotated with @JoinedWith(..) to refer to a class with a no-arg constructor and a join(..) method.

B. Reductions

The final result from a multicast method is reduced to a single value using a Reducer class. A set of default reducers is provided for Unions, Sums and Averages, and the programmer is free to name their own reduction class in the @Multicast(reduce = ..) annotation.

IV. Implementation

We have implemented a prototype of *Cloud Objects* based on OpenJPA [2] and Hadoop [6]. We briefly describe the key elements of the implementation in this section.

A. Collections Proxies

Instance variables of *Cloud Objects* which are specified as Lists or Maps are automatically proxied with a HadoopList or a HadoopMap class which reads and writes from the Distributed Filesystem when the object needs to be persisted. This is done by the EntityManager when retrieving the object, and by the generated Mapper classes when they create an object instance to process partitions of a Multicast Method's input data.

The distributed collection classes support two modes of operation. When used as input to a Multicast Method (on the master machine), the collection classes provide a configureInput(Job job) method which configures a Hadoop Job with the input directory containing the collection's data and an appropriate InputFormat which can parse the data. Each Map job is then provided with a single shard of the partitioned data. The shard for the Map job is created using the createShard(Class<T>, Mapper.Context) method of the HadoopMap and HadoopList classes, which converts the input key/value pairs for the current map task in to the type of collection required by the mapper method.

When accessed as an instance variable rather than as a parameter to a MultiCast method, a distributed collection reads and writes its object data from the distributed filesystem. This is potentially inefficient as the object data is unlikely to be local, but is useful for tasks such as printing out the final results of a computation.

B. Multicast Methods

Proxied objects are obtained using a custom JPA EntityManager. We use the open-source OpenJPA [2] implementation of the JPA standard which uses byte-code rewriting to extend plain java objects with persistence information. When the custom EntityManager returns a persistent object or collection, it is scanned for methods annotated with @Multicast and if these are present they are overriden to be dispatched via Hadoop.

The MulticastInvoker class is responsible for dispatching Multicast methods to run on the Map/Reduce cluster using Hadoop. A single Mapper and Reducer (DefaultMapper and DefaultReducer) are used for every job. These classes are configured using job configuration variables set by MulticastInvoker. For example, the DefaultMapper consults the 'com.ibm.cloudlabs.cloudobjects.multicast.target.class' variable; this variable records the class which will be used on each node to run the 'meat' of the job. MulticastInvoker delegates to the input collection to set the job input path based on the location of the passed object on the distributed filesystem.

Each Map job creates a new copy of the delegate object using the no-arg constructor, which must be present in the class. Any @Shared or Joinable variables in the class are initialized from the distributed cache, and HadoopCollection.createShard(..) is used to create a shard of the multicast variable to be passed to the method from the input pairs. Output is saved to a directory configured by a HadoopCollection or HadoopMap and the client automatically creates and returns a proxy collection wrapping the output directory.

C. Joinables

Joinable variables are initialised before a method is run using the distributed cache, so that each node has the same initial value. During the multicast method, the joinable variable maintains any values set during the method. This preserves the independence of the map jobs. The Mapper implementation writes both updated joinable values and the results of the multicast method to the map output, prefixing a 0 or 1 to the stream to differentiate each case. These outputs are sorted by the framework and passed to the reducer. The Reducer merges Joinable variables using the appropriate Joiner class for the variable and delegates to the configured Reduction class to create the final result of the method.

V. Related Work

The Hadoop [6] implementation of the Map/Reduce algorithm [1] provides a Java API to Map/Reduce. This API is, however, a low-level API which requires the programmer to express computations as collections of Map jobs, Reduce jobs and Driver classes. All of these interact to perform a computation and collect results over a distributed, partitioned datastore. Map/Reduce is typically not object-oriented because it requires programmers to express jobs in a functional way. *Cloud Objects* allows applications to use an object oriented style while taking advantage of the scale and power of Map/Reduce. While *Cloud Objects* does not have the full generality of Map/Reduce - in particular, many Map/Reduce algorithms are in practice tuned using techniques such as In-Mapper Combiners, Pairs and Stripes (see e.g., [7]), which rely on a tight coupling between Mapper and Reducer - we believe it is a promising method for creating large scale applications. While Map/Reduce programs tend to rely on tight coupling between Mapper and Reducer, *Cloud Objects* favours the use of standard, reusable reducers - though custom reducers are supported - and higher-level concepts such as Joinable types and Counters.

 ISBN: 978-1-61208-216-5

Cloud Objects follows a trend of higher-level and domain-specific languages such as Pig [8], Hive [9] and JAQL [10] built on top of Map/Reduce. The aim of these languages is to retain the performance, reliability and scalability benefits of Map/Reduce, while presenting a more familiar, simpler or high-level style to programmers.

Sawzall [11] runs on top of Map/Reduce and, similarly to our approach, uses a set of standard reducers to aggregate outputs from custom map methods. Sawzall uses a custom scripting language which processes a single input and emits values to output types such as sum tables and maximum tables which retain the total of their input and the largest of all of their inputs, respectively. Sawzall is however different from the programming language in which the rest of the application is coded. *Cloud Objects* solves this issue.

Pig [8] is an imperative language with a number of group-based built-in functions such as co-joins, projections and restrictions. The aim of Pig is to provide an easy way for programmers familiar with imperative programming to query distributed data using Map/Reduce. It does not provide any way of writing scripts in an object oriented style and focuses on ad-hoc querying of existing data.

JAQL [10] is closer in spirit to *Cloud Objects*, providing a pure functional language for querying Javascript Object Notation (JSON) objects using Map/Reduce. JAQL is designed for ad-hoc queries of large data rather than writing maintainable programs, and while it allows querying serialised objects, it does not provide features to allow its own programs themselves to be written in an object oriented style.

Hive [9] presents an SQL-like declarative interface for querying large-scale data using Map/Reduce. This lacks the generality of the *Cloud Objects* approach.

Collection-style interfaces such as FlumeJava [12] and Crunch [13] have advantages over domain-specific languages such as Pig and JAQL in that they allow the program to be expressed in a single language and are perhaps closest to our approach. These interfaces allow complicated pipelines of operations on collections of objects to be efficiently optimised in to a set of Map/Reduce jobs. These systems focus on manipulating object collections rather than on adding data-parallel methods to existing object-oriented programs.

Other tools exist which provide JPA bindings from Java objects to partitioned data stores such as HDFS. DataNucleus [2] is an open-source JPA provider with support for a variety of backends including Hive. Users of Google's AppEngine [13] environment can use JPA to persist objects to the AppEngine data store, and can separately use the Map/Reduce functionality of AppEngine to run map jobs over entities in the data store.

Alternatives to Map/Reduce also exist. For example, in the .Net ecosystem, Dryad [14] and DryadLINQ [15] have become popular frameworks for expressing data-parallel computations. Dryad provides a more generic model than Map/Reduce, allowing arbitrary directed acyclic graph computations, and DryadLINQ provides a language-integrated query language which can compiles to Dryad jobs. Another example is Sky-writing [16] which provides a functional coordination language to describe computations to be run on CIEL [17], a Map/Reduce-like system for cluster computation. *Cloud Objects* are at a higher level of abstraction and could be applied on top of these as well.

VI. Conclusion

This paper introduced *Cloud Objects*, an object-oriented programming model which exposes the power of Map/Reduce in a simple, encapsulated, modular way. To use *Cloud Objects*, programmers only need to add a couple of annotations to a regular Java program.

Our prototype implementation of *Cloud Objects* uses Hadoop [6] to distribute the actual code and data and extends OpenJPA [2] to store and retrieve persistent objects to a distributed filesystem. We have benchmarked the prototype using EC2. Initial experiments show that the overhead induced is negligible compared to the cost of computation and network.

References

[1] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008. [Online] Available: http://labs.google.com/papers/mapreduce-osdi04.pdf [Accessed: 14 May 2012].

[2] "Datanucleus," http://www.datanucleus.org/ [Accessed: 14 May 2012].

[3] L. DeMichiel, "Jsr 317: Java persistence 2.0," 2009.

[4] L. DeMichiel and M. Keith, "Jsr 220: Enterprise javabeans," 2006.

[5] S. Burckhardt, A. Baldassin, and D. Leijen, "Concurrent programming with revisions and isolation types," in *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, ser. OOPSLA '10. New York, NY, USA: ACM, 2010, pp. 691–707.

[6] "Apache hadoop," http://hadoop.apache.org [Accessed: 14 May 2012].

[7] J. Lin and C. Dyer, *Data-Intensive Text Processing with MapReduce*. Morgan & Claypool Publishers, 2010.

[8] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins, "Pig latin: a not-so-foreign language for data processing," in *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, ser. SIGMOD '08. New York, NY, USA: ACM, 2008, pp. 1099–1110.

[9] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy, "Hive: a warehousing solution over a map-reduce framework," *Proc. VLDB Endow.*, vol. 2, pp. 1626–1629, August 2009.

[10] "Jaql: Query language for javascript object notation," http://code.google.com/p/jaql/ [Accessed: 14 May 2012].

[11] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan, "Interpreting the data: Parallel analysis with Sawzall," *Sci. Program.*, vol. 13, pp. 277–298, October 2005.

[12] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. R. Henry, R. Bradshaw, and N. Weizenbaum, "FlumeJava: easy, efficient data-parallel pipelines," in *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, ser. PLDI '10. New York, NY, USA: ACM, 2010, pp. 363–375.

[13] "Google app engine," http://code.google.com/appengine/ [Accessed: 14 May 2012].

[14] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: distributed data-parallel programs from sequential building blocks," in *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*. ACM, 2007, pp. 59–72.

[15] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. K. Gunda, and J. Currey, "DryadLINQ: a system for general-purpose distributed data-parallel computing using a high-level language," in *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, ser. OSDI'08. Berkeley, CA, USA: USENIX Association, 2008, pp. 1–14.

 ISBN: 978-1-61208-216-5

[16] D. Murray and S. Hand, "Scripting the cloud with skywriting," in *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*. USENIX Association, 2010, pp. 12–12.

[17] D. Murray, M. Schwarzkopf, C. Smowton, S. Smith, A. Madhavapeddy, and S. Hand, "Ciel: a universal execution engine for distributed data-flow computing," in *Proceedings of NSDI*, 2011.

 ISBN: 978-1-61208-216-5