An Evaluation of Alternative Architectures for Transaction Processing in the Cloud

Donald Kossmann Tim Kraska Simon Loesing

Systems Group, Department of Computer Science, ETH Zurich, Switzerland
{firstname.lastname}@inf.ethz.ch

ABSTRACT

Cloud computing promises a number of advantages for the deployment of data-intensive applications. One important promise is reduced cost with a pay-as-you-go business model. Another promise is (virtually) unlimited throughput by adding servers if the workload increases. This paper lists alternative architectures to effect cloud computing for database applications and reports on the results of a comprehensive evaluation of existing commercial cloud services that have adopted these architectures. The focus of this work is on transaction processing (i.e., read and update workloads), rather than analytics or OLAP workloads, which have recently gained a great deal of attention. The results are surprising in several ways. Most importantly, it seems that all major vendors have adopted a different architecture for their cloud services. As a result, the cost and performance of the services vary significantly depending on the workload.

Categories and Subject Descriptors

H.3.4 [**Systems and Software**]: Performance evaluation (efficiency and effectiveness); H.2.4 [**Systems**]: Distributed databases; K.6.0 [**General**]: Economics

General Terms

Experimentation, Measurement, Performance, Economics

Keywords

Cloud Computing, Benchmark, Performance Evaluation, Cloud Provider, Cloud DB, Transaction Processing, Cost

1. INTRODUCTION

Recently, there has been a great deal of hype about cloud computing. Cloud computing is on the top of Gartner's list of the ten most disruptive technologies of the next years [14]. All major software vendors and many start-ups have jumped on the bandwagon and claim that they are either cloud-enabled or cloud-enabling.

Cloud computing makes several promises. It promises a reduced time-to-market by removing or simplifying the time-consuming hardware provisioning, purchasing, and deployment processes. It promises cost reductions in several ways. First, it promises to turn capital costs into operational costs by adopting a pay-as-you-go business model. Second, it promises a better (close to 100 percent) utilization of the hardware resources. Cloud computing is, therefore, often considered a critical technology for green computing. Furthermore, cloud computing reduces operational cost and pain by automating IT tasks such as security patches and fail-over. In terms of performance, cloud computing promises (virtually) infinite scalability so that IT administrators need not worry about peak workloads. Finally, cloud computing promises improved flexibility in the utilization and management of both software and hardware which translates into savings in both time-to-market and cost.

As of today, a number of products have been launched. In particular, three of the big players of the IT industry, namely Amazon, Google, and Microsoft, have made product offerings. All these offerings have in common that they are available to a general audience by packaging cloud computing technology as a service, which can be activated from any personal computer via a simple REST interface. Also, all these offerings are geared towards delivering on the key promises of cloud computing and their adoption in the IT market place is rapidly growing.

The goal of this paper is to set a first yardstone in evaluating the current offerings. Using the database and workload of the TPC-W benchmark, we assessed Amazon, Google, and Microsoft's offerings and compared the results to the results obtained with a more traditional approach of running the TPC-W benchmark on a Java application server and an off-the-shelf relational database system. In particular, we wanted to address the following questions:

- How well do the offerings scale with an increasing workload? Can indeed a (virtually) infinite throughput be achieved?
- How expensive are these offerings and how does their cost / performance ratio (i.e., bang for the buck) compare?
- How predictable is the cost with regard to changes in the workload?

Obviously, the results reported in this paper are just a snapshot of the current state-of-the-art. The contribution is to establish a framework that allows vendors to gradually improve their services and allows users to compare products.

As will be shown, our experiments resulted in a number of surprises. Even though, many services look similar from the outside (e.g., Microsoft Azure and Amazon Web Services price matrixes are almost identical in terms of network bandwidth, storage cost, and CPU cost), the services vary dramatically when it comes to end-to-end performance, scalability, and cost. Maybe even more surprising are the differences in the architectures that effect large-scale data management and transaction workloads in the cloud.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGMOD'10, June 6–11, 2010, Indianapolis, Indiana, USA.
Copyright 2010 ACM 978-1-4503-0032-2/10/06 ...$10.00.

SHOPIFY 1024

While most (traditional) general-purpose database systems (e.g., DB2, MySQL, Oracle 11, Postgres, SQL Server) share roughly the same "textbook" architecture and data structures (e.g., dynamic programming, B-tree indexes, write-ahead logging) [26], the differences in the implementation of cloud services are immense. While it is too early to come up with a textbook architecture for cloud services, this paper tries to look behind the scenes, classify the architectural variants, and set the stage for a comparison of the different architectural variants with regard to performance and cost.

The remainder of this paper is structured as follows: Section 2 summarizes related work. Section 3 describes alternative database architectures for transaction processing in the cloud. Section 4 gives an overview of the services and variants that we have used for the experiments and which represent the different architectures described in Section 3. Section 5 details the benchmark and experimental environment. Section 6 presents the results of the experiments. Section 7 contains conclusions and avenues for future research.

2. RELATED WORK

This work follows a long tradition in the database community to benchmark new breeds of data management systems as soon as the first products appear on the market place. The first work in that direction was the famous Wisconsin benchmark [10] which eventually resulted in the series of standardized TPC benchmarks for assessing database system performance and cost for different workloads; e.g., TPC-C and TPC-E for OLTP, TPC-H for OLAP, and TPC-W and TPC-App for whole web application stacks. Furthermore, a number of benchmarks have been developed for special-purpose database systems; e.g., OO7 for object-oriented databases [6], Bucky for object-relational databases [7], XMark for XML databases [21], and Sequoia for scientific databases [25]. Of course, there have also been numerous performance studies on various aspects of application servers, database systems, distributed database systems, and specific components of cloud computing infrastructures (e.g., DHTs). In a recent paper, the performance of relational database systems which run in a virtual machine has been studied [16]. Obviously, all these results are relevant. Rather than assessing individual components, however, the goal of our project was to measure the *end-to-end* performance of alternative architectures for the whole web application stack. One paper that particularly inspired our work is the classic paper on client-server database architectures [11].

With the emergence of cloud computing, several studies have assessed the performance and scalability of cloud computing infrastructures. In the database community recent work compared the performance of Hadoop versus the more traditional (SQL-based) database systems [19]. That work focusses on read-only, large-scale OLAP workloads whereas our work is focussed on OLTP workloads. The results of a related study on cost-consistency tradeoffs for OLTP workloads in the cloud have been reported in [15]. Berkeley's Cloudstone project is the most relevant related work. Cloudstone specifies a database and workload for studying cloud infrastructures [23] and defines performance and cost metrics to compare alternative systems. Indeed, we could have used the Cloudstone workload for our experiments but we chose the TPC-W benchmark because of its popularity and wide-spread acceptance in the community. This work is based on two previous position papers: [12] suggests to study the cost in addition to latency and throughput as part of performance experiments. [2] proposes a series of experiments in order to evaluate cloud computing infrastructures. This work can be seen as an initial step towards implementing the agenda proposed in [2]: It carries out the "scalability" and "cost" experiments proposed in [2]. Carrying out the "peak" and "fault tolerance" experiments proposed in [2] is left for future work. One paper that particularly inspired our work is the classic paper on client-server database architectures [11].

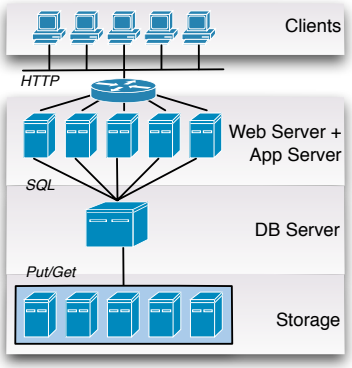

Figure 1: Classic Database Architecture

3. DISTRIBUTED DATABASE ARCHITECTURES

This section revisits distributed database architectures as they are used in cloud-computing today. First, the *classic* multi-tier database application architecture is described as a starting point. Then, four variations of this architecture are described. These variations are based on simple principles of distributed databases such as replication, partitioning, and caching. The interesting aspect is how these concepts have been packaged and adopted by commercial cloud services (Section 4).

3.1 Classic

As a starting point, Figure 1 shows the *classic* architecture used for most database applications today (e.g., SAP R/3 [5]). Requests from clients are dispatched by a load balancer (depicted as a carousel in Figure 1) to an available machine which runs a web and application server. The web server handles the (HTTP) requests from clients and the application server executes the application logic specified, e.g., in Java or C# with embedded SQL (or LINQ or any other database programming language). The embedded SQL is shipped to the database server which interprets this request, returns a result, and possibly updates the database. For persistence, the database server stores all data and logs on storage devices. The interface between the database server and the storage system involves shipping physical blocks of data (e.g., 64K blocks) using *get* and *put* requests. Traditional storage systems use disks which can be attached locally to the machine that runs the database server or which can be organized in a storage area network (SAN). Figure 1 shows the variant in which the storage system is separate from the database server (e.g., a SAN). Instead of disks, next generation storage systems could use solid-state disks, main memory, or a combination of different storage media.

The classic architecture has a number of important advantages. First, it allows to use "best-of-breed" components at all layers. As a result, a healthy market with a number of competing products has emerged at each layer. Second, the classic architecture allows *scalability* and *elasticity* at the storage and web/application server layers. For instance, if the throughput of the application needs to be adjusted due to an increased interest of clients, then it is easy to add machines at the web/application server layer in order to handle

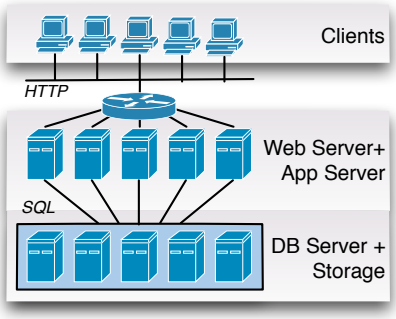

Figure 2: Partitioning

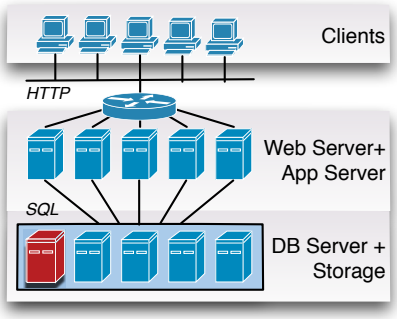

Figure 3: Replication

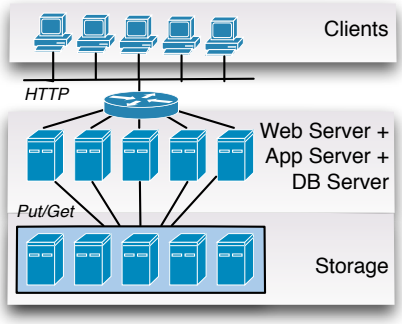

Figure 4: Distributed Control

the additional workload. Likewise, machines at that layer can be turned off or used for a different purpose, if the workload drops. At the storage layer, machines (or disks) can be added and removed in order to increase the bandwidth of the storage system for increased workloads and/or to deal with changes in the size of the database.

The potential bottleneck of the classic architecture is the database server. If the database server is overloaded, the only way out is to buy a bigger machine. The machines used as database servers tend to be quite expensive because they must be provisioned for peak workloads. Therefore, the classic architecture of Figure 1 has limitations in both scalability and cost, two important goals of cloud computing. The remainder of this section lists the architectures that cloud providers have chosen in order to overcome these limitations at the database server layer.

3.2 Partitioning

Figure 2 shows how the classic database architecture can be adapted in order to make use of partitioning. The idea is simple: Rather than having one database server control the whole database, the database is logically partitioned and each partition is controlled by a separate database server. In the database literature, many partitioning schemes have been studied; e.g., vertical partitioning vs. horizontal partitioning, round-robin vs. hashing vs. range partitioning [8]. All these approaches are relevant and can be applied to data management in the cloud.

In addition to the partition scheme, there are several variants of the architecture of Figure 2. First, the partitioning can be transparent or visible to the application programmer (obviously, transparency is desirable.) Second, the storage can be attached to the machines that run the database servers or dissociated in, say, a storage area network (as shown in Figure 1). Figure 2 depicts the variant in which the access to the distributed database is transparent and the storage is attached to each database server. In practice, other variants can also be found (Section 4).

For cloud computing, the architecture of Figure 2 was first adopted by Force.com, the platform that runs the Salesforce application and was opened to run custom-made applications. In Force.com, the partitioning key is the *tenant*. That is, the data is distributed according to the application that generated and owns the data. The partitioning involves the whole server-side application stack, including the web and application servers. All requests to the same tenant are handled by the same web, app, and database server. As a result, Force.com is tuned to scale with the number of applications. However, the Force.com architecture does not support the scalability of a *single* application beyond a single database server. Therefore, we did not include Force.com in our performance study. We expect that Force.com would show similar behavior as the variants that have adopted the classic architecture in our experiments.

Partitioning and the architecture of Figure 2 are a viable solution towards achieving the promises of cloud computing. The database servers can run on cheap machines, thereby using many machines that each operate on a fairly small data set in order to sustain the load. Partitioning, however, has scalability limitations with regard to dealing with a fluctuating workload: Adding or removing machines in order to deal with a higher (or lower) query/update workload involves repartitioning the data and, therefore, moving data between machines. In order to achieve better scalability and fault-tolerance, partitioning needs to be combined with replication.

3.3 Replication

Figure 3 shows how replication can be used in a database architecture. Again, the idea is simple and has been studied extensively in the past. As with partitioning, there are several database servers. Each database server controls a copy of the whole database (or partition of the database, if combined with partitioning). Furthermore, there are many variants conceivable. Figure 3 shows a variant in which the replication is transparent and the storage is associated to the database servers. The most important design aspect of replication is the mechanism to keep the replicas consistent. The most prominent protocol is ROWA (read-once, write all) based on a Master copy [20]. If replication is not transparent, applications direct all update requests to the database server which controls the Master copy, and the Master server propagates all committed updates to the *satellites* when these updates have been successfully committed. Applications can issue requests of read-only transactions to any database server (Master or satellite). If replication is transparent, then requests are routed automatically to the Master or a satellite. In Figure 3, transparent replication is depicted. It shows the Master server in red (the most left server).

Cheap hardware can be used in order to run the database servers. In particular, the satellites can run on cheap off-the-shelf machines. Furthermore, the architecture of Figure 3 can scale-out and down nicely with the workload, if the workload is read-mostly. At any point in time, a satellite server can be dropped in order to deal with a decreasing workload. Adding a satellite server for an increasing query workload involves copying the database from the Master (or a satellite) to the new server. For update-intensive workloads, the Master can become the bottleneck, as shown in Section 6.

Replication can be used in order to increase both the scalability and the reliability of a system. A specific protocol to ensure reliability based on replication was devised by Oracle as part of the Oracle RAC product [18].

3.4 Distributed Control

Figure 4 shows an architecture that models the database system as a distributed system. At first glance, this architecture looks sim-

ilar to the Partitioning and Replication architectures shown in Figures 2 and 3. The differences are subtle, but they have huge impact on the implementation, performance, and cost of a system. The Distributed Control architecture can also be characterized as a *shared-disk architecture* [24] with a loose coupling between the nodes in order to achieve scalability.

In this architecture, the storage system is separated from the database servers and the database servers access concurrently and autonomously the shared data from the storage system. In order to synchronize read and write access to the shared data, distributed protocols which guarantee different levels of consistency can be applied. Again, a large variety of different protocols are conceivable and the classic textbook that gives an overview of such protocols and consistency levels is [27]. In order to reduce overheads, the database tier is merged with the web and application server tier; that is, the database access is affected as a library as part of the application server, rather than providing separate database server processes.

This architecture is potentially the best match for cloud computing. It provides full scalability and elasticity at all tiers. Each HTTP request can be routed to any (web/app/DB) server so that full scalability can be achieved at that level. Furthermore, the data can be replicated and partitioned in any way at the storage layer so that scalability can be achieved at that level, too. Another feature of this architecture is that cheap hardware can be used at all tiers. This scalability, however, comes at a cost: Because of the CAP theorem [4], it is not possible to achieve consistency, availability, and resilience to network partitioning at the same time. In the variant of this architecture that we studied (Amazon S3, Section 4), consistency was sacrificed and only a consistency level known as *eventual consistency* [29] was achieved. In database terms, eventual consistency achieves *durability* and it can be tweaked to achieve *atomicity*, but it does not comply with the *isolation* requirements of database transactions (i.e., serializability).

3.5 Caching

Figure 5 shows how caching can be integrated at the database server layer. Caching can be combined with any other architecture (partitioning, replication, and distributed control). Again, the principle is simple: The results of database queries are stored by dedicated cache servers. Typically, these servers keep the query results in their main memory so that accessing the cache is as fast as possible. Correspondingly, the set of caching servers is typically referred to as *MemCache*. Memcached [9] is the most widely used open-source software to support such distributed main-memory caches.

As for replication, there are many different schemes in order to keep the cache consistent with regard to updates to the database. Figure 5 depicts an approach in which the application controls cache consistency. This approach has been adopted by Google AppEngine which is the only studied cloud provider that operates a farm of dedicated MemCache servers. Unfortunately, Google has not published any details on its implementation of a MemCache.

Caching can also help the cloud computing promises with regard to cost and scalability. Cheap machines can be used for caching. Furthermore, adding and dropping *MemCache* machines is trivial at any point in time.

4. CLOUD SERVICES

This section describes the alternative services offered by three of the big players of cloud computing; namely Amazon (AWS), Google, and Microsoft. Since there are no standards yet, these services differ in many aspects: Business model, software components used at all tiers, and the programming model. An overview

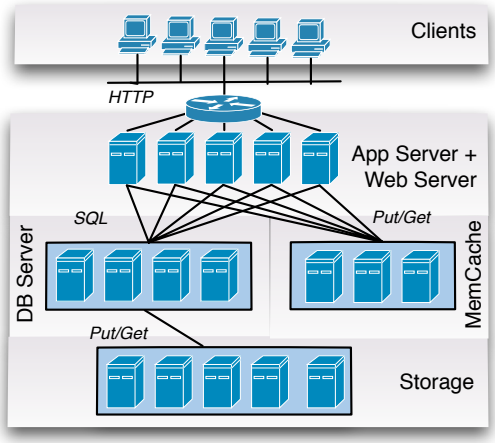

Figure 5: Caching

of the differences and key characteristics is given in Table 1. For the performance experiments presented in Section 6, the *Architecture* is most relevant; this line is, therefore, highlighted in Table 1. Another relevant category is the *HW configuration*. For several services, the user must configure how many virtual machines are used and on which kind of servers these virtual machines are deployed. Only Google AppEngine allocates fully automatically hardware resources for all tiers depending on the workload. SimpleDB and Azure automatically provision and adapt resources at the "DB server" and "Storage" layers, but require manual configuration of the HW resources at the web/app server layer. Amazon provides a service, called AutoScaling, that can be used in order to automatically scale-out and scale-down EC2 machines for the web/app tier. In order to better control the experiments and focus on the scalability at the database tier, however, we did not make use of this service in our experiments.

4.1 Amazon (AWS)

Amazon is a so-called *infrastructure as a service (IaaS)* provider. That is, Amazon provides a set of basic services to use computing infrastructure (CPU cycles, storage, and network), install a platform (e.g., a Tomcat web/app server and MySQL database server), and run an application on that platform. Amazon, therefore, provides a great basis to implement different architectural variants. The basic Amazon services used in the experiments reported in this paper are EC2 (for CPU cycles), EBS (for a storage service that can be mounted like a disk), and S3 (for storage that can be used as a key-value store). Since these services could easily be provided by other cloud providers, all architectural variants implemented on top of Amazon's infrastructure are declared as *flexible* in Table 1. In the last two years, Amazon has also provided richer services such as RDS and SimpleDB. These services are only available on the Amazon cloud. All the Amazon services are described in full detail in [1] which also lists the prices for using each service. The remainder of this section describes how we implemented five different architectural variants using the Amazon services (AWS).

4.1.1 AWS MySQL

The first variant studied follows the classic architecture of Figure 1. This variant can be seen as a baseline for all experiments because it follows a traditional (non cloud-enabled) model to deploy an enterprise web application. In our implementation, we used a varying number of EC2 machines in order to run the web/app

	AWS MySQL	AWS MySQL/R	AWS RDS	AWS SimpleDB	AWS S3	Google AppEng	MS Azure
Business Model	IaaS	IaaS	PaaS	PaaS	IaaS	PaaS	PaaS
Cloud Provider	Flexible	Flexible	Amazon	Amazon	Flexible	Google	Microsoft
Web/app server	Tomcat	Tomcat	Tomcat	Tomcat	Tomcat	AppEngine	.Net Azure
Database	MySQL	MySQL Rep	MySQL	SimpleDB	none	DataStore	SQL Azure
Storage / File Sys.	EBS	EC2 & EBS	-	-	S3	GFS	Windows Azure
Consistency	Repeatable Read	Repeatable Read	Repeatable Read	Eventual Consistency	Eventual Consistency	Snapshot Isolation	Snapshot Isolation
App-Language	Java	Java	Java	Java	Java	Java/AppEngine	C#
DB-Language	SQL	SQL	SQL	SimpleDB Queries	low-level API	GQL	SQL
Architecture	**Classic**	**Replication**	**Classic**	**Part.+Repl.**	**Distr. Contol**	**Part.+Repl.(+C)**	**Replication**
HW Config.	manual	manual	manual	manual/automatic	manual	automatic	manual/automatic

Table 1: Overview of Cloud Services

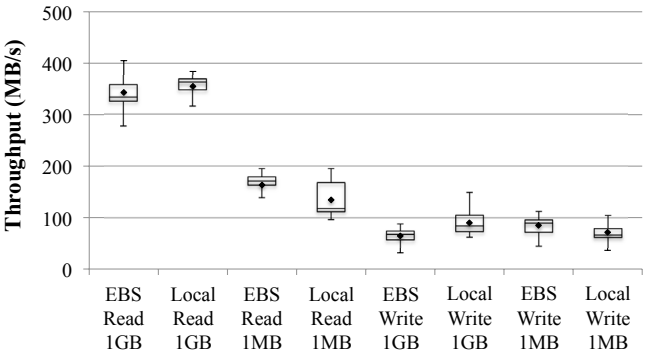

Figure 6: Read/Write Performance EBS

servers and execute the (TPC-W benchmark) application logic. We varied the number of EC2 machines depending on the workload. As a combined web and (Java) application server, we used Tomcat Version 6.0.18. For the database server tier, we used MySQL Version 5.0.51 with InnoDB, running on Ubuntu 8.04. The MySQL server was run on a separate EC2 machine. As a storage system, we used EBS for both the database and the logs. EBS guarantees persistence (the EBS data is replicated). In theory, the database could also be stored on the local disk of the EC2 machines that runs the MySQL server. However, in that approach all data is lost if the EC2 machine fails. That is why this option was not considered in the experiments. With regard to performance, Figure 6 compares the read and write performance of EBS with that of a locally attached disk. It can be seen that the read/write bandwidth of both options is in the same ballpark.

4.1.2 AWS MySQL/R

In order to study the Replication architecture (Figure 3), we used MySQL Replication Version 5.0.51 on a set of EC2 machines. In this variant we used the (cheaper) local disks of the EC2 machines for storing the database because the durability of the data was guaranteed by the Replication architecture. EBS was only used for the logs of the Master copy. These logs are needed in order to fail-over when the Master fails.

MySQL Replication uses the ROWA / Master copy protocol described in Section 3.3 in order to synchronize all update requests, whereas read requests from the applications can be processed by any satellite. The replication is not transparent. Consequently, each application server maintains a connection to the Master copy and connections to one satellite. Requests of updating transactions are handled by the Master, whereas requests of read-only transactions are issued to the satellite associated to the application server. As for *AWS MySQL*, Tomcat was used as an integrated web and application server and the number of EC2 machines for that tier was varied, depending on the workload.

We also experimented with MySQL Cluster Versions 5.0.51 and 7.0.8a. MySQL Cluster promises improved scale-out for a large number of servers [17]. Unfortunately, both of these versions of MySQL Cluster showed worse performance than the simple MySQL system (AWS MySQL) in our experiments so that we do not show the results in this paper. We have no explanation why we could not reproduce the results of [17] in the Amazon cloud.

4.1.3 AWS RDS

In late 2009, Amazon released the *relational database service*, RDS. In essence, RDS implements the same platform as provided by the AWS MySQL approach described above. Therefore, we expect both approaches to perform and cost similarly. The difference is that RDS is pre-packaged so that users do not need to worry about managing the deployment, software patches, software upgrades, and backups. RDS comes in five "sizes" ranging from small to quadruple extra large database servers. Obviously, a small server is sufficient for light workloads, whereas large servers are needed for heavy workloads with high query and update throughputs or complex queries. Accordingly, the prices for RDS vary from USD 0.11 per hour (small) to USD 3.11 per hour (quadruple extra large). This way, RDS implements scale-up on the database tier. Since RDS is based on a Classic architecture, however, RDS is not able to scale-out by adding database servers.

4.1.4 AWS SimpleDB

As mentioned at the beginning of this section, Amazon has also its own database service, called SimpleDB. SimpleDB provides a simple interface which allows to insert, update, and delete records. Furthermore, it allows to retrieve records based on their key values or based on ranges on primary and secondary keys. Details of the implementation of SimpleDB have not been published. From personal communication with Amazon engineers, we have learnt that the SimpleDB architecture can be best characterized as a combination of partitioning (Figure 2) and replication (Figure 3). Unlike MySQL Replication, however, it does not synchronize concurrent read and write accesses to different copies of the same data so that it only supports a low level of consistency called eventual consistency [29]. As of March 2010, SimpleDB also supports “consistent read” as a higher level of consistency. Unfortunately, this release came too late for consideration in our experiments.

At the application layer, the *AWS SimpleDB* variant is implemented using the same configuration as the other AWS variants; i.e., Tomcat with a varying number of EC2 machines. Since SimpleDB does not support SQL, SQL operators such as joins and aggregation had to be implemented at the application level. To do so, we implemented a (Java) library with these SQL operations and manually optimized SQL queries (i.e., join orders and methods). Obviously, this approach resulted in shipping all the relevant base data from SimpleDB to the application servers and resulted in poorer performance as the query shipping approach supported by full-fledged SQL database systems.

4.1.5 AWS S3

As a fourth architectural variant, we implemented the benchmark directly on top of S3. This variant corresponds to the Distributed Control architecture depicted in Figure 4. S3 only provides a low-level *put/get* interface so that any higher-level services such as SQL query processing and B-tree indexing had to be implemented as part of the application. We did that as part of a library that provided the basic database features for implementing the TPC-W benchmark. In order to improve performance, that library stored several tuples in a single S3 object. Furthermore, the library implemented a protocol in order to synchronize concurrent accesses from multiple application servers to the same S3 objects. For this purpose, we used the *basic protocol* proposed in [3]. This protocol can be implemented in a straightforward way on top of S3. As in the *AWS SimpleDB* variant, this protocol only supports eventual consistency. Higher levels of consistency can be implemented using the other protocols of [3] but were not considered as part of our study in order not to lose focus. To improve performance, caching of S3 objects was carried out in the application servers. In particular, S3 objects which represent B-tree index pages were aggressively cached across transactions in the application servers. The basic protocol described in [3] is also applicable to ensure correctness when data and index pages are cached.

[3] uses SQS, a persistent queue provided by Amazon, in order to implement the basic protocol for durability and eventual consistency. We did not follow that recommendation and implemented the basic protocol on our own implementation of queues. Our queues were deployed on a varying number of EC2 machines and EBS in order to store the logs of the queues persistently in the event of failures. We varied the number of EC2 machines for this purpose from one to five, depending on the workload. The TTL period for caching was set to 120 seconds and the checkpoint interval (defined in [3]) was set to 10 seconds. Checkpoints were effected by a watchdog [3].

The integrated web/app/DB server made use of Tomcat and the library to implement basic SQL constructs and consistency. We varied the number of EC2 machines, again depending on the workload.

4.2 Google

Unlike Amazon, Google follows a *platform as a service (PaaS)* only strategy. Google AppEngine is a service which enables to deploy whole applications without providing control over the computing resources. Google AppEngine automatically scales the resources consumed by an application out and down, depending on the workload. Google AppEngine supports Python and Java as programming languages, both with embedded SQL for accessing the database. We used the Java version of Google AppEngine with Google SDK 1.2.4 and Data Mappings with JPA. Unfortunately, Google only supports a simplified SQL dialect, referred to as GQL. Whenever GQL was not sufficient, we implemented the missing functionality in Java as part of a library in the same way as for the *AWS S3* and *AWS SimpleDB* variants. For instance, GQL does not support group by, aggregate functions, joins, or *LIKE* predicates. As for SimpleDB, Google has not published any details on its implementation of GQL and a distributed database system. According to [13], Google AppEngine has adopted a combined Partitioning and Replication architecture (Figures 2 and 3).

One nice feature of Google AppEngine is that it supports a MemCache. Thus, Google AppEngine provides a convenient interface that allows application programmers to put and retrieve objects into a MemCache. If the MemCache is used, Google AppEngine closely follows the architecture shown in Figure 5. We experimented with both variants (caching and non-caching).

4.3 Microsoft

Microsoft has recently launched Azure, a set of cloud services based on Windows, SQL Server, and .Net. To experiment with Azure, we implemented the TPC-W benchmark in C# with embedded SQL. In theory, other technology such as Java can also be deployed on the Azure cloud, but then the libraries for accessing the Azure database service and other Azure services are not available. Like Amazon and Google, Microsoft has not yet published full details on the implementation of Azure. As stated in [22], Azure adopted a Replication architecture (with Master-slave replication) as shown in Figure 3. Therefore, Azure should be directly comparable to the AWS MySQL/R variant.

All three cloud providers charge for storage, network traffic, and CPU hours. They also have similar rates for several categories (e.g., CPU hours). There are, however, also subtle differences. Azure, for instance, differs from Amazon and Google with regard to the pricing of the SQL Azure database service: Rather than paying as you go, Azure charges a monthly flat fee depending on the database size with unlimited database connectivity.

5. EXPERIMENTAL ENVIRONMENT

This section describes the benchmark and experimental environment used for the experiments. The results of the experiments are presented in the next section.

5.1 The TPC-W Benchmark

Since we were interested in the end-to-end performance of enterprise web applications that involve transaction processing, we used the TPC-W benchmark [28]. Other benchmarks for OLTP (such as TPC-C or TPC-E) emphasize the impact of the database system on the overall performance and do not involve any sophisticated application logic. Although the TPC organization has deprecated the TPC-W benchmark, it is still popular both in industry and academia.

The TPC-W benchmark models an online bookstore with a mix of fourteen different kinds of requests. These requests involve queries such as searching for products, displaying products and update functions such as the placement of an order. Furthermore, the TPC-W benchmark specifies three kinds of workload mixes: (a) browsing, (b) shopping, and (c) ordering. A workload mix specifies the probability for each kind of request. In all the experiments reported in this paper, the ordering mix was used because it is the most update-intensive mix: About one third of the requests involve an update of the database in the ordering mix. Finally, the TPC-W benchmark allows to study different workloads with regard to the request throughput. To this end, the TPC-W benchmark models *emulated browsers (EBs)*. Each EB simulates one user who issues a request (according to the probabilities of the workload mix), waits for the answer, and then issues the next request after a specified waiting time. We varied the EB parameter from 1 ($\approx$ 500 requests per hour) to 9000 ($\approx$ 1250 requests per second). In the ordering mix a TPC-W request involves 6.6 HTTP requests on an average because TPC-W web pages contain several embedded components (e.g., images).

The TPC-W benchmark has two metrics. The first metric is a throughput metric and measures the number of valid TPC-W requests per second. This metric is abbreviated as WIPS and this paper follows this notation. It is important that a request is only valid and, thus, counted if it meets the response time requirements. Depending on the kind of request, the allowed response time varies

from 3 seconds to 20 seconds. Benchmark results may only be reported if 90% of all requests of every category are valid and return the complete answer within the allowed response time. It is illegal, for instance, to drop all updates and achieve high WIPS with reads only.

The second metric defined by the TPC-W benchmark is Cost/WIPS. This metric relates the performance (i.e., WIPS) to the total cost of ownership of a computer system. To this end, the TPC-W benchmark gives exact rules on how to compute the cost of a system. These rules had to be relaxed for this study because they were not applicable to any of the cloud services studied. In all our experiments, cost was computed by considering the bills we had to pay.

In addition to the computation of the Cost/WIPS metric, we made the following adjustments to the TPC-W benchmark:

- Benchmark Database: The TPC-W benchmark specifies that the size of the benchmark database grows linearly with the number of EBs. Since we were specifically interested in the scalability of the services with regard to the transactional workload and elasticity with changing workloads, we carried out all experiments with a fixed benchmark database which complies to a standard TPC-W database for 100 EBs. This database involved 10,000 items and had 315 MB of raw data which typically resulted in database sizes of about 1GB with indexes (Section 6.5).

- Consistency: The TPC-W benchmark requires strong consistency with ACID transactions. As shown in Table 1, several variants do not support this level of consistency.

- Bestseller Query: One kind of TPC-W request, the so-called bestseller query, could not be implemented using S3, SimpleDB, and Google AE. Implementing this query at the application level would have been prohibitive because almost the whole database needs to be scanned for this query. To avoid that all results are biased by this kind of request, we replaced this query with a query that randomly returns a set of products and that could be implemented efficiently on all platforms.

- HTTP: The TPC-W benchmark requires the use of the HTTPS protocol for secure client / server communication. As a simplification, we used the HTTP protocol (no encryption).

5.2 Methodology, Metrics, and Implementation

The goal of this performance evaluation was to study the scalability (with regard to throughput) and cost of alternative cloud service offerings under different workloads. To this end, we implemented and ran the TPC-W benchmark on the alternative services listed in Section 4 and measured WIPS and cost, thereby varying the EBs (i.e., number of simulated concurrent users). As mentioned in the previous section, we varied the load from 1 EB (light workload) to 9000 EBs (heavy workload). We did not evaluate the other promises of cloud computing such as availability, time-to-market, or flexibility because these metrics are difficult to measure. In summary, the following metrics were measured:

- WIPS(EB): The throughput of *valid requests per second* depending on the number of emulated browsers (EBs). The higher, the better. (Valid means "meeting the response time goal" as explained in the previous subsection.)

- Cost/WIPS(EB): The cost per WIPS, again depending on the number of EBs. The lower, the better.

- CostPerDay(EB): The (projected) total cost of running the benchmark with a certain number of EBs for 24 hours. The lower, the better.

- s(Cost/WIPS): The standard deviation of the Cost/WIPS for a set of different EB settings (from EB=1 to EB=max where max is the EB value for which the highest throughput could be achieved). This metric is a measure for the predictability of cost of a service provider. Ideally, the Cost/WIPS does not depend on the load and is therefore predictable. Therefore, the lower s, the better.

In addition to these metrics, we measured the time and cost to bulk-load the benchmark database as well as the size and monthly cost to store the benchmark database.

Depending on the variant, we had two different experimental setups in order to determine the cost and WIPS for each EB setting. For the SimpleDB, S3, Google AppEngine (w/o caching) variants, we measured the WIPS for a number of EB settings (EB=1, 250, 500, 1000, 2000, 3000, 4000, and 9000, if possible) during a period of 10 minutes. For these variants, we were not able to measure the whole spectrum of EB settings for budget reasons. For the three MySQL variants (MySQL, MySQL/R, and RDS) and Azure we measured all possible EB settings in the range of EB=1 to EB=9000. This was done by starting with EB=1 and increasing the workload by one EB every 0.4 seconds. In all cases, we did a warm-up run of two minutes before each experimental run; the cost and throughput of this two minute warm-up phase were factored out in the results presented in this paper.

We did a number of additional experiments and measures in order to guarantee the stability of the results. For MySQL, MySQL/R, RDS, and Azure, all experiments were repeated seven times and the average WIPS and cost of these seven runs are reported in this paper. For the SimpleDB, S3, Google AE, and Google AE/C variants, all experiments were repeated only three times, again, because of budget constraints during this project. Furthermore, we ran several data points for longer periods of time (up to thirty minutes) with a fixed EB setting in order to see whether the providers would adjust their configuration to the workload. However, we could not detect any such effects. Only for Microsoft Azure, we observed a small discontinuity. In our first experiments with Azure, Azure became shortly unavailable for EB=2000 and EB=5500. We believe that at these points, Azure migrated the TPC-W database to bigger machines so that the increased workloads could be sustained. This effect happened only for the very first experiment with Azure. It seems that Azure does not migrate databases back to less powerful machines when the workload decreases so that all subsequent experiments on Azure were carried out on the (presumably) big database machine. Overall, however, the results were surprisingly stable and we had only one outlier in one of the MySQL experiments. It is well known that the quality of service of cloud computing providers varies, but a long term, detailed study on these variances was beyond the scope of this work.

The experiments with AWS RDS and Microsoft Azure were carried out in February 2010. The experiments with all other variants were carried out in October 2009. (In October 2009, Azure and RDS were not yet generally available.) Obviously, all providers will make many changes that affect the results, just as hardware and technology trends affect the results of any other performance study. Nevertheless, we believe that the results of this study reveal important insights into the properties of alternative architectures for cloud computing platforms (Section 3).

In all experiments, the emulated browsers (EBs) of the TPC-W benchmark were run on EC2 machines in the Amazon cloud. For

the experiments with Google AppEngine and Azure, therefore, the client machines were located naturally in different data centers than the server machines that handled the requests. For fairness, we made sure that EC2 client machines and EC2 server machines were located in different data centers for all Amazon variants. This was done by explicitly choosing a data center when starting the EC2 instances for clients and servers.

We used *medium HIGH-CPU* EC2 machines to run the web/app servers in the Amazon cloud. Accordingly, we used *medium* machines to run the web/app servers in the Azure cloud. The medium EC2 and Azure machines have roughly the same performance and cost so that the results are directly comparable. Furthermore, we adjusted the number of machines that ran the web/app servers manually in the Amazon and Azure clouds. With the help of separate experiments (not reported here), we discovered that one medium server was able to sustain the load of 1500 EBs. Correspondingly, we provisioned one web/app server per 1500 EBs, up to six machines for the maximum workload of 9000 EBs. In the S3 variant, an integrated web/app/DB server was only able to sustain 900 EBs so that up to ten EC2 machines were used for this variant. We pre-allocated EC2 and Azure machines in order to make sure that they were available when needed (as the load increased). In the experiments with Google AppEngine, we had no influence on the choice and number of machines used for the web/app layer because the servers were automatically provisioned by Google AppEngine (Table 1).

We also used *medium* EC2 machines in order to run database servers for AWS MySQL and AWS MySQL/R. If not stated otherwise, we used a *large* machine for AWS RDS because the large RDS machine has similar performance characteristics as a medium EC2 machine. For all other variants, the database machines could not be configured.

As mentioned in Section 2, we did not carry out any "peak" experiments, as proposed in [2]. The purpose of "peak" experiments is to evaluate how quickly a provider adapts to rapid changes in the workload. As mentioned above, we could not observe any significant adjustments by the providers (with the noteworthy exception of Azure in the first experimental run) so that we believe that our results represent the steady-state behavior of the systems. A more detailed analysis of "peak" performance and adaptability, however, is an important avenue for future research.

In all experiments, the images used as part of the TPC-W benchmark (e.g., pictures of products) were stored in a separate file system and not inside the database. In the Amazon cloud, all images were stored on the local EC2 filesystem to save cost. In Azure, the images were stored as part of the web project.

6. EXPERIMENTS AND RESULTS

This section presents the performance (WIPS) and cost ($) results of running the TPC-W benchmark on the eight cloud service variants described in Section 4. In addition, this section shows the running times and costs of bulkloading the benchmark database for each variant.

6.1 Big Picture

Table 2 summarizes the overall results of this study. More detailed analyses are given in the subsequent subsections. The first column gives the maximum throughput (WIPS) that could be achieved for each variant. The second and third columns list the Cost per WIPS for low workloads (EB=1, second column) and high workloads (EB=max, third column). The fourth column gives the mean and variance of the cost for the whole range of workloads, from EB=1 to EB=max. Here, EB=max refers to the maximum number of EBs that a variant could sustain; i.e., the EBs at which the maximum throughput was achieved. In all columns, the winner is high-lighted in bold and italics.

	Throughput (WIPS)	Cost/WIPS (m$) Low TP	Cost/WIPS (m$) High TP	Cost Predictability (mean $\pm s$)
MySQL	477	0.635	***0.005***	0.015 ± 0.077
MySQL/R	454	2.334	***0.005***	0.043 ± 0.284
RDS	462	1.212	***0.005***	0.030 ± 0.154
SimpleDB	128	0.384	0.037	0.063 ± 0.089
S3	***>1100***	1.304	0.009	0.018 ± 0.098
Google AE	39	0.002	0.042	0.029 ± 0.016
Google AE/C	49	***0.002***	0.028	***0.021 ± 0.011***
Azure	***>1100***	0.775	***0.005***	0.010 ± 0.058

Table 2: Throughput, Cost/WIPS, Cost Predictability

Looking at the first column (for throughput), it becomes clear that only S3 and Azure are able to sustain high workloads of 9000 EBs. For all other variants, the database server becomes a bottleneck with a growing number of EBs. We believe that the S3 variant with a *Distributed Control* architecture is able to scale even beyond 9000 EBs. This architecture is the only architecture which has no potential bottleneck. Since it is based on a "Replication" architecture (Section 3.3), Azure reaches its limits as soon as the Master database server is overloaded. It seems, however, that Microsoft makes use of high-end machines for the SQL Azure database layer so that this limit was not reached for 9000 EBs.

Turning to the "Cost/WIPS" results, it can be observed that all services, except Google AE, have lower Cost/WIPS at high workloads than at low workloads. Ideally, the "Cost/WIPS" should be constant and should not depend on the workload. If the "Cost/WIPS" is higher for a low workload than for a high workload, then the service has fixed costs that need to be paid for, independent of the usage of the service. For instance, all Amazon variants need to pay for at least one EC2 instance in order to be able to respond to client requests, even if there is no load at all. In addition, some Amazon instances must pay for machines at the database layer. Likewise, a monthly flat fee for SQL Azure and at least one machine for a web/app server must be paid in order to keep a web application online in the Azure cloud. Google AppEngine is the only variant that does not have any fixed cost and is free if there is no load. Obviously, such fixed costs are not compliant with the *pay-as-you-go* paradigm promised by cloud computing.

The fourth column of Table 2 shows the mean and standard deviation of the Cost/WIPS metric over the whole range of EBs that a service could sustain. With the exception of Google, all services have a high variance which means that the cost for the service is highly dependent on the load and, thus, becomes unpredictable (unless the system has a constant load).

6.2 Scale-out

Figure 7 shows the WIPS achieved by each variant as a function of EB. For readability, Figure 7 does not show the results for Google AppEngine without caching and MySQL/R. Google AppEngine without caching showed almost the same performance as Google AppEngine with caching; a little worse in all cases, but the differences were marginal and the shape of the curve is almost identical. MySQL/R showed almost the same performance as MySQL, again, slightly worse in all cases, but the shape of the curve was almost the same. For update-intensive workloads such as the ordering mix of the TPC-W benchmark, the Master becomes a bottleneck and scaling out read-only transactions to satellites does not result in any throughput gains.

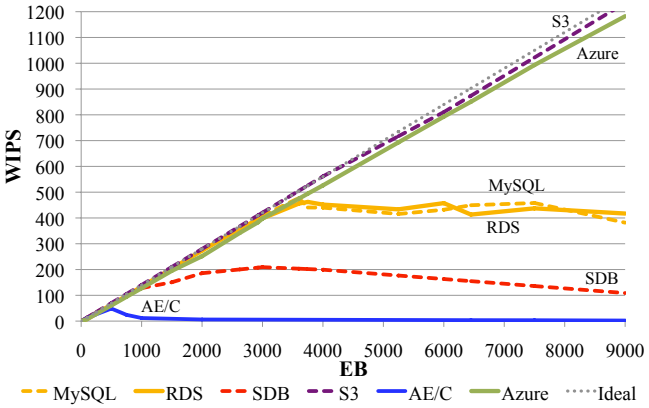

Figure 7: Comparison of Architectures [WIPS]

Figure 7 shows that the only two variants that scale are S3 and Azure. As mentioned in the previous sub-section, S3 is the only variant that is based on an architecture that has no bottlenecks. Azure scales well for using powerful machines to run the database servers. Both of these variants scale almost ideally up to 9000 EBs and achieve the maximum throughput at 9000 EBs. For reference, the *ideal* throughput of a perfect system is shown as a dotted line in Figure 7. All other variants have scalability limits and cannot sustain the load after a certain number of EBs. The baseline variants, MySQL and RDS, reach their limits at about 3500 EBs; SimpleDB at about 3000 EBs; and Google AE/C at a few hundred EBs.

The behavior of the alternative variants in overload situations is surprisingly different. Figures 8 to 10 depict the throughput behavior of AWS RDS, SimpleDB, and Google AppEngine in more detail. These figures show the *ideal* (dotted) and *WIPS* (green solid) lines (as in Figure 7) and, in addition, the number of *issued requests* that were submitted (yellow line). Recall from Section 5 that the TPC-W benchmark specifies that a client (i.e., emulated browser) waits for a response before issuing the next request. Thus, if a system cannot sustain the load and does not produce responses anymore, then the number of issued requests is lower than in an ideal system. Obviously, the WIPS line must be below the issued requests line. Figures 8 to 10 show the following effects in overload situations for the different variants:

- RDS (Figure 8): The throughput of RDS plateaus after 3500 EBs and stays constant. That is, all requests return answers, but a growing percentage of requests are not answered within the response time constraints specified by the TPC-W benchmark. Recall that the MySQL variant is technically the same as RDS and therefore has the same behavior (not shown for brevity).
- SimpleDB (Figure 9): Figure 7 shows that the WIPS grow up to about 3000 EBs and more than 200 WIPS. In fact, SimpleDB was already overloaded at about 1000 EBs and 128 WIPS in our experiments. At this point, all write requests to *hot spots* failed. In the TPC-W benchmark, *item* objects are frequently updated and these update requests were dropped by SimpleDB. According to the rules of the TPC-W benchmark, dropping more than 10% of the requests of any category is illegal (Section 5). As a result, Table 2 reports on a peak throughput of 128 WIPS for SimpleDB. In an overload situation, SimpleDB simply drops requests and returns errors. As failure is immediate, the issued requests grow linearly with the number of EBs.
- Google AE/C (Figure 10): Like SimpleDB, Google AE (with and without caching) drops requests in overload situations. This effect can be observed by a linearly growing *issued requests* curve in Figure 10. Unlike SimpleDB, Google AE is fair and drops requests from all categories. That is, both read and write requests of all kinds are dropped in overload situations. In these scale-out experiments, Google AppEngine performs worst among all variants. This phenomena can be explained by Google's focus on supporting low-end workloads for the lowest possible price (see next section).

	1	10	100	500	1000
MySQL	0.635	0.072	***0.020***	***0.006***	***0.006***
MySQL/R	2.334	0.238	0.034	0.008	***0.006***
RDS	1.211	0.126	0.032	0.008	***0.006***
SimpleDB	0.384	0.073	0.042	0.039	0.037
S3	1.304	0.206	0.042	0.019	0.011
Google AE	0.002	0.028	0.033	0.042	0.176
Google AE/C	***0.002***	***0.018***	0.026	0.028	0.134
Azure	0.775	0.084	0.023	***0.006***	***0.006***

Table 3: Cost per WIPS [m$], Vary EB

6.3 Cost

6.3.1 Cost/WIPS

Table 3 details the Cost/WIPS for the alternative variants with varying EBs. As discussed in Section 6.1, Google AE is cheapest for low workloads (below 100 EBs) whereas Azure is cheapest for medium to large workloads (more than 100 EBs). The three MySQL variants (MySQL, MySQL/R, and RDS) have (almost) the same cost as Azure for medium workloads (EB=100 and EB=3000), but they are not able to sustain large workloads (Section 6.2).

The success of Google AE for small loads has two reasons. First, Google AE is the only variant that has no fixed costs. There is only a negligible monthly fee to store the database (see Table 6). Second, at the time these experiments were carried out, Google gave a quota of six CPU hours per day for free. That is, applications which are below or slightly above this daily quota are particularly cheap.

Azure and the MySQL variants win for medium and large workloads because all these approaches can amortize their fixed cost for these workloads. Azure SQL server has a fixed cost per month of USD 100 for a database of up to 10 GB, independent of the number of requests that need to be processed by the database. For MySQL and MySQL/R, EC2 instances must be rented in order to keep the database online. Likewise, RDS involves an hourly fixed fee so that the cost per WIPS decreases in a load situation. It should be noted that network traffic is cheaper with Google than with both Amazon and Microsoft.

This cost analysis is more an artifact of the business models used by Amazon, Google, and Microsoft. Obviously, all providers can change their pricing any time, and Amazon has frequently done so in the past. The cost analysis, however, indicates for which kind of workload a service is optimized for: Google is obviously targeting the low end market whereas Microsoft seems to be focussing more on enterprise customers. Furthermore, we believe that cost is indeed a good indicator for the efficiency of an implementation.

6.3.2 Cost per Day

Table 4 shows the total cost per day for the alternative approaches and a varying load (EBs). (A "-" indicates that the variant was not able to sustain the load.) These results confirm the observations made in the previous subsection: Google wins for small workloads;

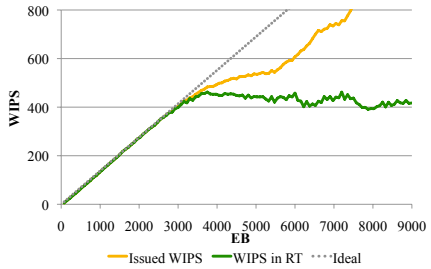

Figure 8: RDS: WIPS(EB)

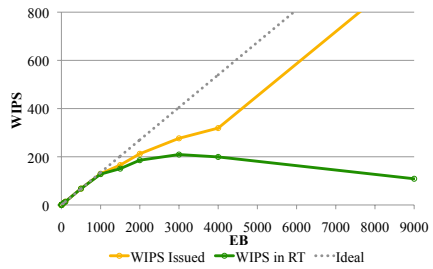

Figure 9: SimpleDB: WIPS(EB)

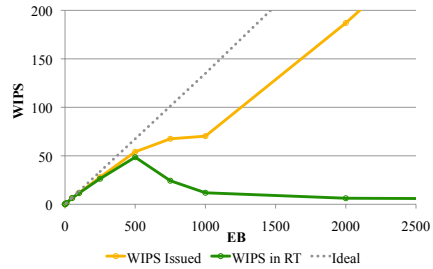

Figure 10: AE/C: WIPS(EB)

	1	250	1000	3000	9000
MySQL	8.25	***22.85***	64.48	183.66	-
MySQL/R	16.40	31.01	72.90	194.44	-
RDS	14.75	29.47	70.92	187.88	-
SimpleDB	4.42	116.52	412.25	-	-
S3	16.02	71.96	131.78	346.60	914.87
Google AE	***0.02***	82.25	-	-	-
Google AE/C	***0.02***	60.57	-	-	-
Azure	9.15	22.86	***61.22***	***176.87***	***521.13***

Table 4: Total Cost per Day [$], Vary EB

Azure wins for medium and large workloads. All the other variants are somewhere in between. The three MySQL variants come close to Azure in the range of workloads that they sustain. Again, Azure and the three MySQL variants roughly share the same architectural principles ("Classic" and "Replication" with Master copy architectures). SimpleDB is an outlier in this experiment. With the current pricing scheme, SimpleDB is an exceptionally expensive service. For a large number of EBs, the high cost of SimpleDB is particularly annoying because users must pay even though SimpleDB drops many requests and is not able to sustain the workload.

Turning to the S3 cost in Table 4, the total cost grows linearly with the workload. This behavior is exactly what one would expect from a pay-as-you-go model. For S3, the high cost is matched by high throughputs so that the high cost for S3 at high workloads is tolerable. This observation is in line with a good Cost/WIPS metric for S3 and high workloads (Table 3). Nevertheless, S3 is indeed more expensive than all the other approaches (except SimpleDB) for most workloads. This phenomenon can be explained by Amazon's pricing model for EBS and S3. For instance, a write operation to S3 is hundred times more expensive than a write operation to EBS which is used in the MySQL variants. Amazon can justify this difference because S3 supports concurrent updates with an eventual consistency policy whereas EBS only supports a single writer (and reader) at a time.

6.3.3 Cost Analysis

Figure 11 shows the cost per day spent on network traffic, CPUs, and storage. The network traffic is depicted in red. Network traffic costs are purely variable costs which entirely depend on the workload. The CPU cost has a variable and a fixed component. In Figure 11, the fixed CPU cost is shown in orange, and the variable CPU cost is shown in yellow. Fixed CPU costs are required to reserve machines. Variable CPU costs are incurred by surcharges of actual usage. For instance, the cost of an EC2 instance depends on the usage. Storage costs also have a fixed and variable component. The fixed storage costs are the monthly costs for storing the database. The variable storage costs are the costs per request to fetch and put data into the database. As will be shown in Section 6.5, the fixed storage costs were neglible for our experiments so that Figure 11 shows the aggregated storage cost (fixed + variable) as a single

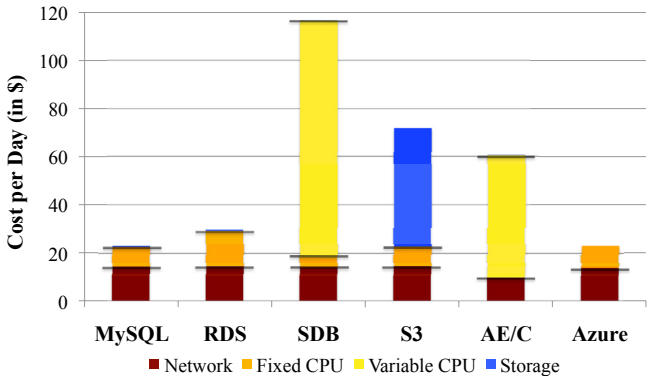

Figure 11: Cost Factors, 250 EBs [$]

storage cost metric in blue. Figure 11 shows this cost breakdown by resource (network, CPU, and storage) for each variant for EB=250. Again, the results for MySQL/R and Google AE without caching are omitted for brevity (they are similar to MySQL and Google AE/C, respectively).

Figure 11 shows that the total cost for the MySQL variant is dominated by network costs for EB=250. Another significant factor is the (fixed) cost for the reservation of EC2 instances for the web/app and database servers. All other cost factors are negligible. For SimpleDB, the total cost is dominated by the (variable) cost for the compute hours spent in order to query and update SimpleDB. As mentioned in the previous subsection, SimpleDB is an expensive service. For the SimpleDB variant, all other cost factors are in the same order as for the other variants. The S3 variant has a significant cost factor for storage because the cost of retrieving and putting data to S3 is significant compared to the cost of retrieving and putting data to EBS (as mentioned in the previous subsection). All other cost factors of the S3 variant are comparable to the cost factors of the other variants implemented on the Amazon cloud.

Looking at the Google cloud, it can be seen that Google AE is the only variant that has no fixed costs for reserving CPUs. All the CPU cost is variable, but even at a moderate workload of 250 EBs, it can be considerable. Nevertheless, the absence of any fixed cost (there is only a negligible fixed cost factor for storing the database in the Google cloud), makes Google so attractive for small applications.

Again, Azure seems to be targeting a complementary market segment as compared to Google. Microsoft charges a fixed price to reserve machines for the application and database servers. However, once this fee has been paid, there are no variable costs for actually using these machines.

Figures 12 to 17 confirm these results. These figures visualize the percentage of each cost factor in each variant, depending on the workload. Figures 12 and 13 for MySQL and RDS are mostly red, as network cost dominates the overall cost for these two variant almost independent of the workload. Only for small workloads (EB

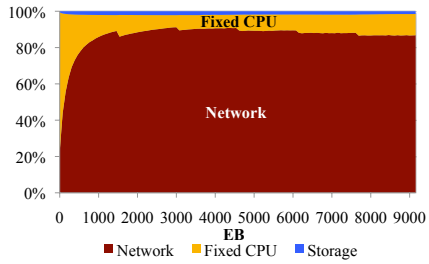

Figure 12: MySQL: Cost Fact., Vary EB

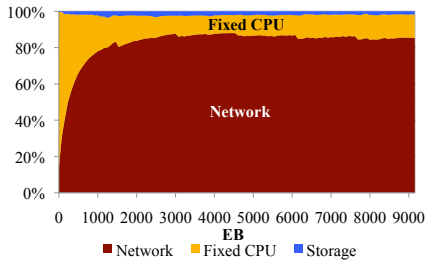

Figure 13: RDS: Cost Fact., Vary EB

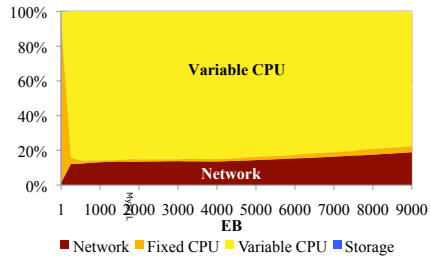

Figure 14: SimpleDB: Cost Fact., Vary EB

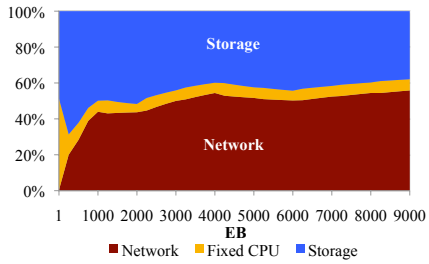

Figure 15: S3: Cost Fact., Vary EB

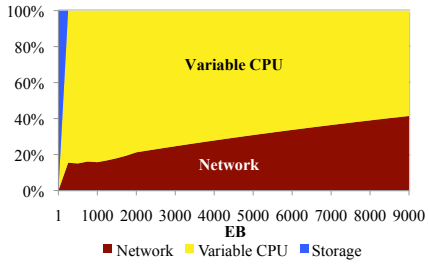

Figure 16: AE/C: Cost Fact., Vary EB

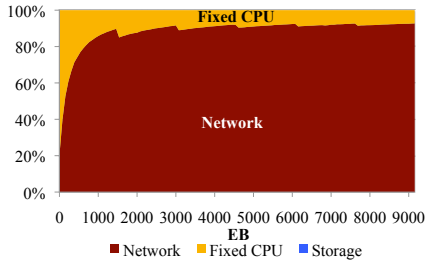

Figure 17: Azure: Cost Fact., Vary EB

< 100), the cost for reserving machines is significant (depicted orange). An interesting effect is the zig-zag in both figures: Every step (at a spacing of 1500 EBs) indicates that an additional EC2 instance was added at the web/app layer in order to sustain the additional workload.

Figure 14 confirms that the cost of the SimpleDB variant is dominated by the cost of querying and updating the SimpleDB service. Correspondingly, Figure 15 shows that for high workloads (EB $>$ 1000), the cost of the Distributed Control architecture implemented on top of S3 is dominated by the usage of the AWS S3 service. Only for small workloads, the cost is dominated by the fixed cost of reserving EC2 machines and network traffic.

Figure 16 confirms that the dominant cost for Google AE is the (variable) CPU cost. Only for very small workloads (EB $<$ 10), the cost of Google AE/C is dominated by storage costs (blue), but for these workloads the overall cost is negligible (in the order of a few cents). In contrast, Figure 17 is mostly red, confirming that the cost of Azure is dominated by network traffic - just as for the MySQL variants. Figure 17 also has the same zig zag as Figures 12 and 13 for adding a new web/app server for each 1500 EBs.

Again, these cost results are an artifact of the pricing models of the providers. The results are likely to change soon as the competition between the three cloud providers tightens. Nevertheless, it is important to understand the cost factors of a system. In the past, for instance, the (variable) cost for CPU time has dropped faster than the cost for network traffic.

6.4 Scale-up

One of the goals of cloud computing is to scale-*out*. That is, provide increased throughput with a growing number of machines. Nevertheless, some cloud providers also support *scale-up* options. For the "Classic", "Partitioning", and "Replication" architectures described in Section 3, such scale-up options are important because the database server can become the bottleneck and scaling-up the database server is the only way to achieve higher throughput. Concretely, Amazon RDS provides a scale-up option. Table 5 shows the overall results in terms of throughput, cost/WIPS, and cost predictability. It can be seen that a larger database machine is able to sustain a higher workload, as expected. However, even the biggest RDS machine is not able to sustain 9000 EBs and a throughput of more than 1000 WIPS. Obviously, a bigger machine increases the fixed cost of the system (poor cost/WIPS for low workloads) and consequently, decreases the cost predictability (i.e., increases the variance).

	Throughput (WIPS)	Cost/WIPS (m$)		Cost Predictability (mean $\pm s$)
		Low TP	High TP	
RDS Small	171	***0.564***	***0.005***	***0.033 ± 0.119***
RDS Large	462	1.212	***0.005***	0.030 ± 0.154
RDS 4 XLarge	***767***	6.431	0.006	0.071 ± 0.613

Table 5: RDS Scale-Up: Throughput, Cost/WIPS, Cost Predictability

6.5 Bulkloading

For completeness, Table 6 shows the bulkloading times and costs as well as the database size and monthly storage costs for the alternative variants. Again, the winners are shown in bold and italics. In all cases, the best possible offering for bulkloading the database was used. For MySQL, the data was inserted using a JDBC connection. For SimpleDB, we used the batch-put operation which is able to store 25 records at once. S3 has no specific bulkloading support so that the data was loaded using the standard protocol of [3]. For Google AE, we made use of a Python script to enable Google's batch import for the DataStore. For Azure, the SQL Azure Migration Wizard was used. In all, MySQL was the clear winner in this category. MySQL/R had exactly three times the time and cost of MySQL because we had a replication factor of three for MySQL/R.

Turning to the size of the database and storage cost per month (the last two columns of Table 6), it can be seen that Azure is the winner. Overall, however, the storage costs are negligable and are not an important factor for the overall cost (Section 6.3). It should be noted that for all variants, except MySQL, the monthly storage cost grows linearly with the size of the database. The MySQL variant is based on EBS for storing the database (Section 4.1.1) and the cost function of EBS is more complex. In a prudent setup, EBS resources are overprovisioned because if an EBS device is full, then a new EBS device with larger capacity must be provisioned and the data must be copied from the old to the new EBS

	BL Time (h:mm)	BL Cost ($)	DB Size (GB)	Storage Cost / Month ($)
MySQL	0:10	***0.17***	0.81	> 0.1
MySQL/R	0:30	0.51	2.43	***0.1***
RDS	***0:07***	0.46	-	***0.1***
SimpleDB	1:50	5.16	1.32	0.33
S3	1:33	0.44	0.99	0.15
Goolge AE	30:10	11.69	4.28	0.64
Google AE/C	30:10	11.69	4.28	0.64
Azure	0:23	0.36	***0.38***	0.0

Table 6: Bulkloading Time and Cost; Database Size and Cost

device. Obviously, copying data is expensive so that the storage must be overprovisioned in order to avoid these incidents.

7. CONCLUSION

This paper presented the results of a first study of the end-to-end performance and cost of running enterprise web applications with OLTP workloads on alternative cloud services. Since the market is still immature, the alternative services varied greatly both in cost and performance. Most services had significant scalability issues. An interesting observation was to see how the alternative services behave in overload situations. With regard to cost, it became clear that the alternative providers have different business models and target different kinds of applications: Google seems to be more interested in small applications with light workloads whereas Azure is currently the most affordable service for medium to large services. Public clouds are often criticized for a lack of support to upload large data volumes. This observation could be confirmed. It is still difficult to upload, say, 1 TB or more of raw data through the APIs provided by the providers.

The more fundamental question of what is the right data management architecture for cloud computing could not be answered. It is still unclear whether the observed results are an artifact of the level of maturity of the studied services or fundamental to the chosen architecture. We hope that this work has pathed the way to a continuous monitoring of progress on alternative approaches and products for data management in the cloud.

8. REFERENCES

[1] Amazon. Amazon WebServices. http://aws.amazon.com/, October 2009.

[2] C. Binnig, D. Kossmann, T. Kraska, and S. Loesing. How is the Weather Tomorrow? Towards a Benchmark for the Cloud. In *Proc. of DBTest*, pages 1–6, 2009.

[3] M. Brantner, D. Florescu, D. A. Graf, D. Kossmann, and T. Kraska. Building a Database on S3. In *Proc. of SIGMOD*, pages 251–264, 2008.

[4] E. A. Brewer. (Invited Talk) Towards Robust Distributed Systems. In *Proc. of PODC*, page 7, 2000.

[5] R. Buck-Emden. *The SAP R/3 System*. Addison-Wesley, 2nd edition, 1999.

[6] M. J. Carey, D. J. DeWitt, and J. F. Naughton. The 007 Benchmark. In *Proc. of SIGMOD*, pages 12–21, 1993.

[7] M. J. Carey, D. J. DeWitt, J. F. Naughton, M. Asgarian, P. Brown, J. Gehrke, and D. Shah. The BUCKY Object-Relational Benchmark (Experience Paper). In *Proc. of SIGMOD*, pages 135–146, 1997.

[8] S. Ceri and G. Pelagatti. *Distributed databases principles and systems*. McGraw-Hill, Inc., 1984.

[9] Danga. MemCached. http://www.danga.com/memcached/, October 2009.

[10] D. J. DeWitt. The Wisconsin Benchmark: Past, Present, and Future. In J. Gray, editor, *The Benchmark Handbook for Database and Transaction Systems* , 2nd edition. Morgan Kaufmann, 1993.

[11] D. J. DeWitt, P. Futtersack, D. Maier, and F. Vélez. A Study of Three Alternative Workstation-Server Architectures for Object Oriented Database Systems. In *Proc. of VLDB*, pages 107–121, 1990.

[12] D. Florescu and D. Kossmann. Rethinking Cost and Performance of Database Systems. *SIGMOD Rec.*, 38(1):43–48, 2009.

[13] J. Furman, J. Karlsson, J. Leon, A. Lloyd, S. Newman, and P. Zeyliger. Megastore: A Scalable Data System for User Facing Applications. In *Proc. of SIGMOD*, 2008.

[14] Gartner. Gartner Top Ten Disruptive Technologies for 2008 to 2012. Emerging Trends and Technologies Roadshow, 2008.

[15] T. Kraska, M. Hentschel, G. Alonso, and D. Kossmann. Consistency Rationing in the Cloud: Pay Only when it Matters. In *Proc. of VLDB*, volume 2, pages 253–264, 2009.

[16] U. F. Minhas, J. Yadav, A. Aboulnaga, and K. Salem. Database Systems on Virtual Machines: How Much Do You Lose? In *ICDE Workshops*, pages 35–41, 2008.

[17] MySQL-AB. Benchmarking Highly Scalable MySQL Clusters. http://www.mysql.com/why-mysql/white-papers/mysql_cge_benchmarks_wp_april2007.php, October 2009.

[18] Oracle. Oracle Real Application Clusters. http://www.oracle.com/technology/products/database/clustering/, October 2009.

[19] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker. A Comparison of Approaches to Large-Scale Data Analysis. In *Proc. of SIGMOD*, pages 165–178, 2009.

[20] C. Plattner and G. Alonso. Ganymed: Scalable Replication for Transactional Web Applications. In *Proc. of Middleware*, pages 155–174, 2004.

[21] A. Schmidt, F. Waas, M. L. Kersten, M. J. Carey, I. Manolescu, and R. Busse. XMark: A Benchmark for XML Data Management. In *Proc. of VLDB*, pages 974–985, 2002.

[22] S. Sengupta. SQL Data Services: A Lap Around. In *Microsoft Professional Developers Conference (PDC)*, 2008.

[23] W. Sobel, S. Subramanyam, A. Sucharitakul, J. Nguyen, H. Wong, A. Klepchukov, S. Patil, A. Fox, and D. Patterson. Cloudstone: Multi-Platform, Multi-Language Benchmark and Measurement Tools for Web 2.0. In *Proc. of CAA*, 2008.

[24] M. Stonebraker. The Case for Shared Nothing. *IEEE Database Eng. Bull.*, 9(1):4–9, 1986.

[25] M. Stonebraker, J. Frew, K. Gardels, and J. Meredith. The Sequoia 2000 Benchmark. In *Proc. of SIGMOD*, pages 2–11, 1993.

[26] M. Stonebraker and J. M. Hellerstein, editors. *Readings in Database Systems*. Morgan Kaufmann, 4th edition, 2005.

[27] A. Tanenbaum and M. van Steen. *Distributed Systems: Principles and Paradigms*. Prentice Hall, 2002.

[28] TPC. TPC-W 1.8. TPC Council, 2002.

[29] W. Vogels. Eventually Consistent. *Commun. ACM*, 52(1):40–44, 2009.