Enabling an Enhanced Data-as-a-Service Ecosystem

Michael Smit, Mark Shtern, Bradley Simmons, and Marin Litoiu
York University, Canada
{msmit|bsimmons|mlitoiu}@yorku.ca *and* mark@cse.yorku.ca

***Abstract*—The sharing of large and interesting Big Data in cloud environments can be achieved using data-as-a-service, where a provider offers data to interested users. In *enhanced data-as-a-service*, the data provider also supplies compute infrastructure, allowing users to run analytics tasks local to the data and reducing the (expensive and slow) transmission of data over networks. This paper describes a services-based ecosystem that allows providers to precisely share portions of their data with users, using a model where users submit MapReduce jobs that run on the provider's Hadoop infrastructure. Providers are given mechanisms to filter, segment, and/or transform data before it reaches the user's task. The ecosystem also allows for intermediaries who offer value-added filtrations, segmentations, or transformations of the data (for example, pre-filtering a dataset to only include high-income users). We describe the RESTful services required to enable this ecosystem, introduce a prototype to demonstrate the concept, and present experiments using this ecosystem to both provide and analyze different segments of a single large data set.**

***Index Terms*—Big Data, service system, RESTful services, data-as-a-service**

I. Introduction

As data is generated in larger volumes, interest is growing in determining how best to process and extract useful, actionable knowledge from large data sets. This so-called *Big Data* is important to many domains, including biological and life sciences [1], [2], the physical sciences [3], and digital humanities [4].

One approach to distributed large-scale data processing is MapReduce [5], often in the implementation included in the Hadoop[1] system. Although Hadoop and the MapReduce paradigm offer users substantial power, there is an investment of both time and capital to create a Hadoop infrastructure. Acquiring computation resources; installing Hadoop; loading, transmitting, and managing large scale data; and navigating the data storage options (Hadoop file system, HBase, Amazon S3) all require technical ability beyond the skills needed to author MapReduce jobs. Some elements of this complexity are addressed by cloud services; using a cloud-based Hadoop infrastructure (e.g. Amazon Elastic MapReduce) or a tool to simplify installation (e.g. Cloudera) can streamline some parts of the process, but only for users with a strong understanding of Hadoop.

In previous work [6], we introduced an approach to precision, segmented sharing of Big Data in a cloud environment, while ensuring compliance with the information and privacy policies of the sharing organization. A user submits a MapReduce job to the provider to be executed; this job's access to data is overseen by a series of *modifying maps* that can filter and transform data before it reaches the user's job. The original data provider may filter out private information, or transform data from a proprietary format to one more usable. Various other data *resellers* may filter data to produce restricted subsets that are made available at reduced cost, or simply to reduce computation by pre-filtering data not relevant to the user. All data processing occurs on the provider's infrastructure, as a service, with only the results transmitted to the user. In addition to the security and privacy benefits, there is a substantial improvement in usability, as significant technical complexity is assumed by the provider, reducing the technical knowledge required.

We call this set of participants and their interactions the *DaaSPatcher ecosystem*. It is enabled by a suite of services implemented by each participant in compliance with a set of specifications. These standardized RESTful services enable loose coupling and runtime composition, allowing users to submit sophisticated data processing jobs to providers and resellers without systems integration effort. The ecosystem includes a marketplace that provides a straightforward interface for non-technical users to find and launch MapReduce jobs on available datasets.

In contrast to previous work which focused on the provider's data segmentation and privacy aspects of this data-sharing model, this paper describes the services that enable the user-facing portions of the ecosystem. We discuss the requirements of these services, the service specifications each participant must comply with, and our implementation of these specifications in a functional prototype implementation. We use the illustration of a data provider offering access to a set of past tweets from Twitter, and various resellers offering (at lower cost) subsets of that data: only tweets in English, only tweets from popular users, etc. Of particular interest is the ability to compose multiple resellers at run-time; for example, composing a reseller removing non-English tweets with one removing unpopular users would create a "new" dataset, English tweets from popular users, without requiring a second copy of the data, without requiring human intervention, and without requiring a provider or reseller to explicitly offer such a dataset.

We begin by describing the data sharing ecosystem in more detail in §II, then describe the services each participant in the ecosystem must implement §III. We introduce our proof-of-concept implementation and demonstrate it using a Twitter dataset (§IV), before offering thoughts on future directions (§V) and the relationship to related work (§VI). Finally, we

[1]http://hadoop.apache.org

conclude the paper in §VII.

II. The DaaSPatcher Ecosystem

Our approach [6] allows *analytics users* to run MapReduce (MR) jobs on some portion of a *data provider*'s Big Data, while affording the data provider total, fine-grained control over access to each piece of data, and allowing run-time transformation of the data. *Resellers* can act as intermediaries in the interaction, offering their own value-added transformations or subsets of the provider's data before it is processed by the user's MapReduce job. These various modifications of an original data set are referred to as *data sources*.

This run-time mediation is provided by prefixing the user's MapReduce job with an additional Map step (a Map-Map-Reduce job) where the provider can implement access control, data segmentation, and/or data transformation. This *Modifying Map* can also control the user's Map task at a low-level, including measuring or limiting execution time. The analytics user is one interested in performing analytics tasks on large-scale data in the cloud, a task of significant interest [7], [8], perhaps for providing services to end users [9]. Conceptually, the provider and reseller maps mediate the user's MapReduce access to data (Fig. 1a).

A data provider may choose to delegate the actual processing to an *infrastructure provider*, which stores a copy of the data on an infrastructure separate from the data provider's production environment, but updates with sufficient frequency to be considered perpetually current. For example, Twitter has delegated most public access of all Tweets to three certified providers[2]. A data provider may also keep such a copy in-house, or it may be acceptable to allow outside access to a single copy of the data. We'll use the term *provider* to refer to an entity responsible for accepting and running MapReduce jobs, regardless of the actual ownership of the data.

The data provider identifies the data sources they wish to provide, and implements a Modifying Map to enforce their information policies (access control), to offer different views on the same data (transformation), and to offer valuable selective subsets of the data (segmentation). They then provide a RESTful API to accept incoming jobs to run on those data sources.

The analytics user submits their compiled code to the provider's web service along with any required parameters. They monitor the status of their job or retrieve the results through the same web service. The user can run their own client for communicating with the web service, or use a client offered through a Software-as-a-Service (SaaS) delivery model, where they submit and monitor jobs through a user interface with the actual communication handled behind-the-scenes.

A *reseller* offers further segmentations or transformations of the data set. They may establish relationships with providers and sell access to the provider's infrastructure, accepting MR jobs from users and running them on the provider. A reseller could offer additional segmentation or transformation to produce value-added data sets, or smaller, more affordable data sets. In the Twitter example, one reseller might segment Tweets by estimated household income based on geographic information; another might augment Tweets with a popularity metric; a third might sell subsets of the overall data set where only Tweets mentioning politics or certain products are included. An analytics user could choose one of these smaller data sets to reduce costs. Resellers can be chained together in (theoretically) unlimited series; for example, a fourth reseller might sell segmented access to the first reseller's Tweet+Income data set, by income tax bracket.

To achieve this second (and third, and nth) layer of runtime data mediation, resellers add their own Modifying Maps between the provider's Modifying Map and the user's Map. Because the data provider is the sole arbiter of which data is passed to the reseller, and the reseller then decides which data is sent to the user, each participant retains the control they need. If any provider or reseller determines the consumer of their service should not have access to a given data record, the Map code is never invoked on that data record. Each `map()` invocation may transform the data from the original key-value pair provided to the provider's map method.

A reseller offers the same API as all the providers, allowing resellers to be easily composed with providers and other resellers. Incoming compiled Map code is augmented with the reseller's Modifying Map, then passed to the next reseller in the chain (or the provider) via their API. Requests for status updates or results are similarly passed on, and the result returned to the requester.

There is also the option of including *enhanced resellers*, which host their own infrastructure, acquire data from multiple providers, and run MR jobs directly on their infrastructure where this data is aggregated, filtered, or otherwise combined and transformed. In essence, they act as both a provider and a reseller. Managing these multiple roles is the responsibility of the reseller; throughout this text, when we refer to providers we include Enhanced Resellers when in their provider role, and likewise for reseller and user roles.

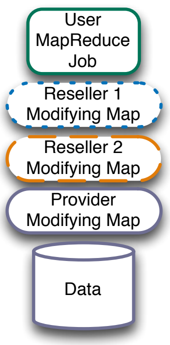
Finally, using a marketplace, users can discover available data sources and submit jobs to them easily, and providers can authorize users to run MR jobs without knowing all of the details about the user or having an established relationship. This is enabled by Attribute-based Access Control (ABAC), as described in [6]. In an open data environment, the ABAC mechanisms can be omitted.

III. Ecosystem Services

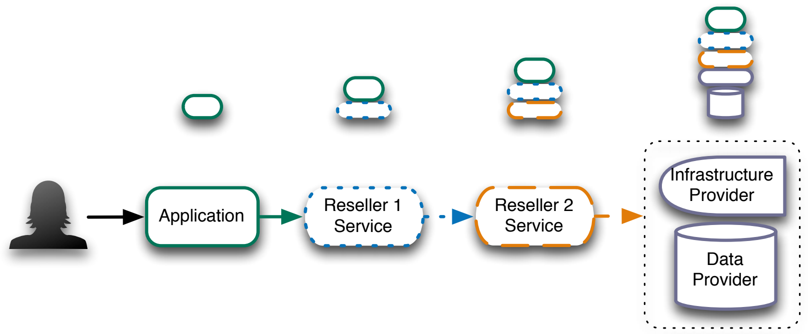
The DaaSPatcher ecosystem includes several participants, each implementing a RESTful service that complies with the given service specification. How these services may be composed, and how the service interactions relate to the conceptual filters, is shown in Fig. 1b. The specifications themselves are expressed in the Web Application Description Language (WADL)[3] but will be described in this section more

[2]https://dev.twitter.com/programs/twitter-certified-products/products

[3]http://www.w3.org/Submission/wadl/



(a) A conceptual view of how a user's Map Reduce job accesses data: only through the filtering code provided by the various resellers.



(b) Tracing how the conceptual view is built over the life of a service composition. Note that each reseller is optional, and there could be many resellers. All computations occur local to the data, with the results passed to the user.

informally for each participant in the ecosystem.

A. Provider

The data (or infrastructure) provider offers mechanisms to discover available datasets, to submit and track jobs submitted, to retrieve the results from completed jobs, and to view the status of the server. All service responses are sent in JSON unless otherwise specified, and the WADL included in the specification is made available to clients. No requirement is expressed regarding the implementation, and additional services may be included in the implementation; the required resources are the following:

GET /datasources: This resource lists all available data sources with metadata about each resource, in the form of a JSON array of JSON objects. Each object includes the unique name of the data source (a name must be unique to the provider but not globally unique), the interface that client code must implement when defining their Map step and the package this implementation must belong to, a plain-text description of the data source, and the price of running jobs. Prices may be expressed as a fixed rate, per CPU-hour, or per GB. Optionally, the provider may include the total size of the dataset, and a pointer to a data source with a subset of the data for free testing of MapReduce jobs.

To enable ABAC protection, the provider must also include a set of attributes they require users[4] will be used to submit to verify their identity and their permission to access the data. These attributes come from various authorities and are signed to verify authenticity. For example, a provider might require the attribute `facebook.age`, the user's age from the authority Facebook, which the provider might user to filter out users under the age of 13. Optionally, the provider may specify a similar set of attributes that the marketplace requires and verifies before providing access to metadata about the data source.

[4]Throughout this section, we use "submitter" to refer to whichever entity is submitting a request to the given service; recall that this may be the analytics user or the reseller. When "user" is used, it refers specifically to the analytics user.

POST /jobs/submit: The job submission endpoint expects a file upload (containing the MapReduce code to run, including a class in the package specified implementing the interface specified by the provider). It is expected the file is typically a jar file, and the provider must at minimum accept jar files, but this is not an explicit restriction. It also requires three parameters: `jobname`, the submitter's unique identifier for the job; `jobreduce`, the submitter's Reduce class; and `jobdatasource`, the provider-assigned name for the datasource on which to run this job.

The provider is responsible for accepting and processing the uploaded file, including injecting its own Modifying Map around the Map task, and submitting the job to its Hadoop infrastructure. It must retain an association between the Hadoop job and the submitter-assigned job name, in order to handle subsequent requests from the submitter referring to the job.

The response in the event of a successful submission is an HTTP status of 200, with the content a simple JSON object with `jobsubmitted` set to the value of the job name. In the event of an error, an HTTP status of 400 (various submitter input errors), 500 (server error), or 401 (permissions error) will be returned, along with explanatory text.

GET /jobs/status/{jobname}: For the given `jobname` in the URI, this endpoint returns the basic status of the job as a JSON object with one key, `status`, with the value one of PENDING, EXECUTING, COMPLETE, or FAILED. A pending job has not yet been submitted to Hadoop; an executing job is currently running on Hadoop; a complete job has been successfully run on Hadoop; and a failed job has encountered an error from which no recovery is possible, and the job should be submitted.

In the event of an error, an appropriate HTTP status code and explanatory message will be returned (400 for a missing or invalid jobname, 404 when the jobname does not exist, 500 for server errors, 401 for permissions errors).

GET /jobs/tracker/{jobname}: Given the `jobname`, this service returns information about the job in HTML format for display. Is is intended to offer features similar to the Hadoop job tracker, which allows submitters to track the progress of their job (percent complete, resources used, etc.). The exact information supplied is up to the provider. It is not permitted to request this resource for a job with the status PENDING. The tracker is expected to offer more information describing the failure for jobs with status FAILED. Errors will be HTTP status codes and explanatory text similar to the `/job/status/jobname` endpoint.

GET /jobs/results/{jobname}: Once a job reaches the COMPLETE status, it is valid to request this resource. Prematurely requesting this service (or errors on user input) will produce errors similar to the previous services. The result of a successful request will typically be of type `text/plain`, though if an alternate type is obvious given the data source, the alternate is permitted. The response may be very large depending on the job submitted.

GET /server/status: This resource returns the status of the server in JSON format, with the key `status` and a value of HEALTHY, AILING, or CLOSED required. Optional keys are `message` for more information, `estimated_wait` for the estimated queue time, `utilization` for the current load of the server, and `next_maintenance_window` to identify the next scheduled down time. Submitting jobs to a CLOSED service is not permitted and to an AILING service is discouraged. The provider determines in what circumstances their server is AILING.

B. Reseller

The reseller provides the same services as a provider, though the requirements of the implementation are slightly different. The reseller is an intermediate service in a composition, in contrast to the user and provider who are by definition at the beginning and end of the composition. The reseller receives inbound requests from submitters, invokes the next service in the composition and retrieves a response, then returns a response to the submitter.

GET /datasources: As for providers, available data sources offered by the provider with all associated metadata are provided here, in the same format. The key difference is internally, the reseller must maintain an association between the data sources they offer, and which data source they are transforming / segmenting / filtering; that is, the next reseller or provider in any composition.

POST /jobs/submit: Like the provider, the reseller accepts the uploaded code, checks it for compliance with the specification, and then injects their own Modifying Map. They may create a new `jobname`, for example by prefixing the submitter's `jobname` with the user ID of the submitter. They pass on the submitter's Reduce class without modification. Finally, they use their map of incoming data source names to which reseller/provider the job should now be submitted.

GET /jobs/(status | results | tracker)/{jobname}: The `jobname` must be mapped to the next reseller/provider in the composition *for that job*, a request sent to that entitty (with a modified `jobname` if required), and the response returned to the submitter. As each reseller in the composition will pass on the request, ultimately the response will be provided by the provider.

GET /server/status: For the reseller, a parameter `datasource` is required for this request, and the contents of this resource from the next reseller/provider in the composition for that data source is returned.

C. Marketplace

A marketplace may provide many services and integrate with other APIs and services as needed; here we focus on those required to establish an ecosystem. Some services (e.g. user login)

POST /datasources/register: A provider defining a new datasource must register their endpoint URI with the marketplace. The minimum requirement is the URI; other optional parameters may be included, such as the requirements of the marketplace on revenue sharing. This service is not rigorously specified as it is not required for runtime composition; a trust relationship must be established, which may build on existing approaches to the automated establishment of trust or be established offline.

POST /authorities/register: Authorities for the ABAC approach to managing access rights register using this service. Again, offline trust establishment is required; as each authority provides their own API and they are not subject to these specifications, the mechanisms for verifying information acquired from an authority are not described here in detail.

IV. Implementation & Demonstration

As a proof of feasibility, we implemented the specification given in §III to create the DaaSPatcher ecosystem. This implementation goes beyond the implementation described previously [6], with a new motivating example and set of experiments.

A. Provider

In this implementation, we consider a single provider, with a repository of tweets to which they wish to provide access. Users are typically not interested in the entire data set, and may not have the resources to purchase, store, or process that volume of data. By providing access to segmented portions, the provider expands their user base; they delegate the details of this task to resellers who can imagine many different filters. This allows the provider to focus on their core competencies (gathering tweets), and allows them to delegate the imagining of interesting filters to resellers. The users gain access to a resource that would otherwise not be accessible to them, without requiring up-front investment in infrastructure, and with faster processing time than processing the entire data set [6].

We implemented the provider's services according to the given specifications, using the PHP Slim Framework[5]. The results of the `GET /datasources` command are shown in Figure 1. Applications using this framework list the REST resources they offer, and define a *handling function* to process

[5]http://www.slimframework.com/

```
[{
    "name": "AllTweets",
    "interface": "org.myorg.MapInterface",
    "interfaceUrl": "http://ceraslabs.com/
        hadoop/MapInterface.java",
    "package": "org.myorg",
    "description": "A set of tweets collected
        in March 2013, including approximately
         50 million tweets.",
    "accessAttributes": [
        "*.emailaddress"
    ],
    "price": {
        "amount": ".04",
        "model": "dollars/cpu-hour"
    }
}]
```

Fig. 1: JSON returned for a provider defining a single data source.

requests arriving at that resource. Resources are identified by URIs, where each element of the URI can be a string constant or a variable. The variable elements of the API are passed as arguments to the handling function. For example, the line `$app->get('/jobs/status/:jobname', 'get_job_status')` instructs the framework to invoke the function `get_job_status()` when it receives requests of the form `GET /jobs/status/{jobname}`, passing the value of `{jobname}` as an argument. This framework has previously been used successfully to offer cloud-related web services [10].

The provider service connects to a Hadoop infrastructure launched using Cloudera's Distribution Including Apache Hadoop version 4 (CDH4). The `POST /jobs/submit` resource unpacks the submitted JAR file, and creates a few file with the submitter's Map code wrapped in the provider's Map code. The CDH4 client is installed on the same machine as the web service, and used to submit jobs to the configured JobTracker for the running Hadoop installation[6]. The other options provided to the service are validated and passed as arguments to the Hadoop job.

We used a MySQL database to store persistent information, such as the mapping from submitter's job names to Hadoop jobs. Whenever a job is launched, all of the information about that job is persisted to the database for (potential) future use, including the Hadoop-assigned id. The database also stores information about configured data sources, and is automatically queried to produce the response to `GET /datasources` using a previously-authored extension to the Slim Framework [10].

To respond to `GET /jobs/status`, the job progress is retrieved and parsed, with the correct status returned based on the presence or absence of keywords in the Hadoop JobTracker. The response to `GET /jobs/tracker` is the HTML produced by the Hadoop JobTracker for that specific Hadoop job, with sensitive information removed. The results are returned in response to `GET /jobs/results` by reading the results file(s) from the Hadoop Distributed Filesystem (HDFS) and passing the contents through to the client (never reading the entire contents into memory).

[6]The exact configuration of the Hadoop infrastructure varies based on expected load and the location and size of the data; our case study describes in more detail the properties of the infrastructure used

B. *Reseller*

The reseller's service is also implemented using the Slim Framework, but behaves differently. As before, a MySQL database maintains a list of datasources offered (from which the `GET /datasources` response is constructed), but also includes a mapping to the information of the reseller or provider whose data is being resold. This mapping is used to correctly route incoming submitter requests.

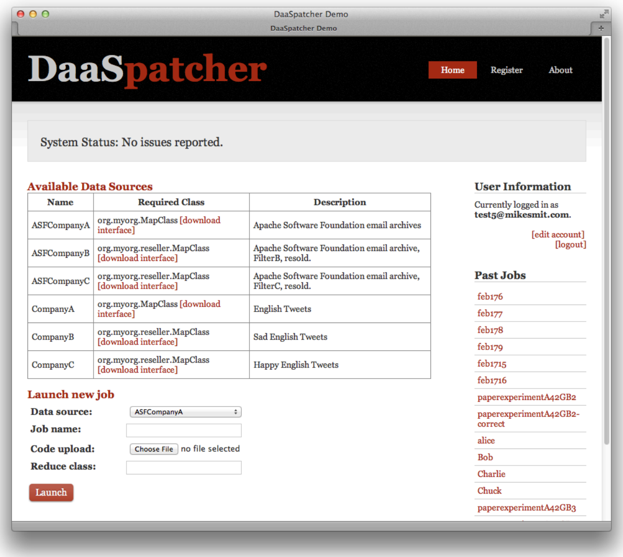
When jobs are submitted, the reseller extracts the contents of the JAR file, wraps the Map class with their Modifying Map, and passes it to the submission endpoint of the next reseller/provider with the same job name. This job name is persisted to the database along with the correct endpoint of the next reseller/provider for that job. For the job status/tracker/results requests, the appropriate resource is requested from the given endpoint, and the response returned exactly to the submitter.

While the provider installation requires additional complexity, the reseller service is implemented in approximately 120 source lines of code.

Modifying Maps: The reseller offers data sources that are segmented, filtered, or in general transformed from the provider's data source (or from another reseller in the chain). We implemented a series of Modifying Maps that can be composed in various ways at runtime to produce refined subsets of the Twitter dataset offered by the tweet provider in our case study.

- English: Returns tweets where the user's language is set to 'en'.
- StrictEnglish: Starts by testing the user's language setting ('en'), but also discards tweets with characters not in ASCII. This filter will produce false negatives, as our examination of the raw data suggests english speakers do make some use of the full range of UTF-8 characters.
- Happy: Returns all tweets containing happy emoticons like :), :D, :P, and variations as a rudimentary sentiment analysis.
- Sad: Returns all tweets containing sad emoticons, like :(, :|, and variations as a rudimentary sentiment analysis.
- Popular: returns tweets from people with more than 1,000 followers.
- TweetToText - transforms the input from Twitter JSON to simple text: the @ sign, the username, a colon, and the status in plain text.

One can then compose resellers offering these Modifying Maps to produce variations to the original data set not explicitly envisioned by the data provider; for example, to produce Tweets from English-speaking users with more than 1000 followers that are happy, in text format (UserMapClass $\leftarrow$



System Status: No issues reported.

Available Data Sources

Name	Required Class	Description
ASFCompanyA	org.myorg.MapClass [download interface]	Apache Software Foundation email archives
ASFCompanyB	org.myorg.reseller.MapClass [download interface]	Apache Software Foundation email archive, FilterB, resold.
ASFCompanyC	org.myorg.reseller.MapClass [download interface]	Apache Software Foundation email archive, FilterC, resold.
CompanyA	org.myorg.MapClass [download interface]	English Tweets
CompanyB	org.myorg.reseller.MapClass [download interface]	Sad English Tweets
CompanyC	org.myorg.reseller.MapClass [download interface]	Happy English Tweets

Launch new job

Data source:	ASFCompanyA
Job name:	
Code upload:	Choose File no file selected
Reduce class:	

Launch

User Information

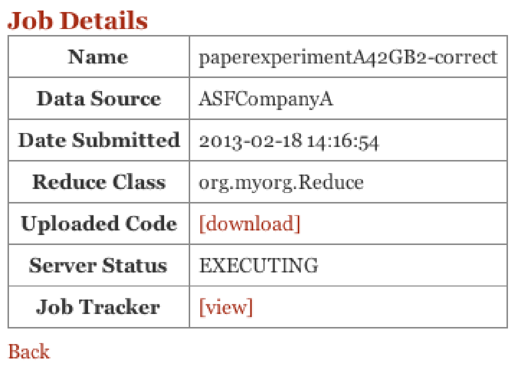
Currently logged in as **test5@mikesmit.com**.

[edit account]
[logout]

Past Jobs

feb176
feb177
feb178
feb179
feb1715
feb1716
paperexperimentA42GB2
paperexperimentA42GB2-correct
alice
Bob
Charlie
Chuck
paperexperimentA42GB3

Fig. 2: On view of the user interface offered by the Marketplace, showing available data sources, past jobs, and a form to launch new jobs.



Job Details

Name	paperexperimentA42GB2-correct
Data Source	ASFCompanyA
Date Submitted	2013-02-18 14:16:54
Reduce Class	org.myorg.Reduce
Uploaded Code	[download]
Server Status	EXECUTING
Job Tracker	[view]

Back

Fig. 3: The job details portion of the user interface.

TweetToText ← Happy ← English ← Popular ← AllTweets). The order of the composition affects the order of the Modifying Maps that wrap the user's map; the outer-most filter (Popular, in this case) sees all of the tweets, and therefore should be the one that filters the most tweets and is the most efficient at doing so.

Based on the actual compositions used by users, resellers may offer "shortcut" data sources that combine the filters into a single Modifying Map for slight performance gains. In our experiments, we used these aggregate Modifying Maps.

C. Marketplace / User Interface

The marketplace registers and stores data source information, persisting to a MySQL database. We registered the various reseller data sources and the provider data source with the marketplace.

In our implementation, the marketplace is also the user interface for analytics users. A web user interface allows users to access the analytics-as-a-service offering using a software-as-a-service client. Through this interface, users can browse available data sources, submit jobs, track job progress and status, and retrieve job results. Users are asked to provide a valid email address and enter their country when registering, and must click a confirmation link in the email to activate their account. These two pieces of information (email address, country) are the only attributes available for access control; in this proof-of-concept, distributed authorities are not included. The marketplace retains information about all submitted jobs, offering features like submitting the same MapReduce code to another data source and remembering past job names to users track jobs. A screenshot is shown in Figure 2; the user may submit new jobs at the bottom of the page, and use the history to the right to track old jobs (Figure 3).

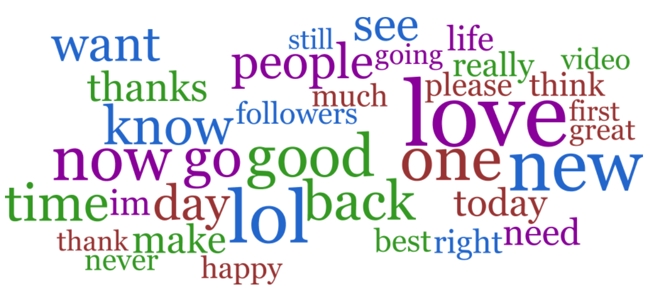
D. User

For the user, we created two alternate word counting MapReduce jobs. The first works on plain text, removing URLs, efficiently splitting on most punctuation, converting all text to lower case, and counting the number of occurrences of each word. The second works on Twitter's JSON format, extracting the text of the tweet, running the same algorithm as above, but weighting the counts with the number of followers of that Twitter user to produce a count of how many users viewed the word (rather than how many users wrote the word).

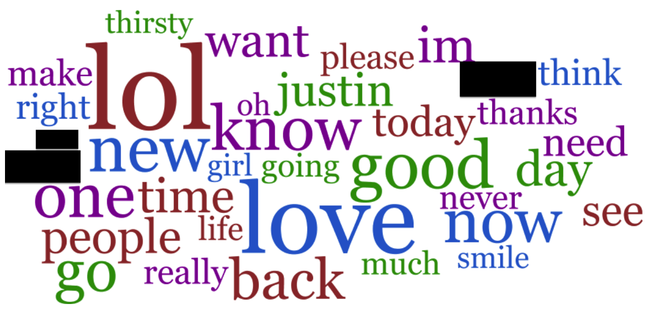
E. Case Study

To validate our proof-of-concept implementation, we deployed a Hadoop infrastructure with access mediated by our ecosystem. We used CDH4 to create a Hadoop cluster with four Amazon c1.medium spot instances (two cores offering 5 ECUs, 1.7 GB of memory). We used a 100 GB file containing Tweets acquired from Twitter's API over a period of several weeks; these Tweets are sampled from the overall userspace (often from more popular or prolific users). Though not entirely in the realm of Big Data, the size of the data is enough to suggest success on data of much larger scale. This data was stored on Amazon S3 in 64 MB files, and retrieved by Hadoop as needed. Temporary data storage and the results used HDFS using instance storage space on the four instances in the cluster.

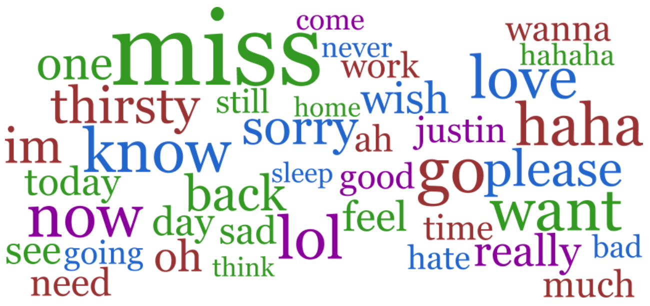
We registered a user account with access to several data sources: English Tweets as Text, Popular English Tweets as JSON, Happy Tweets as Text, Sad Tweets as Text. We submitted the text word count MapReduce to all except the Popular English data source, to which we submitted the JSON-ready word count MapReduce. The jobs were all submitted simultaneously. We downloaded the results, and used post-processing to remove a custom set of the most common words (stop words); standard English stop words, and some Twitter-specific stop words (like "rt" and "follow"). For the results that included non-English words (Happy and Sad), we manually removed words from other languages. We then visualized the



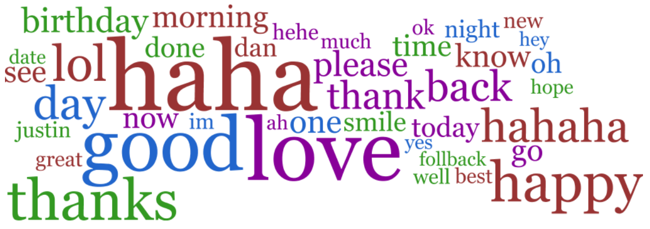
(a) Tweets from English-speaking users, otherwise unfiltered.



(b) Tweets from popular English-speaking users, with word counts weighted by the number of followers. Three words the American Federal Communications Commission (FCC) bans from network television are blacked out.



(c) Sad English Tweets.



(d) Happy English Tweets.

Fig. 4: Word clouds visualizing the most common non-stopword English words in various segments of two weeks of tweets (approx. 50m tweets). Size indicates relative frequency within the given set; colours are assigned randomly.

top 35 words using the Word Cloud visualization, using the same generation code that powers Wordle.net [4]. The results of the visualization are shown in Fig. 4.

The processing took more or less time depending on how many records were permitted to reach the user's MapReduce code (Fig. 5). In all cases all of the data had to be read from S3, but only once. Recall that all filtering and transforming happens locally on the data provider's infrastructure. The English data source permitted access to the largest number of records. There were more Popular tweets than either Happy or Sad tweets, but even the rudimentary sentiment analysis required processing the text of the Tweet, whereas the popularity of the Tweet was present in the metadata.

The CPU time required to produce these four sets of results cost \$0.24. In contrast, based on the size of the outputs produced by each Map, the 240 possible combinations of the currently implemented Modifying Maps[7] would required an additional 1,800 GB of storage space if each segmentation was stored (totalling \$0.2375 per hour in storage charges). Given the vast possibilities for segmentation far beyond only 240 different segments, there is clear value in producing these segments at runtime. There may be a trade-off for the most common segmentations where it is advantageous to store a copy, though this introduces problems of keeping multiple copies of data up to date as new Tweets are generated.

[7]5 unique modifying maps that can be combined in any order, plus the option of ending with TweetToText or not, is $5! * 2$.

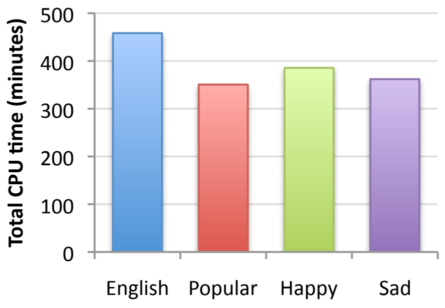


Fig. 5: The CPU time required to complete each of the four jobs.

V. Future Directions

The motivation for this work was simplifying the data analytics tasks for domain experts. It has not escaped our notice that this approach would be very useful for **education** when teaching a course on analytics, business intelligence, or other subjects where the details of installing Hadoop are not included in the learning objectives. One advantage is students using this service could focus on developing the MapReduce logic and not on technology, but especially important is the transformation and segmentation feature which allows the instructor to give every student slightly different datasets to encourage mutual collaboration and problem-solving while still ensuring individual effort is assessed. (Or, in other words, helping to assure academic integrity.)

For data sources with high usage, it is feasible for the provider to maintain a Hadoop installation and use the revenue from frequent requests to cover the expense of this infrastructure. If jobs are infrequent or not intended to generate revenue, for example research data, we are actively developing **on-demand Hadoop clusters**. In response to an incoming request on this service, the Hadoop installation is created per job, based on the budget of the submitting user. Existing approaches to Hadoop-as-a-service are not sufficient here, as they assume the user is supplying their own data (or using a public dataset).

VI. Related Work

Known limitations exist with regards to security of the Hadoop framework [11]. Much work is being done to address this. For example, [12] considers the introduction of hooks/callbacks throughout MapReduce to afford various granularities of control. Alternatively, Airavat [14] combines mandatory access control and differential privacy to augment the security and privacy of MapReduce jobs.

At present, vendors offer services to facilitate running MapReduce jobs on the cloud (e.g., Amazon Elastic MapReduce[8], Microsoft Windows Azure HDInsight[9], etc.). While the potential to run MapReduce on the cloud is beneficial these services were not designed to effectively share Big Data. The DaaSPatcher ecosystem has been introduced to address this requirement.

VII. Conclusion

In this paper we specified the services required to support enhanced data-as-a-service, where a data provider offers not just Big Data but also capacity to process that data locally on their infrastructure via MapReduce jobs. A core technology is the ability to filter, transform, and segment data before it reaches a user's Map task, allowing for a single dataset to be the basis of many data sources, each offering different segmentation or views on the dataset. This increases the market for the provider's data while making data more accessible to users.

The DaaSPatcher ecosystem covers various roles required to compose services at runtime to offer varied data sources to analytics users. We implemented the specified services to create a proof-of-concept ecosystem, and demonstrated the use of this ecosystem to offer segmented and transformed Twitter datasets in a scalable manner.

Acknowledgment

This research was supported by IBM Centres for Advanced Studies (CAS), the Natural Sciences and Engineering Council of Canada (NSERC) under the Smart Applications on Virtual Infrastructure (SAVI) Research Network, and the Ontario Research Fund Research Excellence project Connected Vehicles and Smart Transportation (CVST).

[8]http://aws.amazon.com/elasticmapreduce/

[9]http://www.windowsazure.com/en-us/manage/services/hdinsight/get-started-hdinsight/

References

[1] S. Vijayakumar, A. Bhargavi, U. Praseeda, and S. Ahamed, "Optimizing sequence alignment in cloud using hadoop and mpp database," in *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*, june 2012, pp. 819–827.

[2] E. Waltz, "1000 genomes on amazon's cloud," *Nat Biotech*, vol. 30, no. 5, pp. 376–376, 2012.

[3] S. Toor, R. Toebbicke, M. Resines, and S. Holmgren, "Investigating an open source cloud storage infrastructure for CERN-specific data analysis," in *Networking, Architecture and Storage (NAS), 2012 IEEE 7th International Conference on*, 2012, pp. 84–88.

[4] H. Vashishtha, M. Smit, and E. Stroulia, "Moving text analysis tools to the cloud," in *IEEE Congress on Services*. Los Alamitos, CA, USA: IEEE Computer Society, 2010, pp. 107–114.

[5] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[6] M. Shtern, B. Simmons, M. Smit, and M. Litoiu, "Toward an ecosystem for precision sharing of segmented Big Data," in *IEEE 6th International Conference on Cloud Computing (CLOUD)*, 2013.

[7] A. Cuzzocrea, I.-Y. Song, and K. C. Davis, "Analytics over large-scale multidimensional data: the big data revolution!" in *Proceedings of the ACM 14th international workshop on Data Warehousing and OLAP*, ser. DOLAP '11. New York, NY, USA: ACM, 2011, pp. 101–104.

[8] I. Konstantinou, E. Angelou, D. Tsoumakos, and N. Koziris, "Distributed indexing of web scale datasets for the cloud," in *Proceedings of the 2010 Workshop on Massive Data Analytics on the Cloud*, ser. MDAC '10. New York, NY, USA: ACM, 2010, pp. 1–6.

[9] H. Vashishtha, M. Smit, and E. Stroulia, "Migrating a legacy web-based document-analysis application to Hadoop and HBase: An experience report," in *Migrating Legacy Applications: Challenges in Service Oriented Architecture and Cloud Computing Environments*. IGI Global, 2012, pp. 226–247.

[10] M. Smit, P. Pawluk, B. Simmons, and M. Litoiu, "A web service for cloud metadata," in *IEEE Congress on Services*. Los Alamitos, CA, USA: IEEE Computer Society, 2012, p. To Appear.

[11] O. O'Malley, K. Zhang, S. Radia, R. Marti, and C. Harrell, "Hadoop security design," Yahoo, Inc., Tech. Rep., 2009.

[12] S. Saklikar, "Embedding security and trust primitives within map reduce," http://www.emc-china.com/rsaconference/2012/en/download.php?pdf_file=TC-2003_EN.pdf, 2012.

[13] M. Singhal, S. Chandrasekhar, T. Ge, R. Sandhu, R. Krishnan, G.-J. Ahn, and E. Bertino, "Collaboration in multicloud computing environments: Framework and security issues," *Computer*, vol. 46, no. 2, pp. 76–84, 2013.

[14] I. Roy, S. T. Setty, A. Kilzer, V. Shmatikov, and E. Witchel, "Airavat: Security and privacy for MapReduce," in *Proceedings of the 7th USENIX conference on Networked systems design and implementation*. USENIX Association, 2010, pp. 20–20.