INTERNATIONAL JOURNAL OF CLOUD COMPUTING TECHNOLOGICAL IDEAS
ISSN: 0974-8656 / www.sijccti.weebly.com / VOL3-NO1-PP 1020-1024 (2013)

Implementaion On Amazon Web Services

K.Rama Krishna Rao*1, Dr.A.Srinivasa Rao*2

[1]*Associate professor, CSE Department, Ideal Engg. College, Kakinada., India.*

[2]professor, CSE Department, KL University, Guntur Dt., India

Abstract---Cloud computing has seen tremendous growth, particularly for commercial web applications. The on-demand, pay-as-you-go model creates a flexible and cost-effective means to access compute resources. For these reasons, the scientific computing community has shown increasing interest in exploring cloud computing. However, the underlying implementation and performance of clouds are very different from those at traditional supercomputing centers. It is therefore critical to evaluate the performance of HPC applications in today's cloud environments to understand the tradeoffs inherent in migrating to the cloud. This study was carried out on Amazon web services platform, as infrastructure as a service in the geo-spatial areas. Under this environment, cloud instance, a web and mobile system being previously implemented in the multi-layered structure for geo-spatial open sources of database and application server, was generated. Judging from this example, it is highly possible that cloud services with the functions of geo-processing service and large volume data handling are the crucial point, leading a new business model for civilian remote sensing application and geo-spatial enterprise industry. The further works to extend geo-spatial applications in cloud computing paradigm are left. Key Words : Amazon platform, Amazon web service, Cloud computing

INTRODUCTION

This section describes the implementation of the API of using services available from Amazon. Although we restrict it to Amazon, other providers offer similar features and allow for adopting the discussed methods.

Storage Service

The API for the reliable storage service has already been designed in the lines of storage services from Google (BigTable), Amazon (S3) and others. It just requires a persistent store with eventual consistency properties with latest time-stamp conflict resolution or better for storing large objects. The mapping of the API to Amazon S3 is straightforward . However, note that Amazon makes use of a concept called Bucket. Buckets can be seen as big folders for a collection of objects identified by the URI. We do not require buckets and assume the bucket name to be some fixed value.

Machine Reservation

The Machine Reservation API consists of just two methods, start and stop of a virtualized machine. Consequently, Amazon Elastic Compute Cloud (EC2) is a direct fit to implement the API and additionally offers much more functionality. Furthermore, EC2 also enables us to experiment with alternative cloud service implementations and to build services which are not offered directly in the cloud. This not only saves a lot of money as transfer cost between Amazon machines is for free, but also makes intermediate service requests faster due to reduced network latency.

Simple Queues

Simple Queues do not make any FIFO guarantees and do not guarantee that all messages are available at all times. One way to implement Simple Queues is using Amazon SQS although SQS is not reliable because it deletes messages after 4 days. If we assume that for any page a checkpoint happens in less than 4 days, SQS is suitable for implementing Simple Queues. Unfortunately, another overhead was introduced with.

Building a Database on Top of Cloud Infrastructure

the SQS version from 2008-01-01; that is, it is not possible to delete messages directly with the MessageID. Instead, it is required to receive the message before deleting it. Especially for the atomicity this change of SQS made the protocol to cause more overhead and consequently to be more expensive. Alternatively, Simple Queues can be implemented by S3 itself. Every message is written to S3 using the page URI, a timestamp and ClientID to guarantee a unique MessageID. By doing a prefix scan per page URI, all messages can be retrieved. S3 guarantees reliability and has ”no” restriction on the message size, which allows big chunks (especially useful for atomicity). It is therefore also a good candidate for implementing Simple Queues. Last but not least, Simple Queues can be achieved by the same mechanism as the Advanced Queues

Advanced Queues
Advanced Queues are most suited to be implemented on top of EC2. The range of possible implementations is huge. A simple implementation is to build a simple in memory queue system with a recovery log stored on EBS and time-to time check pointing to S3. Amazon claims an annual failure rate (AFR) for a 20 GB EBS drive of 0.1% - 0.5%, which is 10 times more reliable than a typical commodity disk drive. For log recovery with regular snapshots typically not more than 1 GB is required, resulting in an even better AFR. If higher reliability and availability is desired (EBS is a single data center solution), the queue service has to be replicated itself. Possibilities include replication over data-center boundaries using Paxos protocols [4] or multi-phase protocols [5]. Next to the reliability of the queuing service, load balancing is the biggest concern.
Usage frequencies of queues tend to differ substantially. We avoid complex load balancing by randomly assigning queues to servers. This simple load balancing strategy works the best if none of the queues is a bottleneck and is better the bigger the overall number of queues is. Unfortunately, the collection queues for inserts are natural bottlenecks.
However, small changes to the protocols allow for splitting those queues to several queues and hence, avoid the bottleneck. Still, more advanced load balancing strategies for the queuing service might be required. Solutions include queue handover with Paxos or multi-phase protocols. However, building more reliable and autonome queues is research for itself and we restrict our implementation to a simple in-memory solution with EBS logging.

Implementation on Amazon Web Services

Locking Service
AWS does not directly provide a locking service. Instead, it refers to SQS to synchronize processes. Indeed, SQS can be used as a locking service, as it allows to retrieve a message exclusively. Thus, locks can be implemented on top of SQS by holding one queue per lock with exactly one message. The timeout of the message is set to the timeout of the lock. If a client is able to receive the lock message, the client was able to receive the lock. Unfortunately, Amazon states that it deletes messages older than 4 days and even queues that have not been used for 30 consecutive days. Therefore, it is required to renew lock messages within 4 days. This is especially problematic as creating such a message is a critical task. A crash during deletion/creation can either result in an empty queue or in a queue with two lock messages. We therefore propose to use a locking service on EC2. Implementations for such a locking service are wide-ranging: The easiest way is to have a simple fail-over lock service. Thus, one server holds the state for all locks. If the server fails it gets replaced by another server with a clean empty state. However, this server has to reject lock requests, until all the lock timeouts since the failure have expired. Hence, all locks would have been automatically returned anyway. The lock manager service does not guarantee 100 percent availability, but it guarantees failure resilience. Other possibilities include more complex synchronization mechanisms like the ones implemented in Chubby [Bur06] or ZooKeeper [Apa08]. Again, we implement the simplest solution and build our services on top of EC2 using the described fail-over protocol. Developing our own locking service on EC2 gives the additional advantage of easily implementing shared, exclusive locks and the propagation from shared to exclusive without using an additional protocol. It seems quite likely that such a locking service will be offered by many cloud providers in the near future. Literature already states that Google, Yahoo already use such a service internally.

Advanced Counters
As for the locking service, using EC2 is the best way to implement advanced counters. If a server which hosts one (or several) counter(s) fails, then new counter(s) must be established on a new server. To always ensure increasing counter values, counters work with epoch numbers. This implies that if the counter fails, the epoch number is increased. Every counter value is prefixed with this epoch number. If a machine instance fails, a new machine replaces the service with a clean state, but with a higher epoch number. In other words, the counter service is not reliable and when it fails it .

Building a Database on Top of Cloud Infrastructure
loses its state. Again, like for the lock service, the fail-over time has to be longer than the counter-validation time. This ensures for GSI that requesting the highest validated value does not reveal an inconsistent state.

Indexing
For hosted indexes we make use of Amazon's SimpleDB service. Every primary and secondary index is stored in one table inside SimpleDB. The translation of key searches and range queries to SimpleDBs query language is

straightforward and not further discussed. Results from SimpleDB are cached on the client-side to improve performance and reduce the cost. SimpleDB is only eventual consistent. As a consequence, the Atomicity and Snapshot protocol together with SimpleDB do not provide exactly the specified consistency level. In the case of the primary index, the eventual consistency property can be compensated by considering the messages in the collection queues (assuming that a maximum propagation time exist). However, for secondary indexes it is not possible without introducing additional overhead. Thus, the Atomicity and Snapshot protocol do not provide the specified level of consistency if an secondary index access is made. This problem does not exists with client-side indexes, which are implemented similar to [Lom96] as discussed several times before. This behavior is similar to Google's MegaStore implementation [Ros09].

Performance Experiments and Results

Software and Hardware Used

We have implemented the Cloud API of AWS (S3 and EC2) as discussed in the previous section. Furthermore, we have implemented the protocols presented in on top of this Cloud API and the alternative client-server architecture Variants. This section presents the results of experiments conducted with this implementation and the TPC-W benchmark. More specifically, we have implemented the following consistency protocols: _ Na¨ıve: As in [BFG+08], this approach is used as a baseline. With this protocol, a client writes all dirty pages back to S3 at commit time, without using queues. This protocol is subject to lost updates because two records located on the same page may be updated by two concurrent clients. As a result, this protocol does not even fullfil eventual consistency. It is used as a baseline because it corresponds to the way that cloud services like S3 are used today.

TPC-W Benchmark

To study the trade-offs of the alternative consistency protocols and architecture variants, we use a TPC-W-like benchmark [2]. The TPC-W benchmark models an online bookstore and a mix of different so-called Web Interactions (WI). Each WI corresponds to a click of an online user; e.g., searching for products, browsing in the product catalog, or shopping cart transactions. The TPC-W benchmark specifies three kinds of workload mixes: (a) browsing, (b) shopping, and (c) ordering.

Building a Database on Top of Cloud Infrastructure

probability for each kind of request. In all the experiments, we use the Ordering Mix as the most update-intensive mix: About one third of the requests involve an update of the database. The TPC-W benchmark applies two metrics. The first metric is a throughput metric and measures the maximum number of valid WIs per second (i.e., requests per second). The second metric defined by the TPC-W benchmark is Cost/WIPS with WIPS standing for Web Interactions Per Second at the performance peak. This metric tries to relate the performance (i.e., WIPS) with the total cost of ownership for a computer system. To this end, the TPC-W benchmark gives exact guidance concerning the computation of a system's cost. For our experiments, both metrics have been relaxed in order to apply to cloud systems. Ideally, there exists no maximum throughput and the cost is not a fixed value but depends on the load. Therefore, we measure the average response time in secs for each WI and the average cost in milli-$ per WI. In contrast to the WIPS measures of the TPC-W benchmark these metrics allow for comparing the latency and cost trade-offs of the different consistency protocols and architecture variants. For the scalability experiments, however, we report WIPS to demonstrate the scalability. In all experiments, response time refers to the end-to-end wallclock time from the moment a request has been initiated at the end-user's machine until the request has been fully processed and the answer has been received by the end-user's machine. Throughout this section, we do not present error bars and the variance of response times and cost. Since the TPC-W benchmark contains a mix of operations with varying profile, such error bars would merely represent the artefacts of this mix. Instead, we present separate results for read WIs (e.g., searching and browsing) and update WIs (e.g., shopping cart transactions) whenever necessary. As shown in [BFG+08], the variance of response times and cost for the same type of TPC-W operation is not high using cloud computing. In this performance study, we have made the same observation. As already mentioned, the TPC-W benchmark is not directly applicable to a cloud environment . Thus, next to the metrics we apply the following changes: Benchmark Database: The TPC-W benchmark specifies that the size of the benchmark database grows linearly with the number of clients. Since we are specifically interested in the scalability of the services with regard to the transactional workload and elasticity with changing workloads, all experiments are carried out with a fixed benchmark database size and without pre-filling the order history. Consistency: The TPC-W benchmark requires strong consistency with ACID transitions. As shown before, not all protocols support this level of consistency._ Queries:

We change the bestseller query to randomly select products. Else, this query is best solved by a materialized view which is currently not supported in our implementation. _ We concentrate on measuring the transaction part and do not include the cost for picture downloads or web-page generation. Pictures can be stored on S3, so that they just add an additional fixed dollar cost for the transfer.
_ Emulated Browsers: The TPC-W benchmark specifies that WIs are issued by emulated browsers (EB). According to the TPC-W benchmark, emulated browsers use a wait-time between requests. Here, we do not use a wait-time for the browser emulation for simplicity and in order to scale more quickly to higher loads. Again, this does not influence the relations between the results for the different configurations.
_ HTTP: The TPC-W benchmark requires the use of the HTTPS protocol for secure client / server communication. As a simplification, we use the HTTP protocol (no encryption).

Architectures and Benchmark Configuration

There are several alternative client-server architectures and ways to implement indexing on top of cloud services like AWS. In this study, the following configurations are applied:

EU-BTree: The whole client application stack is executed on “enduser” machines; i.e., outside of the cloud. For the purpose of these experiments, we use Linux boxes with two AMD 2.2 GHz processors located in Europe. In this configuration, client-side B-Tree indexes are used for indexing; that is B-Tree index pages are shipped between the EU machines and S3 as specified . EU-SDB: The client application stack is executed on “end-user” machines. Indexing is done using SimpleDB. Again, the consistency levels for Atomicity and Snapshot are not directly comparable to the EC2-BTree configuration as secondary indexes suffer from the eventual consistency property of SimpleDB.

Building a Database on Top of Cloud Infrastructure

EC2-BTree: The client application stack is installed on EC2 servers. On the “enduser” side, only TPC-W requests are initiated. Indexing is done using a “clientside” B-Tree.
EC2-SDB: The client application stack is installed on EC2 servers. Indexing is carried out using SimpleDB. All five studied consistency protocols (Na¨ıve, Basic, Atomicity, Locking, and Snapshot Isolation) support the same interface at the record manager. Furthermore, all four client-server and indexing configurations support the same interfaces. As a result, the benchmark application code is identical for all variants. All experiments on the EU side are initiated from a single machine located in Europe. This machine, which simulats “end-user” clients, is a Linux box with two AMD processors and 6 GB RAM. For all EC2 variants, we use Medium-High-CPU EC2 instances. A TPC-W application installed on EC2 acts as web server and application server at the same time. Clients can connect to the server which in turn communicates with the cloud services. For the scale-out experiment, we have experienced problems to generate the high load from our cluster in Europe.18 Hence, we have decided to simulate the user requests, by running the client directly on the web server and application server and add a fixed latency (0.1s) and cost (0.020752 milli-$) for each generated page to simulate transfer cost ($) and latency (ms). These average latency times and network cost have been calculated upfront in a separate experiment. As the network latency and cost per page is independent of the used protocol or index, adding a fixed latency and cost value per transaction does not impact the significance of the experiments. The data-size for all basic and tuning experiments is set to 10,000 products and accessed by 10 concurrent emulated browser issuing permanent requests without any wait-time. Unless stated otherwise, the page size in all experiments is set to 100 kb, the checkpoint interval to 45 seconds and the time-to-live of data pages to 30 seconds. For all our experiments we use a US Amazon data center and thus, cost refers to the charges of Amazon for using EC2 and S3 inside a US location. Furthermore, for the sake of uniformity, we use for all experiments our Advanced Queue Service implementation.
Otherwise the response times for the different protocols would not have been comparable. Requests to advanced services such as Advanced queues, counters, and locks are priced at USD 0.01 per 1000 requests. The alternative would have been a dynamic pricing based on the utilization of the system using the EC2 instance pricWith today’s choices the possibilities of building applications inside the cloud are unlimited. Different services, possibly even from different providers, can be combined to achieve the same goal. Furthermore, by using virtualized machines, any kind of service can be created inside the cloud. Although there is no agreement on how to design applications inside the cloud, certain best practice recommendations exist. The simplest and probably fastest way to deploy an application inside the cloud is by use of a PaaS . The platform hides the complexity of the underlying infrastructure (e.g. load balancing, operating system, fault tolerance, firewalls) and provides all the services, typically through libraries, which

are required to build anbnapplication (e.g., tools for building web-based user interfaces, storage, authentication, testing, etc.). As invoking services outside the platform provider is rather expensive with respect to latency, best practice is to use the services and tools from a single PaaS provider. Although PaaS offers many advantages, the biggest drawback is the incompatibility between platforms and the enforced limitations of the platform. In addition, by today, the PaaS software is proprietary and no open-source systems exist, thus preventing the use of this approach for private cloud installation. Developers requiring more freedom and/or not willing to restrict themselves to one provider, may directly use IaaS offerings. These enable a more direct access to the underlying infrastructure. Clients connect by means of a web browser through a firewall to a web server. The web server is responsible for rendering the web-sites and interacts with a (possibly distributed) file system and an application server. The application server hosts and executes the application logic and interacts with a database, and possibly with the file system and other external systems. Although the general structure is similar for all web-based database applications, a huge variety of extensions and deployment strategies exists [3]. By means of virtualized machines all of those variations are also There are quantitative as well as qualitative benefits in maintaining quality assurance. The Quantitative benefits are reduced costs, greater efficiency, better performance, less unplanned work and fewer disputes. The Qualitative benefits are improved visibility and predictability, better control over contracted products, improved customer confidence, better quality, problems show up earlier and reduced risk.

Conclusion

Cloud computing is not something new now, and is a apidly emerging technology that almost every industry that rovides or consumes software, hardware, and infrastructure can leverage. The positive prediction on this theme is dominant with the expectations of the promise of the cloud as a compelling argument for leveraging off-premises hardware resources and on-demand services to reduce costs and improve workflow efficiencies. Technology progress will be advanced in the areas of infrastructure commoditization, open standards, linkage to crowd sourcing, and enterprise project decentralization. Penetration to all areas of business management is also accentuated. While cloud computing has proven itself useful for a wide range of e-Science applications, its utility for more tightly coupled HPC applications has not been proven. In this paper we have quantitatively examined the performance of a set of benchmarks designed to represent a typical HPC workload run on Amazon EC2.

References

[1] S. Hazelhurst, "Scientific computing using virtual high-performancecomputing: a case study using the Amazon elastic computing cloud," in Proceedings of the 2008 annual research conference of the South African Institute of Computer Scientists and Information Technologists on IT research in developing countries: riding the wave of technology. ACM, 2008, pp. 94–103.

[2] E. Deelman, G. Singh, M. Livny, B. Berriman, and J. Good, "The cost of doing science on the cloud: the montage example," in Proceedings of the 2008 ACM/IEEE conference on Supercomputing. IEEE Press, 2008, pp. 1–12.

[3] K. Keahey, R. Figueiredo, J. Fortes, T. Freeman, and M. Tsugawa, "Science clouds: Early experiences in cloud computing for scientific applications," Cloud Computing and Applications, vol. 2008.

[4] K. Keahey, "Cloud Computing for Science," in Proceedings of the 21st International Conference on Scientific and Statistical Database Management. Springer-Verlag, 2009, p. 478.

[5] J. Napper and P. Bientinesi, "Can cloud computing reach the top500?" in Proceedings of the combined workshops on UnConventional high performance computing workshop plus memory access workshop. ACM, 2009, pp. 17–20.