Hybrid Cloud Support for Large Scale Analytics and Web Processing

Navraj Chohan Anand Gupta Chris Bunch Kowshik Prakasam
Chandra Krintz
Computer Science Department
University of California, Santa Barbara, CA

1 Abstract

Platform-as-a-service (PaaS) systems, such as Google App Engine (GAE), simplify web application development and cloud deployment by providing developers with complete software stacks: runtime systems and scalable services accessible from well-defined APIs. Extant PaaS offerings are designed and specialized to support large numbers of concurrently executing web applications (multi-tier programs that encapsulate and integrate business logic, user interface, and data persistence). To enable this, PaaS systems impose a programming model that places limits on available library support, execution duration, data access, and data persistence. Although successful and scalable for web services, such support is not as amenable to online analytical processing (OLAP), which have variable resource requirements and require greater flexibility for ad-hoc query and data analysis. OLAP of web applications is key to understanding how programs are used in live settings.

In this work, we empirically evaluate OLAP support in the GAE public cloud, discuss its benefits, and limitations. We then present an alternate approach, which combines the scale of GAE with the flexibility of customizable offline data analytics. To enable this, we build upon and extend the AppScale PaaS – an open source private cloud platform that is API-compatible with GAE. Our approach couples GAE and AppScale to provide a hybrid cloud that transparently shares data between public and private platforms, and decouples public application execution from private analytics over the same datasets. Our extensions to AppScale eliminate the restrictions GAE imposes and integrates popular data analytic programming models to provide a framework for complex analytics, testing, and debugging of live GAE applications with low overhead and cost.

2 Introduction

Cloud computing has revolutionized how corporations and consumers obtain compute and storage resources. Infrastructure-as-a-service (IaaS) facilitates the rental of virtually unlimited IT infrastructure on-demand with high availability. Service providers, such as Amazon AWS [1] and Rackspace [28], consolidate and share vast resource pools across large numbers of users, who employ these resources on demand on a pay-per-use basis. Customers provision virtual machines (VMs) via API calls or browser portals, which they then configure, connect, monitor, and manage manually according to their software deployment needs.

Platform-as-a-service (PaaS) offerings, such as Microsoft Azure [2] and Google App Engine [16], automate configuration, deployment, monitoring, and elasticity by abstracting away the infrastructure through well-defined APIs and a higher-level programming model. PaaS providers restrict the behavior and operations (libraries, functionality, and quota-limit execution) of hosted applications, both to simplify cloud application deployment, and to facilitate scalable use of the platform by very large numbers of concurrent users and applications. Google App Engine (GAE), the system we focus on herein, currently supports over 7.5 billion page views per day across over 500,000 active applications [15] as a result of their platform's design. As is the case for public IaaS systems, public PaaS users pay only for the resources and services they use.

A key functionality lacking from the original design of PaaS systems is online analytics processing (OLAP). OLAP enables application developers to model, analyze, and identify patterns in their online web applications as users access them. Such analysis helps developers target specific user behavior with software enhancements (code/data optimization, improved user interfaces, bug fixes, etc.) as well as applying said analysis for commercial purposes (e.g. marketing and advertising). These improvements and adaptations are crucial to building a customer base, facilitating application longevity, and ultimately commercial success for a wide range of companies. In recognition of this need, PaaS systems are in-

creasingly offering new services that facilitate OLAP execution models by and for applications that execute over them [17, 26, 3]. However, such support is still in its infancy and is limited in flexibility, posing questions as to what can be done within quota limits and how the service connects with the online applications they analyze.

In this paper, we investigate the emerging support of OLAP for GAE, identify its limitations, and its impact on the cost and performance of applications in this setting. We propose an alternate approach to OLAP, in the form of a hybrid cloud consisting of a public cloud executing the live web application or service and a remote analytics cloud which shares application data. To enable this, we build upon and extend AppScale, an open source PaaS offering that is API-compatible with GAE. AppScale executes over a variety of infrastructures using VM-based application and component isolation. This portability gives developers the freedom and flexibility to explore, research, and tinker with the system level details of cloud platforms [9, 10, 22]. Our hybrid OLAP solution provides multiple options for data transfer between the two clouds, facilitates deployment of the analytics cloud over Amazon's EC2 public cloud or an on-premise cluster, and integrates the popular Hive distributed data warehousing technology to enable a wide range of complex analytics applications to be performed over live GAE datasets. By using a remote AppScale cloud for analytics of live data, we are able to specialize it for this execution model and avoid the quotas and restrictions of GAE, while maintaining the ease of use and familiarity of the GAE platform.

In the sections that follow, we first provide background on GAE and AppScale. We then describe the design and implementation of our hybrid OLAP system. We follow this with an evaluation of existing solutions for analytics, our Hive processing, and an analysis of the cost and overhead of cross-cloud data synchronization. Finally, we present related work and conclude.

3 Background

Google App Engine was released in 2008, with the goal of allowing developers to run applications on Google's infrastructure via a fully managed and automatically scaled system. While the first release only supported the Python programming language, the GAE team has since introduced support for the Java and Go languages. Application developers can access a variety of different services (cf., Table 1) via a set of well-defined APIs. The API implementations in the GAE public cloud are optimized for scalability, shared use, and fault tolerance. The APIs that we focus on in this paper are the Datastore (for data persistence), URL Fetch (for communication), and Task Queues (for background processing).

Table 1: Google App Engine APIs.

Name	Description
Datastore	Schemaless object storage
Memcache	Distributed caching service
Blobstore	Storage of large files
Channel	Long lived JavaScript connections
Images	Simple image manipulation
Mail	Receiving and sending email
Users	Login services with Google accounts
Task Queues	Background tasks
URL Fetch	Resource fetching with HTTP request
XMPP	XMPP-compatible messaging service

AppScale is an open source implementation of the GAE APIs that was released in early 2009, enabling users to run GAE applications on their local cluster or over the Amazon EC2 public IaaS cloud. AppScale implements the APIs in Table 1 using a combination of open source technologies and custom software. It provides a database-agnostic layer, which multiple disparate database/datastore technologies (e.g. Cassandra, HBase, Hypertable, MySQL cluster, and others) can plug into [6]. It implements the Task Queue API by executing a task on a background thread in the same application server as the application instance that makes the request. This support, though simple, is inherently inefficient and not scalable, because it is neither distributed nor load-balanced. Moreover, it does not share state between application servers, which leads to incorrect application behavior when more than one application server is present. We replace this API implementation as part of this work, addressing this limitation.

3.1 App Engine Analytics Libraries

The Task Queue API facilitates the use of multiple, independent user-defined queues, each with a rate limit of 100 tasks per second (which can be increased in some cases [16]) in GAE. A task consists of an application URL, which is called by the system upon task dequeue. A 200 HTTP response code (OK) indicates that the task completes successfully. Other HTTP codes cause re-enqueuing of the task for additional execution attempts. The number of retries, a time delay, and a task name can be optionally specified by developers as part of the task when it is enqueued. Use of task names is important to prevent the same task from being enqueued multiple times (the lack of such measures can result in a task fork bomb, in which a task is infinitely enqueued). One way to circumvent the 10 minute time limit for a task is to chain tasks, in which the initial task performs a portion of the work, and enqueuing another task to resume where

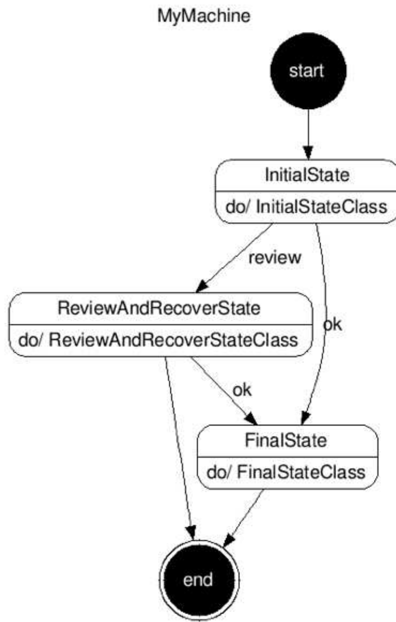

Figure 1: An example state machine in Fantasm.

it has left off. Tasks should be idempotent, or only perform side effects (e.g., updating shared, persistent data) as the final operation – since any failure of a previous statement will cause the task to be re-enqueued (potentially updating shared state incorrectly).

GAE application developers are responsible for program/task correctness when failures occur. This requires that developers make proper use of task names and chaining, and implement tasks that are idempotent. Doing so for all but the most trivial of applications can be a challenging undertaking for all but expert developers. To address this limitation, there are libraries that provide a layer of abstraction over the GAE task queue interface and implementation. These libraries are Fantasm [14], GAE Pipeline [26], and GAE MapReduce [17]. Each automates naming and failure handling by saving intermediate state via the Memcache and the Datastore APIs.

Fantasm, based on [18], employs a programming model that is based on finite state machines (FSM). A programmer describes a state machine via the YAML markup language by identifying states, events, and actions. The initial state typically starts with a query to the datastore, to gather input data for analysis. Fantasm steps through the query and constructs a task for each entity (datastore element) that the query processes in each state. Optionally, there can be a fan-in state, which takes multiple previous states and combines them via a reduction method. Figure 1 shows an example FSM. A limitation of Fantasm is how it iterates through data. It does not shard datasets, but instead, pages through a query serially, leading to inefficient execution of state machines.

```
class WCUrl(pipeline.Pipeline):
  def run(self, url):
    r = urlfetch.fetch(url)
    return len(r.data.split())

class Sum(pipeline.Pipeline):
  def run(self, *values):
    return sum(values)

class MySearchEngine(pipeline.Pipeline):
  def run(self, *urls):
    results = []
    for u in urls:
      # Do word count on each URL
      results.append((yield WCUrl(u)))
    yield Sum(*results) # Barrier waits
```

Figure 2: Code example of Pipeline parallellizing work.

The GAE Pipeline library facilitates chaining of tasks into a workflow. Pipeline stages (tasks) yield for barrier synchronization, at which point the output is unioned and passed onto the next stage in the pipeline. Figure 2 shows an example of parallel processing via Pipeline that counts the number of unique words on multiple web pages. The *yield* operator spawns background tasks, whose results are combined and passed to the *Sum* operation. Implementing similar code via just the Task Queue API is possible, but is more complicated for users.

The GAE MapReduce library performs parallel processing and reductions across datasets. Mapper functions operate on a particular kind of entity and reducer functions operate on the output of mappers. Alternative input readers (e.g. for use of Blobstore files) and sharding support is also available. The GAE MapReduce library uses the Task Queue API for its implementation, as opposed to using Google's internal MapReduce infrastructure or Hadoop, an open source implementation. Both are more flexible than GAE MapReduce, and allow for a wider range of analytics processing than this library. Currently, a key limitation of GAE MapReduce is that all entities in the Datastore are processed, even when they are not of interest to the analysis.

Each of these abstractions for background processing and data analytics in GAE introduce a new programming model with its own learning curve. Moreover, analytics processing on the dataset is intertwined with the application, (that users use to produce/access the dataset) which combines concerns, can introduce bugs, and can have adverse affects on programmer productivity, user experience, and monetary cost of public cloud use. To address these limitations, we investigate an alternate approach to performing online data analytics for applications executing within GAE that employs a combination of GAE and AppScale concurrently.

4 Hybrid PaaS Support for Web Application Data Analysis

In this work, we investigate how to combine two PaaS systems together into a hybrid cloud platform that facilitates the simple and efficient execution of large-scale analysis of live web application data. Our hybrid model executes the web application on the GAE public cloud platform, synchronizes the data between this application/platform and a remote AppScale cloud, and facilitates analysis of the live application data using the GAE analytics libraries, as well as other popular data processing engines (e.g. Hadoop/Hive) using AppScale. Users can deploy AppScale on a local, on-premise cluster, or over Amazon EC2. In this section, we overview the two primary components of our hybrid cloud system: the data synchronization support and the analytics processing engine. We then discuss our design decisions and how our solution works within the restrictions of the GAE platform.

4.1 Cross-Cloud Data Synchronization

The key to our approach to analytics of live web applications is the combined use of GAE and AppScale. Since the two cloud platforms share a common API, applications that execute on one can also do so on the other, without modification. This portability also extends to the data model. That is, given the compatibility between AppScale and GAE, we can move data between the two different platforms for the same application. We note that for vast datasets such an approach may not be feasible. However, it is feasible for a large number of GAE applications today. The cross-platform portability facilitates and simplifies our data synchronization support, and makes it easier for developers to write application and analytics code, because the runtime, APIs, and code deployment process is similar and familiar.

We consider two approaches to data synchronization: bulk and incremental data transfer. For bulk transfer, GAE currently provides tools as part of its software development kit (SDK) to upload and download data into and out of the GAE datastore en masse. We have extended AppScale with similar functionality. Our extensions provide the necessary authentication and data ingress/egress support, as well as support for the GAE Remote API [16], which enables remote access to an application's data in the datastore. The latter must be employed by any application for which hybrid analytics will be used. Using the Remote API, a developer can specify what data can be downloaded (the default is all). Bulk download from, and upload to, is subject to GAE monetary charges for public cloud use.

There are several limitations to bulk data transfer as a mechanism for data synchronization between the two application instances. First, in its current incarnation, transfer is all or nothing (of the entities specified). As such, we are able to only perform analytics off-line or post-mortem if we are to copy the dataset once (the most inexpensive approach). To perform analytics concurrently with web application execution, we are forced to download the same data repeatedly over time (as the application changes it). This can be both costly and slow. Finally, the data upload/download tools from GAE are slow and error prone, with frequent interruptions and data loss.

To address these limitations, we investigate an alternative approach to synchronizing data between GAE and AppScale: incremental data transfer. To enable this, we have developed a library for GAE applications that runs transparently in both GAE and AppScale. Our incremental data transfer library intercepts all destructive operations (writes and deletes) and communicates them to the AppScale analytics cloud. In our current prototype, we do not support the limited form of transactions that GAE applications can perform [13]. As part of our ongoing and future work, we are considering how to reflect committed transactional updates in the AppScale analytics cloud. Developers specify the location of the AppScale analytics cloud as part of their GAE application configuration file. Since the library code executes as part of the application in GAE, it must adhere to all of the GAE platform restrictions. Furthermore, communication to the AppScale analytics cloud is subject to GAE charges for public cloud use.

Our goal with this library is to avoid interruption or impact on GAE web application performance and scale, from the users' perspective. We consider two forms of synchronization with different consistency guarantees: eventual consistency (EC) and best effort (BE). EC incremental transfer uses the Task Queue API to update the AppScale analytics cloud. Using this approach, the library enqueues a background task in GAE upon each destructive datastore operation. The task then uses the URL Fetch library to synchronously transmit the updated entity. In GAE, tasks are retried until they complete without error. Thus, GAE and AppScale data replicas for the application are eventually consistent, assuming that both the GAE and AppScale platforms are available.

Our second approach, best effort (BE), for incremental transfer implements an asynchronous URL Fetch call to the AppScale analytics cloud for the application upon each destructive update. If this call fails, the GAE and AppScale replicas will be inconsistent until the next time the same entity is updated. The BE approach can implement potentially fewer transfers since failed transfers are not retried. This may impact the cost of hybrid cloud analytics using our system. BE is useful for settings in which perfect consistency is not needed.

To maintain causal ordering across updates we employ a logical clock (a Lamport clock [23]), ensuring that only the latest value is reflected in the replicated dataset for each entity. Using this approach, it is possible that at any single point in time there may be an update missing (still in flight due to retries in EC or failed in BE) in the replicated dataset. We transmit entity updates as Protocol Buffers, the GAE transfer format of Datastore entities.

4.2 Analytics Processing Engine within AppScale

We next consider different implementations of the AppScale analytics processing engine. We first extend AppScale to support each of the three analytics libraries that GAE supports, described in Section 3.1. We start by replacing the TaskQueue API implementation in AppScale, from a simple, imbalanced approach, to a new software layer, similar to that for the Datastore API implementation and transaction support [9], that is implementation-agnostic and allows different task queue implementations to be plugged in and experimented with.

The GAE Task Queue API includes the functions:

```
AddTask(name, url, parameters)
DeleteTask(name)
PurgeQueue()
```

We emulate the GAE behavior of this API (that we infer using the GAE SDK and by observing the behavior of GAE applications) in our task queue software layer within AppScale. Each task that is added to the queue specifies a *url* that is a valid path (URL route) defined in the application, to which a POST request can be made using the *parameters*. The *name* argument ensures that a task is only enqueued once given a unique identifier. If a name is not supplied, a unique name is assigned to it. The *PurgeQueue* operation will remove all tasks from a queue, resetting it to an initial, empty state, whereas *DeleteTask* will remove a named task if it is still enqueued. Task execution code is within the application itself (a relative path), or can be a fully remote location (a full path). Successful execution of a task is indicated by a HTTP 200 response code. The task queue implementation retries failed tasks up to a configurable number of times, defaulting to ten attempts.

The AppScale Task Queue interface for plugging in new messaging systems is as follows: This API includes the functions:

```
EnqueueTask(app_name, url, parameters)
LocateTask(app_name, task_name)
AddTask(app_name, task_name)
AckTask(app_name, task_name, reenqueue)
PurgeQueue(app_name)
```

The *AddTask* function stores the given task name and state in the system-wide datastore. Possible task states are ‘running’, ‘completed’, or ‘failed’, and states can be retrieved via *LocateTask*). *AckTask* tells the messaging system whether the task should be re-enqueued, and if it should be, the messaging system increments the retry count associated with that task. Each function requires the application name because AppScale supports multiple applications per cloud deployment, isolating such communications.

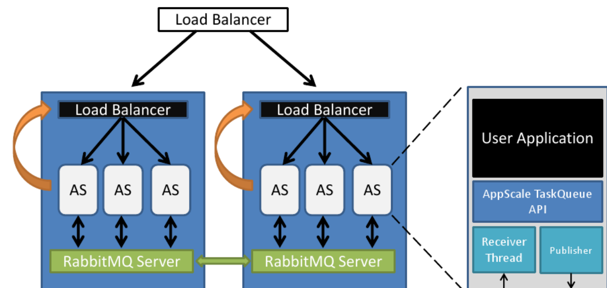

Figure 3: Overview of RabbitMQ implementation in AppScale.

Using the AppScale task queue software layer, we plug-in the VMWare RabbitMQ [27] technology and implement support for each of the GAE analytics libraries (GAE MapReduce, GAE Pipeline, and Fantasm) described in Section 3.1 on top of the Task Queue API. We have chosen to integrate RabbitMQ due to its widespread use and multiple useful features within a distributed task queue implementation, including clustering, high availability, durability, and elasticity. Figure 3 shows the software architecture of RabbitMQ as a task queue within AppScale (two nodes run a given application in this figure). Each AppScale node that runs the application (load-balanced application servers) runs a RabbitMQ server. Each application server has a client that can enqueue tasks or listen for assigned tasks (a callback thread) to or from the RabbitMQ server. We store metadata about each task (name, state, etc.) in the system in the cloud datastore. A worker thread consumes tasks from the server. Upon doing so, it issues a POST request to its localhost or full path/route (if specified), which gets load-balanced across application servers running on the nodes. Tasks are distributed to workers in a round-robin basis, and are retried upon failure. RabbitMQ re-enqueues failed tasks and is fault tolerant.

In addition to the Task Queue, MapReduce, Pipeline, and Fantasm APIs, we also consider a processing engine that is popular for large-scale data analytics yet that is not available in GAE. This processing engine employs a combination of MapReduce [12] (not to be confused with GAE MapReduce, which exports different semantics and behavioral restrictions) and a query processing engine

that maps SQL statements to a workflow of MapReduce operations. In this work, we employ Hadoop, an open source implementation of a fully featured MapReduce system, and Hive [29, 25, 20], an open source query processing engine, similar in spirit to Pig and Sawzall. This processing engine (Hive/Hadoop) provides users with ad-hoc data querying capabilities that are processed using Hadoop, without requiring any knowledge about how to write or chain MapReduce jobs. Moreover, using this AppScale service, users can operate on data using the familiar syntax of SQL and perform large-scale, complex data queries using Hadoop.

AppScale integrates multiple datastore technologies, including Cassandra, Hypertable, and HBase [6, 7]. All of these datastores are distributed, scalable, fault-tolerant, and provide column-oriented storage. Each datastore provides a limited query language, with capabilities similar to the GAE Datastore access model: entities, stored as Protocol Buffers, are accessed via keys and key ranges. We focus on the currently best performing datastore in this work, Cassandra [9].

Our extensions swap out the Hadoop File System (HDFS) in AppScale and replace it with CassandraFS [5], an HDFS-compatible storage layer, that interoperates directly with Cassandra, with the added benefit of having no single points of failure within its NameNode process. Above CassandraFS, we deploy Hadoop; above Hadoop, we deploy Hive. Developers can issue Hive queries from the command line, a script issued on any AppScale DB node [22], or via their applications through a library, similar to the GAE MapReduce library implementation in AppScale.

To enable this, we modified the datastore layout of entities in the AppScale datastore. Previously, we employed a single column-family (table) for all kinds of entities in an applications dataset. We shared tables across multiple applications and we isolated datasets using namespaces prepended to the key names. In this work, we store column-families for each kind of entity. The serialization and deserialization between Hadoop, CassandraFS, and Cassandra happens through a custom interface, which enables Hadoop mappers and reducers to read and write data from Cassandra. We extended the AppScale Datastore API with a layer that translates entities to/from Protocol Buffers. Our extensions eliminate the extract-transform-load step of query processing so that entities can be processed in place.

This support enables Hive queries to run SQL statements which are partitioned into multiple mapper and reducer phases. Hive compiles SQL statements into a series of connected map and reduce jobs. Analysts can perform queries that are automatically translated to mappers and reducers, rather than manually writing these functions and chaining them together. Take for example the task of getting the total count of entities of a certain kind. A Hive query is as simple as:

```
SELECT COUNT(*) FROM appid_kind;
```

us-east-1	Northern Virginia, USA
eu-west-1	Dublin, Ireland
ap-southeast-1	Singapore
ap-northeast-1	Tokyo, Japan
sa-east-1	Sao Paulo, Brazil
us-west-1	Oregon, USA
us-west-2	California, USA

Table 2: EC2 Regions for Amazon Web Services.

To to the same thing in GAE, the entities are paged through and a counter incremented. Note that the Google Query Language for GAE applications limits the number of entities in a single fetch operation to 1000. If the dataset is large enough, then the developer must use a background task or manually implement task queue chaining. Another alternative approach is to use sharded counters to keep a live count; multiple counter entities are required if the increment must happen at a rate faster than once per second. Both methods are foreign to many developers and are far more complex and non-intuitive than simple SQL Hive statements.

5 Evaluation

In this section, we evaluate multiple components of our hybrid web application and analytics system. We first start with an evaluation of the cross-cloud connectivity within a hybrid cloud deployment. For this, we analyze the round-trip time (RTT) between a deployed GAE application in Google datacenters and virtual machines deployed globally across multiple regions and availability zones of Amazon EC2. We next evaluate the performance of the GAE libraries for analytics using the GAE public cloud. We then evaluate the efficacy of our extensions to the AppScale TaskQueue implementation. Lastly, we show the efficiency of using the AppScale analytic solution running Hive over Cassandra.

5.1 Cross Cloud Data Transfer

To evaluate the performance of cross-cloud data synchronization between GAE and AppScale, we must first understand the connectivity rate between them for incremental data transfer (cf Section 4.1). To measure this, we deploy an application in the GAE public cloud that we access remotely from multiple Amazon EC2 micro instances in 16 different availability zones, spanning seven

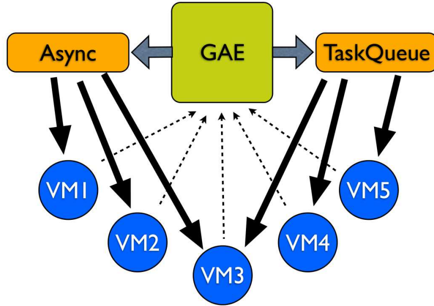

Figure 4: Experimental Setup for Measuring Round-trip Time and Bandwidth Between a GAE Application and VMs in Multiple EC2 Regions.

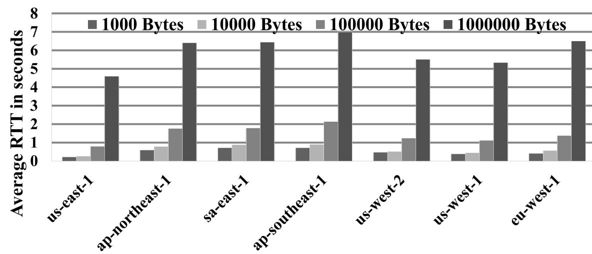

Figure 5: Round-trip Time Per Different Packet Size.

regions. Figure 2 shows the regions we consider, and Figure 4 depicts our experimental setup.

Our experiment issues a HTTP POST request from the EC2 instances, each with a data payload of a particular size, a destination URL location, a unique identifier, and the type of hybrid data synchronization to employ: eventually consistent (EC) or best effort (BE). The sizes we consider are 1KB, 10KB, 100KB, and 1MB (the maximum allowed for GAE's Datastore API). The EC2 instances host a web server, which receives the data from the GAE application (either from a task via EC or from the application itself via BE) and records the current time and request identifier. Figure 5 shows the average RTT for different packet sizes, for each availability zone. The data indicates that it is advantageous to batch updates when possible since there is not a linear relationship between size and RTT, as sizes grow.

We next consider whether the geographical location of the AppScale cloud (different EC2 regions) makes a significant difference in the communication overhead on data synchronization. To evaluate this, we consider the average round-trip time (RTT) and bandwidth across payload sizes to the GAE application for the different regions (Figure 6). The US East region had the RTT

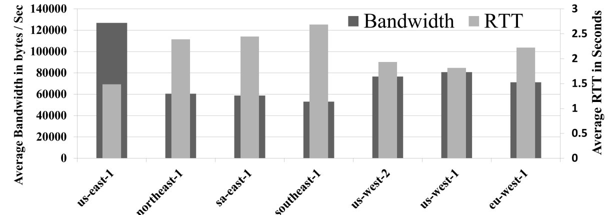

Figure 6: Round-trip Time and Bandwidth Between a GAE Application and Different EC2 Regions.

with the highest bandwidth, by a factor of two. Both US regions have the next best performing communication behavior. This data suggests that our GAE application is hosted (geographically) in GAE in the Eastern US. Locality to the application shows more than 2x the bandwidth for the US East availability zone than other zones (130KB versus 50KB to 80KB for other zones). We investigated this further and found via traceroutes and pings that the application was located near or around New York. We also found with this experiment that bandwidth over time is generally steady, with the exception of between the hours of 16:00 and 22:00 (figure not shown). It may be possible to take advantage of such information to place the AppScale cloud to enable more efficient data synchronization.

We next investigated the task queue delay in GAE. We are interested in whether the delay changes over time or remains relatively consistent. We present this data in Figure 7, as points at each hour in the day (normalized to Eastern Standard Time) that we connect using lines to help visualize the trends. The left x-axis is RTT in seconds for the region, and the right x-axis is the average queue delay (in seconds) for the region. Queue delays do vary but this variance (impact on RTT) is most perceptible during the early evening hours in all regions.

Finally, we compare our two methods for synchronization: EC and BE. EC uses a combination of the Task Queue API and synchronous URLFetch API; the use of the former ensures that all failed tasks are retried until they are successful. BE uses asynchronous URLFetch for all destructive updates and does not retry upon failures.

We ran the experiment for seven days and sent a total of 1195288 requests. Out of the 597644 packets (half of the total packets) sent via the TaskQueue option, 11679 were duplicates (unnecessary transfers). The asynchronous URLFetch experienced 10 duplicate packets suggesting the URLFetch API will retry in some cases from within the lower layers of the API implementation as needed. We experienced no update loss using EC and 5 updates lost for BE.

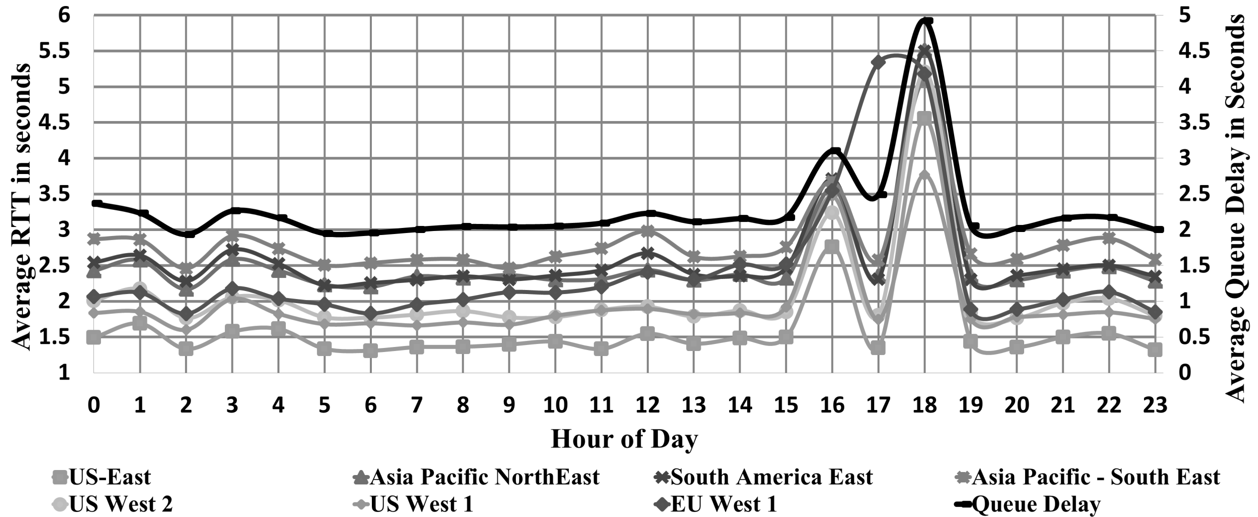

Figure 7: Round-trip time from multiple regions to a deployed GAE application with task queue delay.

5.2 Benchmarks

We next consider the performance of five different and popular analytics benchmarks: wordcount, join, grep, aggregate, and subset aggregate. Wordcount counts the number of times a unique word appears. Join takes two separate tables and combines them based on a shared field. Grep searches for a unique string for a particular substring. Aggregate gives the summation of a field across a kind of entity, while subset aggregate does the same, but for a portion of the entire dataset (one percent for this benchmark). We implemented each benchmark using the Fantasm, Pipeline, and MapReduce GAE libraries, as well as a Hive query.

5.3 Google App Engine Analytics

For the experiments in this section, we execute each benchmark five times and present the average execution time and standard deviation. We use the automatic GAE scaling thresholds, and had billing enabled. We considered experiments with 100, 1000, 10000, and 100000 entities in the datastore. We attempted even higher numbers of entities, but the running time for each trial became infeasible to get complete results.

The tables in 3 shows the results for all of the benchmarks. The Fantasm implementation shows a large latency for a significant numbers of entities, and compared to Pipeline, is 6X to 30X slower. This is due to the fact that Fantasm's execution model has a task for each entity, so it must do paging through the query[1]. Pipeline, by comparison, retrieves a maximum of 1000 entities at a time from the datastore, reducing the amount of time spent querying the database. Pipeline does not see much latency increases from 100 to 1000 entities, because both require only a single fetch from the datastore, and the difference lays in the summation. MapReduce also deals in batches, but the size of the batch depends on the number of shards. When the number of entities went from 100 to 1000 for MapReduce, the growth in latency was over 5X because the number of shards was one. 10000 entities, on the other hand, had 10 shards, and therefore did more work in parallel, seeing an increase in less than half the time. Pipeline has an advantage because of its ability to combine multiple entity values before doing a transactional update to the datastore, whereas both MapReduce and Fantasm are incrementing the datastore transactionally for each entity. For the implementation, the counter was sharded to ensure that there was high write throughput for increments.

Pipeline shows less overhead for Grep as compared to Aggregate (100-1000) because it uses half as many Pipeline stages. In the aggregate Pipeline implementation, there was an initial Pipeline which does the query fetches to the datastore, and another for incrementing the datastore in parallel after combining values. Grep, by comparison, does not need require combining or transactional updates, as required for the counter update in aggregate. Counter updates require reading the current value, incrementing it, and storing it back. Aggregate vs Grep MapReduce has a similar behavior to Pipeline because each mapper does not require transactional updates.

The Join benchmark combines two different entity kinds to create a new table. The Join results show similar trends as Aggregate and Grep. During the exper-

[1] The Fantasm library, since the writing of this paper, has added the ability to do batch fetches for better performance.

	100	1000	10000	100000
Fantasm	13.80 ± 1.61	110.29 ± 4.70	1148.24 ± 86.20	11334.59 ± 1047.57
Pipeline	2.46 ± 0.86	3.05 ± 0.32	11.08 ± 0.50	98.34 ± 3.82
MapReduce	9.34 ± 0.35	57.36 ± 8.96	104.56 ± 17.83	377.70 ± 63.35

Aggregate

	100	1000	10000	100000
Fantasm	10.85 ± 0.77	121.21 ± 21.07	1819.86 ± 1175.19	10360.40± 396.56
Pipeline	2.40 ± 1.26	2.663 ± 0.51	9.77 ± 0.72	98.89 ± 13.76
MapReduce	2.73 ± 0.30	4.56 ± 0.09	24.05 ± 0.30	227.57 ± 20.76

Grep

	100	1000	10000	100000
Fantasm	10.71 ± 1.22	109.83 ± 4.90	977.23 ± 80.34	10147.75 ± 1106.15
Pipeline	4.54 ± 2.34	14.48 ± 5.22	44.11 ± 12.57	159.96 ± 73.30
MapReduce	6.28 ± 1.43	40.18 ± 1.66	66.76 ± 10.92	256.40 ± 11.16

Join

	100	1000	10000	100000
Fantasm	0.58 ± 0.30	3.54 ± 0.28	16.95 ± 1.34	78.28 ± 10.62
Pipeline	1.97 ± 0.05	2.04 ± 0.20	2.01 ± 0.09	3.81 ± 1.60
MapReduce	2.67 ± 0.24	5.42 ± 0.45	27.66 ± 1.74	237.75 ± 12.00

Subset

	100	1000	10000	100000
Fantasm	12.22 ± 3.20	105.82 ± 8.45	1022.96 ± 72.85	10977.50 ± 1258.76
Pipeline	3.63 ± 0.74	4.97 ± 0.92	25.89 ± 8.92	222.14 ± 9.02
MapReduce	6.40 ± 0.96	42.70 ± 0.72	134.88 ± 9.59	840.71 ± 125.15

Wordcount

Table 3: Execution time in seconds for the benchmarks in GAE.

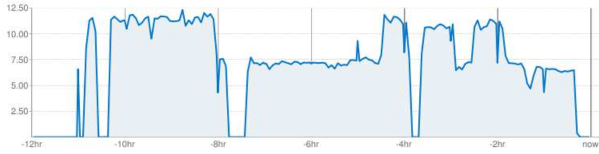

Figure 8: An identical benchmark run three times showing variability in run time.

iments for Join, we experienced high variability in the performance of both the Pipeline and Fantasm libraries. Figure 8 shows a snapshot of three separate trials for Fantasm, in which noticeable differences in processing times occur. Multitenacy could be a primary reason for the fluctuations, yet the exact reasons are unknown and requires further study.

The Subset benchmarks queries a Subset of the entities rather than the entire dataset. Here we see that Fantasm does well, as this scenario was the primary reason for developing the library according to its developers [14]. Pipeline performs best, once again, because of its ability to batch the separate entities, and to not require separate web requests to process individual entities as Fantasm does. MapReduce suffers the most because it must map the entire dataset even though only a Subset is of interest.

For wordcount, MapReduce experiences its largest increase from 10000 to 100000 in this benchmark, which was due to several retries because of transaction collisions. The optimistic transaction support in GAE allows for transactions to rollback if a newer transaction begins before the previous one finishes. This is ideal for very large scale deployments, where failures can happen and locks could be left behind to be cleaned up after a timeout has occurred. Yet it is also possible to bring the throughput of a single entity to zero if there is too much contention. The performance of the wordcount benchmark can be improved by using sharded counters per word as

opposed to the simple non-shared counter per word in our implementation. Built-in backoff mechanisms in the MapReduce library alleviates the initial contention, allowing the job to complete.

5.4 AppScale Library Support

We next investigate the use of the GAE analytics libraries over AppScale using the original Task Queue implementation in the GAE software development kit (SDK) and our new implementation based on the RabbitMQ (RMQ) distributed messaging system. We present only Pipeline results here for brevity (the relative differences between GAE and AppScale are similar). Table 4 shows the average time in seconds for the GAE applications executing over a 3 node Xen VM AppScale deployment. Each VM had 7.5GBs of RAM and 4 cores, each clocked at 2.7GHz. Note, that for the GAE numbers, we do not know the number of nodes/instances or the capability of the underlying physical machines employed.

The left portion of the table shows the RMQ execution time in seconds for each message size. The right portion of the table shows the SDK execution time in seconds for each message size. The SDK implementation enqueues the tasks as a thread locally rather than spreading out load between nodes. In addition, the SDK spawns a thread for each task which posts its request to the localhost. Tasks which originate from the local host will never be run on another node. RabbitMQ, on the other hand, spreads load between nodes, preventing any single node from performing all tasks. We are unable to run the 100K jobs using the SDK because the job fails each time from a lack of fault tolerance. If for any reason the node which enqueues the task fails, that task is lost and not rerun again. RabbitMQ, however, will assign a new client to handle the message, continuing on in the face of client failures. For larger sized datasets we also see a speedup because of the load distribution of tasks.

5.5 AppScale Hive Analytics

We next investigate the execution time of the GAE benchmarks using the Hive/Hadoop system. Figure 5 presents the execution time for the previous benchmarks using the Hive query language on a AppScale Cassandra deployment. There was no discernible difference between the sizes of the datasets, but rather the number of stages, where grep only needed a single mapper phase, while the rest had both mapper and reducer phases. While slower for smaller sizes than the GAE library solutions, the Hive solution is consistently faster when dealing with larger quantities of entities (although it has the same issue as the MapReduce library when dealing with data subsets).

The Hive/Hadoop system in AppScale introduces a constant startup overhead for each phase (map or reduce) of approximately 10s. This overhead is the dominant factor in the performance. Once the startup has occurred, each benchmark completes very quickly. The numbers in the table include this overhead. Each of the benchmarks use a single mapper and reducer phase except for Grep. Our approach is significantly more efficient (enabling much larger and more complex queries) than performing analytics using GAE. Moreover, our approach significantly simplifies analytics program development. Each of our GAE benchmarks requires approximately 100 lines each to implement their functionality. Using our system, a developer can implement each of these benchmarks using a single line with fewer than 50 characters.

5.6 Monetary Cost

The cost of transferring data in GAE is dependent on two primary metrics: bandwidth out which is billed at .12 USD per gigabyte, and frontend instances, at .08 USD per hour. For low traffic applications, these costs can be covered by the free quota. For higher traffic, it is possible to adjust two metrics to keep cost down; the first is the maximum amount of time waiting before a new application server is started (where it will be billed for a minimum of 15 minutes), and the second is the number of idle instances that can exist (lowers latency to new requests in exchange for higher frontend cost).

We can compress data and work in batches to lower the bandwidth cost, seeing as how the additional latency for sending updates is between 4 and 7 seconds on average for the largest possible entity of 1MB. The compression execution time is added to frontend hour cost, and the level of compression is very dependent on the application's data (images, for example, may already be highly compressed). The average daily cost of the data transfer was 12.41 USD for frontend hours, 1.03 USD for datastore storage (went up over time), 2.55 USD for bandwidth, and 15.63 USD for datastore access. As future work, we are leveraging our findings to improve our datastore wrapper to minimize cost while still maintaining low latency overhead.

The cost for on-site analytics such as Fantasm and Pipeline is based on datastore access, both for reading the data which is needed for operation, and metadata for tracking the current progress of a job. The other cost associated is the frontend instance hours. The cost for running Pipeline for wordcount on 100000 entities was 0.34 USD (not accounting for the free quota), where 0.056 USD was frontend hours, 0.13 USD was datastore writes, and 0.154 USD on datastore reads. The cost of datastore writes is highly dependent on the number of indexed en-

	100 RMQ	1000 RMQ	10000 RMQ	100000 RMQ	100 SDK	1000 SDK	10000 SDK
Aggregate	3.02	5.72	183.93	610.12	3.77	6.14	N/A
Grep	5.37	16.90	205.53	862.36	6.11	28.88	260.03
Join	2.72	5.16	165.03	455.31	3.78	5.90	305.82
Subset	2.45	3.12	12.61	786.53	2.55	3.20	12.11
Wordcount	7.41	11.43	311.52	635.28	8.38	17.40	411.12

Table 4: Execution time in seconds for benchmarks using the Pipeline library on AppScale with RabbitMQ (RMQ) and the SDK implementation.

	100	1000	10000	100000
Aggregate	20.59 ± 1.41	21.14 ± 0.55	20.30 ± 0.88	20.94 ± 0.59
Grep	11.90 ± 1.32	11.00 ± 0.58	11.17 ± 1.30	10.69 ± 0.44
Join	20.52 ± 1.01	20.71 ± 0.84	20.43 ± 0.57	23.41 ± 0.64
Subset	19.93 ± 0.54	20.07 ± 1.34	20.26 ± 0.86	20.66 ± 0.45
Wordcount	21.73 ± 1.50	22.13 ± 1.51	22.19 ± 0.96	21.54 ± 0.95

Table 5: Execution time in seconds for benchmarks using Hive.

tities, and therefore if the entities have more properties, the writes can multiply quickly as would cost (each index write counts as a datastore write). In general, it is difficult to predict the cost of GAE analytics. Our approach allows developers to perform analytics repeatedly without being charged at the cost of data transfer.

Our other option for downloading the data is via bulk transfer using tools provided by the SDK. We investigated the use of such tools but we ran into difficulties where exceptions arose and the connection would drop. Multiple attempts were needed, driving cost up as much to 5 to 6 times the cost of a daily experimental run (from 15 USD to 86 USD) before being able to complete a full download of the data. It took 9520 seconds on average for the three successful downloads of a dataset of 202MB. This option is clearly not acceptable for hybrid analytic clouds.

6 Related Work

OLAP and data warehousing systems have been around since the 1970s [8], yet there is no system available for GAE which is currently focused providing OLAP for executing web applications. AppScale, with its API compatibility and our extensions herein, brings OLAP capabilities (as well as its testing and debugging) to this domain.

TyphoonAE is the only other framework which is capable of running GAE applications outside of GAE. TyphoonAE however is a more efficient version of the SDK (executes the system serially) and only supports the Python language. AppScale and our work supports Python, Java, and Go languages and is distributed and scalable. TyphoonAE does not have the same facility as AppScale to run analytics, as it does not support datastores capable of Hive support. Private PaaS offerings such as Cloud Foundry [11] offer an open source alternative to many proprietary products and offer automatic deployment and scaling of applications, yet do not support GAE APIs.

There are many cloud platforms which allows for analytics to be run on large scale datasets. Amazon's Elastic MapReduce is one such service, where machines are automatically setup to run jobs, along with customized interfaces for tracking jobs [24]. The Mesos framework is another cloud platform which can run a variety of processing tools such as Hadoop and MPI, and does so with a dynamically shared set of nodes [19]. Helios is yet another framework that simplifies the application deployment process.

In [21], the authors measured data-intensive applications in multiple clouds including GAE, AWS, and Azure. Their application was a variant of the TPC-W benchmark, similar to an online bookstore. Our benchmarks, by comparison, are analytics driven rather than online processing. Furthermore, since the time of publication Google–as well as the other cloud providers–have continuously improved functionality and added features. Our work provides a new snapshot in time of the current system, which has since come out of preview and become a fully supported service.

Data replication across datacenters is a common method for prevention of data loss and to enable disaster recovery if needed. Currently GAE implements three-

plus times replication across datacenters using a variant of the Paxos algorithm [4]. Extant solutions, such as [30], however, are not applicable because of the restrictions imposed by the GAE runtime. To overcome this limitation, we provide a library wrapper around destructive datastore operations, to asynchronously update our remote AppScale analytic platform. As part of future work, we are investigating how to provide disaster recovery using our hybrid system.

7 Conclusion

Cloud computing has seen tremendous growth and wide spread use recently. With such growth comes the need to innovate new methods and techniques for which extant solutions do not exist. Online analytics processing systems are such an offering for Google App Engine, where current technology has focused on web application execution at scale and with isolation, and existing solutions have operated within the restrictions imposed.

In this paper we have described, implemented, and evaluated two systems for running analytics on GAE application, running current libraries in AppScale through the implementation of a distributed task queue, and the ability to run SQL statements on cross-cloud replicated data. Future work will carry forward our findings to optimize cross-cloud data synchronization as well apply our system to another use case: disaster recovery.

References

[1] Amazon Web Services. http://aws.amazon.com/.

[2] Microsoft Azure Service Platform. http://www.microsoft.com/azure/.

[3] AZURE, M. Business Analytics, 2011. http://www.windowsazure.com/en-us/home/tour/business-analytics/.

[4] BAKER, J., BOND, C., CORBETT, J., FURMAN, J., KHORLIN, A., LARSON, J., LEON, J., LI, Y., LLOYD, A., AND YUSHPRAKH, V. Megastore: Providing Scalable, Highly Available Storage for Interactive Services. In *5th Biennial Conference for Innovative Data Systems Research* (2011).

[5] Brisk Datastax. http://www.datastax.com.

[6] BUNCH, C., CHOHAN, N., KRINTZ, C., CHOHAN, J., KUPFERMAN, J., LAKHINA, P., LI, Y., AND NOMURA, Y. An Evaluation of Distributed Datastores Using the AppScale Cloud Platform. In *IEEE International Conference on Cloud Computing* (Jul. 2010).

[7] BUNCH, C., KUPFERMAN, J., AND KRINTZ, C. Active Cloud DB: A RESTful Software-as-a-Service for Language Agnostic Access to Distributed Datastores. In *ICST International Conference on Cloud Computing* (2010).

[8] CHAUDHURI, S., AND DAYAL, U. An overview of data warehousing and olap technology. *SIGMOD Rec. 26* (March 1997), 65–74.

[9] CHOHAN, N., BUNCH, C., KRINTZ, C., AND NOMURA, Y. Database-Agnostic Transaction Support for Cloud Infrastructures. In *IEEE International Conference on Cloud Computing* (July 2011).

[10] CHOHAN, N., BUNCH, C., PANG, S., KRINTZ, C., MOSTAFA, N., SOMAN, S., AND WOLSKI, R. AppScale: Scalable and Open AppEngine Application Development and Deployment. In *ICST International Conference on Cloud Computing* (Oct. 2009).

[11] Cloud Foundry. http://cloudfoundry.com/.

[12] DEAN, J., AND GHEMAWAT, S. MapReduce: Simplified Data Processing on Large Clusters. *Proceedings of 6th Symposium on Operating System Design and Implementation(OSDI)* (2004), 137–150.

[13] ENGINE, G. A. App Engine Transaction Semantics, 2010. http://code.google.com/appengine/docs/python/datastore/transactions.html.

[14] Fantasm. http://code.google.com/p/fantasm/.

[15] Google app engine blog. http://googleappengine.blogspot.com.

[16] Google App Engine. http://code.google.com/appengine/.

[17] Google App Engine MapReduce. http://code.google.com/p/appengine-mapreduce/.

[18] GURP, J. V., AND BOSCH, J. On the implementation of finite state machines. In *in Proceedings of the 3rd Annual IASTED International Conference Software Engineering and Applications, IASTED/Acta* (1999), Press, pp. 172–178.

[19] HINDMAN, B., KONWINSKI, A., ZAHARIA, M., GHODSI, A., JOSEPH, A., KATZ, R., SHENKER, S., AND STOICA, I. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In *Networked Systems Design and Implementation* (2011).

[20] HIVE. Hive Query Processing Engine, 2010. https://cwiki.apache.org/confluence/display/Hive/Home.

[21] KOSSMANN, D., KRASKA, T., AND LOESING, S. An evaluation of alternative architectures for transaction processing in the cloud. In *Proceedings of the 2010 international conference on Management of data* (New York, NY, USA, 2010), SIGMOD '10, ACM, pp. 579–590.

[22] KRINTZ, C., BUNCH, C., AND CHOHAN, N. AppScale: Open-Source Platform-A s-A-Service. Tech. Rep. 2011-01, University of California, Santa Barbara, Jan. 2011.

[23] LAMPORT, L. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM 21*, 7 (1978).

[24] MAPREDUCE, A. E. Amazon Elastic MapReduce. http://aws.amazon.com/elasticmapreduce.

[25] MURTHY, R., AND JAIN, N. Talk at ICDE 2010. Hive–A Petabyte Scale Data Warehouse Using Hadoop., Mar. 2010.

[26] Google App Engine Pipeline. http://code.google.com/p/appengine-pipeline/.

[27] RabbitMQ. http://www.rabbitmq.com.

[28] Rackspace Hosting. http://www.rackspace.com.

[29] THUSOO, A., SARMA, J. S., JAIN, N., SHAO, Z., CHAKKA, P., ANTHONY, S., LIU, H., WYCKOFF, P., AND MURTHY, R. Hive- a warehousing solution over a map-reduce framework. In *VLDB* (2009), pp. 1626–1629.

[30] WOOD, T., LAGAR-CAVILLA, H. A., RAMAKRISHNAN, K. K., SHENOY, P., AND VAN DER MERWE, J. Pipecloud: using causality to overcome speed-of-light delays in cloud-based disaster recovery. In *Proceedings of the 2nd ACM Symposium on Cloud Computing* (New York, NY, USA, 2011), SOCC '11, ACM, pp. 17:1–17:13.