UNIVERSITY OF CALIFORNIA, SAN DIEGO

A developer's survey on different cloud platforms

A thesis submitted in partial satisfaction of the
requirements for the degree
Master of Science

in

Computer Science

by

Dzung Doan

Committee in charge:

Professor Yannis Papakonstantinou, Chair
Professor Alin Deutsch
Professor Geoffrey M. Voelker

2009

The thesis of Dzung Doan is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

Chair

University of California, San Diego

2009

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGEMENTS

I would like to gratefully acknowledge the extensive help and encouragement of Professor Yannis Papakonstantinou. I would also like to acknowledge the help of my committee members, Professor Alin Deutsch and Professor Geoffrey Voelker, as well as my many colleagues at UCSD.

ABSTRACT OF THE THESIS

A developer's survey on different cloud platforms

by

Dzung Doan
Master of Science in Computer Science

University of California San Diego, 2009

Professor Yannis Papakonstantinou, Chair

Cloud platforms are emerging with great potential to support developers to develop and deploy applications in the cloud. This thesis aims at getting a glimpse of this potential, by conducting a survey on several typical cloud platforms from the perspective of a developer.

We found that Amazon Web Services is a very general environment that can support various types of applications, although the generality comes with minimum support for the development process; there are opportunities for third-party add-on frameworks that compensate this shortcoming in specific application categories, with Hadoop being an example in the area of distributed/parallel computing. Sun Grid Compute Utility proposes an attractive model for batch processing programs, but the service needs significant improvement to be a strong competitor. Google App Engine is a promissing platform for web applications, hiding inside appropriately designed features aiming at high scalability and throughput. Microsoft Windows Azure, although at the high level provides roughly similar offerings as a general environment, consists of a very impressive and solid foundation, and promisses to become a strong player in the landscape.

Chapter 1

Introduction

Cloud computing is a computing paradigm in which tasks are assigned to a combination of connections, software and services accessed over a network. This network of servers and connections is collectively known as "the cloud." Computing at the scale of the cloud allows users to access supercomputer-level power. Using a thin client or other access point, like an iPhone, BlackBerry or laptop, users can reach into the cloud for resources as they need them. For this reason, cloud computing has also been described as "on-demand computing."

This vast processing power is made possible though distributed, large-scale cluster computing, often in concert with server virtualization software, like Xen[1], and parallel processing. Cloud computing can be contrasted with the traditional desktop computing model, where the resources of a single desktop computer are used to complete tasks, and an expansion of the client/server model. To paraphrase Sun Microsystems' famous adage, in cloud computing the network *becomes* the supercomputer.

The coming shift to cloud computing is a major change in the industry. One of the most important parts of that shift is the advent of cloud platforms. As its name suggests, this kind of platform lets developers write applications that run in the cloud, or use services provided from the cloud, or both. Different names are used for this kind of platform today, including *on-demand platform* and *platform as a service (PaaS)*. Whatever its called, this new way of supporting applications has great potential.

To see why, think about how application platforms are used today. When a development team creates an on-premises application (for example, one that will run within an organization), much of what that application needs already exists. An operating system provides basic support for executing the application, interacting with storage, and more, while other computers in the environment offer services such as remote storage. If the creators of every on-premises application first had to build all of these basics, wed have many fewer applications today.

Similarly, if every development team that wishes to create a cloud application must first

build its own cloud platform, we wont see many cloud applications. Fortunately, vendors are rising to this challenge, and a number of cloud platforms have been taking shape. The goal of this thesis is to conduct a survey on several platforms from the perspective of a developer.

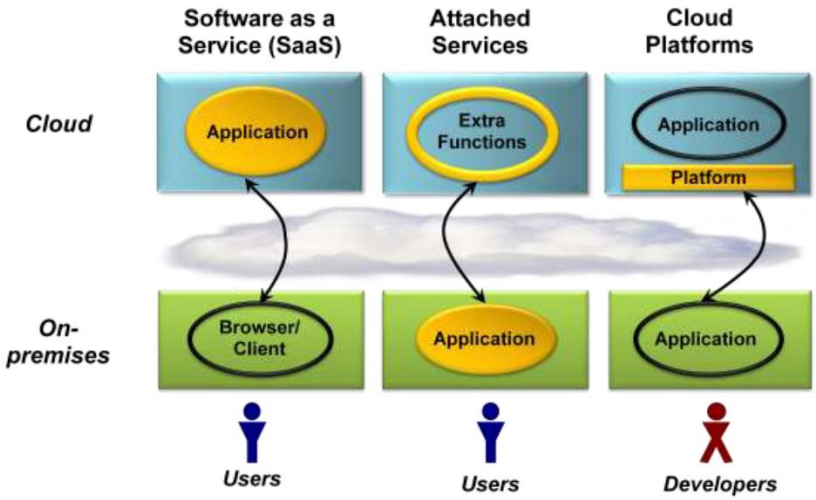


Figure 1.1: Cloud services can be grouped into three broad categories

First, to put in context, its useful to start by looking at cloud services in general. As Figure 1.1 shows, services in the cloud can be grouped into three broad categories. Those categories are:

- Software as a service (SaaS): A SaaS application runs entirely in the cloud (that is, on servers at an Internet-accessible service provider). The on-premises client is typically a browser or some other simple client. The most well-known example of a SaaS application today is probably Salesforce.com[2], but many, many others are also available.

- Attached services: Every on-premises application provides useful functions on its own. An application can sometimes enhance these by accessing application-specific services provided in the cloud. Because these services are usable only by this particular application, they can be thought of as attached to it. One popular consumer example of this is Apples iTunes[3]: The desktop application is useful for playing music and more, while an attached service allows buying new audio and video content. Microsofts Exchange Hosted Services[4] provides an enterprise example, adding cloud-based spam filtering, archiving, and other services to an on-premises Exchange server[5].

- Cloud platforms: A cloud platform provides cloud-based services for creating applications. Rather than building their own custom foundation, for example, the creators of a new SaaS

application could instead build on a cloud platform. As Figure 1 shows, the direct users of a cloud platform are developers, not end users.

The survey investigates cloud platforms with two broad categories of applications in mind: batch applications and interactive applications. Batch refers to programs that can run to completion without human interaction in a typically long (many minutes to hours or even more) process, and may involve intensive computations and/or handling large datasets. Interactive refers to plenty-of-short-user-requests apps, such as Internet-based OLTP web applications.

The survey identifies typical platforms capable of supporting developers to develop, deploy, and execute applications of these two types on their environments. In evaluating a platform, we are ultimately concerned about its service for developers. This details to a number of major questions we try to address: can it support the type of applications it claims able to? how does it support the development and testing of applications? how about the support for deployment and execution? what is the unique feature(s) that distinguish(es) itself from others (detailed examination of the feature may follow)?

To address these questions, we build applications and/or conduct analysis and experiments on the selected platforms. In building applications, the focus is on developer experience, particularly that of someone relatively new to writing apps on the cloud, since most cloud platforms are relatively new and constantly changing; the experience may relate to support that comes natively from the platform, or that comes from an add-on tool/framework from the same or a third-party vendor that we believe is the most prominent tool/framework for the given type of application on the given platform. In conducting analysis and experiments, the focus surrounds the unique feature(s) in consideration.

The platforms chosen for the thesis are Amazon Web Services[6], a general environment able to support both types of applications, Sun Grid Compute Utility[7], a grid infrastructure mostly geared toward batch applications, and Google App Engine[8], a platform for interactive web applications. Additionally, the thesis presents a technical view on Windows Azure[9], a general platform recently introduced by Microsoft.

For Amazon Web Serices, we successfully built and ran a Hadoop[10]-based distributed program that simulates a bioinformatics app - BLAST[11], and successfully launched RUBiS[12], an Ebay-like auction system. In Sun Grid Compute Utility, we didn't have a very satisfactory experience porting the BLAST-simulating app to this environment. For Google App Engine, we showed that the chosen concurrency control rules result in the highest transaction throughput for the datastore, along with some practices for building scalable apps, and the discussion of developer experience. The technical presentation on Windows Azure is based mostly from (lots of) information available from the vendor.

The remaining content of this thesis is organized as follows. Chapter 2 explores developer experience with Amazon Web Services, focusing on Amazon Elastic Compute Cloud (EC2)[13].

Chapter 3 investigates the building and running of batch applications on Sun Grid Compute Utility. Chapter 4 discusses Google App Engine, with the focus on its unique feature - the App Engine datastore. Chapter 5 presents a technical view on Microsoft Windows Azure. And chapter 6 concludes the survey.

Chapter 2

Amazon Web Services

2.1 Introduction to Amazon Web Services (AWS)

AWS[6] provides companies of various sizes with an infrastructure web services platform in the cloud. With AWS one can requisition compute power, storage, and other services gaining access to a suite of elastic IT infrastructure services as his business demands them. With AWS, one has the flexibility to choose from a number of development platforms or programming models the one suitable for the problems he's trying to solve. The user pays only for what he uses, with no up-front expenses or long-term commitments, making AWS the cost-effective way to deliver the application to customers and clients. With AWS, one can take advantage of Amazon.coms global computing infrastructure that powers their renowned online business over the years.

Currently AWS consists of the following component infrastructure services:

- **Amazon Elastic Compute Cloud (Amazon EC2)**[13]: A web service that provides resizable compute capacity in the cloud. Configure an Amazon Machine Instance (AMI) and load it into the Amazon EC2 service. Quickly scale capacity, both up and down, as the computing requirements change.
- **Amazon Simple Storage Service (Amazon S3)**[14]: A simple web services interface that can be used to store and retrieve large amounts of data, at any time, from anywhere on the web. It gives developer access to the same data storage infrastructure that Amazon uses to run its own global network of web sites.
- **Amazon SimpleDB**[15]: A web service for running queries on structured data in real time. This service works in close conjunction with Amazon S3 and Amazon EC2, collectively providing the ability to store, process and query data sets in the cloud. Amazon SimpleDB is easy to use and provides the core functionality of a database real-time lookup and simple querying of structured data without the operational complexity.

- **Amazon Simple Queue Service (Amazon SQS)**[16]: A reliable, highly scalable, hosted queue for storing messages as they travel between computers. By using Amazon SQS, developers can simply move data between distributed components of their applications that perform different tasks, without losing messages or requiring each component to be always available.

2.2 Amazon EC2

The overall AWS offerings, particularly Amazon EC2 and Amazon S3 (EC2 and S3 for short, respectively), enable developers to develop and deploy their applications on Amazon infrastructure for use by themselves or end users. Since it provides the computing capabilities as a processing engine, we consider EC2 as the core service from the developer perspective. Thus, we investigate EC2 in details. Along the way, we also touch upon S3 as a backup repository for data of applications running on EC2 instance(s).

To investigate EC2 as a developer, we build applications to run on this cloud service. These applications are roughly divided into two categories: batch and interactive. We will evaluate the experience building and running applications of these two types on EC2 via steps from development, testing, to deployment and execution of the constructed programs.

First, a detailed description of the EC2 service is presented.

2.2.1 Detailed description of Amazon EC2

Amazon Elastic Compute Cloud is a web service that provides resizable compute capacity in the cloud. It is designed to make web-scale computing easier for developers. EC2 presents a true virtual computing environment, allowing developer to use web service interfaces to launch instances, load them with custom application environment, manage the networks access permissions, and run the image using as many or few systems as the developer desires [1].

To use EC2, one:

- Create an Amazon Machine Image (AMI) containing applications, libraries, data and associated configuration settings. Or use pre-configured, templated images to get up and running immediately. An AMI is simply a packaged-up environment that includes all the necessary bits to set up and boot the instance. AMIs are the unit of deployment.

- Upload the AMI into Amazon S3. Amazon EC2 provides tools that make storing the AMI simple. Amazon S3 provides a repository to store images.

- Use Amazon EC2 web service to configure security and network access.

[1] currently, by default, Amazon allows a developer to run up to 20 instances at a time; if one wishes to run more than 20 simultaneous instances, he needs a one-time approval from Amazon

- Choose which instance type(s) desired, then start, terminate, and monitor as many instances of the AMI as needed, using the web service APIs or the variety of management tools provided.
- Pay only for the resources actually consumed, like instance-hours or data transfer.

Note that data stored on a specific instance persists only as long as that instance is alive; once the instance terminates, data are lost. To persist the data, prior to terminating the instance, one needs to back up the data to persistent storage, either over the Internet, or to Amazon S3.

EC2 provides several different instance types falling into 2 categories: standard instances and high-cpu instances. Configurations are as follows:

Standard Instances

- *Small Instance (Default):* 1.7 GB of memory, 1 EC2 Compute Unit (1 virtual core with 1 EC2 Compute Unit), 160 GB of instance storage, 32-bit platform.
- *Large Instance:* 7.5 GB of memory, 4 EC2 Compute Units (2 virtual cores with 2 EC2 Compute Units each), 850 GB of instance storage, 64-bit platform.
- *Extra Large Instance:* 15 GB of memory, 8 EC2 Compute Units (4 virtual cores with 2 EC2 Compute Units each), 1690 GB of instance storage, 64-bit platform.

High-CPU Instances

- *High-CPU Medium Instance:* 1.7 GB of memory, 5 EC2 Compute Units (2 virtual cores with 2.5 EC2 Compute Units each), 350 GB of instance storage, 32-bit platform.
- *High-CPU Extra Large Instance:* 7 GB of memory, 20 EC2 Compute Units (8 virtual cores with 2.5 EC2 Compute Units each), 1690 GB of instance storage, 64-bit platform.

Note that one EC2 Compute Unit (ECU) provides the equivalent CPU capacity of a 1.0-1.2 GHz 2007 Opteron or 2007 Xeon processor.

Also note that all our experiments in the following sections use the default standard instance-small instance, unless noted otherwise.

2.2.2 Batch applications running on EC2

How does AWS support the development and testing of Batch applications?

To a developer, his EC2 instances are essentially remote machines that he has full control on. To develop applications on an instance, he needs to have the development environment and all necessary software installed on the instance while the instance is alive or packaged into the AMI from which the instance starts. From all the AMI templates globally available from Amazon

or publicly shared by other developers, one may or may not find one that suits his needs; if not available, the developer always has the option of bundling the AMI by himself, or installing additional software to his live instance.

Nevertheless, doing development on a remote machine over the Internet usually doesn't give the comfort as doing that locally, unless one's local workstation doesn't meet the systems requirement of the software needed for the development activity, while an EC2 instance does. The preferred way is to develop application locally, then run it on EC2. EC2 instances are thus viewed as an execution environment for running programs to get results, and probably for testing programs during development.

Developing program locally, then running the completed application on one single EC2 instance probably only makes sense if the instance provides extra power (memory, CPU power, or storage capacity) to meet the need of the running job which the local workstation is not capable of. This usually means an instance of the type large/extra large/high-cpu is the choice in this case.

The development/testing activity in this scenario is not different than the conventional manner. One develops program on his own machine, tests it with some (typically small) datasets; once finished, he starts an EC2 instance of the appropriate type from an appropriate AMI, installs required software if needed, uploads the program and the real-life datasets to the instance, then has it run and waits for the results; he could also use the instance for the testing purpose (for larger test datasets, for example) during development. There's no need of any type of special support from AWS for this type of development/testing activity; and in fact, such a support doesn't exist.

In the context of batch applications being discussed here, there's usually the need to run jobs that are so computationally intensive and/or handles a so large dataset that one single EC2 instance, no matter what type it is, is not a match. Given that one can start as many instances as he wants, it's a natural choice to use the combined power of a cluster of instances to handle such massive batch jobs. The issue now centers around the application itself: can it be designed in such a way that it can exploit the power of a cluster of nodes? Further, what's the support provided by AWS to develop/test distributed/parallel applications that run on many instance-cluster?

Currently AWS has no such support; it only provides infrastructure API to facilitate tasks such as start up, shutdown instances; no distributed computing API is available.

However, there's an open source distributed computing platform, named Hadoop[10], that provides good support for EC2. Hadoop features a distributed file system, called HDFS[17], and a MapReduce[18] compliant distributed computing framework that facilitate writing and running programs in a parallel fashion on a multi-node cluster. Hadoop has been adopted in production systems by organizations of various sizes, with several well known names, and to the best of our knowledge, is the most prominent tool that supports developing batch applications

running on EC2 clusters. In the following section, an experience of writing/testing a Hadoop application on an EC2 cluster is presented.

Writing/testing a Hadoop program to run on an EC2 cluster

We develop a program that simulates BLAST[11], a popular tool in the biology community allowing running a query against a biological sequence database. Unlike BLAST, which uses probability to prune the search range early on, our program implements a brute force method, which searches the entire database and compares all database sequences with the query sequence. Each comparison, from the algorithmic perspective, is a string matching operation that finds the most similar (modified) sub-parts from the two strings according to a set of scoring criteria. Depending on the datasets and scoring criteria, the program can become computationally intensive. It's not typically handling a large dataset per se, since a sequence database for a specie usually is in the order of tens to more than a hundred Mbs. Nevertheless, it's a program suitable to run on a multi-node cluster, because the comparisons between the query and the database sequences can be performed independently in parallel.

The comparison itself, essentially a matching operation between two strings, implements the combination of the Smith-Waterman[19] and the Hirschberg[20] algorithms and was written as a separate library on a previous occasion; only a minor modification to it was needed for this project (see the appendix for further information about this program). The major effort was put in bringing this operation in a large number into the Hadoop context.

There may be more than one way to approach Hadoop EC2. Here the way that worked for us is presented.

Getting yourself familiar to the concepts

The original paper about the MapReduce programming model[18] published by Google is a good start. We got the basic idea and spirit of this distributed computing framework from this paper. Hadoop essentially is an open source version of this framework, implemented in Java. It'd probably be even better to also read the GFS paper about Google File System[21] on which Google's MapReduce framework functions; however, this seems optional since for us, the HDFS documents[17][22] were sufficient enough.

Setting up and launching a single-node cluster

The Hadoop quickstart guide[23], as the name implies, is a good place to start. It suggests getting up and running a single node cluster containing only one's single workstation so that user can get a flavour of the HDFS and MapReduce. This pseudo-cluster, as we will show later, is valuable in testing/debugging the application locally during development. We were able to download Hadoop[24], install the package, launch this single-node cluster on our own machine and run sample programs provided by Hadoop without any trouble. The Hadoop version of

choice is 0.16.4, which is claimed the latest stable release at the time of the experiment (the latest release is 0.17.1).

Setting up and launching a true EC2 cluster

We wanted to make sure we could get a Hadoop EC2 cluster up and running before investing any effort in developing the application. On the Hadoop project wiki, there's a section solely dedicated to EC2[25] that provides useful guide for this task. In the library of public AMIs for EC2, there are available several AMIs already bundled with different versions of Hadoop; one needs to pick the desired version, and launch the cluster from that selected AMI. Note that the launch and shutdown of a cluster are controlled by scripts on the local workstation.

Here are some further noteworthy notes of what we did:

- The local Hadoop installation we did in the previous step comes with some scripts/programs to control the remote EC2 cluster.
- The file hadoop-ec2-env.sh from the installation contains variables consulted by the scripts controlling the cluster. One needs to set relevant variables correctly. Some further notes:
  - Variables identifying yourself to EC2 (names, values and locations of keys used to access AWS in general and EC2 in particular) could easily be set wrong. Should check these if you fail to launch the cluster.
  - The version of Hadoop running on the cluster may be different from that in the local workstation. In our case, we chose version 0.16.1 because among the public Hadoop AMIs available, that's the latest stable release.
  - The host name of the master node is specified here. The master node is the master for both HDFS and the MapReduce framework. When launching the cluster, DNS for this host is set to point to its IP address.
  - The cluster size (number of EC2 instances) is also specified here.

We could get a trial two-node cluster up and running sample programs after a couple of attempts setting the variables. Overall, up to this point, we can say Hadoop has good support for EC2; configuring, launching and terminating a cluster is made relatively easy with Hadoop provided scripts.

Now, we were confident digging into writing the Hadoop program.

Developing the application

IBM provides an Eclipse plug-in[26] that facilitates the creation and deployment of Hadoop MapReduce applications. Yet, its documents and instructions on where to get it and its compatibility with other software on both Hadoop wiki and IBM web site are not very up-to-date;

only after the successful completion of our application, we became confident that it worked with our version of Eclipse (3.3.2, latest release at time of experiment), JDK (1.6, latest version at time of experiment) and Hadoop (0.16.x).

With the development environment in place, we started by reading the document introducing the architecture and design of HDFS[17], then moved on to the Hadoop MapReduce tutorial[27]. These gave us a general sense of the system.

We had an enjoyable experience developing the application. The API documentation[28] is not comprehensive; in fact, one could say it's still at a relatively early stage, even pretty confusing at places. But all the source code of the platform is available and comes with the distribution, and this proved really valuable at times; consulting it when needed, combined with the documentation and experiments helped us get various details, connecting all together gave a big picture of how the platform actually functions behind the scene, an understanding that is vital to writing a good application. Moreover, the source code of non-core add-on components also served as good examples for our coding.

Another good source of support is the Hadoop user mailing list[29]. This list seems growing rapidly and very active. There are usually tens of question and answer messages posted daily. We got several of our issues resolved via this channel.

Coming from a centralized programming environment, we found that moving to a distributed one was initially somewhat a learning curve. One example is how to deal with files in the file system. With a single-machine programming style mindset, we started out trying to open the database file using the memory mapped file method, because that's how large files (many Mbs in size) are brought into memory in the traditional programming environment. This didn't work, because the database file exists in the HDFS context, whereas the standard Java libraries-provided memory mapped file facilities searched for the file to open in the local file system. It turned out that Hadoop-provided file handling facilities deal well with large files on HDFS with low level details transparent to programmers, and well suited our purpose, so we used them instead.

The story about file handling experience doesn't end here. It turned out distributed APIs didn't mean to replace local APIs in all cases. As mentioned previously, the query sequence is matched against all database sequences; and these operations are performed in parallel in all cluster nodes. While the database sequences are automatically shipped to the nodes by the framework, the query arrives at the nodes by an explicit direction of the programmer using a "cache" feature of Hadoop, and cached locally. To read the query from the cache, we used the distributed API, and failed. We later found out the correct usage is with the regular Java file handling utility.

And the same "cache" feature needed to be used to distribute custom libraries to the nodes. Each task at a node runs as a separate local Java process, and all the libraries consulted need to be visible to the process.

These lessons, while may seem obvious to an experienced Hadoop programmer, took us sometime to figure out, attributable to the brevity and ambiguity of the documentation. Nevertheless, with the hands-on trial-and-fail approach, with testing and debugging again and again, one gains a good understanding of the underlying mechanism.

The debugging facility, one could say, still needs significant improvement. In Hadoop 0.16.4, a common means to show debugging information is via the Reporter feature, displaying information on the web report for each cluster-node. But it took us a while to locate this report through several layers of other reports. In Hadoop 0.17.0, it's heard that it comes with a more convenient debugging tool, which enables the analysis of log files on each cluster node.

Testing was made pretty handy with the plug-in. It enables the connection between the IDE and a Hadoop cluster so that testing can be done from within Eclipse. The connection, along with configuration parameters, is saved and managed within the IDE. We established connections to both the pseudo-cluster and the true EC2 cluster from our development environment.

We did most of the testing locally against the pseudo-cluster. At the beginning, when fairly new to HDFS, we got into an issue that took about one day to resolve. After any reformatting of the HDFS (reformatting whenever there's something "out-of-control" seems a habit of a beginner, particularly when this operation takes just seconds), everything messed up mysteriously, and we couldn't test the program. Spending hours digging deep into the system, we found that reformatting caused an inconsistency in the ID of the namespace between the master node and the data nodes; and this could simply be fixed by a manual editing of a text file. Later, people in the mailing list confirmed that was a bug that had not been resolved even in the latest release.

Finally, after some learning curve, we got the application completed and working.

How does AWS support the deployment and execution of Batch applications on its environment?

As mentioned, the developer has complete control over his EC2 instances, so typically, the deployment process involves uploading/installing the application along with all necessary software/resources to the instance(s). If the application is to be used over and over again, he may choose to bundle an AMI containing it so that later, he only needs to upload the new dataset to be processed by that particular execution while the program itself is readily runnable in the instance.

Having said that, deployment is solely the developer's responsibility. Support may come in the form of an AMI out there provided by AWS or a third party that already bundles some software required by the runtime environment (some runtime engine, some database software, and so on). The developer needs to pick one that is most suitable to his needs, if he doesn't want to build the instance from scratch.

Deploying and executing our application on EC2 cluster

Having a live cluster, this task, at the simplest manner, involves just copying from the local workstation the jar packaged application file, the libraries, the database file and the query file to a local directory in the master node; then putting them (except the application file) to the HDFS; and running the program just as you would with any jar executable file and waiting for the result.

However, to run a program efficiently, it may not be that simple. It requires configuring the cluster appropriately, a task that typically may need the expertise of an experienced administrator. A Hadoop cluster has a myriad of parameters that may stun a beginner looking at it; some may be set specifically for a MapReduce job programmatically, while some must be set globally by the admin. Nevertheless, one may give a job some trial runs and tune the parameters accordingly.

Initially, we ran our program, posing a mouse sequence query against the human sequence database that contains 72340 sequences, using default values for the parameters on a 20-node cluster. It took about 4 hours to finish. Since our program didn't use the reduce phase, we then got rid of all the reduce tasks, and re-ran the program with the same dataset; this time, it took slightly less than 2 hours to finish.

Realizing that the current setting only allows at most 2 map tasks to run simultaneously on a node, and observing that the CPU of nodes didn't seem to be used up, we changed a parameter so that each node could run as many as 300 simultaneous tasks, hoping the application would run in a more efficient manner. It turned out that it still took almost the same amount of time on the same dataset, particularly slightly less than 2 hours to finish the job. And by no means we can guarantee that this is the optimal execution on the given cluster.

During the execution, the master node console where the job is triggered reports the job progress. This information can also be obtained from a web report accessing port 50030 of the master node via HTTP. In fact, this web report is relatively comprehensive and contains links to more detailed reports, for example reports for every node which display information such as the progress of ongoing tasks, listing of completed tasks with task duration, failed tasks, and so on. Once the job finishes, this web report provides useful summary and detailed information of the execution.

Scalability of AWS+Hadoop for Batch applications

Having mentioned Hadoop as a prominent tool to develop batch applications that can exploit the combined power of an EC2 cluster given that the developer can start as many EC2 instances as he wants, we are interested in investigating the scalability of the AWS+Hadoop solution. We use the *Sort* program[30] available from the Hadoop distribution. It sorts a set of data consisting of random key-value records. Key size ranges from 10 to 1000 bytes, whereas

value size ranges from 0 to 20000 bytes. The program simply uses the MapReduce framework to sort the input directory into the output directory. The mapper is the predefined IdentityMapper and the reducer is the predefined IdentityReducer, both of which just pass their inputs directly to the output. The sorting effect is achieved by the sort operation inherent in the framework. This program features a typical batch application that handles a large dataset involving both heavy read and write operations.

The random input data is generated by *Randomwriter*[31], which is also a MapReduce program shipped with the Hadoop distribution. This program enables the control on how much random data is generated.

We varied the cluster size from 2 to 6 nodes, and the dataset to sort from 1Gb to 10Gb. Note that one node solely dedicates to the job of the master of HDFS and MapReduce framework, orchestrating the entire cluster; thus the actual work of sorting is done in the other nodes; so for a 2 node-cluster, the sorting is done in just 1 node, whereas for a 6 node-cluster, the sorting is done in 5 nodes. All clusters in the test used default values for their parameters.

Each cluster must have access to 10Gb of random input data, because it would have to sort up to that amount. Generating 10Gb of input data in a cluster then storing them only there thus would probably be inefficient, because when the experiments for the current cluster are done, the cluster needs to be shutdown [2] and data would be lost, and the new, one node larger cluster would have to repeat the time-consuming [3] input data generation step. Here, the advantage of Amazon S3 is realized. As a preparation step, we used an 11 node-cluster to generate 10Gb of data; 10 nodes actually performed the generation with each creating an approximately 1Gb file. We then backed up these 10 files to S3. After that, at the beginning of the sorting experiments for each cluster, we copied all 10 files to the cluster. Besides the benefit of having the back up data for possible future experiments, given the good network bandwidth[32] between EC2 and S3, we believe what we did was more efficient than otherwise.

For the experiments, we timed the operation in seconds, and came up with the following graphs (note that in the graphs, the cluster size only reflects the number of nodes actually doing the sorting).

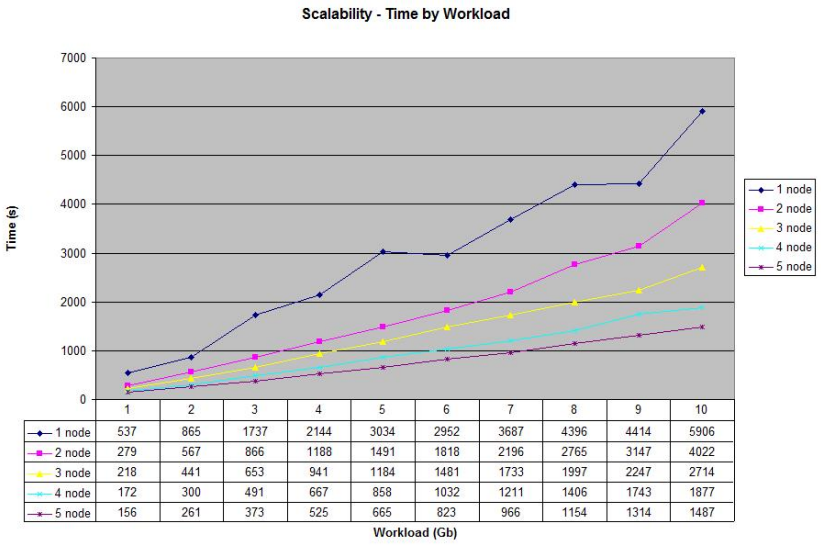
As seen in graph 2.1, each cluster can scale almost linearly with the workload; and as in graph 2.2, the operation scales with the cluster size.

It'd be interesting to see how the system behaves when the ratio of workload (in Gb) to cluster size (number of nodes) is fixed. Graph 2.3 shows the result with ratios of 1 and 2.

Because with some certain ratio, no matter what the cluster size is, each node roughly processes the same amount of data, we expected to see the sorting time to stay roughly the same for a certain ratio, hoping Hadoop would do a good job of distributing data evenly across the

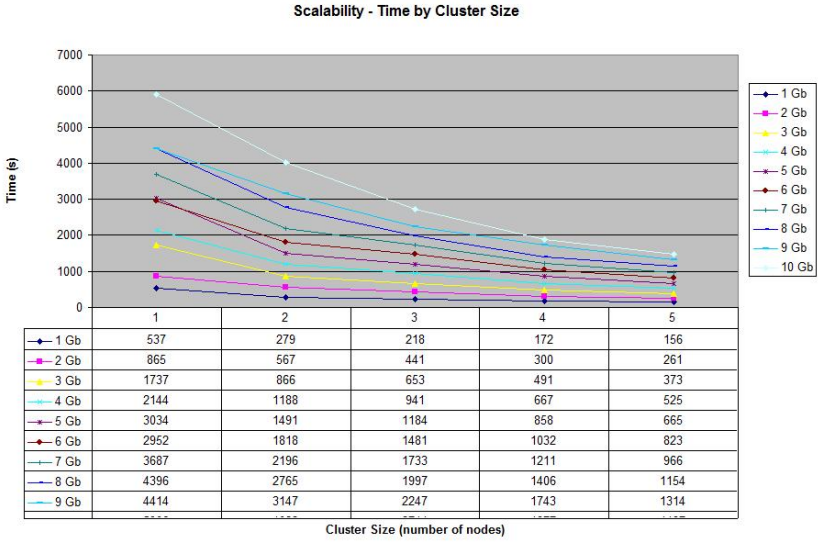
[2] there's a claim that a newer Hadoop version enables the addition of a new node on the fly to a cluster without the cluster having to restart

[3] in the order of tens of minutes for clusters with size up to 6 nodes; we in fact initially did try generating 10Gb of data in a 6 node-cluster, it took 809 seconds



	1	2	3	4	5	6	7	8	9	10
1 node	537	865	1737	2144	3034	2952	3687	4396	4414	5906
2 node	279	567	866	1188	1491	1818	2196	2765	3147	4022
3 node	218	441	653	941	1184	1481	1733	1997	2247	2714
4 node	172	300	491	667	858	1032	1211	1406	1743	1877
5 node	156	261	373	525	665	823	966	1154	1314	1487

Figure 2.1: Scalability - Time by Workload



	1	2	3	4	5
1 Gb	537	279	218	172	156
2 Gb	865	567	441	300	261
3 Gb	1737	866	653	491	373
4 Gb	2144	1188	941	667	525
5 Gb	3034	1491	1184	858	665
6 Gb	2952	1818	1481	1032	823
7 Gb	3687	2196	1733	1211	966
8 Gb	4396	2765	1997	1406	1154
9 Gb	4414	3147	2247	1743	1314

Figure 2.2: Scalability - Time by Cluster Size

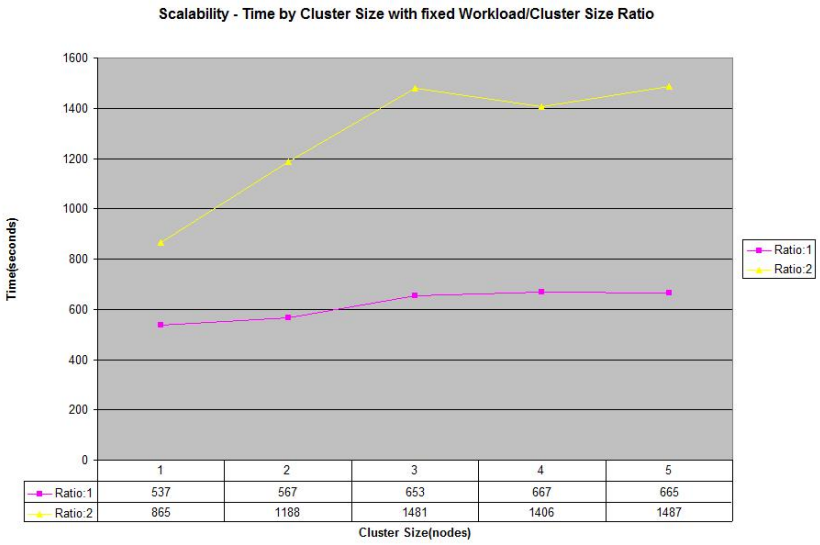


Figure 2.3: Scalability - Time by Cluster Size with fixed Workload/Cluster Size ratio

cluster. That seems to be the case, at least with clusters with sizes of 3 and higher, with time difference within margin of errors. The operations on the two smaller clusters were noticeably faster. We suspect it's because of less network overhead, particularly with higher ratios (because the amount of data that travels across the network in larger clusters is higher). However, all of these speculations need more thorough experiments with more data to be supported.

2.2.3 Interactive applications running on EC2

How does AWS support the development and testing of Interactive applications?

Similarly to batch applications, AWS does not provide direct support for the development and testing of interactive applications. The development activity can be performed directly on the instance, in which the support might come in the form of an available AMI facilitating the development effort, or preferably at the local workstation. In the latter case, AWS is viewed as a platform for execution and optionally, testing purposes; if solely done locally, there's no support by AWS for this activity.

How does AWS support the deployment and execution of Interactive applications on its environment?

As mentioned with batch applications, support for the deployment of interactive applications may come in the form of an AMI available out there that already bundles software supporting the application. The developer needs to pick the right one, then uploads/installs

the application along with any additional artifacts. Alternatively, he can build his own AMI containing the application and boot the instance from there.

The most significant support AWS provides for the execution of interactive applications is the network bandwidth. With this support, an application, typically a Web-scale application, can sit there in the AWS infrastructure and serve users world wide via the Internet.

We want to get a proof of this, thus decide to deploy a web application on AWS. The system of choice is a free, open source auction system, called RUBiS[12], developed by a team at Rice University. The version we use complies the LAMP model, with Linux (Fedora 8) as the operating system, Apache 2.2.8 as the web server, MySQL 5.0.45 as the database server, and PHP 5.2.4 as the dynamic web page generation engine.

RUBiS is modeled after eBay and implements the core functionality of an auction site: selling, browsing, and bidding. Complementary services like instant messaging or newsgroups are not incorporated. There are three kinds of user sessions: visitor, buyer, and seller. For a visitor session, users need not register but are only allowed to browse. Buyer and seller sessions require registration. In addition to the functionality provided during visitor sessions, during a buyer session users can bid on items and consult a summary of their current bids, rating and comments left by other users. Seller sessions require a fee before a user is allowed to put up an item for sale. An auction starts immediately and lasts typically for no more than a week. The seller can specify a reserve (minimum) price for an item. All user actions such as browsing items, bidding, buying or selling, and so on are defined as 26 interactions that can be performed from the client's web browser.

The MySQL database contains 7 tables: users, items, categories, regions, bids, buy_now and comments. The users table records contain the users name, nickname, password, region, rating and balance. Besides the category and the sellers nickname, the items table contains the name that briefly describes the item and a more extensive description, usually an HTML file. Every bid is stored in the bids table, which includes the seller, the bid, and a max_bid value used by the proxy bidder (a tool that bids automatically on behalf of a user). Items that are directly bought without any auction are stored in the buy_now table. The comments table records comments from one user about another. As an optimization, the number of bids and the amount of the current maximum bid are stored with each item to prevent many expensive lookups of the bids table. This redundant information is necessary to keep an acceptable response time for browsing requests. As users only browse and bid on items that are currently for sale, the item table is split in a new and an old item table. The very vast majority of the requests access the new items table, thus considerably reducing the working set used by the database.

The database is sized according to some observations found on the eBay Web site. There are about 33,000 items for sale, distributed among eBay's 20 categories and 62 regions. The history of 500,000 auctions is kept in the old-items table. There is an average of 10 bids per item, or 330,000 entries in the bids table. The buy_now table is small, because less than 10% of the

items are sold without auction. The users table has 1 million entries. Users are assumed to give feedback (comments) for 95% of the transactions. The comments table contains about 506,500 comments refering either to items or old items. The total size of the database, including indices, is about 1.4Gb.

We built a RUBiS server on our local machine. We installed and configured all the components of the LAMP stack, set up and configured RUBiS including loading the 1.4Gb database. Then we bundled the RUBiS server into an AMI and uploaded it to S3, using it to launch the auction site in a small EC2 instance. The process of bundling an AMI, uploading to S3, and booting an instance from there was facilitated by AWS-provided scripts and went smoothly.

With the live auction site standing in the Amazon cloud, we had a fun time browsing/selling/bidding via the system using our web browser. And anyone in the world with Internet access and knowledge of our systemś AWS-generated URL could do the same.

Scalability of AWS for Interactive applications

In the context of interactive applications, for scalability we are concerned with the ability to respond to the increase in the number of concurrent users. This further drills down to the ability of one instance to respond to the increase, and once one instance exhausts, the ability of the environment to allocate additional instance to keep up with the workload.

The auction site deployed earlier was used for scalability evaluation. RUBiS comes with a benchmarking tool which is a client program that emulates usersb́ehavior for various workload patterns. There are two workload mixes available for the tool: a browsing mix made up of only read-only user interactions and a bidding mix that includes 15% read-write user interactions. In our experiments, we use the bidding mix, which is the most representative of an auction site workload.

The client browser emulator program spawns threads that initiate customer sessions - a customer session is a sequence of interactions for the same customer. For each session, the client emulator opens a persistent HTTP connection to the Web server and closes it at the end of the session. Each emulated client waits for a certain think time before initiating the next interaction. The next interaction is determined by a state transition matrix that specifies the probability to go from one interaction to another one. The think time and session time for all benchmarks are generated from a negative exponential distribution with a mean of 7 seconds and 15 minutes, respectively.

For the bidding mix we choose, each thread initiates two customer sessions. We vary the load on the site by varying the number of threads. Comparing the number of completed sessions, reported after each experiment, with the number of threads spawned, we could tell if the server was overloaded during the experiment.

In configuring the RUBiS server before bundling the AMI, we set the maximum number of simultaneous connections allowed to MySQL database to a very high value, 4096; for our experiments, this practically sets no limit at the database side for the client emulator program. We set the maximum number of simultaneous connections allowed to the Apache Web server to 256; for a small EC2 instance, when the number of concurrent HTTP connections approaches this threshold, there should be a lot of memory swap happening, which is one cause for the saturation point. That was basically all the preparation done at the server side for the scalability test.

First, we used one client machine to initiate 200 threads. There were 360 sessions successfully completed, meaning the success ratio which is the number of completed sessions divided by the number of threads is 1.8, considered a high number. In our experiments, according to our own observation and definition, a success ratio is considered high if it's greater than 1.5, indicating the saturation point was not hit. An ideal success ratio would be 2.

We raised the number of threads by 100 for each new experiment, and found out that at 600 threads, the ratio dropped to 1.443, an indicator of the occurring of the saturation point. The ratio dropped further with more threads in further experiments, confirming that the saturation range had been entered.

To determine whether the client or the server saturates at 600 threads, we divided this workload to two identical client machines, each hosting 300 threads. An earlier experiment verified that the client machine is well capable of handling 300 threads. Experiment with this two client machines - one EC2 server machine configuration yielded a success ratio of 1.54, within the margin of error of the saturation point, suggesting that this point lies at the EC2 server, and the server will not scale very well beyond the load of 1200 customer sessions.

To verify this hypothesis, we planned to add one more EC2 server from the same AMI but playing the role just as the Web server connecting to the database hosted in the first EC2 server. That would give a RUBiS configuration with 2 web servers accessing the same database server, which is hosted on the same physical machine with one of the web servers. The two client machines, each hosting 300 threads, then would target these two servers. We would expect to see a boost in the success ratio, indicating that by adding one more web server, the saturation point of 1200 customer sessions at the server side would have been cleared.

We would further expect that having n web servers, RUBiS will scale well up to roughly 1200n customer sessions, provided that the connection limit at the MySQL database side is raised accordingly if possible when necessary. With two web servers - two client machines, we would expect to raise the number of threads for new experiments to verify this. We would further expect to repeat the experiments for a configuration of three web servers and possibly more and plot results on a graph.

It was a pity that we were hit by some technical issues lying in the LAMP stack setting up multiple-server configurations. We were not yet able to resolve the issues, thus we didn't have a reliably working multiple RUBiS server system to prove the hypothesis. Graph 2.4 shows only

the result for the case of one RUBiS EC2 server.

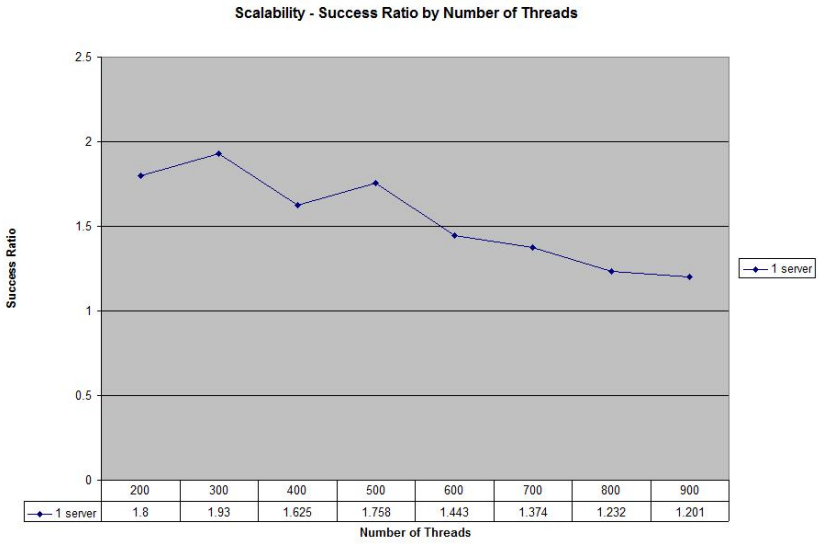


Figure 2.4: Scalability - Success Ratio by Number of Threads

2.3 Conclusion

AWS, particularly EC2 and S3, provides a simple interface for accessing infrastructure services in the cloud. The overall offering is quite basic and thus very general and doesn't impose any programming model. EC2 essentially provides Linux-based instances [4] in the cloud that lay out the platform for running any type of Linux-supported applications. AWS itself basically doesn't provide any development support and doesn't care how you develop your app; you can use any tool/model/framework/skill of your choice, as long as the resulted program can run on the given Linux platform, you are good to go. S3 provides storage repository in the cloud that could be used alone as a data backup service or in conjunction with other services, as in our experiment in which it is used as a permanent storage of data of the EC2-based app.

The generality and openness of the service let third-party vendors step in and bundle various AMIs with their products. Available AMIs out there, free or charged, can be found with software such as databases (Oracle, MySQL), Web hosting (Apache HTTP), development environments (JBoss, Ruby on Rails), among others.

With good application execution support, such as the network bandwidth and one's ability to fire up many EC2 instances, AWS promisses a true potential. Large-scale web sites

[4] More operating systems have been recently supported, as mentioned in chapter 6.

could be deployed with less pain at lower cost yet having good infrastructure scalability, high-performance computing type batch jobs are no longer out of reach of low-budget groups [5]. In fact, we learnt that a number of organizations started to use AWS for serious purposes, particularly the Hadoop+EC2 solution, among which are The New York Times, Veoh (http://www.veoh.com/, the online video provider), among others [6].

[5] All our expenses for AWS, including those paying for EC2 instances running both applications and the sort experiment many times, those paying for months of storage of the RUBiS AMI and the input data for the sort experiment, and those paying for the bandwidth consumed in and out of the AWS network, are affordable by the restricted budget of a single graduate student.

[6] For a more comprehensive list of Hadoop+EC2 users, go to http://wiki.apache.org/hadoop/PoweredBy, search for the keyword "EC2".

Chapter 3

Sun Grid Compute Utility

3.1 Introduction

Sun Microsystems provides access to a set of infrastructure Web services at http://network.com. One of the offerings here is the pay-as-you-go Sun Grid Compute Utility, which provides customer access to standardized parallel computing resources. An application developer accesses this service through the Web-based portal or through programs that invoke the sungrid API. Through the portal, developers can upload applications and data, or users can select an application from a catalog of pre-installed applications. Applications execute on a networked infrastructure of compute nodes and can distribute processing tasks across nodes, allowing execution to occur in tandem or in parallel. When execution completes, the user downloads run results, either through the Web-based portal or the API. Software developers can also provision application services on the Network.com infrastructure, enabling immediate access to services via the Internet.

All Network.com execution nodes providing this service are identically configured with a rich software environment featuring the Solaris Operating System on x64, the Java programming language, and C, C++, and FORTRAN compilers and libraries. The Sun N1 Grid Engine software[33] acts as a Distributed Resource Manager for execution nodes, and application scripts can invoke Sun N1 Grid Engine software commands to create multiple jobs and distribute processing tasks. Applications that invoke Message Passing Interface (MPI)[34] calls can also execute through a pre-installed MPICH[35] implementation.

Generally speaking, there are three use cases for applications that access the Sun Grid Compute Utility:

- User applications: Developers can write or port applications in a local development environment, executing them through the Web-based portal interface or via the sungrid API.

- Catalog applications: Network.com offers on-demand access to a set of popular, preinstalled, and ready-to-use applications. Independent software vendors (ISVs) and other developers can make their applications available for shared use on Network.com. Optionally, they can restrict application access, compelling users to purchase and obtain license tokens prior to execution.

- Long-running services: Application providers can provision long-running services on Network.com to give customers ready access to business services.

From the developer perspective, the user applications is the fundamental use case of the service. Figure 3.1 illustrates the sequence of steps for running an application developed for upload and execution on the Sun Grid. As the figure shows, using the Web portal to upload and execute an application parallels the process of using the API to run an application on the site.

Before application resources are uploaded to Network.com, it is assumed that all code development, debugging, and testing happens in a local development environment. The local environment can be a platform for Java language development, or it may include C, C++, or FORTRAN compilers and the Solaris OS on an x86/x64 platform. In either case, applications can invoke Sun N1 Grid Engine software directives to spawn multiple jobs or tasks that can execute in parallel.

Once an application works properly in a local environment, it can be bundled with related scripts, libraries, executable binaries, and input data into resource archive files, which are uploaded to Network.com. The user then defines a job for execution, specifying the name of the executable, arguments, resource files, and so forth. Finally, the user initiates a job run.

The Sun Grid Compute Utility manages the job run, moving resources to a job-specific, temporary home directory to isolate and protect user data. It unpacks resource archive files and stages the run, submitting the executable to the Sun N1 Grid Engine master, which in turn distributes processing tasks to execution nodes. When the job run completes, the Sun Grid Compute Utility collects any files that were created or modified in the user's home directory hierarchy during execution. The modified files are placed into a results archive for the job, which the user can download via the portal.

The program flow of an application that uses the API to submit jobs for execution mimics the sequence of steps in using the Web-based portal. The sungrid API allows developers to create Java client applications that create a session on Network.com, request user authentication, create job resources and definitions, submit jobs for execution, and download results after run completion.

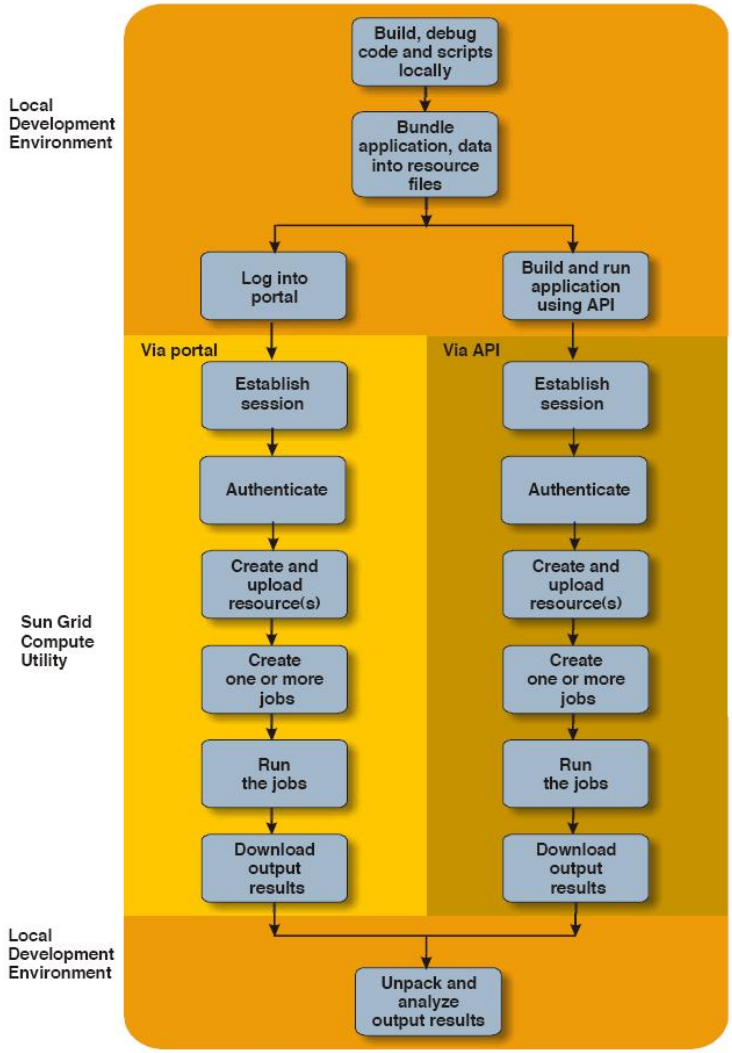


Figure 3.1: Sequence of steps for running an application on Sun Grid

3.2 Building a user application

Since it's a fundamental use case of the Sun Grid Compute Utility, we choose to implement this use case to evaluate the environment. The language chosen is Java. To facilitate the development of Java applications that can fully exploit the potential of this grid based infrastructure, Sun provides a technology called Compute Server[36] as an open-source project. It is intended to help Java developers more easily and efficiently use the Sun Grid Compute Utility for the distributed execution of parallel computations. It is designed to be used by Java developers who may be more familiar with their application's subject matter than with parallel programming environments.

In its current early access state, the Compute Server technology comprises:

- Compute Server programming model: to simplify the design and development of parallel computing applications.
- Compute Server execution framework: to support the efficient execution of Compute Server applications on the Sun Grid Compute Utility
- Grid Compute Server plug-in for the NetBeans IDE: to ease local development, testing, and packaging of Compute Server applications for uploading to the Compute Utility, and to facilitate off-grid pre- and post-processing of the grid job's input and output.

The programming model employed by this technology is the generic master-worker pattern and variations of it. In this pattern, a master process distributes independent computational tasks to multiple worker processes. The worker processes operate on their tasks independently and in parallel. As workers complete their tasks, they may return their results to the master for final aggregation and/or reduction.

Supporting this master-worker computing paradigm, Compute Server technology includes generic master and worker components, as well as other infrastructure elements that manage flow of work through the system. Application-specific code is plugged into this framework: customized task generator code runs within the master to create application-specific tasks, that are then distributed to and executed by the workers.

Expanding the applicability of the basic master/worker computing pattern, the Compute Server programming model allows developers to construct applications as sequences of parallel execution phases, with the output of one phase becoming the input to the next. This simple chaining capability allows developers to create more sophisticated parallel computing applications while preserving the simplicity of the overall programming model.

Compute Server technology also features a deployment architecture, recognizing the need to support local pre- and post-processing of job input and output, as well as on-grid execution of the Compute Server job. It defines an Application class and deployment APIs to help developers

address these needs. Developers create a locally executable application that may, optionally, collect and prepare input to the grid job. Within this application, developers use Compute Server deployment APIs to package input data and application code as needed for uploading to Sun Grid. Once this local application has been run, the grid resources that is produced can be uploaded to the grid, the grid job can be executed, and the job results can be downloaded. The deployment APIs can also be used to extract results from the downloaded grid job output file, enabling developers to perform results post-processing within their local application.

The Grid Compute Server Plug-in for the NetBeans IDE extends this development environment with specific features that further simplify the creation of Compute Server applications. It provides project and class templates and online help to ease application development, as well as tools that address the unique demands of Compute Utility development, such as the local debugging support for both the local application and the on-grid executable code. The Plug-in also provides GUI-based mechanisms for specifying numerous application configuration properties, providing an alternative to manually coding these options.

The entire process of developing and executing a Compute Server technology enabled application on Sun Grid is summarized in figure 3.2:

3.2.1 How does ComputeServer support the development and testing of applications for Sun Grid?

First, we set up the development environment. Besides the Grid ComputeServer Plug-in, we used another tool from Sun called Sun Grid Module Suite Plug-in[37] for NetBeans IDE that enables grid job upload/execution/download directly from NetBeans. We found a little trouble installing and assembling all of these pieces of software (NetBeans, the two plug-ins, and the JDK) together because their latest versions were not compatible, and the documents of the two plug-ins were not consistent. After some trials and consultation from the Compute Server technology mailing list[38], finally we were all set with setting up the development environment. We used the latest version of the two plug-ins (Grid ComputeServer 0.8 and Sun Grid Module Suite 1.0.44) with NetBeans5.5 and JDK5 (the latest versions at the time of the experiment were NetBeans6.0 and JDK6).

We adopt the same batch program that we use for Amazon Web Services for Sun Grid. Recall that this program executes a query against a biological sequence database that contains many individual sequences. This execution consists of the matching operations between the query sequence and all individual sequences in the database. Since all of these matching operations are independent, this program is a perfect fit for running on Sun Grid (see the appendix for further information about this program).

MapReduce[18] is supported by the ComputeServer's master-worker programming model, so retaining this design pattern is a natural choice. Like in AWS, we only use the map phase. Be-

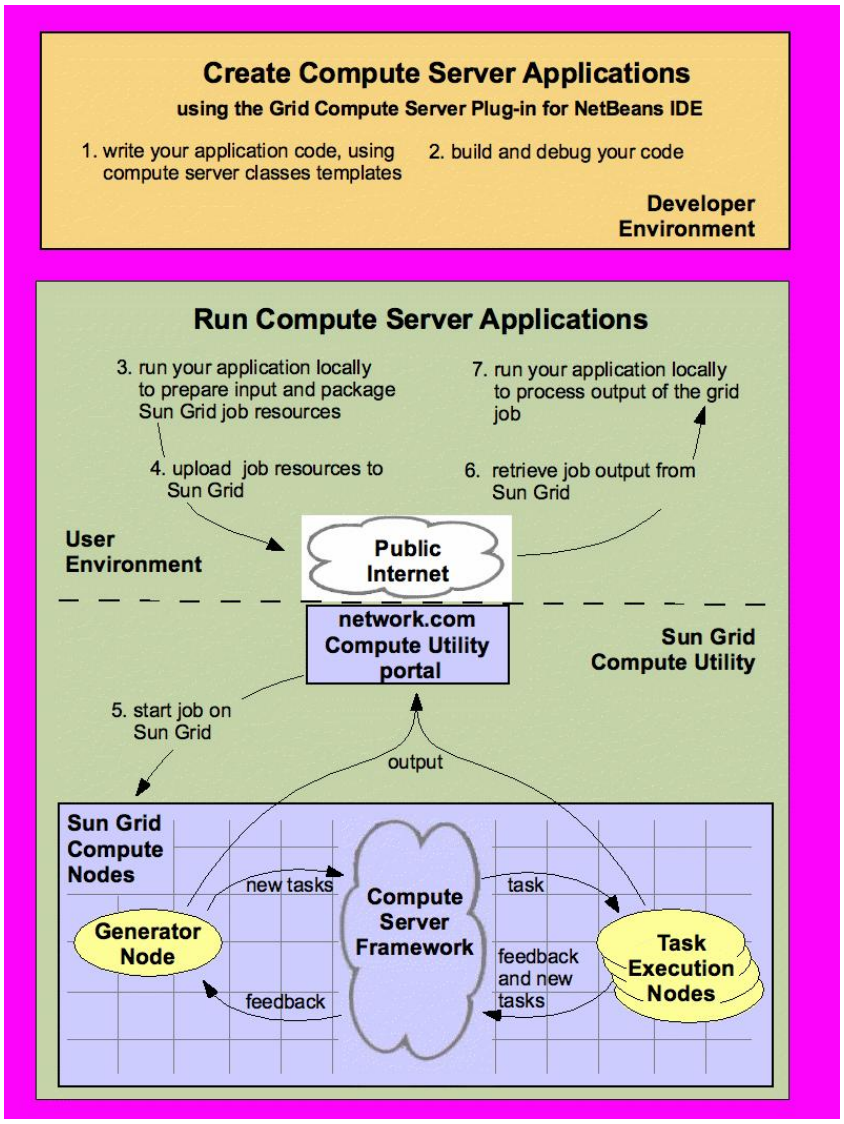


Figure 3.2: Create and run Compute Server applications

sides this on-grid program, we need to develop the off-grid programs for pre- and post-processing of the grid job's input and output. Additionally to class templates, the plug-in also provides the canonical WordCounter sample project that features an on-grid MapReduce compliant program and off-grid input/output processing programs; these were valuable in getting us understand various elements of ComputeServer and in speeding up the development process.

We didn't find much of a learning curve in the development process. A developer without prior exposure to MapReduce and distributed programming environments should experience the same. For the on-grid program, major elements that may be unfamiliar to such a developer are the master component and the accompanying map generator that creates map tasks, worker components and map tasks executed there, and other components, most notably the Input/Ouput Maps, that support the overall flow of work through the distributed system. These elements, in fact, are relatively straightforward to apprehend and somewhat don't bear the distributed nature to the extent similar frameworks such as Hadoop[10] do. ComputeServer gives developer an impression that given a network of individual nodes, it's basically a plugin to connect the nodes together without providing an abstraction that aggregates the entire network as a single resource, such as the HDFS[17] in Hadoop. In fact, ComputeServer does not feature any distributed file system. As a result, the awareness of physical nodes emerges pretty clearly in writing the program. For example, we had to code a fully contained map tasks generator class that locates sequences in the database file for further processing; in coding it, we were fully aware that this class was to be executed solely in one single node, which is the master node. In comparison, in Hadoop, the same task is accomplished by implementing methods of several classes and interfaces and these computations are distributed across the cluster behind the scene by the framework without the developer being aware of the physical location of the execution.

The awareness of individual node execution results in a familiar single-machine programming experience. Except for a few cases, when writing code to serve a particular purpose, for the most part, the developer uses the same APIs that work for stand-alone programs given all the capability and resource limitation of the single machine environment. For example, in the map tasks generator class that runs on the master node, the way to deal with the large database file is to map it to the node's memory using the standard Java libraries-provided memory mapped file facilities. In comparison, in Hadoop, this large file exists in the HDFS context, and meant to be processed by Hadoop-provided file handling facilities, not the traditional memory mapped file method. The few cases that feature practices specific to the distributed nature of the network mostly concern with inter-node communication, such as the distribution of map tasks to worker nodes via queues (in fact, this is almost transparent to developer by the framework), and job input/output, such as the retrieval of input data from the original resource files and writing of output data when the job is done (this is facilitated by a familiar map-type structured, simple data abstraction called Input/Output Maps).

Given the traditional way of file handling as mentioned in the previous example, dealing

with trulely large datasets is a challenge in this framework. Without an accompanied storage infrastructure that collectively accumulates the storage capacity of all the nodes under a single abstraction layer, such as a distributed file system, ComputeServer can't handle large files in many Gbs to Tbs in size, as what Hadoop can do with HDFS. This fact probably aligns with, or may be the consequence of, Sun Grid, as marketed by Sun, being an environment for compute-intensive applications, not data-intensive ones. In fact, the current limit on the amount of data uploaded for a job execution is only 10 Gbs.

Nevertheless, given the familiar single-machine programming style, a limited set of distributed APIs exposed by ComputeServer, the learning curve is small, we found little trouble in implementing the programs, on-grid and off-grid. The local testing was done within the NetBeans IDE following the same steps as running the real job: the pre-processing program ran first, packaging the sample test data (sample database file, sample query file) and other parameters along with the on-grid executable into a single resource archive file; the Grid ComputeServer plugin then enabled taking this resource archive file and running the executable against the given test dataset on the local machine as if on the true grid, creating the grid output file; finally, the post-processing program handled the resulted output file, printing out human-readable outcome of the grid job. The local debugging took full advantage of the debugging facilities of NetBeans, added by some handy features of the Grid ComputeServer plugin, easing the spotting out and handling of issues in the coding process.

3.2.2 How does Sun Grid support the deployment and execution of applications?

With the application in place, we were ready to deploy and run it on Sun Grid. We used the same dataset used in the experiment with AWS EC2: a mouse sequence query posed against the human sequence database containing 72340 individual sequences. The local pre-processing program run packaged this dataset and appropriate parameters along with the on-grid application into a resource zip file of 18.2 Mbs in size. Via the Web-portal, we uploaded this file to Sun Grid, created a new job based on it, initiated the job execution and waited for the result. While the job was running, there was constant update on the portal about its progress, such as the number of CPU hours consumed. One thing to note is that the execution is charged by CPU hours, so this information may be of concern; the Grid Compute Server plugin allows developer to specify the limit for CPU hours as well as job duration when packaging the resource file for upload. When the job finished, an output file was generated, and a short summary about the execution was available in the portal for later review. We downloaded the output file, and ran the local post-processing program to verify the result.

We repeated this process a couple of times at different times of the day during the course of several days. The job duration varied from around 2.5 hours to less than 4 hours. This could

be due to several factors, for example the job could take longer when the network is busier serving more concurrent jobs resulting in lower task throughput of our job. Surprisingly, the number of CPU hours also differed, although less, between different runs on the same dataset and parameters. This could also be due to several factors, for example the differring locality resulted from the different number of workers assigned to our job because of different workloads imposed on the network by other jobs, differing conditions such as network contention and so on.

In fact, our experiment was not a smooth process. Before reaching a reliable point, we ran into several issues. First, there was a problem of reading the output data. Output of the execution was a set of match results between the query and the sequences in the database. This set was placed on the Output Map by the on-grid program as a set of key-value pairs, so that later the off-grid post-processing program could retrieve the data. For each pair, essentially the key is the identification of the database sequence, and the value contains detailed result of the match. However, somehow our off-grid program couldn't at all read values in this Ouput Map, while keys were retrievable. This issue didn't occur for smaller datasets. Finally, we had to opt to a workaround in which both the sequence identification and the detailed match result were combined into the key in the Output Map, while the value of the Output Map stayed empty. This workaround helped retrieve the data wanted.

The second issue concerned with job statistics. Each of our grid run was supposed to return this, including information such as the number of tasks generated, average time taken to process a task, number of tasks failed, task throughput, and so on; but they never did. We contacted the Compute Server team for help, sending them our output zip files. Examining these files, they said many files that were supposed to be there were missing. The way Sun Grid works is that for each job, upon completion, it collects all the files created or modified in the job's home directory hierarchy since the job started to include in the output archive. The Compute Server team suspected the system clocks in some nodes were off, causing the problem, and suggested us to contact Sun Grid support. After some investigation, Sun Grid support explained the issue was due to a limit on the zip file size imposed by their system, and suggested us to use tar instead as the format for the job output file; they said they were able to re-run our job and the resulting tar file contained all the missing files.

There was one execution in which not only job statistics but all job results were missing. We were not able to re-create the problem. Sun Grid support attributed this to the same zip file size limit issue.

Talking about their support, the Compute Server team was very helpful, but the Sun Grid team was not very responsive. Their policy stated that customer requests usually are responded within 48 hours, but most of the time we had the feeling that we were forgotton. They usually only responded after a significantly longer period and further urges from us.

We finally didn't get chance to perform more experiments and try the tar format for output file to see if job statistics are included. Sometime later, after some (pretty long) delay in

correspondence with Sun Grid support and us working on other cloud computing environments, when we came back, to our surprise, we couldn't log in to the Sun Grid network, being explained that our account violated some "legal issue". After further (pretty long) delay in correspondence, they explained the issue was due to their new changes made to the system, and suggested a temporary solution in which we had to create a new account and have them move all current data from the existing non-working account to this new one, while waiting for the upgrading process to complete. After some more time, when we once more came back, the service was totally closed. They announced they are in transition and adding some "exciting new options". As of the time of this writing, the service is not back yet.

In discussing with the Compute Server team, they revealed that when a Computer Server-implemented job runs, initially the framework allocates 20 slots (a slot represents a CPU core on Sun Grid) to the job. Then it automatically varies the number of slots, with the max of 198, to match the actual workload. We were not able to observe this because we were never able to obtain the job statistics.

3.3 Conclusion

Sun Grid is a promissing grid infrastructure for developers to exploit the compute power in the cloud. The current offering suits compute-bound applications whose hardware demand, particularly CPU demand, exceeds one's infrastructure capability.

Compute Server is a software framework built upon Sun Grid and provides a simple yet powerful programming model for applications to exploit the parallel nature of the grid. Developer finds this tool easy to approach, but it mostly targets compute-intensive applications only.

We believe in today's world, more and more compute-bound programs couple with the requirement to handle large datasets. To attract more users, Sun Grid needs to address this need. The grid's storage resource needs to be exposed to users, and a storage abstraction layer over the entire network probably needs to be added, along with more physical storage capacity to each node if necessary. It'd be interesting to see if Compute Server or a similar framework could retain the simplicity of its programming model in this case.

Despite the attractive nature, Sun Grid proves to be still in an early development stage. It needs significant improvement of features as well as better customer service to reach its full potential. As claimed by the vendor, Sun Grid is in a transition period. We'd be interested in seeing how it gets better after it reopens.

Chapter 4

Google App Engine

4.1 Introduction

4.1.1 What is Google App Engine (GAE)

GAE[8] lets one run web applications on Google's infrastructure. App Engine applications are easy to build, easy to maintain, and easy to scale as traffic and data storage needs grow. With App Engine, there are no servers to maintain: one just uploads the application, and it's ready to serve users.

The GAE environment includes the following features:

- dynamic web serving, with full support for common web technologies.
- persistent storage with queries, sorting and transactions.
- automatic scaling and load balancing.
- APIs for authenticating users and sending email using Google Accounts.
- a fully featured local development environment that simulates GAE on the developer's computer.

Currently Python[39] is the only programming language supported by GAE. The runtime environment includes the full Python language and most of the Python standard library. More languages are expected to be supported in the future.

4.1.2 The Sandbox

The runtime environment features a sandbox so that applications run in a secure environment that provides limited access to the underlying operating system. These limitations

allow App Engine to distribute web requests for the application across multiple servers, and start and stop servers to meet traffic demands. The sandbox isolates the application in its own secure, reliable environment that is independent of the hardware, operating system and physical location of the web server.

Examples of the limitations of the secure sandbox environment include:

- An application can only access other computers on the Internet through the provided URL fetch and email services and APIs. Other computers can only connect to the application by making HTTP (or HTTPS) requests on the standard ports.
- An application cannot write to the file system. An app can read files, but only files uploaded with the application code. The app must use the App Engine datastore for all data that persists between requests.
- Application code only runs in response to a web request, and must return response data within a few seconds. A request handler cannot spawn a sub-process or execute code after the response has been sent.

4.1.3 The Python Runtime Environment

The Python runtime environment currently uses Python version 2.5.2.

The environment includes the Python standard library[40]. Of course, calling a library method that violates a sandbox restriction, such as attempting to open a socket or write to a file, will not succeed. For convenience, several modules in the standard library whose core features are not supported by the runtime environment have been disabled, and code that imports them will raise an error.

Application code must be written exclusively in Python. Code with extensions written in C is not supported.

The Python environment provides rich Python APIs for the datastore[41], Google Accounts[42], URL fetch[43] and email[44] services. App Engine also provides a simple Python web application framework called webapp[45] to make it easy to start building applications.

For convenience, App Engine also includes the Django web application framework[46], version 0.96.1. Note that the App Engine datastore is not a relational database, which is required by some Django components. Some components, such as the Django template engine, work as documented, while others require a bit more effort.

Developer can upload other third-party libraries with the application, as long as they are implemented in pure Python and do not require any unsupported standard library modules.

4.1.4 The Datastore

App Engine provides a distributed data storage service that features a query engine and transactions. Just as the distributed web server grows with the traffic, the distributed datastore grows with the data.

The App Engine datastore is not like a traditional relational database. Data objects, or "entities," have a kind and a set of properties. Queries can retrieve entities of a given kind filtered and sorted by the values of the properties. Property values can be of any of the supported property value types[47].

The Python API for the datastore includes a data modeling interface that can define a structure for datastore entities. A data model can indicate that a property must have a value within a given range, or provide a default value if none is given. The application can provide as much or as little structure to the data as it needs.

The datastore uses optimistic locking (details in section xyz) for concurrency control. An update of an entity occurs in a transaction that is retried a fixed number of times if other processes are trying to update the same entity simultaneously. The application can execute multiple datastore operations in a single transaction which either all succeed or all fail, ensuring the integrity of data.

The datastore implements transactions across its distributed network using "entity groups". A transaction manipulates entities within a single group. Entities of the same group are stored together for efficient execution of transactions. Application can assign entities to groups when the entities are created.

4.1.5 App Engine Services

App Engine includes a service API[42] for integrating with Google Accounts. Using Google Accounts lets the user start using the application faster, because the user may not need to create a new account. It also saves the developer the effort of implementing a user account system just for the application. The Users API can also tell the application whether the current user is a registered administrator for the application. This makes it easy to implement admin-only areas of the site.

Applications can access resources on the Internet, such as web services or other data, using the URL fetch service[43]. This service retrieves web resources using the same high-speed Google infrastructure that retrieves web pages for many other Google products.

Applications can send email messages using the mail service[44]. This service uses Google infrastructure to send email messages.

The Memcache service[48] provides the application with a high performance in-memory key-value cache that is accessible by multiple instances of the application. Memcache is useful for data that does not need the persistence and transactional features of the datastore, such as

temporary data or data copied from the datastore to the cache for high speed access.

The Image service[49] lets the application manipulate images. With this API, one can resize, crop, rotate and flip images in JPEG and PNG formats.

4.1.6 Quotas and Limits

App Engine imposes various resource limits on application. Application resource limits, or "quotas," are refreshed continuously. If the application reaches a time-based quota, such as bandwidth, the quota will refresh at the beginning of each calendar day. Fixed quotas such as storage use are only relieved when the application decrease usage.

Some features impose limits unrelated to quotas to protect the stability of the system. For example, when an application is called to serve a web request, it must issue a response within a few seconds. If the application takes too long, the process is terminated and the server returns an error code to the user. The request timeout is dynamic, and may be shortened if a request handler reaches its timeout frequently to conserve resources.

Another example of a service limit is the number of results returned by a query. A query can return at most 1,000 results. Queries that would return more results only return the maximum. In this case, a request that performs such a query isn't likely to return a request before the timeout, but the limit is in place to conserve resources on the datastore.

4.2 The Datastore

From the developer perspective, the App Engine datastore is the most unfamiliar component in all the offerings. Thus, this is the focus of our GAE investigation.

The datastore provides persistent storage at the data layer in the multi-tier web architecture. Traditionally, a DBMS, typically a RDBMS, functions at this layer. However, being a platform potentially serving millions of developers with possibly even more applications, utilizing DBMS at the data layer for GAE seems a poor choice.

Assigning one DBMS instance for each application or developer account would create an impractical administration burden and miserably poor utilization of resources. Creating one schema for each application or developer account would blow out the data dictionary of any DBMS. Somehow putting data of all applications into one schema? One DBMS instance would not scale to thousands of machines and thousands of disks, which is the potential demand for GAE's user base.

The core issue here is scalability. GAE tackles this by inventing their own data storage technology. The datastore bases on Google's proprietary distributed storage system for managing structured data, called BigTable[50]. Thus, it inherits scalability of the BigTable. The following section examines the underlying design of the datastore on the BigTable.

4.2.1 The datastore design on the BigTable

The BigTable is a scalable, structured storage which could essentially be considered as a giant sharded, sorted two dimentional array. Figure 4.1 reflects the core idea of its structure.

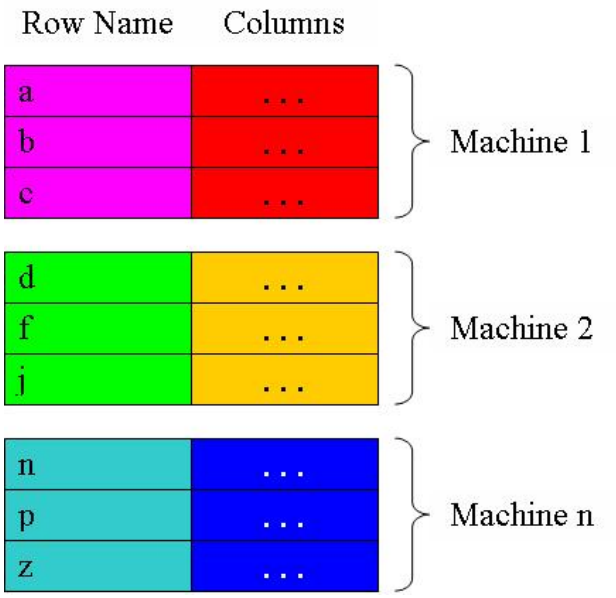


Figure 4.1: BigTable's structure at a high level

The BigTable consists of a set of rows, each has a row name and a set of columns. The rows are sorted by their row name and split across machines (thousands or more machines). Atomic operations provided by the BigTable for each row are read, write, delete, and single-row transaction (read a row, perform some operation and write it back, for example the increment); another important efficient operation is row scan by prefix or range of row names.

In GAE, each unit of access by applications in the datastore, a data object, is called an entity. An entity is in fact a row in the Bigtable. Particularly, the BigTable maintains one single table that stores all entities in all applications in GAE, called the *Entities* table. This *Entities* table is generic, schemaless. Besides the row name, it has only one column which stores the serialized data of entities. Entity key that uniquely identifies the entity functions as the row name.

With this organization, the App Engine datastore is scalable with the BigTable to a large number of machines when data grow.

4.2.2 Entity and Entity Group

From this section, we will look at entities from the perspective of App Engine applications. Instead of consisting of a row name and a single column as what can be seen from the

BigTable's perspective, an entity now consists of a key and one or more properties, named values of one of several supported data types[47].

The concurrency control mechanism used by the datastore is a variant of the multiversion timestamp control[51] (next section). Timestamp values are not real time, but counter-based. Multiple versions of each entity with their associated timestamps are physically stored with the entity. Following, figure 4.2, is a illustration of this. For simplicity, the entity in this figure has only one property, and timestamp values are consecutive integers. As illustrated, the key's value is 'A'; the entity has two versions currently stored whose values are 'X' and 'Y' and their timestamps are 0 and 1, respectively. Note that 'Y' is a newer version than 'X'.

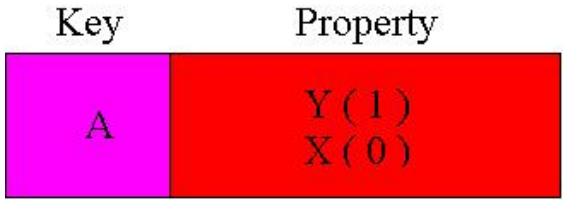


Figure 4.2: An entity

Entities may stand alone or be grouped into groups. Each group has a root entity, simply called root, which has children. Each child in turn has children, and so on. This way each group is shaped like a tree. The grouping of entities (creation of a new root, assignment of new entities into a group, and so on) is done by the application. Keys of entities in the same group are structured such that these entities are physically contiguous in the BigTable. Stand alone entities are in fact roots of single-entity groups.

A root, besides the key and properties, has a transaction journal to record changes made by ongoing transactions and a last committed timestamp to reflect which transaction last committed. These information are used for the root's entire group (detailed in the next section). Following, figure 4.3, is a simplified illustration of a root. Here the journal has one entry indicating that the value 'Y' is to be applied to an entity whose key is 'A' (which in this case is the root) by a transaction whose timestamp is 1; the last committed timestamp indicates that the transaction last committed on this entity group is the one with timestamp 1 (so in this example, the change recorded in the journal has been successfully applied to the entity and committed, and now has become the official value of the entity).

The versions of an entity in general and the journal entries of a root in particular that are no longer needed are garbage collected by separate processes.

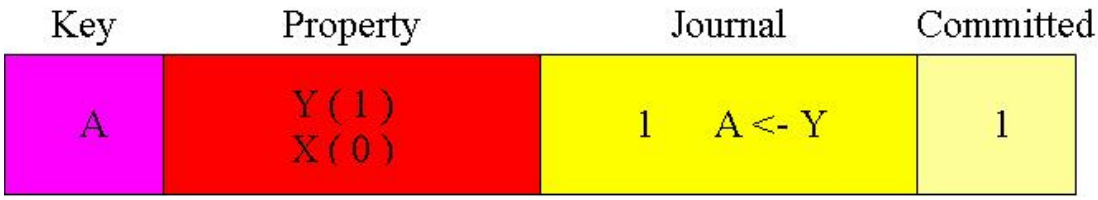


Figure 4.3: A root entity

4.2.3 Concurrency control rules

Each transaction can manipulate entities in only one group. There can never be any two entities belonging to two different groups that are accessed (inserted, updated, deleted, retrieved) in one single transaction. Thus, grouping entities into groups can actually be thought of as organizing data to facilitate transactions. Data related in some way, for example a checking and a saving account of a person that may potentially involve in a money transfer transaction between each other that requires the credit and the debit operations to eitheir both succeed or fail, should be considered to be put into the same group.

Datastore operations in a transaction consist of read and write actions against entities of a group. Before touching on rules governing the concurrent execution of transactions, let's examine what are underlying these datastore operations.

The read action

When a read is performed on an entity E (E may or may not be the root), the scheduler may choose to retrieve any of the versions available in the entity. There are cases in which the version retrieved is not the current version of E (detailed in the next section)

The write action

Illustrated in figure 4.4 is a group where a write is about to happen. We simply refer to entities by their key's value. The root A and entity B have current values 'Xa' and 'Xb', committed by transactions with timestamp 0 and 2, respectively. The last transaction that committed on this group, however, is 3 that updated 'Xc' to entity C (not shown). The arrow from B to A denotes that B is an entity in the group whose the root is A.

Now a new transaction with timestamp 4 performs the write to update 'Yb' to B. Firstly, a new entry is created in the journal as in figure 4.5.

Finally, the new journal entry is applied to B (but not persisted yet). The write action is complete as in figure 4.6.

If the write action is performed against the root A, the process is similar.

Figure 4.4: An entity group where a write is about to happen

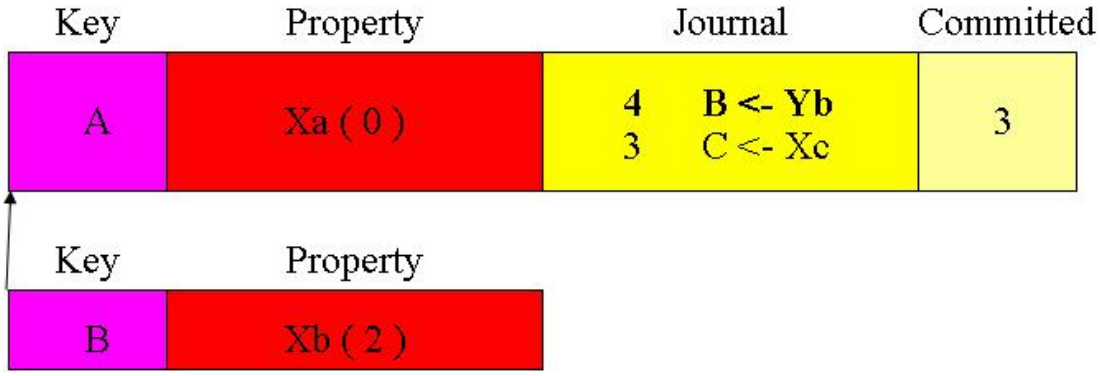


Figure 4.5: Transcaction with timestamp 4 writes the journal

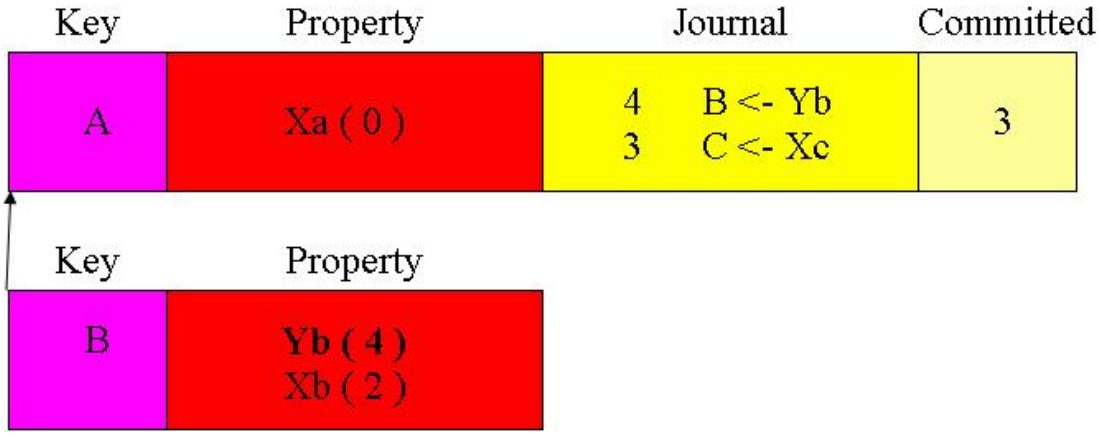


Figure 4.6: After the new journal entry has been applied to B

The commit action

When transaction 4 finishes all its actions and valid to commit (detailed on the next section on when a transaction is considered valid to commit), it makes all write entries it created persisted and sets the last committed timestamp field of A to its timestamp. The transaction effectively gets committed as in figure 4.7.

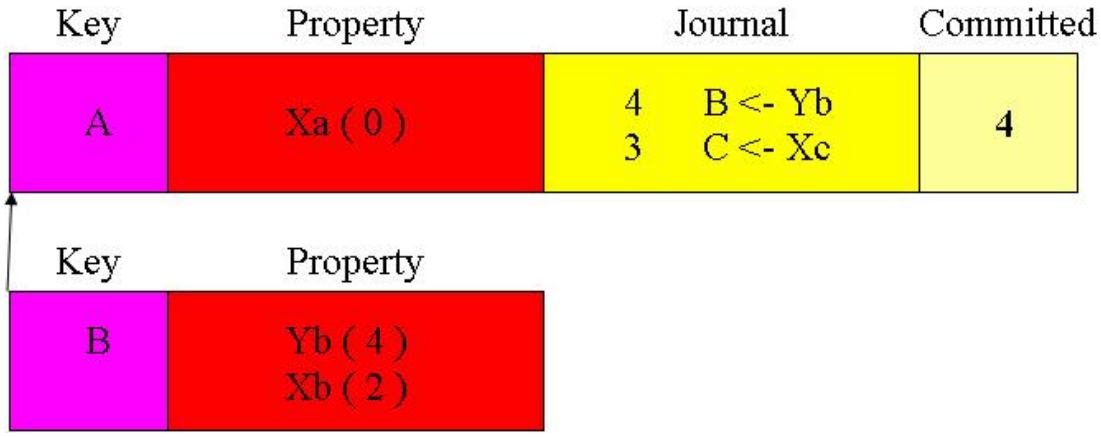


Figure 4.7: Transaction 4 commits

As can be seen, the transaction journal and last committed timestamp fields located at the root serve the entire group. All changes anywhere in the group are recorded in the journal, and the timestamp of the last transaction that committed is recorded in the last committed timestamp field, even if the transaction made no change to the root.

Concurrency control rules

When a transaction T starts, it is assigned a timestamp TS(T) generated based on the last committed timestamp CTS(A) of the root A of the group it targets. It also remembers CTS(A).

1. T wants to write entity B in the group (B may or may not be the same as A): it performs the write action as outlined in the previous section, including writing the journal in A and applying that journal to B.

2. T wants to read entity B: if T has modified B, it retrieves data from B as of TS(T) (this is non-persistent data). If T has not performed any write to B, it retrieves data from B as of the original CTS(A) that it remembers from the beginning, or as of the latest committed timestamp in B before CTS(A) if the entry that corresponds to CTS(A) doesn't exist in B.

3. T completes all its actions and now wants to commit. It compares the original CTS(A) that it remembers from the beginning with the current value at this point; if they are still the

same, T is considered valid to commit and the commit action takes place as outlined in the previous section; otherwise, T must be rolled back (aborted and restarted with a new, larger timestamp). T may be rolled back at most a fixed number of times before giving up.

Note that if T is a read-only transaction, the step 3 essentially gets omitted (and T always succeeds).

4.2.4 Implications on applications

Dirty Read and Deadlock

As can be seen in the rule for read, a transaction only reads committed data; it never reads work in progress of other transactions; in other words, dirty reads[51] do not occur in this system.

Deadlocks[51] also do not occur, because this is a timestamp-based system, in which no transaction ever has to wait for any other transaction before it can proceed with its actions, as in the case of lock-based systems[51].

Serializability principle and implications

Unlike the general timestamp-based scheduler[51] where the timestamp order of transactions dictates the serial schedule, the datastore rules result in serializability that does not necessarily comply the timestamp order. There are cases in which the datastore has a different way of serializing transactions. Technically, this difference is resulted from how the two approaches track actions performed on data elements: in the general approach, write time and read time are maintained for each element and they are used in comparison with the timestamp of the transaction about to perform an action against the element to decide whether the action is allowed to be carried out; in the datastore, only committed time is maintained for each element (each entity, or more precisely, each entity group), and it's not used to check individual actions, but instead the entire transaction at the point of commit if the transaction has performed some write.

The following example illustrates the difference between the two approaches. Suppose there are two transactions running concurrently; T1 is a write transaction while T2 is a read-only one; particularly, T1 writes element A and T2 reads that same element; T2 starts when T1 is already ongoing but not yet committed (in other words, T2 has higher timestamp than T1).

In the general timestamp-based approach, according to the timestamp order, any serializable schedule must be equivalent to T1T2; in other words, a schedule is considered serializable if and only if conflicting actions comply to this order. So the schedule in figure 4.8 in which T1 performs the write to A w1(A) before T2 performs the read from A r2(A) is serializable.

But the schedule in which w1(A) comes after r2(A) is not serializable because these conflicting actions violate the timestamp order. It is illustrated in figure 4.9.

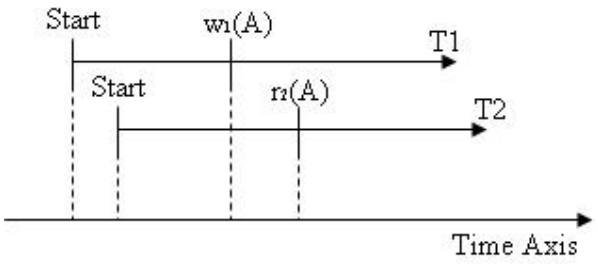


Figure 4.8: Serializable schedule

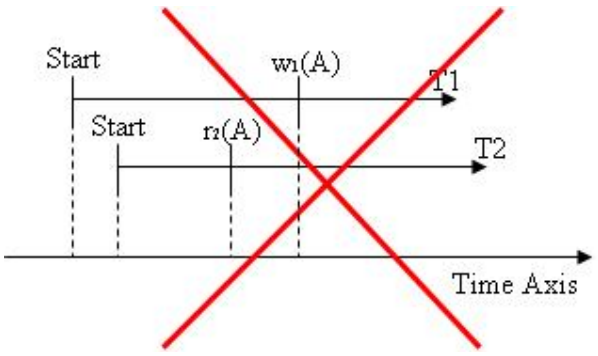


Figure 4.9: Non-serializable schedule

To enforce serializability, write time and read time for each element are maintained along with appropriate rules by the scheduler. When r2(A) is performed, read time of A RT(A) is set to the timestamp of the transaction carrying out the action TS(T2). Later when w1(A) is about to be performed, the timestamp of the transaction requesting that action TS(T1) is compared against RT(A) which at this point is TS(T2), and since TS(T1) < TS(T2), this write action is detected as physically unrealizable, and T1 must be rolled back.

In the datastore approach, both of these schedules are serializable. In fact, w1(A) and r2(A) are not even conflicting actions. T2 doesn't look at T1's work in progress. Instead, it reads the committed version of A as of when it starts, which is also the committed version of A as of when T1 starts because T1 has not reached the commit point when T2 starts. Thus, no matter what the ordering between w1(A) and r2(A) is, the net effect always appears as if T2 performs r2(A), then T1 follows with w1(A). In other words, both schedules are equivalent to the serial schedule T2T1, which is the reverse of the timestamp order.

The advantage of the datastore approach in this scenario is that transactions always succeed, while in the general timestamp-based approach, rollback may occur.

A real-life situation in which this advantage may be realized is the following: given the same setting with T1 and T2, these transactions now operate on two bank accounts A and B; T1 transfers 10USD from A to B while T2 queries these accounts. Suppose these transactions attempt the following schedule (the actions appear in chronological order):

- T1 adds 10USD to B
- T2 reads A and B
- T1 deducts 10USD from A

In expressing in terms of read and write actions only, this schedule could be re-written as: w1(B)r2(A)r2(B)w1(A).

This schedule is not possible under the general approach and T1 would have to roll-back because the actions on A violate the timestamp order. In fact, several different ways of interleaving of actions between these two transactions are not possible due to the same reason.

However, this schedule is possible under the datastore approach. In fact, any way of interleaving of actions between these two transactions is possible; T2 always reads the balances of A and B as of before the transfer operation begins.

For serializable schedules under the general approach, there might be the benefit of T2 reading the up-to-date information. For example, in the schedule w1(B)r2(B)w1(A)r2(A), T2 would see the effect of the transfer while the transfer is being performed. However, because T2 reads work in progress of T1, this approach might have to deal with the issue of dirty reads, which does not occur in the datastore approach because only committed data are visible to transactions.

Avoiding rollback is also generally the benefit of lock-based system[51]. So how does locking compare with the datastore approach? It turns out while the attempted schedule in our example is possible in the datastore, it's not serializable in the locking system. T1 has to hold

locks on A and B while doing the transfer, thus T2 can't interleave its reads in between; instead, T2 has to delay its actions until after the transfer is done; in other words, the schedule has to be transformed to w1(B)w1(A)r2(A)r2(B) (and T2 may have updated information provided by dirty reads). Another serializable schedule under the locking scheme is r2(A)r2(B)w1(B)w1(A) in which T2 holds the locks first that causes the delay for T1's actions.

Through these analyses, we see that in this particular situation in which there's a read-only transaction that executes concurrently with a write transaction that accesses common elements, the datastore combines the benefits of both worlds: it avoids rollback (locking scheme's benefit), and at the same time avoids delay (timestamping scheme's benefit). In fact, the datastore concurrency control rules dictate that delay never happens; rollback may happen when two concurrent transactions try to write the same entity group.

This observation may have more general implications in terms of transaction throughput. In an environment where all transactions are read-only, all three approaches perform equally ideally. When write transactions start to occur, the locking scheme starts to lag behind due to delays taking place. When there are more write transactions trying to access common elements with reads, the datastore approach starts to rise to the top because the general timestamp-based scheme starts to introduce more rollbacks. But when write becomes so prevalent that the likelihood of different transactions concurrently writing common elements is high, the datastore approach may loose its advantage; in this situation, locking may perform best because rollbacks are too frequent in the other two approaches.

The situation in which read-only transactions are the majority, and more importantly, it's not very common that concurrent transactions write the same elements, is probably typical for many Web applications. As a platform for Web applications, thus, the set of concurrency control rules chosen for the datastore is probably the right choice because it yields the highest transaction throughput in this situation.

This also has some implications for GAE application designers who want to build scalable apps that can serve many users. They should try to avoid writes to the datastore, when possible. For inevitable writes, they should design such that these writes target different entity groups. The idea is to try to avoid high write-contention entity groups that may cause a lot of rollbacks and thus, failed transactions, creating scalability bottlenecks for applications. When such high-conflict groups seem to emerge, a variety of techniques should be considered to handle the situation, such as sharding.

Another somewhat more subtle implication is that application designers should try to avoid long transactions, especially long write transactions. Generally, long transactions is not a good practice in Web applications; GAE does have a time limit on how long a session can be processed, so technically transactions can't exceed this limit. Besides this, the implication from the concurrency control rules is that the longer a write transaction, the more likely rollback will happen because it's more likely that another write to the same group will jump in the middle of

the process. Once rollback happens, the long transaction is restarted and again gets exposed to potential interferences by other writes, which could eventually leads to failure after the maximum number of retry has been used. Given the session time limit, a rolled back long transaction may even fail before reaching the number of retry allowed. Thus, transactions should only include operations truely necessary to do its jobs; others, particularly non-datastore calculations, should be considered to be put outside.

4.2.5 Experiments

We implement some experiments verifying the implications of the concurrency control rules of the datastore. This verification suggests good/bad practices with respect to the datastore to build scalable applications on the App Engine. Although principly, the datastore inherits scalability of the underlying BigTable, due to its rules governing the execution of concurrent transactions, scalability doesn't come for granted to your application. Some good practices should be employed, while some "bad" ones should be avoided if you want your application to become popular that can serve many concurrent users while maintaining reasonable data integrity and consistency.

Global Variable

As mentioned, the use of high-conflict entity groups may create scalability bottlenecks for applications. A special case of this is the use of high-conflict single-entity groups. A typical example is a global variable maintained as a single entity in the datastore, and updated by many user sessions. This creates a high write contention on the entity. Because all the simultaneous writes to this entity are serialized, many update attempts may exhaust their fixed number of retry, or their fixed amount of timeout for a HTTP request set by the App Engine, potentially resulting in failed user requests, leading the application to an inconsistent state.

One concrete example of a global variable is a counter. Suppose the application needs to keep track of the count of something, such as the number of items in the inventory, the number of orders placed, and so on. When a user session enters a new item, or places a new order, the counter is increased by one, and updated to the datastore.

We simulate this scenario using a simpler example: our application counts the number of HTTP requests it receives. We use httperf[52], a tool for measuring web server performance. Httperf is capable of spwaning simultaneous HTTP GET requests to a web address for a specified period of time. The application maintains a single global counter in the datastore, which is retrieved, increased by one, and then put back into the datastore by the Python script that handles each HTTP request.

We spawned a total of 2700 HTTP requests under the rate of 20 requests/second. After the experiment, the count was found to be 2228, instead of 2700. The log showed exactly the

discrepancy of 472 failed requests, all falling into the same category of failed transactions with the particular message *"TransactionFailedError: The transaction could not be committed. Please try again."* after five unsucessful retries that consumed an alarmingly high amount of CPU resource. Each unsucessful retry did indicate the failure reason as transaction collision. One notice is that even among successful requests, retrying of transaction for more than one time is pretty prevalent. Another notice is that for the most part, the CPU time consumed by requests is proportional to the number of transaction retries.

So if the global variable approach is that troublesome, why not use the simpler and familiar count() function as in the relational database world? Why not just issue a statement similar to "select count(*) from items" whenener the count needs to be examined? Such a statement is quite possible in the App Engine environment, given the count() method in the Model class provided by the datastore API.

We implemented this alternative for our experiment. For each HTTP request, a new root entity is entered into the datastore. The same experiment setting was applied, with 2700 HTTP requests under the rate of 20 requests/second. A separate Python script with the count() method afterward returned the correct number of requests that hit the app.

The issue with this approach lies in the complexity of the count() operation. Unlike in the relational database world, this operation in App Engine is not constant time, but O(N). Entities in the datastore are physically maintained in the BigTable, and this distributed data structure has no knowledge of the count of its elements. To perform the count of N entities, it has to physically scan through all those entities. Thus, if complexity of the operation is of the developer's concern, which is usually the case, especially for popular apps that require high scalability, this approach is not optimal.

Sharded Variables

The use of global variables, as seen in the previous section, is a bad practice in a distributed environment in general. Back to our example of the counter, the use of the count() method is also not optimal. Is there a good solution for this problem?

There is. The fundamental flaw of the global variable approach is that it creates one single point of bottleneck that suffers from high write contention. If this one single point is sharded (partitioned) into multiple pieces that can be independently processed in parallel, then combined when the total is needed, the contention is cleared, or at least relieved.

In our example of the counter, we partitioned the global counter into n sharded counters, which are n separate root entities. Whenever a request comes in, one of the n sharded counters is randomly picked, retrieved, increased by 1, then put back to the datastore. The Python script also sums up the n counters, and reports the total back to the client.

Using the same experiment setting, we varied the number of sharded counters n. With n = 2, the total count was not much different than the case of the single global counter, and often

off by about 500. With n = 4, the total count was often off by 10 - 20. With n = 8, the error was about only 1 or 2. N = 16 always reported the correct count of 2700. In all the experiments, whenever error occurred, there was always corresponding entry in the log with the same message as in the case of the global counter.

We make the conclusion that sharded counters is a good practice that should be utilized. Using the right configuration for the shards, the count can be retrieved accurately in constant time. The job of the administrator is to figure out the right configuration; too few shards may still lead the app to the inconsistent state, whereas too many shards will make the retrieval operation take longer than neccessary.

Large Entity Groups

Besides global variable, large entity groups (groups with many entities) may also likely create problems. Recall that all concurrent writes to a group, even to different member entities, are serialized at the point of commit (among two simultaneous writes, the one that is about to commit later is rolled back, in effect being serialized). Thus, large entity groups, which likely attract a large number of writes, likely become write-bottlenecks, and is a practice that should be avoided if the app is to scale to a large number of users.

To prove this claim, we implemented an app that had n entity groups. Each HTTP request to the app would create one entity in a randomly selected group among those of the app; the new entity would become a direct child of the root of the group. We then spawned 1000 requests under the rate of 20 requests/second, and examined the number of entities successfully created. We performed several experiments varying the number of entity groups n. With n = 1, there were only 176 entities created. With n = 2, 639 entities inserted. We ran the experiment with n = 4 a couple of times and the number of entities was usually off by almost 10 (the number reported ranged from 988 to 994). N = 8 always resulted in all 1000 entities successfully created. For all the experiments, whenever there was a missing entity, there existed a corresponding entry in the error log with the message *"TransactionFailedError: too much contention on these datastore entities. please try again."*

The conclusion is that entity groups facilitate the execution of transactions; but the design should dictate small, localized entity groups. Large groups, particularly those that attract a lot of writes, likely create problems in terms of scalability and could potentially result in inconsistent state.

4.3 User Experience

As with other cloud computing environments, we want to get some insights as how it is like to build, maintain, and monitor applications on GAE. We do this via the development,

deployment and execution of a series of small programs used for experiments in the previous section.

4.3.1 Developer Experience

Development of App Engine applications is done locally. GAE provides the App Engine SDK[53] for this activity. Developer needs to download then install Python 2.5 and the SDK and link them together to setup the development environment.

We use Eclipse as the IDE for writing the app. There's an article[54] in the GAE community guiding the configuration of Eclipse to use with GAE. As part of the configuration, the PyDev extension[55] needs to be setup and associated with the Python runtime installed earlier to customize Eclipse for Python developers.

The App Engine SDK includes a web server application, called the development web server, that emulates all of the App Engine services on the local computer. The SDK includes all of the APIs and libraries available on App Engine. The development web server also simulates the secure sandbox environment, including checks for imports of disabled modules and attempts to access disallowed system resources. Once running, the development web server listens for requests on the selected port on the local machine, facilitating the testing of applications being developed. Eclipse can also be configured to integrate with the development web server so that this server can be started and stopped from within the IDE.

App Engine applications are easy to build. They communicate with the web server using the simple CGI standard. When the server receives a request for the application, it runs the application with the request data in environment variables and on the standard input stream (for POST data). To respond, the application writes the response to the standard output stream, including HTTP headers and content. To write an application, fundamentally the developer needs to code a set of handler scripts that will be invoked to process requests and return responses. The mapping between handler scripts and URLs so that appropriate scripts are triggered upon the arrival of requests is specified by the developer in a configuration file.

To further facilitate development, App Engine supports any web application framework written in pure Python that speaks CGI (and any WSGI[56]-compliant framework using a CGI adaptor), including Django[46], CherryPy[57], Pylons[58], and web.py[59]. These frameworks relieve developers from the burden of manually writing all the details code required by the CGI standard. The developer can bundle a framework of his choosing with the application code by copying its code into the application directory. App Engine includes a simple web application framework of its own, called webapp[45]. This framework is already installed in the App Engine environment and in the SDK, thus automatically available for use. We use this framework for our application.

Debugging and unit testing take advantage of the facilities provided by Eclipse, and the

development web server. During unit testing locally with the browser directing requests to the development web server, errors, uncaught exceptions, stack traces and so on show information within the Eclipse console, thus the developer can take advantage of the IDE's facilities to spot out the problems.

There's a community of GAE developers[60] in which people share experience, ask and answer questions, contribute sample code, applications and articles. One may find this community helpful when he faces issues. In fact, we find the community at this moment somewhat not yet very active, but it's growing. A pretty impressive point about the community's forum is that Google engineers, the people who themselves are building GAE, are very responsive and helpful here. Our questions posted on this forum, if no one else has the answers, are always responsed responsibly in a timely manner by Googlers.

4.3.2 Admin Experience

App Engine provides the Administration Console at http://appengine.google.com/ for administrative tasks. One needs a google account [1] to access this. The console enables the creation of new applications [2], and the set up of a free appspot.com sub-domain, or a top-level domain name of one's choosing.

Once the set up step is done, the application can be uploaded to App Engine. This task is facilitated by a script that comes with the SDK. The developer only needs to run the script from the local machine and the application will be uploaded to the right place in the cloud.

App Engine enables the co-existence of different versions of one application. This is useful for testing. One could expose a new version for a restricted group of users, while the old version still serves the rest. The enabling/disabling/switching between versions is done in the admin console. Note that all different versions share the same datastore; in other words, all versions of an application, including the test version, use the same data in the production system.

The admin console also allows for inviting other people to be developers for the application, so they can access the console and upload new versions of the code. Note that this does not offer a source control system; all people in a project have full control of the code.

On the console, the admin can browse, manage, modify data and indexes in the datastore.

The console features a dashboard with charts that help analyze traffic and relevant information, such as requests/second, miliseconds/request, errors/second, bytes sent/second, and so on. The dashboard also reports summary information about the current load (accumulated recently, such as in the last 18 hours) for each URI such as the number of requests, average CPU, % CPU, as well as the count and percentage of errors. Besides that, summary and details about

[1] an account with Google's unified sign-in system; to sign up for one, all a user needs is a valid email address which doesn't need to be a Gmail address

[2] at the moment, each account can have at most 10 applications

all types of resources that have been consumed with respect to application quotas are provided, alerting the admin about potential problems that may arise due to the excessive use of some resource.

The logs are really helpful in our experiments. Each error log entry details the traceback, the error message, and the real time as well as the CPU time the request takes, with an adequate level of alert if the CPU time is considered unusually high, besides other common information. If several application versions are in use, the logs for all are recorded accordingly. These logs can be downloaded for further analysis at the local machine.

Besides the user logs, there are also admin logs. Admin actions, such as the upload of a new version, the modification of data in the datastore, and so on, are recorded here.

4.4 Conclusion

Google App Engine presents a small and compact offering that targets a specific group of applications, namely web applications. The service aims at simplicity for developers. Development, deployment and administration of applications are straightforward, enabling teams to quickly introduce new apps/new versions to customers. However, simplicity comes with some limitation. Given the very specific and narrowly scoped of the offering, applications that require more flexibility at the server side, such as interaction with the file system, backend processing of data, do not seem a good fit. It'd be interesting to see how Google addresses this with the evolution of the platform, for example with more languages supported as stated in their roadmap.

Nevertheless, the current offering is suitable for a variety of applications, such as certain kinds of consumer apps. With a limited number of specific goals, the vendor can afford to achieve the goals with thoughtful design. Particularly, the platform hides inside carefully architected features, such as the datastore, that aim at scalability and high throughput. Thus, applications suitable for Google App Engine have the potential to scale to a very large number of users while still maintaining reasonable integrity and consistency. However, the potential does not free application designers from employing certain good practices if they want their apps to fully reach that potential.

Chapter 5

Microsoft Windows Azure

5.1 Introduction to Windows Azure

Microsoft recently released the Azure Services Platform[9] for limited preview, called the CTP period. It's a cloud computing environment that consists of the foundational Windows Azure and add-on services. Windows Azure can be thought of as a cloud services operating system that abstracts Microsoft data centers into a cloud computing platform that provides customers with on-demand compute and storage to host, scale, and manage internet or cloud applications and services. Several add-on services are also included with Azure. In this chapter, we will take a quick look at this new offering from Microsoft. There will be no experiments, but instead an overall technical view at the offering based on information from the vendor is presented.

The terms applications, (user) services, and Azure services all by default refer to software deployed to the environment by customers, and will be used interchangably. Moreover, features discussed in this chapter are what are available in the CTP; future offerings in the commercial release will be noted explicitly.

5.2 Fabric

The Azure fabric is a general term for the logical cluster of machines that Azure services run on, along with the processes that manage the cluster, provision and deploy services, monitor system status, and recover from failures.

The fabric consists of nodes, which may be physical machines, virtual machines, or (eventually) partial virtual machines.

5.2.1 Services

An Azure service is defined by a service definition, called a model. It's an XML definition of the different components of the app, the resources needed, and how they interact with each other and the outside world.

Currently only a few model templates are provided. In the commercial release, the full XML definition language will be available and arbitrary service definitions will be supported.

Components

A role is an individual executable component of an app. Typically it comprises a binary or piece of code that runs as an OS process, such as a web frontend, backend, worker, garbage collector, daemon, and so on. A role may be instantiated many times. Roles run on fabric nodes. Currently each role runs in its own Hyper-V[61]-based virtual machine, and may only run managed code. In the commercial release, roles may be written in native code and even execute on bare hardware. In the other direction, they may also run in a virtual machine shared by more than one role.

A service definition enables defining communication for components. Connection between two roles is expressed by a channel. Channels may also be used to connect load balancers together or with roles. An endpoint is a generic network endpoint, either internal or external, for serving requests. An interface is an externally accessible endpoint. Supported protocols include HTTP(S), SMTP, ATOM, and SOAP.

Provisioning

A role definition dictates resources needed, including CPU, memory, disk, network bandwith, and even specific OS features. When the role is deployed, it's limited to these resources. The role definition also includes the number of instances that the fabric should deploy, either a specific number or a range. Finally, constraints may be specified, including how many role instances may run in the same node, whether instances of different roles should be co-located, and how to allocate instances across update domains and fault domains.

Update domains and fault domains are optional but useful features. Update domains are used to partition the service during upgrades. A role may specify the number of update domains its instances should be deployed in.

During rolling OS upgrades and service updates, only one update domain will be upgraded at a time. While an update domain is being upgraded, the fabric won't route live traffic to roles or load balancers inside it. After the upgrade finishes successfully, the fabric returns all elements of that domain back to service and starts on the next update domain.

Fault domains are somewhat similar. They're essentially disjoint failure zones within a single data center and specified according to the data center topology, based on factors like

rack and switch configurations, which make certain classes of failures likely to affect well-defined groups of nodes. As with update domains, a role may specify the number of fault domains its instances should be deployed in.

Hosting

Virtual machines with a limited choice of Windows OS is the container to run role code. It has a standard Microsoft stack available: IIS7, .NET CLR, ASP.NET, .NET services, Live services, SSL. Several .NET CLR managed code languages are available, and more are coming, along with native code. Role code is subject to a custom Azure CAS (code access security) policy for managed code.

Role instances interact with the fabric via APIs. This may include examining the service definition and the current node, accessing configuration settings, and reporting health and other status info. Role instances may also use the local disk on their nodes as temporary storage space, within a quota limited, chrooted directory. Moreover, there's a distributed logging system that collects logs and makes them available for user access.

Configuration and APIs

Services include configuration settings, which are similar to the Windows registry. Some are system-defined, such as the instance id, update domain id, and fault domain id, while some are declared by the developer/deployer in the service definition, along with default values. These settings may be changed at runtime. Roles can define callbacks to be called when a change event happens; this allows developer/deployer to change them dynamically at runtime without a restart. Of course, most of the system-defined settings will only change on a restart.

Geodistribution and Georeplication

Currently Azure runs in a single data center on the US west coast. In the commercial release, it is expected to have multiple centers functioning and customers may have some control over geodistribution and georeplication, such as where and how their services and data live.

5.2.2 Implementation

The Azure fabric implementation consists of two main components, the fabric controller and the fabric controller agent.

Fabric Controller (FC)

The FC manages the life cycle of Azure services. It allocates resources, provisions, deploys, monitors them, and maintains their goal state. It also manages and monitors both virtual and physical machines in the fabric.

State machine

The FC maintains an internal state machine that it uses for managing the life cycle of a service. It maintains the current state of each virtual and physical node, which it updates by talking with FC agents residing on each machine. The FC also knows the goal state of each published service.

The state machine maintains internal data structures for logical services, logical roles, logical role instances, and logical and physical nodes.

The FC's main loop is described as a heartbeat. During each heartbeat, it compares services' goal states with the current state of the nodes hosting them, if any. When a role whose the goal state doesn't match its node's current state is found, the FC tries to move the node's state toward the role's goal state. If impossible, it might move the role instance to another node.

The FC also monitors for certain events that move nodes out of goal state, for example failure.

Service life cycle

The life cycle of a service is divided into four stages by the FC: resource allocation, provisioning, deployment and upgrading, and maintennance.

Resource allocation is largely a graph mapping problem. The service definition is modeled as a graph in which roles are nodes and network channels between roles and load balancers are edges. This graph is mapped with the graph maintained in the FC representing the logical cluster managed by the FC.

Resource allocation is transactional at the service level; either all of the necessary resources are allocated or none.

Other provisioning and standard data center operations are handled either automatically by the FC, or offline, such as burn-in, diagnostics, replacing parts, capacity planning, and so on.

Deployments and upgrades are done mostly automatically by the FC, one update domain at a time. Within each update domain, new OS or service images are copied to all nodes. Then, all nodes are marked as out of service (which moves them out of the goal state), so live traffic no longer reaches them. Then, the upgrade takes place. Finally, the nodes are marked in service and moved back into the goal states after the upgrade successfully finishes.

Upgrades may be done either full automatically or manually. In the earlier case, each update domain starts immediately after the previous one finishes. In the latter case, humans start each new update domain. Service deployments are usually automatic; OS upgrades are either automatic or manual.

Maintennance consists of the standard monitoring for failures and unhealthiness, as reported by both the FC agents and roles themselves. When certain events occur, the FC moves a service/role out of its goal state, and may mark parts out of service. It then falls back to the deployment process of moving the service back toward the goal state via its heartbeats.

Networking

Virtual and dedicated IPs for services are allocated and programmed in load balancers automatically. The FC moves dedicated IPs as in and out of service based on running role instances with interface when they move in and out of goal state. Load balancers only send traffic to role instances in goal state.

Implementation details

The FC itself is a cluster of 5-7 machines. At any given time, one is the primary, and the others are secondary workers.

The FC internally consists of four main components:

- The *core* is responsible for running the heartbeat, the state machine, the resource allocation constraint solver, and so on.
- The *object model* includes the business logic for roles, services, and so on.
- The *state system* stores, retrieves, and validates state checkpoints. It can roll back and forward in the case of failures.
- The *data replication system* is a distributed file system dedicated to the FC, distributed across all of the FC machines.

Notably, the FC cluster actually runs on a smaller, limited, dedicated version of Azure itself.

Virtualization

Virtual machines on nodes use a hypervisor architecture based on Hyper-V[61], which takes advantage of new hardware features designed to assist virtualization. The hypervisor runs directly on the hardware, and multiple virtual machines run on top of it. Among which one virtual machines is the host; it includes device drivers which talk directly to the hardware. The hypervisor routes system calls and other direct hardware accesses from the other (guest) virtual machines to the host, which performs them on behalf of the guests.

The OSes for both the host and the guests also include a feature called VMBus, which is an optimized, high-bandwidth direct data bus between virtual machines.

Image-based deployment

Deployments and upgrades are based on virtual hard disk images, which provide several benefits. They're homogenous, stabler and more predictable, faster to copy and install, cacheable, easier to repare via replacement, and may be constructed and serviced offline, which is automated. OSes and services are built on separate images.

5.3 Monitoring and Alerting

There's little to no support for developer-configurable monitoring. The commercial release will support limited monitoring, as well as health and status reporting from roles. At the minimum, developers will be able to set alerts to fire when role instances become unhealthy, crash, or otherwise fail, when their nodes fail, or when a role logs a message at the CRITICAL level. Alerts may be delivered by email, instant messaging, and/or SMS.

5.4 Storage Services

Besides the foundational Azure, the CTP provides three storage services that run on top of the Azure fabric. They are tables, queues, and blobs. In the commercial release, they'll be joined by lock and cache services. All of these services are intended to be simple and massively scalable.

Each storage service has a programmatic .NET API and an HTTP REST API. Additionally, there are command-line utilities for browsing and modifying data in each service except for tables currently.

5.4.1 Tables

Features

Tables is Azure's scalable, schemaless, structured datastore, similar to the Google App Engine datastore. It's a distributed storage system very similar in design to Bigtable[50]. It's designed to scale to billions of entities and terabytes of data.

Entity Model

Each entity has a table name and a schemaless, key/value property bag. Entities are limited to 1MB and 255 properties each. Several primitive types are supported as property types.

Three properties are special: the partition key, the row key, and the version.

The partition key identifies a partition, which is a group of entities that should be physically stored together. The row key uniquely identifies an entity within a partition. Together, an entity's partition key and row key comprise its primary key. The partition and row key are both strings, up to 64KB, and defined by the developer.

Queries

Beyond the standard CRUD operations, tables supports a limited form of querying. Currently queries supports only equals filters, on one or more properties; results are returned in

partition + row key order. In the commercial release, developer-defined secondary indexes for sorting and inequality filters will be supported.

The programmatic query API uses LINQ[62], a simplified query language; the HTTP REST API uses the ADO.NET Data Services URL-based interface.

Along with the result entities, each query returns a continuation marker, which may be passed back in following query calls to retrieve the next page of results.

Consistency and Transactions

Tables is highly consistent. Single entity writes are synchronous, and there are no dirty reads. ACID transactions are supported on single entities.

Optimistic concurrency governs consistency and transactions, similar to the Google App Engine datastore, using a system-defined version property on each entity that is updated on each write.

In the commercial release, transactional batch writes will be supported, for example all writes in a batch will be committed or none.

One interesting feature is that table-scan-based queries within partitions provide snapshot isolation, since they can use a single base timestamp.

Schemas

Azure tables is schemaless, like the Google App Engine datastore. There's a built-in Tables table that stores the list of tables in an account, but not the properties or their types. It's queryable the same way as normal tables.

5.4.2 Queues

Features

Queues is similar to Amazon SQS[16]. It allows messages to be enqueued and processed later, asynchronously, in a loosely coupled fashion. A service specifies a set of queues in its definition. At runtime, it may enqueue, dequeue, and delete messages from those queues. Messages have a typed payload, up to 8KB.

Supported operations for queue are create/delete queue, enqueue/dequeue message, delete message, clear queue, and get queue length.

The dequeue operation acquires a configurable lease on a message. While that lease is in effect, the message is invisible to other queue clients. When the owner of the lease is finished with the message, it may permanently delete it. If it doesn't delete it within the lease period, the lease expires and the message becomes available to other clients again.

Implementation

The order of dequeueing messages is roughly FIFO, but not exactly. The system makes no guarantees about starvation or fairness within a queue.

5.4.3 Blobs

Opaque blobs are stored in the blobs service, which is similar to Amazon S3[14]. Blobs may be created and retrieved programmatically. They are identified by unique, human-readable URL paths under ¡account¿.blob.core.windows.net, chosen by the developer. Blobs may be up to 50GB, and may have additional metadata consisting of key/value pairs.

Blobs up to 64MB may be uploaded directly. Larger sized blobs must be divided into blocks which are uploaded separately. Blocks may be uploaded out of order, and individual blocks may be repeated. At the end, a Commit call is made with the block ids in order.

The blob namespace is the hierarchical URL path namespace under <account>.blob.core-.windows.net. Blobs may be enumerated at any level in the URL space, non-recursively. Paging is supported with continuation markers, similar to tables.

Blobs are not immutable. They may be modified, appended, and cloned. For block-based blobs, modifications are made at the block level.

5.5 SQL Services

Besides storage services, Azure comes with a schemaless structured database that lives in the cloud, called SQL Services. It's based on sharded SQL Server with entity model similar to Tables.

SQL Services is designed to complement Tables, providing RDBMS features such as support for the demanding traditional OLAP/OLTP usage patterns, data mining, reporting, as well as many of the existing SQL Server tools and features. However, its scalability and ability to serve a high load of end user traffic is in question. Also, it's expected to be much more expensive.

5.6 Developer Experience

Microsoft strongly insists that existing tools, libraries, skills, and knowledge transfer to the cloud. Specifically, Visual Studio, .NET, LINQ[62], and so on are used to write Azure services, just like they are used to write on-premises services.

The service development lifecycle consists of four steps:

- Write the code and service model, done by developer(s).

- Determine the necessary resources and configure the service, done by developer(s) and/or deployer(s).
- Allocate resources, provision, and deploy the service, done by the fabric.
- Maintain the service in the goal state, done by the fabric.

The Azure SDK comes in a file runme.exe that includes a full, stand-alone, stubbed-out implementation of the Azure fabric, APIs, and storage services. It's multi-process, so developer can run multiple roles, and multiple instances of each role, on the local machine.

Visual Studio is not required, but naturally the most supported environment. When using the SDK, all features are available, most notably the debugger. Debuggers may not be attached to services running in the cloud on Azure, of course.

To publish an app to the cloud, first build each role binary with Visual Studio, or any compatible compiler. Then, use CSPack.exe to package a service image that contains the role binaries. Finally, use CSRun.exe to upload the image and publish the service to a staging instances, where the service is available for testing before it serves live traffic. Finally, to start serving live traffic flip the switch to swap the staging and production instances, via either CSRun.exe or the developer portal.

The SDK, CSPack, and CSRun are all command line, with Visual Studio plugins built on top of them. This facilitates third party tools and plugins.

Asymmetric keys are provided for authenticating to the developer portal from local machines, authenticating to the storage systems from live role instances, and signing and encrypting service images before publishing them.

Services get a subdomain of cloudapp.net to serve off of. Currently dedicated top-level domains are not available, but expected to be there in the commercial release.

Finally, the core Azure API provides a logging facility that logs to a reliable, distributed store. The logs are then available for developer access.

5.7 Conclusion

Many cloud platform vendors have their services evolve from existing non-cloud components. Amazon first had their data centers for internal use and when they detected the opportunity, they exposed their infrastructure as web serices to outsiders. Similarly, Google had data centers and components such as the BigTable serving their own business; they then essentially exposed those as a cloud offering to the world.

Microsoft, who came out later in this business, takes a different approach. They build everything from the ground up, from the data centers, the system governing those centers, to add-on services interfacing users. All are built with a cloud-oriented, external users-serving mindset right from day one. The result is a very impressive and solid cloud foundation, at least on paper.

At a high level, Windows Azure does impose its own programming/deployment model, with application components and configuration represented as roles, channels, load balancers, endpoints. Probably time will tell how general this model is and what types of applications it can (and can not) cover.

From a developer perspective, current offerings from Microsoft are roughly similar to what are available in the market. Similarity can be realized between Tables and App Engine datastore or Amazon SimpleDB, Queues and Amazon SQS, Blobs and Amazon S3, Azure SDK and App Engine SDK. However, given the strong foundation and the big commitment and investment, we can expect to see Microsoft to emerge as a key cloud platform vendor with comprehensive services.

Chapter 6

Conclusion

Summary of the survey

With cloud computing, cloud platforms are emerging as a new way for developing, deploying and delivering applications. This is a major shift in the industry that presents a very fast moving landscape with more and more vendors joining and adding innovations to the stream. We conduct a survey that examines several major platforms ranging from general to specific with respect to the types of applications they support.

AWS[6] and Windows Azure[9] are the two general environments with similarities and differences. Their offerings to developers, at a high level, are largely similar. However, looking beyond the surface, the differences are notable that may mean something for a developer in terms of which direction to take. AWS services are somewhat separate and independent, and introduced one by one gradually over time. Amazon seems to follow the Unix philosophy: small, simple tools that do one job, do it well, and fit together in ways that complement each other. We were impressed by their services via our experiments.

Microsoft, on the other hand, throws out all the dishes on the table at once. We believe they managed to do this because they focus on the foundation first, the cloud operating system. Once the solid foundation is in place, their offerings can be relatively quickly built on top, fundamentally the same way third-party developers implement their apps on Azure: write code for roles, declare configuration in the service definition, and have the system deploy the apps. Thus we can expect to see Microsoft cloud services to grow and get comprehensive pretty fast.

Another notable difference is that AWS services are quite basic and open and don't impose any programming model. EC2[13] essentially rents a machine to a user; he can use that machine in whatever way he wishes, using any technology of choice. In Azure, there's no notion of renting a machine just for the sake of renting a machine; principly a service must be defined with some purpose, and machines and other resources will be allocated accordingly; this way, Microsoft does impose its own model for service development/deployment, although this model

is claimed to welcome non-Microsoft technologies in the commercial release.

AWS recently made several advancements in their offerings. Besides various Linux variants, EC2 now supports OpenSolaris and Windows Server 2003. Several features have been introduced aiming at building failure resilient applications, including the ability to place instances in multiple locations engineered to be insulated from failures in each other, a feature called Elastic Block Store (EBS) that provides off-instance storage that persists independently from the life of an instance and can be attached to a running instance, and a feature that allows an EC2 account to obtain a set of static IP addresses that can be automatically remapped to replacement instances once those holding those IP addresses die.

Besides Amazon SimpleDB and SQS getting more mature, a new infrastructure service, Amazon CloudFront[63], has recently been added to the offering. It integrates with S3 intending to help customers distribute content to end users with low latency and high data transfer speeds.

Sun Grid[7] is one of the first cloud platforms available in the market. It's a promissing offering for high performance computing-flavoured applications. In fact, under its catalog listed various applications of this type from Sun or third-party, free or charged for customers, in several domains like Computer Aided Engineering and Life Sciences. However, our experiment on this platform was somewhat not very satisfactory, in terms of both technically and customer service received. The platform as of this writing is in transition and Sun announced it'd be reopened late March, 2009.

Google App Engine[8] offers a platform for CGI-styled web applications. To achieve scalability, they introduce a component that is unfamiliar to most web developers: the BigTable[50]-based datastore[41]. It's meant to replace the familiar RDBMS sitting at the data layer of a web application. Efforts have been made to give it some familiar flavours, such as the SQL-like language for querying, or reference property to model the join operation, but the differences are still notable. Nevertheless, the datastore is well designed and suited for the App Engine as a platform for certain kinds of scalable web applications. The job of the app designers is to employ certain practices for their apps to be truely scalable.

Cloud Architectures

The coming of the cloud platform revolution brings opportunities for applications to reach a whole new dimension. With platform vendor's ability to add a huge number of machines and amount of storage, applications practically have the potential to reach infinite scalability. In fact, infinite scalability may even be put as one of standards for a new type of applications, "true cloud applications", for the lack of a better term.

The auction system used in our survey is not yet at the extent of "true cloud applications". Backed by a MySQL database, apparently RUBiS[12] can't scale infinitely. Another shortcoming is that in response to the increase of the number of users, application admin, which is us, have to step in and constantly reconfigure the system (and still failed, due to the lack of

expertise). We believe in the context of "true cloud applications", this should mostly not be of the application admin's concern; this burden should be shifted to the hands of the platform vendor's system admin, and this guy's action actually should typically have the scalability impact on many applications hosted in his network, not just our single one. Back to RUBiS, the issue is the only thing this system has to do with the cloud is that it was deployed in the cloud; it could not exploit the potential because it was designed in the "old fashioned".

The answer to the RUBiS problem from Google and Microsoft is the App Engine datastore and Azure Tables, respectively. [64] and [65] go one step further: given the context of the new world of cloud computing, redefine the database optimization problem with new constraints among which is infinite scalability, and different metrics to optimize among which is consistency. This work suggests an initial step towards a long-term vision to implement full-fledged database systems on top of cloud services. In this initial step, it abandons strict consistency and DB-style transactions for the sake of scalability and availability. It proposes a new data management system using AWS cloud infrastructure services including S3, EC2, SQS, SimpleDB, and an accompanying new architecture for a Web-based interactive application that works on top of this data management system. This web application is used to study the trade-offs of the proposal varying different parameters, among which is the level of consistency guaranteed by the system, with fixed assumptions, among which is infinite scalability.

Amazon recently suggests tips for cloud architectures[66] for general "true cloud applications" with examples. Some highlights include use scalable ingredients, have loosely coupled systems, think parallel, on-demand requisition and relinquishment, and use designs that are resilient to reboot and relaunch.

Conclusion

A new kind of application platform doesnt come along very often. But when a successful platform innovation does appear, it has an enormous impact. Think of the way PCs and servers shook up the world of mainframes and minicomputers, for example, or how the rise of platforms for N-tier applications changed the way people write software. While the old world doesnt go away, a new approach can quickly become the center of attention for new applications.

Cloud platforms arent yet at the center of most peoples attention. The odds are good, though, that this wont be true five years from now. The attractions of cloud-based computing, including scalability and lower costs, are very real. If you work in application development, whether for a software vendor or an end user, expect the cloud to play an increasing role in your future. The next generation of application platforms is here.

Appendix A

The string matching (local alignment) problem

In biology, various cell-related components like DNA or protein are modeled as strings composed of characters from subsets of the English alphabet. These representations are often used for computational purposes, for example, if two DNA strings from two species, by a computation, are determined "similar", this might give some hint that these species might be relatives.

Determining "similarity", from the algorithmic view point, is a string matching problem, the problem of finding a longest common subsequence of the two strings. To clarify, let's be given an example with two small DNA sequences: ACGTGA and ATGGCC.

The result of the match is a pair of strings with the following form:

XX....XX ((modified) substring 1)

XX....XX ((modified) substring 2)

This pair is the result because it gives the highest score among all possible pairs of (modified) substrings from the two original strings, based on a set of scoring criteria. The set consists of rules on how to score the match between two characters, and includes the following rules:

- match: score given if two characters are the same. For example, match = 1 means if 'A' is paired with 'A', it'll get 1 point.
- mismatch: score given if two characters are different. For example, mismatch = 0 means if 'A' is paired with 'C', it'll get 0 point.
- indel: score given if a character is paired with a special character '−', called the "indel" (essentially, this special character represents an unknown/unconcerned character inserted). For example, indel = 0 means if 'A' is paired with '−', it'll get 0 point.

The score of a result candidate of a match between two strings is the summation of scores of all pairs of characters at the same position from the two candidate substrings, based on the scoring rules.

Back to our example with two DNA sequences ACGTGA and ATGGCC. If given the set of scoring criteria (match=1, mismatch=0, indel=0), the match result is the following:

ACGTGA−−

A−−TGGCC

The score of this result is 3 because there are three pairs of charaters that are the same at positions 1, 4, and 5 (the point given to all the other pairs is 0). As can be seen, to achieve this match, both strings are modified with the insertion of two "indel" characters '−'. There's no other match that can achieve better score, although there may be other matches that achieve the same score of 3. Due to the two "indels", this result is 8 characters long, 2 longer than the length of both original strings.

The insertion of "indel" has some meaning biologically. A sequence modified by "indel" usually puts the original sequence in an evolutionary perspective, in which "indels" may represent some unknown mutations at given positions. Choosing a set of scoring criteria that likely allows "indel" to be inserted, as in this example, is usually motivated by the desire to realize the relevance between the two sequences, and thus the two species, in the context of the evolutionary process. The higher the score of the match result, the more "similar" the two original sequences are.

If given the set of scoring criteria (match=1, mismatch=-30, indel=-20), the match result is the following:

TG

TG

This result's score is 2 and it's 2 characters long. As can be seen, the match consists of only exact substrings from the two original strings. The reason is the chosen set of scoring criteria pays heavy penalty for "indel" (and mismatch). This is usually motivated by stronger emphasis on the current state of the two given original sequences than concerning about how they might have evolved.

Back to the algorithmic perspective, the solution for this string matching problem is based on dynamic programming, and contains two steps: first, compute the score of the match result, then second, construct the match result itself. For our program, we implement step 1 using the Smith-Waterman algorithm[19] and step 2 using the Hirschberg algorithm[20]. Both of these algorithms are quadratic time O(nm) and linear space O(m), in which n and m are the lengths (number of characters) of the two original strings.

We performed a test in which a mouse sequence is matched against a human sequence. Both sequences are 10000 characters long. The test was performed on an Intel CPU 1.73Ghz machine. For a set of scoring criteria (match=1, mismatch=-30, indel=-20), the operation took 37 seconds, the match result was exact match with score of 32 and 32 characters long. For the criteria (match=1, mismatch=-5, indel=-5), the operation took 258 seconds, the match result contains understandably many mismatches and "indels" (because of lighter penalty for mismatch

and "indel") with score of 229 and 1471 characters long. Memory consumption for all the tests did not exceed 20Mb at peak time.

The BLAST-simulating program

BLAST[11] is a tool widely used in the biology community. Given a query string (a biological sequence), a sequence database (a set of biological sequences separated by a special delimiter character), the user chooses a set of scoring criteria and a threshold, then executes the query against the database. BLAST will match the query with the sequences in the database, and output the best results, the results whose scores are greater than or equal to the threshold.

BLAST doesn't match the query against all sequences in the database. It uses some probability information obtained from the metadata of the query and the database to determine regions in the database that likely contain the best results, then the match operations are only carried out against those regions. This way the query is executed much more efficiently. Note that each match operation carried out in BLAST usually implements the combination of the Smith-Waterman and the Hirschberg algorithms.

Our batch program used in the survey simulates BLAST. However, for our own purpose, instead of pruning the search range early on as done in BLAST, we use a brute-force method: match the query against all database sequences, and report the best results. To do this, based on the special delimiter character, we extract all individual sequences from the database, then match them one-by-one with the query. These match operations are carried out independently in parallel using the combination of the Smith-Waterman and the Hirschberg algorithms.

Preparing the dataset

We downloaded the human sequence database that contains known human sequences from an online biological data repository[67]. It's a file conforming to the FASTA format[68]. We converted it to a '*'-delimited sequence database file using the Python script PrepDB.py that comes with the InsPecT software written by the UCSD bioinformatics group, downloadable from the software's home page[69].

This human database contains 72340 sequences, 30445515 characters long, so an average sequence is 421 characters long. After conversion, the database file is approximately 30Mb in size.

The mouse sequence used as the query is 10000 characters long, and there are 72340 match operations involving it that run independently. The set of scoring criteria used is (match=1, mismatch=-5, indel=-5) or (match=1, mismatch=-30, indel=-20). The threshold score is 0, meaning that we want all 72340 match results to be returned.

We used this same set of parameters and dataset for the experiments on both Amazon Web Services and Sun Grid Compute Utility.

Bibliography

[1] Xen, "http://www.xen.org/," .

[2] Salesforce.com, "http://www.salesforce.com/," .

[3] iTunes, "http://www.apple.com/itunes/," .

[4] Microsoft Online Services, "http://www.microsoft.com/online/exchange-hosted-services.mspx," .

[5] Microsoft Exchange Server, "http://www.microsoft.com/exchange/default.mspx," .

[6] Amazon Web Services, "http://aws.amazon.com/," .

[7] Sun Grid Compute Utility, "http://www.network.com/," .

[8] Google App Engine, "http://code.google.com/appengine/," .

[9] Azure Services Platform, "http://www.microsoft.com/azure/default.mspx," .

[10] Apache Hadoop, "http://hadoop.apache.org/," .

[11] Basic Local Alignment Search Tool, "http://blast.ncbi.nlm.nih.gov/blast.cgi," .

[12] RUBiS, "http://rubis.objectweb.org/," .

[13] Amazon Elastic Compute Cloud (EC2), "http://aws.amazon.com/ec2/," .

[14] Amazon Simple Storage Service (S3), "http://aws.amazon.com/s3/," .

[15] Amazon SimpleDB, "http://aws.amazon.com/simpledb/," .

[16] Amazon Simple Queue Service (SQS), "http://aws.amazon.com/sqs/," .

[17] The Hadoop Distributed File System: Architecture and Design, "http://hadoop.apache.org/core/docs/r0.16.4/hdfs_design.html," .

[18] Jeffrey Dean and Sanjay Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, December 2004.

[19] Waterman MS Smith TF, "Identification of Common Molecular Subsequences," *Journal of Molecular Biology*, vol. 147, pp. 195197, 1981.

[20] D. S. Hirschberg, "A linear space algorithm for computing maximal common subsequences," *Comm. A.C.M*, vol. 18(6), pp. 341–343, 1975.

[21] Howard Gobioff Sanjay Ghemawat and Shun-Tak Leung, "The Google File System," *19th ACM Symposium on Operating Systems Principles*, October 2003.

[22] Hadoop DFS User Guide, "http://hadoop.apache.org/core/docs/r0.16.4/hdfs_user_guide.html," .

[23] Hadoop Quickstart, "http://hadoop.apache.org/core/docs/r0.16.4/quickstart.html," .

[24] Hadoop Core Releases, "http://hadoop.apache.org/core/releases.html#Download," .

[25] Hadoop Wiki AmazonEC2, "http://wiki.apache.org/hadoop/AmazonEC2," .

[26] IBM MapReduce Tools for Eclipse, "http://www.alphaworks.ibm.com/tech/mapreducetools," .

[27] Hadoop Map-Reduce Tutorial, "http://hadoop.apache.org/core/docs/r0.16.4/mapred_tutorial.html," .

[28] Hadoop 0.16.4 API, "http://hadoop.apache.org/core/docs/r0.16.4/api/index.html," .

[29] Hadoop Core Mailing Lists, "http://hadoop.apache.org/core/mailing_lists.html," .

[30] Hadoop provided Sort program, "http://wiki.apache.org/hadoop/Sort," .

[31] Hadoop provided RandomWriter program, "http://wiki.apache.org/hadoop/RandomWriter," .

[32] RightScale Blog, "Network performance within Amazon EC2 and to Amazon S3. http://blog.rightscale.com/2007/10/28/network-performance-within-amazon-ec2-and-to-amazon-s3/," .

[33] Sun Grid Engine, "http://gridengine.sunsource.net/," .

[34] The Message Passing Interface (MPI) standard, "http://www.mcs.anl.gov/research/projects/mpi/," .

[35] MPICH-A Portable Implementation of MPI, "http://www.mcs.anl.gov/research/projects/-mpi/mpich1/," .

[36] Compute Server Technology, "https://computeserver.dev.java.net/," .

[37] Sun Grid Plug-in, "https://sungridplugin.dev.java.net/," .

[38] Compute Server mailing lists, "https://computeserver.dev.java.net/servlets/projectmailinglistlist," .

[39] Python programming language, "http://www.python.org/," .

[40] Python standard library, "http://docs.python.org/library/index.html," .

[41] The Python Datastore API, "http://code.google.com/appengine/docs/python/datastore-/," .

[42] The Python Google Accounts API, "http://code.google.com/appengine/docs/python/users-/," .

[43] The Python URL Fetch API, "http://code.google.com/appengine/docs/python/urlfetch-/," .

[44] Python Mail API, "http://code.google.com/appengine/docs/python/mail/," .

[45] The webapp framework, "http://code.google.com/appengine/docs/python/tools/webapp-/," .

[46] The Django web application framework, "http://www.djangoproject.com/," .

[47] Types and Property Classes. Google App Engine, "http://code.google.com/appengine/docs-/python/datastore/typesandpropertyclasses.html," .

[48] Python Memcache API, "http://code.google.com/appengine/docs/python/memcache/," .

[49] Python Images API, "http://code.google.com/appengine/docs/python/images/," .

[50] Sanjay Ghemawat Wilson C. Hsieh Deborah A. Wallach Mike Burrows Tushar Chandra Andrew Fikes Fay Chang, Jeffrey Dean and Robert E. Gruber, "Bigtable: A Distributed Storage System for Structured Data," *OSDI'06: Seventh Symposium on Operating System Design and Implementation*, November 2006.

[51] Jeffrey D. Ullman Hector Garcia-Molina and Jennifer Widom, ," *Database System Implementation*, pp. Prentice Hall, ISBN–10: 0130402648 ISBN–13: 978–0130402646, June 1999.

[52] D. Mosberger and T. Jin, "httperf: A Tool for Measuring Web Server Performance," *Performance Evaluation Review*, vol. 26, pp. 31–37, December 1998.

[53] Google App Engine SDK, "http://code.google.com/appengine/downloads.html," .

[54] Configuring Eclipse on Windows to Use With Google App Engine, "http://code.google.com/appengine/articles/eclipse.html," .

[55] Pydev Extensions, "http://pydev.sourceforge.net/," .

[56] Python Web Server Gateway Interface v1.0, "http://www.python.org/dev/peps/pep-0333/," .

[57] CherryPy, "http://www.cherrypy.org/," .

[58] Pylons, "http://pylonshq.com/," .

[59] web.py, "http://webpy.org/," .

[60] Google App Engine Discussion Group, "http://groups.google.com/group/google-appengine," .

[61] Virtualization with Hyper-V, "http://www.microsoft.com/windowsserver2008/en/us/hyper-v.aspx," .

[62] Language Integrated Query, "http://en.wikipedia.org/wiki/Language_Integrated_Query," .

[63] Amazon CloudFront, "http://aws.amazon.com/cloudfront/," .

[64] Donald Kossmann Daniela Florescu, "Rethinking the Cost and Performance of Database Systems," *Submitted for publication*, December 2008.

[65] David A. Graf Donald Kossmann Matthias Brantner, Daniela Florescu and Tim Kraska, "Building a database on S3," *SIGMOD Conference*, June 2008.

[66] Jinesh Varia. Cloud Architectures, "http://jineshvaria.s3.amazonaws.com/public/cloudarch-itectures-varia.pdf," .

[67] Online biological data repository, "ftp://ftp.ebi.ac.uk/pub/databases/ipi/current/," .

[68] FASTA format, "http://en.wikipedia.org/wiki/FASTA_format," .

[69] InsPecT, "http://proteomics.bioprojects.org/software/inspect.html," .