The Jabberwocky Programming Environment for Structured Social Computing

Salman Ahmad saahmad@stanford.edu
Alexis Battle ajbattle@stanford.edu
Zahan Malkani zahanm@stanford.edu
Sepandar D. Kamvar sdkamvar@stanford.edu

ABSTRACT

We present Jabberwocky, a social computing stack that consists of three components: a human and machine resource management system called Dormouse, a parallel programming framework for human and machine computation called ManReduce, and a high-level programming language on top of ManReduce called Dog. Dormouse is designed to enable cross-platform programming languages for social computation, so, for example, programs written for Mechanical Turk can also run on other crowdsourcing platforms. Dormouse also enables a programmer to easily combine crowdsourcing platforms or create new ones. Further, machines and people are both first-class citizens in Dormouse, allowing for natural parallelization and control flows for a broad range of data-intensive applications. And finally and importantly, Dormouse includes notions of real identity, heterogeneity, and social structure. We show that the unique properties of Dormouse enable elegant programming models for complex and useful problems, and we propose two such frameworks. ManReduce is a framework for combining human and machine computation into an intuitive parallel data flow that goes beyond existing frameworks in several important ways, such as enabling functions on arbitrary communication graphs between human and machine clusters. And Dog is a high-level procedural language written on top of ManReduce that focuses on expressivity and reuse. We explore two applications written in Dog: bootstrapping product recommendations without purchase data, and expert labeling of medical images.

ACM Classification: H5.2 [Information interfaces and presentation]: User Interfaces. - Graphical user interfaces.

General terms: Languages, Human Factors

Keywords: social computing, crowdsourcing

INTRODUCTION

In the last few years, there has been a heightened interest in human computation, where tasks that are difficult for computers (such as image labeling or transcription) are split into microtasks and dispatched to people. Human computation has been used to address large-scale goals ranging from labeling images [23], to finding 3-D protein structures [3], to creating a crowdsourced illustrated book [8], to classifying galaxies in Hubble images [1].

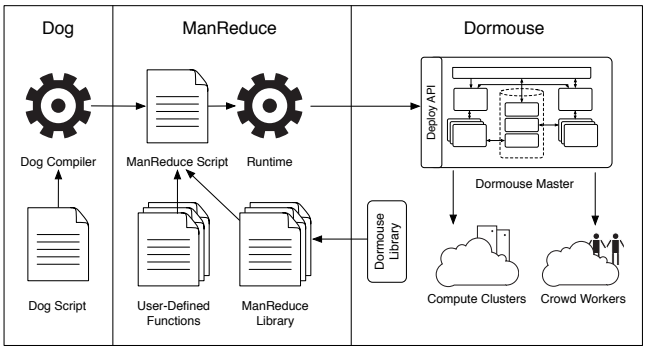

Figure 1: Overview of Jabberwocky

In existing paradigms, human workers are often treated as homogeneous and interchangeable, which is useful in handling issues of scale and availability. However, the limited notions of identity, reputation, expertise, and social relationships limit the scope of tasks that can be addressed with these systems. Incorporating real identities, social structure, and expertise modeling has proven valuable in a range of applications, for example, in question-answering with Aardvark [11]. Building general frameworks for human computation that include these notions will enable complex applications to be built more easily.

A second drawback of existing platforms is that each defines a stand-alone system with rigid structure and requirements, and thus demands significant work in order to integrate human computation into larger applications. Each new application may require building a pipeline from the ground up, and in many cases, a new community. Particularly for complex applications, which may involve several steps of human computation using different crowdsourcing platforms interleaved with machine computation, constructing such a pipeline can be a tedious effort. In practice, complex systems are discouraged, and most uses of human computation avoid multiple interleaved processing steps.

To address these issues, we designed Jabberwocky, a social computing stack that consists of Dormouse, ManReduce, and Dog. Dormouse is the "virtual machine" layer of the Jabberwocky stack, consisting of low-level software libraries that interact with both people and traditional computing machines. Dormouse maintains real identities, rich user pro-

UIST'11, October 16-19, 2011, Santa Barbara, CA, USA.

files, and social relationships for the people who comprise the system, and allows end users to define arbitrary person-level properties and social structures in the system. Further, Dormouse allows programmers to interact with several different crowdsourcing platforms using the same primitives. This enables the development of cross-platform programming languages for social computing. And finally, because Dormouse defines communications protocols for both people and machines, programmers can very naturally interact with both in unified control flows even for complex parallel processing tasks.

On top of Dormouse, we built ManReduce, a programming framework inspired by MapReduce [4] (and related to CrowdForge [13]) to facilitate complex data processing tasks. ManReduce, like MapReduce, gives the programmer the ability to specify *map* and *reduce* steps, but allowing either step to be powered by human or machine computation. The data flow, resource allocation, and parallelization necessary for each step are handled by ManReduce with no onus on the programmer. In addition to combining machine and human computation, ManReduce also provides the ability to choose particular types of people to complete each task (based on Dormouse), and allows arbitrary dependencies between multiple *map* and *reduce* steps. Many interesting social computing applications fit naturally into this paradigm, as they frequently involve the need for parallelization of subtasks across people or machines, and subsequent aggregation such as writing a summary or averaging ratings. As a simple example, conducting a survey and tabulating summary statistics for each question (breaking down according to a variety of demographics) can be expressed using a human *map* step that sends the survey in parallel to many people, and one or more machine *reduce* steps on the output that aggregate the responses keyed by question and/or user demographic.

While ManReduce offers flexibility and power, it can be too low-level for several classes of applications. In many cases, it is useful to trade some flexibility for expressivity, maintainability, and reuse. To that end, we designed Dog, a high-level scripting language that compiles into ManReduce. Inspired by the Pig [19] and Sawzall [21] languages for data mining, Dog defines a small but powerful set of primitives for requesting computational work from either people or machines, and for interfacing with workers (and their properties) through Dormouse. In addition, it defines simple interfaces for using, creating, and sharing functions (human or machine) and microtask templates, making it easy to quickly implement a wide range of applications.

Together, the components of the Jabberwocky stack allow programmers to implement applications in a few lines of code that would otherwise require writing large amounts of ad hoc infrastructure. We explore several such applications in this paper, including a journal transcription application that maintains the privacy of the journal-writer, a recommender system that requires no pre-existing co-occurrence data, and a medical image tagging application that allows experts to leverage low-cost generalists (and generalists to learn from experts).

DORMOUSE

The lowest level of the Jabberwocky stack is Dormouse, a "virtual machine" layer that enables cross-platform social computation. Similar to process virtual machines (such as the Java Virtual Machine) in traditional computing, Dormouse sits on top of existing crowdsourcing platforms, providing a platform-independent programming environment that abstracts away the details of the underlying crowdsourcing platform. Importantly, Dormouse also enables programmers to seamlessly create new crowdsourcing communities and add social features (such as worker profiles and relationships) to existing ones.

Design Goals

Our design goals for Dormouse are to:

Support cross-platform programming languages for social computing. Programming languages run on Dormouse can work with any crowdsourcing platform that hooks into Dormouse, and can support execution across several platforms in the same program. For example, a programmer may, in one step, routes tasks to a large number of inexpensive workers from one crowdsourcing platform, and in a next step, routes tasks to a smaller number of vetted experts from another platform, without the needing to learn the separate (and often complex) API calls from multiple platforms.

Make it easy for programmers to build new crowdsourcing communities. In addition to being able to reside on top of existing crowdsourcing platforms, Dormouse makes it easy for programmers to create new worker communities given a set of e-mail addresses.

Enable rich personal profiles and social structures. Programming languages run on Dormouse allow the programmer to route tasks based personal properties such as expertise and demographic, and also to set and modify expertise levels based on task performance. Further, programmers may route tasks based on social structure (for example, an application that matches technical papers to reviewers may route papers to computer science graduate students to review, and then to their advisor to validate the review).

Combine human and machine computation into a single parallel computing framework. Programming languages run on Dormouse allow the programmer to allocate machines and people in similar ways, leading to more natural control flows for parallelization.

Make it easy for programmers to create and reuse human tasks. Dormouse has a straightforward mechanisms to create new templates for human tasks, and importantly, to reuse those created by others. This minimizes redundant work and allows programmers to focus on control flows rather than creating and optimizing task templates.

Architecture

Dormouse is implemented as a set of software components that reside on a *Dormouse master* machine. These components communicate with one another, as well as an external Dormouse machine cluster and a set of human workers. These software components are described below and shown in Figure 2.

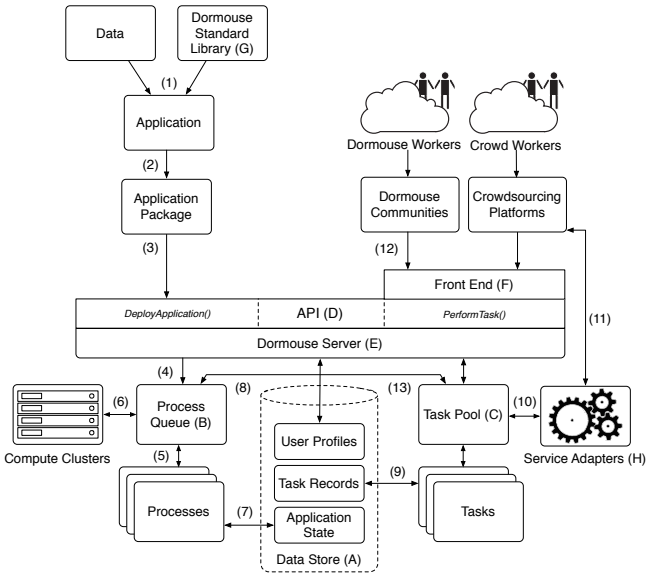

Figure 2: Internal Architecture of Dormouse

The *Dormouse Data Store (A)* is a set of distributed SQL databases that store user profiles, community information, and application-specific information and data for each running process.

The *Dormouse Process Queue (B)* schedules the top level control flow of an application, as well as any computational steps that require the Dormouse machine cluster. All Dormouse applications are registered with the Process Queue.

The *Dormouse Task Pool (C)* maintains a set of tasks that are waiting to be performed by human workers. Each task in the pool is annotated with specifications for the type of workers that are eligible to complete the task. Worker specifications are predicates represented by a binary expression tree and encoded using JSON.

The *Dormouse API (D)* is a low-level API that contains functions to process and update Dormouse system resources (for example, to register a task, to update a profile, to terminate an application, etc.).

The *Dormouse Server (E)* is a continuously running service that manages programs as they are executed, human tasks as they are dispatched, people as they join or leave, etc., by calling Dormouse API functions when appropriate. For example, when the Dormouse Server gets a request to run a new Dormouse package, it calls `add`, which adds the package to the Process Queue.

The *Dormouse Front End (F)* is a Ruby on Rails web application that serves as a UI for both workers and developers. Workers use the Frontend to manage profile information, view tasks for which they are eligible, and select and complete tasks. Developers can use the interface to manage their developer accounts and deploy their applications.

The *Dormouse Standard Library (G)* provides runtime hooks that communicate between an application and Dormouse, convenient data serialization routines, and a set of human tasks (e.g. `Label` and `Compare`) that can be called from any Dormouse application. A human task is an object that consists of a UI template[1], input parameters, and a return value.

The *Dormouse Service Adapters (H)* manage communication between Dormouse and other crowdsourcing platforms, such as Mechanical Turk. To hook into Dormouse, a crowdsourcing platform need only provide an API that minimally allows for the posting of tasks. The Task Pool uses these adapters to invoke the target service's API to post tasks.

Process Deployment

To show how these components interact with one another, we walk through deploying an application on the Dormouse architecture. Figure 2 illustrates the process.

The developer begins by writing an application that links to the Dormouse Standard Library (1), packaging the application using the Dormouse Command-line Utility (2), and deploying the package through the Dormouse Server (3).

The Dormouse Server unpacks the application, reads the manifest file to find the main executable (the program that contains the top-level control flow of the application) and sends that executable to the Process Queue (4).

The Process Queue begins running the executable (5), which may pause for one of two reasons: it either needs access to machine resources to perform a computational step or it needs to ask human workers to perform a task. In either case, the executable automatically saves its state to the Dormouse Data Store using the Dormouse Standard Library serialization routines (7), and temporarily terminates.

The Process Queue exploits the parallelism in computational steps by running them over a compute cluster (6). It does this by copying the application package to each node in the cluster and invoking the runtime hooks provided by the Dormouse Standard Library to selectively execute a single step over a subset of key-value input parameters. Once all of the steps from all the different nodes are finished, the results are sent back to the Process Queue, and then to the executable, which starts to run again.

When the executable reaches a point where no further work can be done without human input, it saves its state, terminates, and outputs a list of human tasks using JSON. The executable will be started again once the human tasks are complete.

The Process Queue takes the JSON output and sends information to the Task Pool (8). The Task Pool creates a record that includes a link back to the process, worker specification information, and the name of the Service Adapter, if any, that should be used to post tasks (9). The Task Pool invokes the "submit" method on the specified Service Adapter (10), which performs the steps necessary to post each task on the appropriate crowdsourcing platform (11). If no Service Adapter is provided, the tasks will be available on Dormouse to existing users (12).

[1]written in ERB, a template language used by Ruby on Rails

Once a task is completed, the Task Pool sends the answer back to the Process Queue (13). When all needed human tasks are finished, the process is re-executed, starting up from where it left off. The process continues until it once again needs human input or it finishes running. When the process is finished, it outputs a return code signaling the Process Queue to mark it as done and sends the developer a notification, which includes a download link to a JSON file containing the results.

MANREDUCE

In order to make Dormouse readily usable for complex data-intensive applications, we specify a programming framework called ManReduce, based on the simple functional programming paradigm and resource management scheme used by MapReduce [4]. ManReduce shares some conceptual similarities with the independently conceived CrowdForge [13], but ManReduce has some important advantages to both MapReduce and Crowdforge that we discuss below.

Design Goals

With ManReduce, while providing power and flexibility, we aimed to maintain a conceptually simple design that can be rapidly understood and easily used. In addition, ManReduce has three key features that, in combination, allow us to develop a wide range of applications for social computation: the ability to dispatch jobs both to machines and people, to utilize social structure and choose which people to whom to send jobs, and to introduce arbitrary dependencies between multiple *map* and *reduce* steps. These features, in particular, are absent in both CrowdForge and MapReduce. We discuss each core contribution below.

Conceptual simplicity and ease of use. We begin with a clean conceptual foundation for our framework. Like in MapReduce, a program is broken down into steps that are each written as a *map* or a *reduce* . A ManReduce *map* step consists of a set of small, equivalent chunks of work, performed in parallel when possible on independent inputs, producing outputs in the form of key-value pairs. A *reduce* step collects several input items (according to a shared key), and performs some computation over all of them to produce a final output. These two steps can be used to encode a wide range of parallelized computational applications.

In addition to the conceptual simplicity, our particular implementation is easy to use and allows a programmer to create full applications in few lines of code. The ManReduce framework is written in Ruby[2]. Internally, *map* and *reduce* are convenience wrappers that instantiate a Ruby `Step` object that accepts an anonymous function. A simple ManReduce script, shown in Figure 3, conducts a survey (utilizing the Dormouse function Survey). In this example, a *map* step `survey_map` sends the survey to human workers, and a *reduce* step `avg_reduce` takes the mean of their responses to each question. Notice that the user does not need to write any scaffolding code, such as parsing input or writing output. The appendix includes the full source code of `Survey` and `Average`

[2]We are also working on Python and Java implementations.

```
1  map :name => "survey_map" do |key, value|
2
3    task = Survey.prepare
4      :task_name => "Respond to survey",
5      :replication => 1000
6
7    task.ask do |answer|
8      for a in answer
9        emit(a["question"], a["response"])
10     end
11   end
12 end
13
14 reduce :step =>"Average",:name =>"avg_reduce"
```

Figure 3: Survey ManReduce Script.

ManReduce can be run simply on the command line, and it is then deployed it on the Dormouse server. Once all the tasks have been completed, ManReduce will write the output to a destination file and terminate the process. Note that ManReduce automatically serializes all worker answers, allowing us to re-run (and even debug) a script many times, automatically using the available answers to compute new (or corrected) statistics without re-doing human tasks. Thus, designing, coding, and executing ManReduce programs is straightforward.

MapReduce with both humans and machines. In the ManReduce framework, either people *or* machines can perform the work necessary for both *map* and *reduce* steps. This allows users to implement complex applications which interleave both human and machine steps.

Many computation tasks fit naturally into this hybrid ManReduce paradigm. For example, ManReduce makes it easy to specify a human-assisted information extraction system. This application has a variety of uses for large corpora of documents, where automatic information extraction cuts down on human work significantly, but alone does not always provide sufficient accuracy. A machine *map* step could specify automatic information extraction of facts from a set of scientific papers, such as genetic and environmental risk factors for a set of candidate diseases. Then, a human *reduce* step could aggregate the proposed facts about each disease, check their accuracy and convert them into a summary. The ManReduce code for this example is shown in Figure 4.

In the example explored in Figure 4, the reduce step includes asking people to aggregate and summarize a set of facts. As shown on line 12, this was accomplished by instantiating a Dormouse Task object in Ruby and calling the ask method. Custom human *map* and *reduce* steps can include any of the human tasks available through the Dormouse libraries, or custom human functions. Likewise, machine *map* and *reduce* steps can call functions from existing Ruby libraries and custom libraries, as used on line 2. In addition, the ManReduce libraries include a range of pre-defined *map* and *reduce* steps, including image labelling, ranking items from a list, and free text-entry. The built-in libraries, and the ability to define and share custom steps, provide a large and growing codebase with which to easily create new ManReduce scripts.

```
map :name => :extract_disease_facts do |key, value|
  facts = RiskExtractor.extract (value)

  for fact in facts do
    emit (fact["disease"], fact["risk_factor"])
  end

end

reduce :name => :summarize do |key, values|

  task = SummarizeFacts.prepare
    :task_name => "Summarize disease risks: #{key}"
  task.facts = values

  task.ask do |answer|
    emit (key, answer)
  end

end
```

Figure 4: Human-Assisted Information Extraction

```
task = ImagePersonTask.prepare
    :task_name => "Tag person: #{key}",
    :replication => 5

task.worker_namespace = "facebook"
task.worker_predicate = Predicate.parse(["friends CONTAINS ?", key])
```

Figure 5: User Specification with Predicate

Utilization of social structure through Dormouse. ManReduce takes advantage of the social structure and worker profiles provided by Dormouse, finding people whose attributes match those needed for a particular task. This is a natural extension from MapReduce, which allows specification of machines by their properties, such as processor speed or memory. Using functionality from Dormouse, we can specify that certain *map* or *reduce* tasks be dispatched only to people with graduate degrees in biology, or expertise in computer science, or simply to people under 25.

Adding identities and relationships opens many possibilities in human computation. For instance, accurately tagging people in photographs is important for image search engines. Using current techniques, search engines can identify a set of images and candidate names associated with each, but many pictures contain multiple people and many names correspond to several real people. The people in these photos can, however, be identified quickly and accurately by their friends. By identifying Facebook users according to each name, we could define a *map* step that asks friends of each user to judge whether an image contains their friend.

A user specification is used in ManReduce by simply setting the worker predicate and namespace properties of a Task object before it is asked. The specification is created using a Predicate object, as shown in Figure 5.

Complex ManReduce dependency graphs. Going beyond

```
map :name => "collect_survery" do |key,value|
  ...
end

reduce :name => "sort_by_gender", :from => "collect_survey"
  ...
end

reduce :name => "sort_by_age", :from => "collect_survey"
  ...
end
```

Figure 6: Complex ManReduce Dependencies

the original MapReduce model, ManReduce allows arbitrary chaining of *map* and *reduce* steps, similar to the Dryad (machine computational) framework [12]. Not only can a programmer specify multiple *map* and *reduce* steps in arbitrary order, she can also specify several different *reduce* steps that operate on the output of a single *map* step, or have a *map* step follow a *map* step directly. The ability to define functions on a generalized graph rather than a single *map* and *reduce* holds particular importance in the social computing domain. For instance, even a simple case like two sequential *map* steps may not fit naturally into single *map*, if a human step follows a machine step, or if a one human step is followed by another with different worker expertise constraints.

In ManReduce, each Step (*map* or *reduce*) may have multiple parents and multiple children. A step receives inputs from its parents and sends its output to each of its children. By default, ManReduce infers dependencies by the order in which they are defined – each Step's parent is assumed to be the Step immediately preceding it in the source code. To specify complex dependencies, we provide the `from` specifier in the `man` and `reduce` declarations, which specifies the name of the step whose output to use as input, as shown in Figure 6.

An interesting example with complex control flow is privacy-preserving journal transcription. Many researchers and designers keep handwritten notebooks, which would be useful to digitize and make searchable. Since OCR has limited accuracy with handwriting, we use human transcription, but split each page up into small chunks to make the tasks manageable and reduce the likelihood of revealing sensitive content. A ManReduce program for this application begins with a machine *map* step that "shreds" each scanned page into small overlapping images, each of which contains just a few words. A human *map* step then classifies the snippet as to whether it contains text, an image, or an equation. The result is sent to three human transcription map steps, one which is public and operates on the text, and two that are sent to people who are proficient in latex or illustrator to transcribe figures and equations. A human reduce step then votes on the best transcription for each snippet, and a machine *reduce* step then pastes the chunks from a single page back together into a PDF. Optionally, a final human *reduce* step, perhaps restricted to a trusted set of workers (e.g. people from your own institution or even just yourself) is used to rapidly ver-

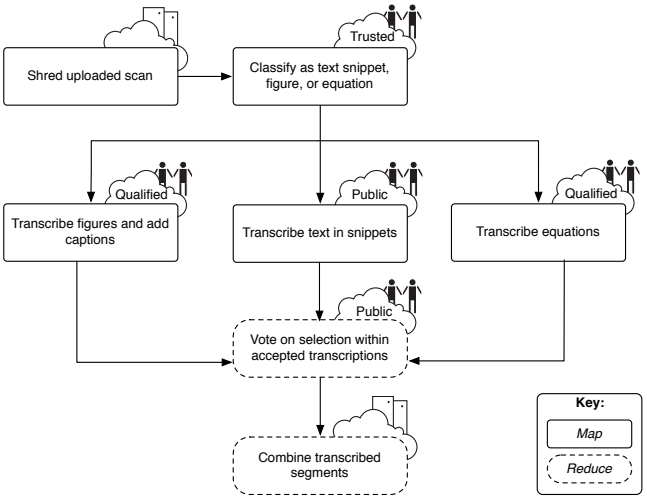

Figure 7: Illustration of dependency graph in the Journal Transcription Application

ify the complete output of several pages at a time and output the final document. The control flow for this application is shown in Figure 7.

Discussion

While ManReduce takes inspiration for its name from MapReduce, it takes inspiration for its design from a number of parallel programming frameworks, including MapReduce, Dryad [12], and GPU shader languages [22]. Each of these frameworks was developed in an environment characterized by data-intensive applications and the availability of parallel computing resources. This characterization also holds for many applications in human computation, for example, in image processing or machine learning.

At the same time, there are some applications where ManReduce is not suitable, for example, for using a single worker in the crowd to control a robot [14]. Data-processing applications are well-suited to ManReduce, while real-time or single-worker sequential applications are not.

In fact, while we believe MapReduce-based models are compelling for variety of human computation applications, we recognize that other paradigms may be more appropriate for certain tasks, or preferred by some programmers. Dormouse makes it possible to implement other such frameworks, and we hope to see a variety of paradigms for social computation implemented for use with Dormouse. These programming paradigms could take advantage not only of the features provided by Dormouse, but also the wide array of human tasks that are written for Dormouse[3].

DOG

While ManReduce is flexible and powerful, one drawback is that it can be too low level for many applications. ManReduce requires programmers to think in terms of explicit *map* and *reduce* steps, even in common cases where it is more natural to think in terms of reusing and assembling basic building blocks. It also requires some knowledge of advanced programming constructs (for example, ManReduce uses callback functions for human steps).

To address these issues, we developed Dog, a high-level procedural programming language that sits on top of ManReduce and focuses on reusability, maintainability, and ease-of-use. Our approach of writing a high-level language that compiles into ManReduce is inspired by similar techniques in the large-scale data-processing world. For example, Pig [19] and Sawzall [21] are high-level languages built on top of MapReduce, and Nebula [12] and DryadLINQ [24] are high-level languages built on top of Dryad.

Design Goals

In creating Dog, we had three main design goals. First, we wanted to make Dog a highly expressive language, so that even people with little knowledge of programming languages could understand and write a Dog program. This is an especially appropriate design goal for a social computing language, where many of the constructs involve specifying people and asking them to do something. Such constructs are understood even by non-programmers, and having a language that reflects the natural way that people express these requests would make social computation accessible to a broad audience.

Our second goal was reusability. In traditional MapReduce, many programs are written by combining pre-existing maps and reduces in ad hoc ways. In human computation, there are a large number of common patterns, for example, human verification, human voting, machine summarization of human inputs. We wanted to make it very easy for Dog programmers to express and combine these common patterns.

And finally, we wanted to achieve these goals without diminishing the efficiency, power and flexibility of ManReduce.

We achieve the first two goals as follows. We define a large number of library functions that express common human and machine functions, such as the human functions Vote, Label, Compare, Extract, and Answer, and the machine functions Histogram, Filter, Median, and Sort. Dog then contains a set of easily-understandable primitives for (a) human and machine resource allocation and (b) and parameterization and execution of these library functions.

This approach abstracts away not only the code for parallelization, but also the code for defining human or machine functions, and lets the programmer focus on defining the control flow. This is an exciting feature of Dog that will become apparent in the code samples later in this paper. Dog scripts are very easily readable and writable. Even with just the functions that are included in default Dog libraries, a programmer can write a large number of powerful programs simply and compactly.

To achieve our third goal of maintaining the power and flexibility of ManReduce, we allow programmers to write their own libraries of human and machine functions in ManReduce, and import those libraries into Dog programs. In this sense, our design of Dog gives the same power of ManRe-

[3] As an analogy, languages like Scala [18] and Clojure [10] are implemented for use with the Java Virtual Machine, allowing these languages to make use of the platform independence of the JVM as well as the function libraries written for Java.

duce behind the scenes, while still achieving our goals of expressivity and reuse.

The Dog compiler is implemented as a recursive descent parser that parses Dog programs and generates ManReduce code. The Dog standard library functions are simply wrappers around mappers and reducers in the ManReduce standard library. The Dog command-line utility includes convenience methods to compile and deploy Dog programs in a single step.

Language Specification

At its core, Dog is organized around four high-level language primitives:

`PEOPLE`, which specifices the type of people to perform some function
`ASK`, which asks a group of people to perform some human function
`FIND`, which instantiates those people
`COMPUTE`, which asks a set of machines to perform some function
as well as default libraries that include a number of human tasks (such as `Label`) and computational steps (such as `Histogram`).

For example, a simple Dog program to review UIST submissions and compute a tag cloud on words in the reviews can be written:

```
students = PEOPLE FROM facebook WHERE university =
     'mit' AND degree = 'computer science'
reviews = ASK students TO Review ON
    uist_submissions USING payment = 0 and
    replication = 3
words = COMPUTE Split ON reviews
histogram = COMPUTE Histogram ON words
```

People

The `PEOPLE` command returns a specification of people. The common use case for the `PEOPLE` command is to specify a certain type of people to perform a given task. `PEOPLE` requires a `FROM` clause that specifies the Dormouse community or crowdsourcing service from which the people will be selected. For example:

```
workers = PEOPLE FROM mechanical_turk;
```

will return a specification for mechanical turk workers.

Each Dormouse community defines properties on the people in the community, and these properties can be accessed through the `WHERE` clause. For example, a Dog programmer may write:

```
workers = PEOPLE FROM gates WHERE expertise
    CONTAINS 'theory' AND advisor='don knuth'
```

Note that the `PEOPLE` command doesn't return actual person ids; it returns a worker specification, stored internally as a predicate. The specification is instantiated when `ASK` or `FIND` is called on the specification.

Ask

The `ASK` command executes a human function. It takes as arguments a human task, a specification of people to perform it, and (optionally) a set of parameters for the human task, and (optionally) a data set on which the human task should operate.

So for example, a Dog programmer may write:

```
labels = ASK workers TO Label ON image_data USING
    layout='game'
```

Each human function has default parameters, so unless a programmer wants to change these default parameters, she can omit the `USING` clause. Further, programmers can omit the `ON` clause for human tasks that don't take input data.

Dog inherits a number of human tasks from Dormouse, for example: Vote, Label, Compare, and Answer. Programmers may also create libraries of other human functions for their own use and reuse by others in Dormouse, and import them into Dog.

Compute

The `COMPUTE` command executes a machine function. `COMPUTE` takes as its arguments a machine function, a data set upon which the function acts, and (optionally) any additional parameters required by the function. For example:

```
tag_cloud = COMPUTE TagCloud ON words USING
    color_scheme = 'random'
```

Like human tasks, the Dog standard libraries inherit a number of machine functions from Dormouse, including `Histogram`, `Average`, and `Filter`. Additionally, Dog programmers may create libraries of machine functions for their own use and reuse by others using Dormouse or ManReduce.

Find

In some cases, a Dog programmer may want to instantiate a specification of people independently of the `ASK` function. For example, she may be interested in computing summary demographic information on a given Dormouse community. The `FIND` command does this. For example, the code snippet:

```
workers = PEOPLE FROM gates WHERE expertise
    CONTAINS 'machine learning'
ids = FIND workers
```

returns the Dormouse ids of machine learning experts in the `gates` community. `FIND` may also be used to return people who have successfully performed a task. For example:

```
workers = PEOPLE FROM facebook
labels = ASK workers TO Label ON data
workers_who_labeled = FIND PEOPLE FROM labels
```

Data Model

Dog is designed to support sequential transformations on large-scale data, either by parallel human or machine functions. Dog is also designed to make it easy to express control flows that involve selecting people and specifying tasks for them to perform.

As such, Dog supports two primary data types: people specifications and data maps. A *people specification* is returned by the `PEOPLE` command, and is stored internally as a predicate[4].

[4]For example, the Dog code `PEOPLE FROM facebook`

A *data map* in Dog is expressed as a wrapper around a key-value store. Key-value stores lend themselves naturally to parallelism, and crowdsourcing is by its nature parallel. They also lend themselves well to serialization, which is an important part of Dog, especially as human steps can be expensive to re-run, and intermediate steps are often too big to fit in memory. And finally, having all data items be a key-value store helps our goal of simplicity, as new human and machine functions can be easily written in ManReduce, and the output of any data transform can be used as the input to any other data transform.

An important design feature of Dog is that all data returned by human functions retains information about who performed that function. This is a useful feature in a number of contexts, including reputation updating and routing data based on social relationships.

Routing Tasks

A key feature of Dog is the ability to route tasks to workers based on expertise, demographic, or social structure. While `FROM` handles basic routing, some applications require routing based on properties of each *individual* task or data item. The `SUCH THAT` clause enables such routing. In the following example, we show a program where students are asked to review papers in their areas of expertise, and their advisors are asked to validate them, matching the particular reviewer with her advisor for each paper.

```
students = PEOPLE FROM gates
reviews = ASK students TO Review ON
    uist_submissions SUCH THAT uist_submission.
    topic IS IN student.areas_of_expertise
reviewers = FIND PEOPLE FROM reviews
advisors = PEOPLE FROM gates WHERE advisees
    CONTAINS reviewers
validated_reviews = ASK advisors TO Validate ON
    reviews SUCH THAT review.reviewer = advisor.
    student
```

Function Libraries and Input Data

To create a library, a programmer simply creates a directory with a .doghouse extension that contains the appropriate Dormouse and ManReduce program files that define the human and machine functions. To import a library, a Dog programmer uses the `REQUIRE` command in the header, followed by a path to the Doghouse file, as well as an optional library namespace. For example:

```
REQUIRE "/dog/lib/statistics.doghouse" AS stats
```

To import a data file, a programmer uses the `IMPORT` command in the header:

```
IMPORT "/dog/lib/image_file.js" AS images
```

Other Dog Features

Dog supports a number of other commands as well. Joining of communities and data is supported through the `MERGE`, `SHIFT`, `UNSHIFT`, and `CROSS` commands. The `PARAMETERS` command is a convenience command that encapsulates parameters for the `ASK` function. The `PRINT` command prints variable values, and the `INSPECT` command prints a small selection of large maps for debugging purposes.

A convenient feature of Dog is that the primitives are composable, allowing Dog programmers to produce compact, readable code. For example, the following code snippet

```
workers =  PEOPLE FROM mechanical_turk
preferred_candidates = ASK workers TO Vote ON
    candidates
```

can be rewritten as:

```
workers = ASK PEOPLE FROM mechanical_turk TO Vote
    ON candidates
```

Composability can be arbitrarily complex, and entire programs can be written in one line:

```
COMPUTE ranking ON (ASK PEOPLE FROM
    mechanical_turk TO Vote ON candidates)
```

Debugging Environment

A challenge when writing programs with human tasks is that it can be expensive to test and debug, as human workers may need to be paid, or may get frustrated by errors in their task templates. This discourages programmers from iterative improvements to their programs. A second challenge to debugging is that many times, programs will need to run over large amounts of input data (for example, labeling a large image corpus) that take time. What programmers will often do is create separate truncated input data sets for debugging, which requires additional work.

The Dog command-line utility has a debug mode that addresses both of these issues in simple but effective ways. In debug mode, Dog programs are deployed on a local version of Dormouse, which uses the programmer's local machine as the Dormouse master, and routes tasks to the programmer or whomever the programmer specifies. Second, in debug mode, input data is automatically truncated to a reasonable size based on a number of heuristics.

This allows programmers to easily test out their control flow and human functions on a small set of people and data before deploying.

EXAMPLES

Bootstrapping Recommendations

As a case study of the Dormouse framework, we developed a prototype pipeline for personalization of product offerings (such as Groupon deals). Based on evaluating demographic preferences for deals, this pipeline would enable significantly improved targeting of deals to people, but without the need for large amounts of proprietary usage data. Small companies and brand-new services rarely have access to the volume of usage data needed for standard personalization approaches such as collaborative filtering, making this an exciting use of human-powered computation.

For this use case, we used the `facebook` community through Dormouse, allowing us to easily access demographic information (location, gender, and age) for each of the people contributing preferences, in addition to the friendship graph for people. We collected a set of 500 Groupon deals, and paired

`WHERE gender = "female"` is equivalent to the Ruby code: `Condition.new(["facebook","gender"],"female","=")`

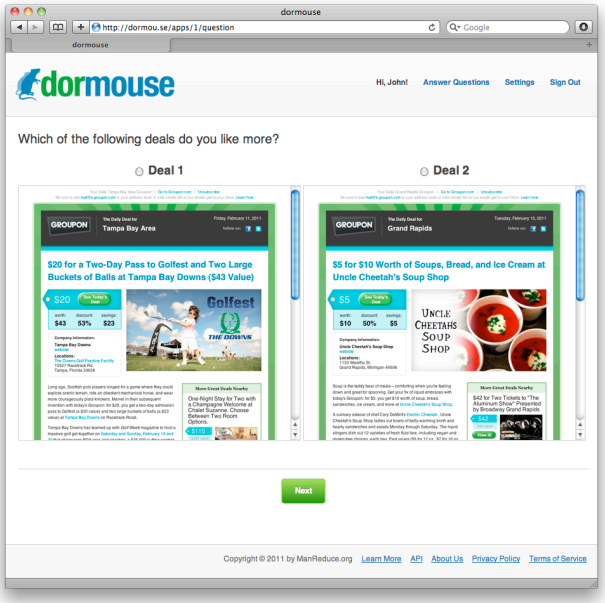

Figure 8: Deals Application

them randomly, creating 12,500 total pairs. We asked workers from the `facebook` community to state which of the two deals in a pair they preferred, and why. Three hundred workers performed an average of 42 comparisons each. Note that soliciting ratings of individual deals would be problematic, as there is no absolute scale of quality or interest. Soliciting preferences between pairs is a simple and robust way around this problem. At the end, we collected a set of preferences for any subset of people (of a certain age group, for instance), and using ordering by the number of "wins" for each item, produce an approximate ranking of all deals for that demographic. We showed ManReduce code in Figure 6 illustrating components of this pipeline, so here we show a simple Dog script that could be used, requiring even fewer lines of code (Figure 10).

This pilot study demonstrated the convenience and simplicity of the Jabberwocky framework. With just a few lines of code and minimal setup, we specified the entire pipeline. In addition, we gained evidence that product targetting could benefit from such a system, as the preferences did indeed vary noticeably by demographic. For example, the top five deals prefered by males over females, and the top five deals prefered by females over males, are shown in the table at the end of this section. Anecdotally, we noticed that the female raters preferred many deals related to beauty, fitness, and home improvement, while the male raters often preferred dining and hobbies. We also note that the male and female rankings showed low Spearman correlation ($r = 0.24$), compared with correlation between rankings segregated by a random split ($r = 0.71$). The difference between these empirical correlations is highly significant ($p < 1e-4$).

To target deals to a particular individual, we can combine the preferences according to each relevant demographic (such as "male" and "18-25" and "San Francisco") using a probabilistic noisy-or model [5]. This would combine the probability of each demographic segment liking a deal with the probabilities that an individual will agree with each of their demographics. Also, we can can incorporate a feedback loop, in which deals with uncertain ranking are re-submitted for further user feedback. Additionally, and importantly, we can incorporate the friend graph directly, asking users which deals they believe their friends would prefer, and up-weighting agreements[5].

Male-preferred	Female-preferred
Artisan Cheese Experience **	Manicure or Deluxe Mani/Pedi **
Shooting range, Dinner, Drinks **	Paddleboard Rental and Lesson **
Chiropractic, Massage, or Allergy Treatment **	Keratin Treatment **
Panoramic Wall Mural **	RedAwning.com (Vacation)**
Wine Tour and Tasting **	Custom Massage *

Figure 9: Deal preferences by gender. ** $p < .01$; * $p < .05$ by Fisher's exact method.

Medical Image Analysis

A second application of Jabberwocky is to use people to improve the performance and evaluation of a machine learning framework for complex medical data. This case study explores the role of expertise in human computation.

In medicine, a common method of clarifying the subcellular location of proteins is immunohistochemical staining, where flourescently tagged antibodies are introduced into a tissue, binding with the protein the scientist wants to localize. The resulting images are then evaluated by pathologists. This technique is used in a wide range of applications, from exploring gene function in Parkinson's [25] to diagnosing cancerous tumors.

There are large quantities of immunohistochemical stains generated each year. Because of this, some researchers have started to explore image-recognition techniques to analyze these stains. However, these techniques are not often used in practice, because the idiosyncracies in how staining methods work make it difficult for a single image-recognition algorithm to work across many different stains.

We used Dog to write a program that routes IHC stains to generalists to localize the stains via a human image-segmentation function in Dormouse (Figure 11), and then routes those non-expert localizations to experts to perform a faster validation step. The expert validation feedback was then rerouted to the original generalists, who could use the feedback to improve. The validated localizations can then be used as input into a machine learning algorithm. An extension of this would be that as the generalists got more right, they increased in reputation in the system.

What is interesting is that what we originally built as a tool to aid machine learning ended up also aiding human learning. One can imagine that such systems can get even more nuanced, with active learning algorithms that have input as to what labeling tasks to route to which people. We call this *social machine learning*, where social learning systems interact with the machine learning systems, and both benefit from

[5] reminiscent of the Newlywed Game.

```
#!/usr/bin/env dog

IMPORT "deals.js" AS deal_pairs
REQUIRE "rank_deals.doghouse"

CONFIG title = "Compare Deals"
CONFIG description = "..."

answers = ASK PEOPLE TO RankDeals ON
    deal_pairs

age = COMPUTE Projection ON answers USING
    key_name = "age" AND value_name = "deals"
gender = COMPUTE Projection ON answers USING
    key_name = "gender" AND value_name = "
    deals"
ethnicity = COMPUTE Projection ON answers
    USING key_name = "ethnicity" AND
    value_name = "deals"
education = COMPUTE Projection ON answers
    USING key_name = "education" AND
    value_name = "deals"

age_ranking = COMPUTE PairwiseRank ON age
gender_ranking = COMPUTE PairwiseRank ON
    gender
ethnicity_ranking = COMPUTE PairwiseRank ON
    ethnicity
education_ranking = COMPUTE PairwiseRank ON
    education
```

Figure 10: Deals Application in Dog

the other. We believe that this will be an area with much opportunity.

RELATED WORK

A number of programming frameworks for human computation have been introduced in recent years. Crowdforge [13], a MapReduce-inspired framework that was developed simultaneously but independently from ManReduce, defines *partition*, *map*, and *reduce* steps, and allows nested *map* and *reduce* steps. TurKit [16] is a toolkit for deploying iterative tasks to Mechanical Turk that maintains a straighforward procedural model. Soylent [2], a word-processing interface that calls Mechanical Turk workers to edit parts of a document on demand, introduces the Find-Fix-Verify crowd programming pattern, which splits tasks into a series of generation and review stages. Each of these specifies and implements a design pattern rather than building a full stack, and are platform dependent.

Recently, a trio of declarative query languages for human computation have been proposed: hQuery [20], CrowdDB [6], and Qurk [17]. These languages view crowdsourcing services as databases where facts are computed by human processors. These languages are different from Dog in that they are declarative rather than imperative. The imperative model of Dog is particularly important for social computation, where we want the programmer to be able to specify the type of person who will compute the result, not just the desired outcome.

Heymann et. al propose hProc [9], a programming environment that focuses on modularity and reuse. It shares with Dormouse the notions of easy reuse of human function templates, and of abstracting out the specific crowdsourcing ser-

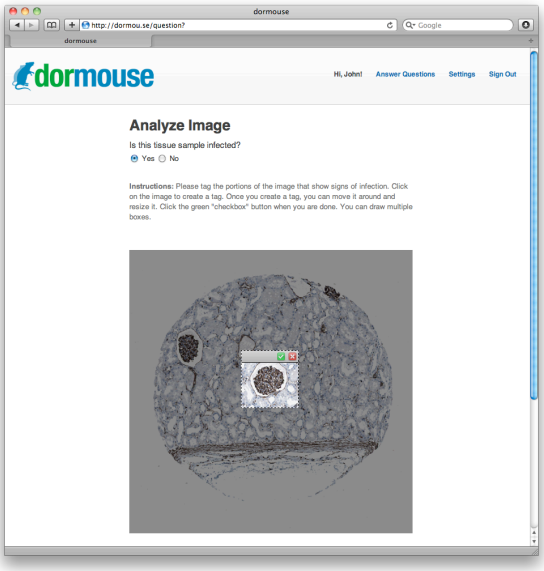

Figure 11: Immunohistochemical Staining Application

vice through its marketplace drivers. However, there are also many key differences; for example, hProc does not have mechanisms for interleaving human and machine computation or abstracting away details of parallel processing. More substantively, since hProc has not been implemented, the design decisions are described as high-level proposals.

Much of the Jabberwocky software stack has taken its inspiration from related constructs in traditional and parallel computing. The Dormouse Virtual Machine is inspired in part from the Java Virtual Machine [15], a platform-independent execution environment that converts Java bytecode into machine language and executes it. In practice, Dormouse is not a true virtual machine in that it operates on top of crowdsourcing platforms rather than microprocessor architectures. In this sense, it is perhaps more reminiscent to Google's Global Workqueue, or some of the cluster management protocols used in scientific computing such as Parallel Virtual Machine [15] or MPI [7]. ManReduce takes its inspiration from MapReduce [4], Dryad [12], and GPU shader languages [22]. And Dog takes its inspiration from Pig [19], Sawzall [21], Nebula [12] and DryadLINQ [24]. The popularity of these existing tools suggests that the paradigms we present here will be useful and natural for many programmers.

CONCLUSION

To date, the programming frameworks for crowd computing have been single-platform frameworks. Further, the programming frameworks for crowdsourcing have viewed the crowd as a collection of largely independent and interchangeable workers, rather than an ecosystem of connected, heterogeneous people. And finally, despite a clearly differentiated domain, no domain-specific programming languages have been developed for social computing, requiring programmers to define control flows for people in languages designed for computers.

The Jabberwocky software stack represents a step forward in

the tools available to programmers for social computation. A programmer may deploy a non-trivial application in Dog without having to build labor-intensive sociotechnical infrastructure, allowing developers to easily tap into people and their heterogeneous skillsets in an organized manner. Jabberwocky puts a wide range of possibilities for data-intensive applications within reach of a broad class of developers, and we believe it holds the potential to change the way programmers interact with people using code.

REFERENCES

1. S Bamford and et al. Galaxy Zoo.
2. M. Bernstein, G. Little, R.. Miller, B. Hartmann, M. Ackerman, D. Karger, D. Crowell, and K. Panovich. Soylent: a word processor with a crowd inside. In *Proc. UIST (2010)*.
3. S Cooper, F. Khatib, A. Treuille, J. Barbero, J. Lee, M. Beenen, A. Leaver-Fay, D. Baker, and Z. Popovic. Predicting protein structures with a multiplayer online game. *Nature*, June 2010.
4. J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Communications ACM*, January 2008.
5. FJ Diez. Parameter adjustment in bayes networks. the generalized noisy or-gate. In *Proc. UAI (1993)*.
6. Michael J. Franklin, Donald Kossmann, Tim Kraska, Sukriti Ramesh, and Reynold Xin. CrowdDB: answering queries with crowdsourcing. In *Proc. SIGMOD (2011)*, pages 61–72.
7. William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 1996.
8. B. Hartmann. Amazing but True Cat Stories. http://bjoern.org/projects/catbook/, April 2009.
9. P. Heymann and H. Garcia-Molina. Human processing. Technical report.
10. R. Hickey. The clojure programming language. In *Proc. DLS (2008)*.
11. D. Horowitz and S.D. Kamvar. The anatomy of a large-scale social search engine. In *Proc. WWW (2010)*.
12. M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. *SIGOPS Operating Systems Review*, March 2007.
13. A. Kittur, B. Smus, and R. E. Kraut. CrowdForge: Crowdsourcing Complex Work.
14. W. Lasecki, K. Murray, S. White, R. Miller, and F. Bigham. Legion: closed-loop crowd control of existing interfaces. In *Proc. UIST (2011)*.
15. T. Lindholm and F. Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., 2nd edition, 1999.
16. G. Little, L. Chilton, M. Goldman, and R. Miller. TurKit: Tools for Iterative Tasks on Mechanical Turk. In *Proc. HCOMP (2009)*.
17. A. Marcus, E. Wu, S. Madden, and R. C. Miller. Crowdsourced Databases: Query Processing with People. In *Proc. CIDR (2011)*.
18. M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Maneth, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger. An overview of the Scala programming language. Technical report, EPFL Lausanne, Switzerland.
19. C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: A Not-So-Foreign Language for Data Processing. In *Proc. SIGMOD (2008)*.
20. A. Parameswaran and N Polyzotis. Answering Queries using Humans, Algorithms and Databases. Technical report.
21. R. Pike, S. Dorward, R. Griesemer, and S. Quinlan. Interpreting the data: Parallel analysis with Sawzall. *Scientific Programming*, October 2005.
22. D. Tarditi, S. Puri, and J. Oglesby. Accelerator: using data parallelism to program GPUs for general-purpose uses. *SIGOPS Operating Systems Review*, October 2006.
23. L. von Ahn. Games with a Purpose. *Computer*, June 2006.
24. Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. Gunda, and J. Currey. DryadLINQ: a system for general-purpose distributed data-parallel computing using a high-level language. In *Proc. OSDI (2008)*.
25. L. Zhang, M. Shimoji, B. Thomas, D. Moore, S. Yu, N. Marupudi, R. Torp, I. Torgner, O. Ottersen, T. Dawson, and V. Dawson. Mitochondrial localization of the Parkinson's disease related protein DJ-1. *Human Molecular Genetics*, June 2005.

Appendix

The source code of `Survey` and `Average` from Figure 3.

survey.rb:

```
1  class Survey < ManReduce::Task
2   def render
3    include_file("survey.erb")
4   end
5
6   def process_response(response)
7    answers = []
8    answers << {"rating" => response["rating"]}
9    answers << {"length" => response["length"]}
10   return answers
11  end
12 end
```

survey.erb:

```
1  <label>
2    How long have you been using the service?
3  </label>
4  <input type="text" name="length">
5
6  <label>
7    How is the quality of your service?
8    (10 is good, 0 is bad)
9  </label>
10 <input type="text" name="rating">
```

average.rb:

```
1  class Average < ManReduce::Reduce
2   def reduce(key, values)
3    count = values.length
4    sum = values.inject(0) {|sum, x| sum += x}
5    if count == 0 then
6     emit(key, 0)
7    else
8     emit(key, sum / count)
9    end
10  end
11 end
```