Building a Cloud for Yahoo!

Brian F. Cooper, Eric Baldeschwieler, Rodrigo Fonseca, James J. Kistler, P.P.S. Narayan,
Chuck Neerdaels, Toby Negrin, Raghu Ramakrishnan, Adam Silberstein,
Utkarsh Srivastava, and Raymie Stata

Yahoo! Inc.

Abstract

Yahoo! is building a set of scalable, highly-available data storage and processing services, and deploying them in a cloud model to make application development and ongoing maintenance significantly easier. In this paper we discuss the vision and requirements, as well as the components that will go into the cloud. We highlight the challenges and research questions that arise from trying to build a comprehensive web-scale cloud infrastructure, emphasizing data storage and processing capabilities. (The Yahoo! cloud infrastructure also includes components for provisioning, virtualization, and edge content delivery, but these aspects are only briefly touched on.)

1 Introduction

Every month, over half a billion different people check their email, post photos, chat with their friends, and do a myriad other things on Yahoo! sites. We are constantly innovating by evolving these sites and building new web sites, and even sites that start small may quickly become very popular. In addition to the websites themselves, Yahoo! has built services (such as platforms for social networking) that cut across applications. Sites have typically solved problems such as scaling, data partitioning and replication, data consistency, and hardware provisioning individually.

In the cloud services model, all Yahoo! offerings should be built on top of cloud services, and only those who build and run cloud services deal directly with machines. In moving to a cloud services model, we are optimizing for human productivity (across development, quality assurance, and operations): it should take but a few people to build and rapidly evolve a Web-scale application on top of the suite of horizontal cloud services. In the end-state, the bulk of our effort should be on rapidly developing application logic; the heavy-lifting of scaling and high-availability should be done in the cloud services layer, rather than at the application layer, as is done today. Observe that while there are some parallels with the gains to be had by building and re-using common software platforms, the cloud services approach goes an important step further: developers are insulated from the details of provisioning servers, replicating data, recovering from failure, adding servers to support more load, securing data, and all the other details of making a neat new application into a web-scale service that millions of people can rely on.

In this paper, we describe the requirements, how the pieces of the cloud fit together, and the research challenges, especially in the areas of data storage and processing. We note that while Yahoo!'s cloud can be used to support externally-facing cloud services, our first goal is to provide a common, managed, powerful infrastructure for Yahoo! sites and services, i.e., to support internal developers. It is also our goal to open source as many components of the cloud as possible. Some components (such as Hadoop) are already in open source. This would allow others outside of Yahoo! to build their own cloud services, while contributing fixes and enhancements that make the cloud more useful to Yahoo!

2 Requirements

Yahoo! has been providing several centrally managed data management services for years, and while these services are not properly "cloud services" they have many of the characteristics. For example, our database of user profiles is run as a central service. Accessing the user database requires only the proper permissions and a client library, avoiding the need to set up and manage a separate user information repository for every application. Experience with these "proto-cloud" services have helped inform the set of requirements we laid out for our cloud:

Multitenancy We must be able to support many applications (tenants) on the same hardware and software infrastructure. These tenants must be able to share information but have their performance isolated from one another, so that a big day for Yahoo! Mail does not result in a spike in response time for Yahoo! Messenger users. Moreover, adding a new tenant should require little or no effort beyond ensuring that enough system capacity has been provisioned for the new load.

Elasticity The cloud infrastructure is sized based on the estimates of tenant requirements, but these requirements are likely to change frequently. We must be able to quickly and gracefully respond to requests from tenants for additional capacity, e.g., a growing site asks for additional storage and throughput.

Scalability We must be able to support very large databases, with very high request rates, at very low latency. The system should be able to scale to take on new tenants or handle growing tenants without much effort beyond adding more hardware. In particular, the system must be able to automatically redistribute data to take advantage of the new hardware.

Load and Tenant Balancing We must be able to move load between servers so that hardware resources do not become overloaded. In particular, in a multi-tenant environment, we must be able to allocate one application's unused or underused resources to another to provide even faster absorption of load spikes. For example, if a major event is doubling or quadrupling the load on one of our systems (as the 2008 Olympics did for Yahoo! Sports and News), we must be able to quickly utilize spare capacity to support that extra load.

Availability The cloud must always be on. If a major component of the cloud experiences an outage, it will not just be a single application that suffers but likely all of them. Although there may be server or network failures, and even a whole datacenter may go offline, the cloud services must continue to be available. In particular, the cloud will be built out of commodity hardware, and we must be able to tolerate high failure rates.

Security A security breach of the cloud will impact all of the applications running on it; security is therefore critical.

Operability The systems in the cloud must be easy to operate, so that a central team can manage them at scale. Moreover, the interconnections between cloud systems must also be easy to operate.

Metering We must be able to monitor the cloud usage of individual applications. This information is important to make provisioning decisions. Moreover, the cloud will be paid for by those applications that use it, so usage data is required to properly apportion cost.
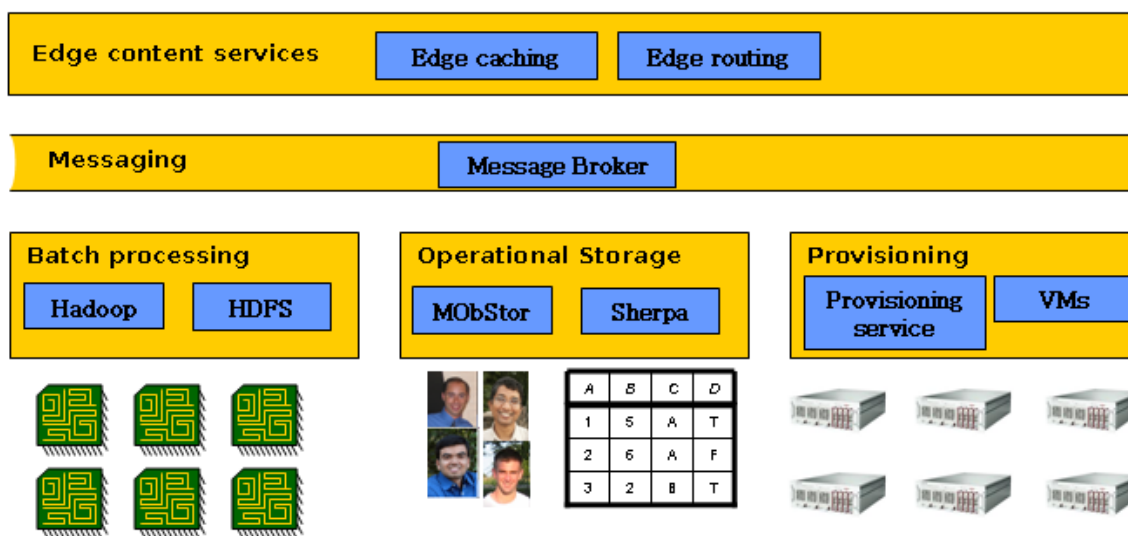
Figure 1: Components of the Yahoo! data and processing cloud.

Global Yahoo! has users all over the world, and providing a good user experience means locating services in datacenters near our users. This means that cloud services must span continents, and deal with network delays, partitions and bottlenecks as they replicate data and services to far flung users.

Simple APIs We must expose simple interfaces to ease the development cost of using the cloud, and avoid exposing too many parameters that must be tuned in order for tenant applications to get good performance.

3 Overall architecture

Yahoo!'s cloud focuses on "horizontal services," which are common platforms shared across a variety of applications. Those applications may themselves be "vertical services," which are task-specific applications shared by a variety of end users. For example, we view Yahoo! Mail as a vertical service, while a blob store (such as our MObStor system) is a horizontal service that can store attachments from Mail, photos from Flickr, movie trailers from Yahoo! Movies, and so on.

Figure 1 shows a block diagram of the main services in our cloud. As the figure shows, there are three tiers of services: core services; messaging; and edge services. While the bottom tier provides the heavy lifting for server-side data management, the edge services help reduce latency and improve delivery to end users. These edge services include edge caching of content as well as edge-aware routing of requests to the nearest server and around failures. The messaging tier helps tie disparate services together. For example, updates to an operational store may result in a cache invalidation, and the messaging tier carries the invalidation message to the cache.

The bottom tier of core services in Figure 1 is further subdivided into three groups of systems. Batch processing systems manage CPU cycles on behalf of large parallel jobs. Specifically, we have deployed Hadoop, an open source version of MapReduce [3], and its HDFS filesystem. Operational storage systems manage the storage and querying of data on behalf of applications. Applications typically have two kinds of operational data: structured records and unstructured blobs. In our infrastructure, structured data is managed by Sherpa (also known as PNUTS [2]), while blobs are stored in MObStor. Provisioning systems manage the allocation of servers for all of the other service components. One way to provision servers is to deploy them as virtual machines, and our provisioning framework includes the ability to deploy either to a VM or to a "bare" machine.

3

The horizontal services in our cloud provide platforms to store, process and effectively deliver data to users. A typical vertical application will likely combine multiple horizontal services to satisfy all of its data needs. For example, Flickr might store photos in MObStor and photo tags in Sherpa, and use Hadoop to do offline analysis to rank photos in order of popularity or "interestingness." The computed ranks may then be stored back in Sherpa to be used when responding to user requests. A key architecture question as we move forward deploying the cloud is how much of this "glue" logic combining different cloud services should be a part of the cloud as well.

In the rest of this article, we focus on the operational storage and batch computation components, and examine these components in more detail.

4 Pieces of the cloud

4.1 Hadoop

Hadoop [1] is an open source implementation of the MapReduce parallel processing framework [3]. Hadoop hides the details of parallel processing, including distributing data to processing nodes, restarting subtasks after a failure, and collecting the results of the computation. This framework allows developers to write relatively simple programs that focus on their computation problem, rather than on the nuts and bolts of parallelization. Hadoop data is stored in the Hadoop File System (HDFS), an open source implementation of the Google File System (GFS) [4].

In Hadoop, developers write their MapReduce program in Java, and divide the logic between two computation phases. In the Map phase, an input file is fed to a task, which produces a set of key-value pairs. For example, we might want to count the frequency of words in a web crawl; the map phase will parse the HTML documents and output a record (`term,1`) for each occurrence of a term. In the Reduce phase, all records with the same key are collected and fed to the same reduce process, which produces a final set of data values. In the term frequency example, all of the occurrences of a given term (say, "cloud") will be fed to the same reduce task, which can count them as they arrive to produce the final count.

The Hadoop framework is optimized to run on lots of commodity servers. Both the MapReduce task processes and the HDFS servers are horizontally scalable: adding more servers adds more compute and storage capacity. Any of these servers may fail at any time. If a Map or Reduce task fails, it can be restarted on another live server. If an HDFS server fails, data is recovered from replicas on other HDFS servers. Because of the high volume of inter-server data transfer necessary for MapReduce jobs, basic commodity networking is insufficient, and extra switching resources must be provisioned to get high performance.

Although the programming paradigm of Hadoop is simple, it enables many complex programs to be written. Hadoop jobs are used for data analysis (such as analyzing logs to find system problems), data transformation (such as augmenting shopping listings with geographical information), detecting malicious activity (such as detecting click fraud in streams of ad clicks) and a wide variety of other activities.

In fact, for many applications, the data transformation task is sufficiently complicated that the simple framework of MapReduce can become a limitation. For these applications, the Pig language [5] can be a better framework. Pig provides relational-style operators for processing data. Pig programs are compiled down to Hadoop MapReduce jobs, and thus can take advantage of the scalability and fault tolerance of the Hadoop framework.

4.1.1 Hadoop in the cloud

Hadoop runs on a large cluster of centrally managed servers in the Yahoo! cloud. Although users can download and run their own Hadoop instance (and often do for development purposes) it is significantly easier to run Hadoop jobs on the centrally managed processing cluster. In fact, the convenience of storing and processing data in the cloud means that much of the data in our cluster is Hadoop from birth to death: the data is stored

in HDFS at collection time, processed using MapReduce, and delivered to consumers without being stored in another filesystem or database. Other applications find it more effective to transfer their data between Hadoop and another cloud service. For example, a shopping application might receive a feed of items for sale and store them in Sherpa. Then, the application can transfer large chunks of listings to Hadoop for processing (such as geocoding or categorization), before being stored in Sherpa again to be served for web pages.

Hadoop is being used across Yahoo by multiple groups for projects such as response prediction for advertising, machine learned relevance for search, content optimization, spam reduction and others. The Yahoo! Search Webmap is a Hadoop application that runs on a more than 10,000 core Linux cluster and produces data that is now used in every Yahoo! Web search query. This is the largest Hadoop application in production, processing over a trillion page links, with over 300 TB of compressed data. The results obtained were 33 percent faster than the pre-Hadoop process on a similar cluster. This and a number of other production system deployments in Yahoo! and other organizations demonstrate how Hadoop can handle truly Internet scale applications in a cost-effective manner[1].

4.2   MObStor

Almost every Yahoo! application uses mass storage to store large, unstructured data files. Examples include Mail attachments, Flickr photos, restaurant reviews for Yahoo! Local, clips in Yahoo! Video, and so on. The sheer number of files that must be stored means they are too cumbersome to store and organize on existing storage systems; for example, while a SAN can provide enough storage, the simple filesystem interface layered on top of a SAN is not expressive enough to manage so many files. Moreover, to provide a good user experience, files should be stored near the users that will access them.

The goal of MObStor is to provide a scalable mass storage solution. The system is designed to be scalable both in terms of total data stored, as well as the number of requests per second for that data. At its core, MObStor is a middleware layer which virtualizes mass storage, allowing the underlying physical storage to be SAN, NAS, shared nothing cluster filesystems, or some combination of these. MObStore also manages the replication of data between storage clusters in geographically distributed datacenters. The application can specify fine-grained replication policies, and the MObStor layer will replicate data according to the policies.

Applications create collections of files, and each file is identified with a URL. This URL can be embedded directly in a web page, enabling the user's browser to retrieve files from the MObStore system directly, even if the web page itself is generated by a separate HTTP or application server. URLs are also virtualized, so that moving or recovering data on the back end filesystem does not break the URL. MObStor also provides services for managing files, such as expiring old data or changing the permissions on a file.

4.2.1   MObStor in the cloud

As with the other cloud systems, MObStor is a centrally managed service. Storage capacity is pre-provisioned, and new applications can quickly create new collections and begin storing and serving data. Mobstor uses a flat domain based access model. Applications are given a unique domain and can organize their data in any format they choose. A separate metadata store provides filesystem-like semantics: users can create, list and delete files through the REST interface.

Mobstor is optimized for serving data to internet users at Yahoo! scale. A key component of the architecture is a caching layer that also supports streaming. This enables the system to offload hot objects to the caching infrastructure, allowing the I/O subsystem to scale. Like other cloud services, Mobstor strives to locate data close to users to reduce latency. However due to the cost of the underlying storage and the fact that users are less sensitive to latency with large files, Mobstor does not have to support the same level of replication as the other cloud services.

---

[1]Thanks to Ajay Anand from the Hadoop team for these statistics.

There are many Yahoo! applications currently using MObStor. Examples include:

- Display ads for the APT platform

- Tile images for Yahoo! Maps

- Files shared between Yahoo! Mail users

- Configuration files for some parts of the Yahoo! homepage

- Social network invites for the Yahoo! Open platform

Each of these use cases benefits from the ability to scalably store and replicate a large number of unstructured objects and to serve them with low latency and high throughput.

4.3 Sherpa

The Sherpa system, also called PNUTS in previous publications [2, 6], presents a simplified relational data model to the user. Data is organized into tables of records with attributes. In addition to typical data types, "blob" is a valid data type, allowing arbitrary structures inside a record, but not necessarily large binary objects like images or audio; MObStor is a more appropriate store for such data. We observe that blob fields, which are manipulated entirely in application logic, are used extensively in practice. Schemas are flexible: new attributes can be added at any time without halting query or update activity, and records are not required to have values for all attributes. Sherpa allows applications to declare tables to be hashed or ordered, supporting both workloads efficently.

The query language of Sherpa supports selection and projection from a single table. We designed our query model to avoid operations (such as joins) which are simply too expensive in a massive scale system. While restrictive compared to relational systems, our queries in fact provide very flexible access that covers most of the web workloads we encounter. The system is designed primarily for online serving workloads that consist mostly of queries that read and write single records or small groups of records. Thus, we expect most scans to be of just a few tens or hundreds of records, and optimize accordingly. Scans can specify predicates which are evaluated at the server. Similarly, we provide a "multiget" operation which supports retrieving multiple records (from one or more tables) in parallel by specifying a set of primary keys and an optional predicate, but again expect that the number of records retrieved will be a few thousand at most.

While selections can be by primary key or specify a range, updates and deletes must specify the primary key. Consider a social networking application: A user may update her own record, resulting in access by primary key. Another user may scan a set of friends in order by name, resulting in range access.

Data in Sherpa is replicated to globally distributed datacenters. This replication is done asynchronously: updates are allowed to a given replica, and success is returned to the user before the update is propagated to other replicas. To ensure the update is not lost, it is written to multiple disks on separate servers in the local datacenter.

Asynchronously replicated data adds complexity for the developer. Sherpa provides a consistency model to simplify the details of reading and writing possibly stale data, and to hide the details of which replica is being accessed from the applciation.

4.3.1 Sherpa in the cloud

Sherpa is a hosted service, and the software and servers are managed by a central group. Applications that wish to use Sherpa can develop against a single-server standalone instance. However, all production data is served from cloud servers. This allows application developers to focus on their application logic, and leave the details

6

of designing, deploying and managing a data architecture to a specialized group. In order to support this hosted model, the Sherpa operations group must provision enough capacity to support all the applications that will use it. Currently, we work with customers to estimate their capacity needs and then pre-provision servers for their use. We are moving to a model with extra servers in a "free pool," and if an application's load on Sherpa begins to increase, we can automatically move servers from the free pool into active use for that application.

Sherpa is designed to work well with the other cloud services. For example, Hadoop can use Sherpa as a data store instead of the native HDFS, allowing us to run MapReduce jobs over Sherpa data. We also have implemented a bulk loader for Sherpa which runs in Hadoop, allowing us to transfer data from HDFS into a Sherpa table. Similarly, Sherpa can be used as a record store for other cloud services. As an example, MObStor is investigating using Sherpa to store metadata about files.

5   Open questions

Many research questions have arisen as we build the cloud, both at the level of individual components and across components. In this section, we discuss some key questions that span cloud components. Although many of our cloud components are in production or nearing completion, these questions will have to be resolved in order for the cloud to reach its full potential.

Interacting with the cloud   How do users interact with the cloud? One possibility is that each application chooses individual cloud systems, and manages their interactions at the application level. For example, suppose a user has a record-oriented data set and an OLTP workload. He therefore loads it into a Sherpa database. Periodically, he does extensive OLAP work. At these times, he loads the data set from Sherpa to HDFS, and runs Hadoop jobs.

However, one of the advantages of using the cloud is that it can provide seamless integration between multiple services. The job of the developer is easier if he does not have to explicitly manage data storage. For applications that will use multiple services, a nicer abstraction may be that data is placed "in the cloud" and is accessible to any service the application needs. In the above example, the data may be stored in Sherpa, but when an OLAP job is submitted, the cloud software decides whether to move the data to HDFS or to run a MapReduce job directly over Sherpa. This approach makes the cloud more complex, as it is an optimization problem which must take into account the query workload as well as information about the current capabilities and load on each of the services; the profile of future queries from the same application; the load from other services; and so on.

Another way we can make it easier for developers to use cloud services is to provide a common API for the various services. For example, we may develop a query language which spans multiple services (e.g., some combination of Pig, SQL, and the simple Sherpa and MObStor access languages) which then compiles to operations on individual services as well as actions to move data between them. If we hide many of the data placement and service selection decisions behind a declarative query language, we may not always make the best decisions without at least some input from the developer. We will likely need a mechanism for profiling the performance of the system, so that developers can readily identify which components are becoming a bottleneck. Such a mechanism will need to monitor an entire application as it interacts across multiple cloude services. In addition, a hint mechanism will be needed which allows the application to guide data placement and allocation decisions based on observations of the bottlenecks.

Quality of service   A key question for any shared infrastructure is how we can isolate the performance of different applications. Applications will place variable load on the cloud, and a spike in one application's workload will affect other applications sharing the same hardware. One approach to this problem is to place quotas on applications' resource usage. This approach is too inflexible, since spare resources cannot be used to

absorb load spike beyond an application's quota. We could also use some version of weighted fair sharing (like that used in networking systems), which allows spare resources to be allocated to needy applications. However, the infrastructure needed to monitor and dynamically allocate resources is complex. A third approach is to have a small number (e.g. two) of application classes. "Gold" applications can use as many resources as they like, while "bronze" applications are served by the remaining resources in a best effort manner. This moves resource allocation decisions into the hands of the business people, who must carefully choose just a few gold applications. Whatever approach we use will have to effectively enforce QoS for an application, even as it crosses over between Sherpa, Hadoop and MObStor, and other components of the cloud.

Other open issues There are several other issues which we are investigating:

- *Automating operations* - Our central operations group will be managing many servers, many different services, and many applications. Tools and processes which automate things like failover and resource allocation will make their jobs significantly easier.

- *Growth* - Applications which start small can grow to become quite large. Although our cloud services are designed to scale elastically, we must investigate how well they tolerate growth of one, two or more orders of magnitude.

- *Privacy* - Each system in the cloud has its own data model and thus its own model of enforcing privacy for that data. When data starts moving between systems, we must ensure that the change to a different data model does not cause a privacy breach. Moreover, we must ensure that multi-tenant applications can only see each other's data if doing so does not violate the user's privacy.

- *Capacity management* - It is difficult to know how much hardware to dedicate to the cloud to meet the anticipated load. Even if one resource (e.g. CPU) is plentiful, another resource (e.g. in-memory cache space) may be scarce. Similarly, a shift in the load, such as a move from sequential to random record access, can create a new bottleneck in a previously plentiful resource. We need to develop effective and comprehensive models for planning the capacity needs of the cloud.

6 Conclusion

We believe that the Yahoo! cloud will be a key to both lower costs and increased innovation. The Cloud Computing and Data Infrastructure division has the charter to develop cloud services such as Hadoop, MObStor and Sherpa, make them elastic, robust and reliable, and integrate them into a comprehensive cloud infrastructure. Many cloud components are already adopted widely, and we are seeing further rapid growth on the horizon.

References

[1] Hadoop. hadoop.apache.org.

[2] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. PNUTS: Yahoo!s hosted data serving platform. In *Proc. VLDB*, 2008.

[3] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, 2004.

[4] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google File System. In *Proc. SOSP*, 2003.

[5] C. Olston et al. Pig Latin: A not-so-foreign language for data processing. In *Proc. SIGMOD*, 2008.

[6] A. Silberstein et al. Efficient bulk insertion into a distributed ordered table. In *Proc. SIGMOD*, 2008.