A Runtime Quality Measurement Framework for Cloud Database Service Systems

Markus Klems, David Bermbach
Institute of Applied Informatics (AIFB)
Karlsruhe Institute of Technology
Karlsruhe, Germany
first.last@kit.edu

René Weinert
Institute of Applied Informatics (AIFB)
Karlsruhe Institute of Technology
Karlsruhe, Germany
rene.weinert@student.kit.edu

***Abstract*—Cloud database services promise high performance, high availability, and elastic scalability. The system that provides cloud database services must, hence, be designed and managed in a way to achieve these high quality objectives. There are two technology trends that facilitate the design and management of cloud database service systems. First, the development of distributed replicated database software that is optimally designed for highly available and scalable Web applications and offered as open source software. Second, the possibility to deploy the system on cloud computing infrastructure to facilitate availability and scalability via on-demand provisioning of geo-located servers.**

We argue that a runtime quality measurement and analysis framework is necessary for the successful runtime management of cloud database service systems. Our framework offers three contributions over the state of the art: (i) the analysis of scaling strategies, (ii) the analysis of conflicts between contradictory objectives, and (iii) the analysis of system configuration changes on runtime performance and availability.

I. Introduction

Cloud computing is a model for building scalable and highly available services on top of an elastic pool of geo-located configurable resources, such as virtual machines, network bandwidth, and storage [1]. Elasticity means that additional resource capacity can be rapidly allocated (and de-allocated) to a system in a matter of seconds or minutes without significantly affecting or interrupting the services that are using these resources. The resources which are located in multiple data centers around the world are payed per use. Thereby, customers have a model for building affordable geo-scalable services that do not slow down or become unavailable under increasing workloads – as long as the customer's credit card permits the payment for resource capacity. Moreover, customers can build dependable services with replicated data storage that tolerate server failures or even entire data center disasters.

A. Terminology

Cloud database services promise high performance data processing, i.e., in the case of Web transaction processing cloud databases, low upper-percentile response time and elastically scalable throughput. Moreover, the services are supposed to be always available. At the same time, we can observe an increasing popularity of so-called *NoSQL databases*, i.e, database software systems that are designed in favor of high performance and high availability by means of data distribution and replication mechanisms. NoSQL databases usually have a flexible schema and provide less generic query capabilities than relational databases. Instead, NoSQL databases are optimized for transactional Web applications that usually only require query capabilities of limited complexity, such as simple HTTP methods. Nonetheless, relational database systems can also be deployed at Web scale if some of their features and design principles, such as data normalization, are sacrificed.

We refer to database software that is designed for high performance and high availability Web applications as *cloud database software*. Compute cloud users can deploy and manage cloud database software on top of a globally distributed compute cloud, e.g., Amazon EC2. We refer to such a system as *cloud database service system*.

B. Motivation

Managing the quality of service of the cloud database service system is necessary because applications on top of the system can usually only function properly if performance guarantees are given by the system. For example, the Amazon Dynamo key-value database negotiates service level agreements (SLAs) with the service clients, particularly SLAs on the request response time that can be guaranteed for a client-specified throughput distribution [2]. Flash crowds are a phenomenon where an application suddenly becomes very popular and must handle quickly growing workloads while maintaining service performance qualities [3].

C. Contributions

The quality of service depends on both the database software design and the runtime management of the system on top of the pool of on-demand servers in the compute cloud. In this paper, we focus on the latter and argue that a *quality measurement and analysis framework* is a necessary tool for the successful runtime management of cloud database service systems. In particular, our framework

(i) can be used for the analysis of scaling strategies,
(ii) allows more balanced database system optimization decisions in the face of conflicting objectives, and
(iii) can be used for the analysis of system configuration changes on runtime performance and availability.

TABLE I
SELECTED PERFORMANCE & AVAILABILITY CONFIGURATION PARAMETERS OF TYPICAL CLOUD DATABASE SERVICES AND SYSTEMS

Cloud DB Service	**System Design**	**Data Model**	**Service Performance & Availability Configuration Parameters**
Amazon SimpleDB	Unknown	Column-oriented	Read consistency levels
Amazon DynamoDB	Peer-to-Peer Ring	Column-oriented	Read consistency levels, provisioned throughput
Amazon RDS	Master-Slave	Relational schema	Multi-AZ deployments, read replicas
Cloud DB System	**System Design**	**Data Model**	**Cluster Performance & Availability Configuration Parameters**
Cassandra	Peer-to-Peer Ring	Column-oriented	Consistency levels, replication factor and replica placement, partitioning
Riak	Peer-to-Peer Ring	No schema	Consistency levels, replication factor
HBase	Master-Slave	Column-oriented	Replication factor and replica placement, distribution via HDFS
MongoDB	Master-Slave	Document-oriented	Replication groups, sharding
MySQL	Master-Slave	Relational schema	Cluster replication, sharding

The use of fine-grained quality metrics allows fine-grained quality control and quality optimization of a running system which is subject to variable workloads as well as variable server and network failures.

D. System configuration

A cloud database service system is set up on a compute cloud, i.e., an elastic scalable pool of virtualized servers. Desirable qualities include high performance and high availability. The qualities are enforced via system *configuration parameters*. We propose a parametric configuration model that is divided into 3 levels:

1) Compute cloud management: capacity management
2) Cluster management: data replication and distribution
3) Server management: operating system and software

In the context of our current framework, the compute cloud management level is primarily concerned with capacity management, i.e., the allocation and de-allocation of (differently sized) virtual machines. This level could be extended to encompass security-related configuration parameters, such as firewall settings, key exchange mechanisms, et cetera. On the cluster management level, our framework currently focuses on data replication and distribution mechanisms, however, other configuration parameters could also be of interest, such as failure detection or access control mechanisms. On the server management level, the framework focuses on operating system and software settings, such as file system settings, threads, caching, data compression, etc.

II. RELATED WORK

Cloud database service systems should provide services with supreme performance, elastic scalability, and availability. At the same time, other qualities that are potentially harmful to performance and availability, such as strong data consistency, are desirable, as well. Moreover, cost of service should be as low as possible and is typically constrained by a budget.

A. Elastic Scalability

Elastic scalability can be measured with a variety of metrics, for example speed-up, size-up, and scale-up [4], [5]. Speed-up refers to the question how response time reacts to new servers that are added, ceteris paribus, to the running system. Scale-up refers to the question how much throughput can be gained by adding more servers and increasing the workload volume, ceteris paribus.

The authors of [6] suggest measuring scalability with a regression function and the corresponding correlation coefficient. The deviation from linear scalability can then be measured with the correlation coefficient of a linear regression function of throughput depending on the number of servers. Furthermore, the authors suggest a metric to measure peak-workload scalability as a percentage of operations per second that complete within a given response time service level in relation to all issued operations per second during a peak load period. The authors of [7] suggest an elasticity metric that measures the response time impact during the bootstrapping phase of a new node into a cluster.

B. Dependability

Cloud database service systems are designed for high availability. In the event of server or network failures, it is desirable that a system does not suddenly stop serving requests. Even when entire data centers go down, the cloud database services should be available. Under such failure scenarios, in distributed replicated systems it is hard or even impossible to provide all qualities, particularly data availability and consistency, to their full extent [8], [9].

The authors of [6] suggest a metric for dependability (fault-tolerance) by specifying a percentage of failing resources and then measuring the number of operations that complete within a given response time service level in relation to all issued operation requests. The authors of [10] differentiate availability into two separate metrics, harvest (precision in terms of the fraction of response data that is returned) and yield (binary availability/unavailability). As discussed in [11], harvest can also be used as a measure of consistency since incomplete responses can appear as inconsistent responses. The interpretation of harvest thus depends on the client-side implementation.

C. Consistency

Literature knows a multitude of consistency definitions. For our purposes, though, strict consistency means that all replica of a given data item are identical. This is a necessary precondition for ACID (Atomicity, Consistency, Isolation, Durability) guarantees. Relaxing the ACID guarantees leads to the eventual consistency model, meaning that in the absence of updates strict consistency will be reached eventually [12]. Here, we focus on eventual consistency metrics which provide more fine-grained quality assessment than the ACID model.

There are two main perspectives on consistency in itself: a client-side perspective and a system-side (or data-centric) perspective. The first depends on the behavior observable by the client while the latter describes the interna of a storage system. For most use cases a client-side perspective suffices as this is the consistency level actually experienced by applications using the storage system. A system-centric view on consistency, in contrast, is a the means to generate the client-side output.

Vogels [13] names several subforms of consistency beyond strict and eventual consistency when taking a client-side perspective. One prominent example is monotonic read consistency (MRC) where a storage system guarantees never to return a version older than n once this version has been returned at least once. Other examples are monotonic write consistency (MWC) or read your writes consistency (RYWC). Bermbach and Tai [14] suggest an approach to experimentally quantify eventual consistency via the length of the inconsistency window, a concept defined as t-visibility by [15]. They also count violations of monotonic read consistency. A similar approach for measuring staleness after batch inserts has been implemented in YCSB++ [16].

III. Cloud Database Service Quality Metrics

Based on the quality metrics discussed in the literature, we develop a comprehensive quality model for cloud database service systems that comprises of multiple fine-grained runtime quality of service metrics.

A. Performance

Performance is typically measured in terms of throughput and request response time. The performance measurement samples are aggregated into estimators of certain parameters of the performance distribution. These estimators are the performance metrics.

1) Average Throughput: A throughput measurement $m_i = ops$ is the number of operations ops over a given time period t_i. Based on multiple throughput measurements, throughput metrics can be formulated for each type of request, such as read or write requests. Typically, users are interested in average throughput measured in operations per second, $T = \bar{m} = \sum_i^n m_i / \sum_i^n t_i$.

2) Average and Upper-Percentile Response Time: In the context of our work, a response time measurement $m_i = t_{req} - t_{resp}$ is defined as the time lag between issuing a service request and receiving a successful response. The success of a response is indicated by the response message status code, such as “HTTP 200”. Based on multiple measurements, response time metrics can be formulated for each type of request, such as read or write requests. Two metrics are of particular interest, namely the *average response time* of n measurements, $RT = \bar{m} = \sum_i^n m_i/n$, and the *upper-percentile response time* $RT_p = F^{-1}(p)$ with F^{-1} as the inverse cumulative response time distribution function. Typical values are $p = 95\%$ or $p = 99\%$.

3) Elastic Scalability: Elasticity is defined via both speed and performance impact of adding new resources into a cloud database service system, and removing resources from a system, respectively. The objective is to add or remove servers as fast as possible without reducing performance and availability metrics below a given level. We later show experimental results where we measured the impact of increasing throughput on response time variability by example of two different scaling strategies.

Scalability is defined by the increase in throughput, and the decrease in response time, respectively, that can be gained by adding more server resources to a database system, as well as the speed of adding new servers to a cluster. On the other hand, it is also important to measure the speed and performance impact of removing servers from a database system if the capacity is not needed any more because workloads have decreased.

The basic metrics for measuring elastic scalability are scale-up and speed-up. Scale-up defines the throughput-processing capacity gained by increasing the number of servers by α percent, i.e., $T(\alpha s)/T(s)$, where s is the number of servers and $T(\cdot)$ is the throughput function. Speed-up similarly defines the decrease of response time that can be gained if servers are added but workloads remain constant, i.e. $RT(s)/RT(\alpha s)$.

B. Consistency

We propose the following metrics to measure degrees of consistency:

- Staleness: This is the time window between the last read of version n and the start of the write of version $n+1$ for most storage systems. If the system guarantees transaction isolation, this value decreases by the write latency for version $n+1$. Given a distribution of these inconsistency windows, one can also calculate a curve describing the probability of stale reads over time which is just a different representation of the same data.
- Violations of MRC: This the likelihood of reading a version n followed by reading an older version some time later.
- Violations of MWC: The likelihood of two consecutive writes by the same client being serialized in reverse chronological order.
- Violations of RYWC: The likelihood of a client writing a version n and the same client reading an older version some time later.

C. Dependability

Dependability encompasses both reliability and availability. A typical reliability metric is the expected failure rate, i.e., the expected number of failures in a given time period. Reliability depends on a variety of factors, such as hardware, software, and human failures, that cannot be measured with our framework. Instead, we assume that empirical failure data is provided as an input into the framework.

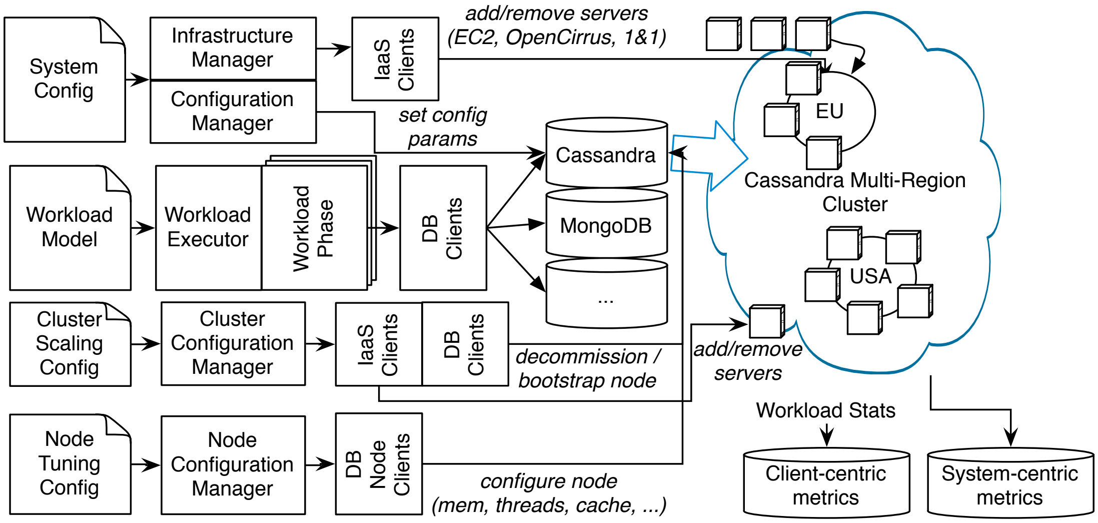


Fig. 1. Experiment-based Runtime Quality Measurement Framework.

Availability is the ratio between uptime and uptime + downtime. In other words, availability can be measured as the ratio between successful requests due to unavailability of service and the number of all requests. Given a failure rate, availability metrics are related to failure detection and recovery time measurements for different systems and failure scenarios. Usually, availability is defined as the ratio $\frac{MTTF}{MTTF+MTTR}$ where MTTF denotes the mean time to failure and MTTR the mean time to recovery.

IV. Quality Measurement Framework

The quality measurement framework manages the setup of cloud database software on top of virtual servers in a compute cloud (figure 1).

A. Architecture

The *Infrastructure Manager* provisions virtual servers that can be placed in different geographic regions. The servers are then configured by the *Configuration Manager*. Thereby, we have a convenient automated mechanism to set up a cloud database service system when we need it – and an economic mechanism to decommission the system when we do not need it anymore.

The benchmarking framework is based on the open source tools Yahoo! Cloud Serving Benchmark (YCSB) and YCSB++ [5], [16]. The *Workload Executors* are deployed and can also be placed in different geographic regions. Workloads are divided into multiple phases which allow testing complex workload scenarios. The abstract workload model is translated into workload instances that are executed by a specific database client. Our contribution is the integration of cloud database service system setup and benchmarking. The benchmarking framework can also be allocated and deallocated on-demand.

The framework provides a *Cluster Configuration Manager* that can scale the database system up by provisioning new virtual servers in the compute cloud, configuring the servers, and bootstrapping the servers as new nodes into the cloud database service system. The Cluster Configuration Manager can also scale the system down by decommissioning nodes, waiting until the nodes have been removed from the cluster, and then terminating the corresponding virtual servers in the compute cloud. The *Node Configuration Manager* is used to apply server-level configurations, such as thread-pool sizes, memory allocation, etc. We envision a *Failure Executor* and *Recovery Executor* to automatically test complex failure and recovery scenarios. These parts of the framework have not been implemented, yet.

The quality data is measured from two perspectives, the client perspective, such as the response time of client requests, and the system perspective, by integrating monitoring tools as demonstrated with [16].

B. YCSB DB Client Adapters

We developed additional YCSB adapters for Amazon Simple Storage Service (S3), SimpleDB, and DynamoDB. Table V show the data model mapping between the generic YCSB data model and the respective database client implementations, as well as the mapping between generic Read, Write, and Scan requests and the respective database client implementations.

1) Simple Storage Service: Simple Storage Service (S3) is not perfectly suitable for the type of workload generated by YCSB. All YCSB operations work with data records. Each record consists of a number of fields (by default 10) of a certain length (by default 100 bytes). In the S3 adapter, we map the generic record data type into a file with field name-value pairs. For each insert operation, a new file is generated and uploaded to S3 as an S3Object, and each read operation downloads a file and writes the requested fields to a buffer stream.

2) SimpleDB: SimpleDB is better suited for mapping the generic YCSB data schema than S3. Similar to DynamoDB and Cassandra, SimpleDB saves each record in a row with a flexible number of columns. We found that the sequential execution of SimpleDB requests results in low throughput because the client becomes a bottleneck. Therefore, we implemented the client to make asynchronous requests and calculate latency upon receiving the callback response.

3) DynamoDB: For all Amazon storage service benchmarks we adjusted the YCSB record field length property to 90 bytes instead of 100 bytes. The reason is that DynamoDB provides a given amount of "Provisioned Throughput" and calculates this capacity based on items that are less than 1KB in size. If the item size is equal or close to 1KB, we would need to provision twice the throughput. Similar to the SimpleDB client, DynamoDB's Java client can become a bottleneck if operations are not executed asynchronously.

4) Cassandra: The Cassandra DB client maps the YCSB table schema to a column family in a keyspace. Records are stored as rows with a flexible number of columns where each column contains the content of one field. With the Cassandra 0.7 client, single records are accessed by a slice query with a predicate that lists the fields that are requested. A write operation (insert or update) maps to a batch mutate requests that writes multiple columns in a single row. The default replication factor is 1, and the read and write consistency levels are ONE, as well; i.e. no replication is used and a single acknowledgement is required per operation. We extended the existing Cassandra client implementation to vary the consistency level with each benchmark run.

V. Experiments

In this section, we discuss some representative results that were gained with the quality measurement framework by example of Cassandra clusters on top of Amazon EC2 servers. Cassandra [17] is a Peer-to-Peer system with replication and distribution techniques inspired by Amazon's Dynamo key-value store [2] and a column-oriented [18] data model.

A. Analysis of the Cassandra Database System

We focus on the five decision parameters stated in table II: "number of servers", "server type", "replication factor", "Read consistency level", and "Write consistency level".

1) Cassandra Scalability: We evaluate two different scaling strategies (table III), vertical and horizontal scaling. The initial setup is a Cassandra cluster on top of four m1.large[1] Amazon EC2 instances. One m1.large instance currently costs $0.36 per hour, plus additional charges for traffic, etc. The cluster is pre-filled with 40 GB synthetic data. With the *vertical scaling* strategy, each of the 4 Cassandra nodes is upgraded to m1.xlarge[2] instances that offer twice the resources. *Horizontal scaling*, on the other side, doubles the number from four m1.large instances to eight m1.large instances. We benchmark the two clusters that represent the two different

Parameter	Range
#servers	$n \in \{3, \dots, 16\}$
EC2 server type	$t \in \{m1.small, m1.large, m1.xlarge\}$
Replication factor	$N \in \{1, \dots, 5\}$
Read consistency	$R \in \{\text{ONE}, \text{QUORUM}, \text{ALL}\}$
Write consistency	$W \in \{\text{ONE}, \text{QUORUM}, \text{ALL}\}$
Objective	$f_i \in \{\text{Throughput}, \text{Read average}, \text{Write average}, \text{Read } 95^{\text{th}}, \text{Write } 95^{\text{th}}, \text{Inconsistency}, \text{Cost}\}$

TABLE II
Ranges of the decision variables and objectives.

Param	Vertical Scaling	Horizontal Scaling
n	4	8
t	m1.xlarge	m1.large
N	1	1
R	ONE	ONE
W	ONE	ONE
f_1	Throughput	Throughput
f_2	Avg. Latency	Avg. Latency
f_3	99-perc. Latency	99-perc. Latency

TABLE III
Experiment design for scalability tests.

Param	Factorial Experiment Design
n	3, ..., 16
t	m1.small
N	1, ..., 5
R	$\{\text{ONE}, \text{QUORUM}, \text{ALL}\}$
W	$\{\text{ONE}, \text{QUORUM}, \text{ALL}\}$
f_1	Throughput
f_2	Avg. Latency
f_3	99-perc. Latency

TABLE IV
Experiment design for replication setting tests.

YCSB DB	Table	Record	Field
S3	Bucket	File, S3Object	Text line
SimpleDB	Domain	Replaceable-Item	Replaceable-Attribute
DynamoDB	Table	Item	Attribute-Value
Cassandra	Keyspace, Column-Family	Row	Column
YCSB DB	**Read**	**Write**	**Scan**
S3	Get Object	Put Object	List Objects
SimpleDB	Select	Put Attributes	Select
DynamoDB	Get Item	Put Item	Scan
Cassandra	Get Slice	Batch Mutate	Get Range Slices

TABLE V
YCSB DB Client Data Model and Operation Mapping.

	Thr	Wr Av	Wr 95	R Av	R 95
C	89 ops/s	101 ms	138 ms	35 ms	38 ms
EC	95 ops/s	88 ms	112 ms	37 ms	36 ms

TABLE VI
Performance values for SimpleDB.

	Thr	Wr Av	Wr 95	R Av	R 95
C	96 ops/s	44 ms	23 ms	44 ms	22 ms
EC	99 ops/s	16 ms	17 ms	15 ms	15 ms

TABLE VII
Performance values for DynamoDB.

[1] Each m1.large instance is equipped with 7.5 GB memory, 850 GB disk storage, and 4 EC2 Compute Units (equivalent to four 1GHz Opteron 2007 cores).

[2] Each m1.xlarge instance is equipped with 15 GB memory, 1700 GB disk storage, and 8 ECU, for $0.72 per hour.

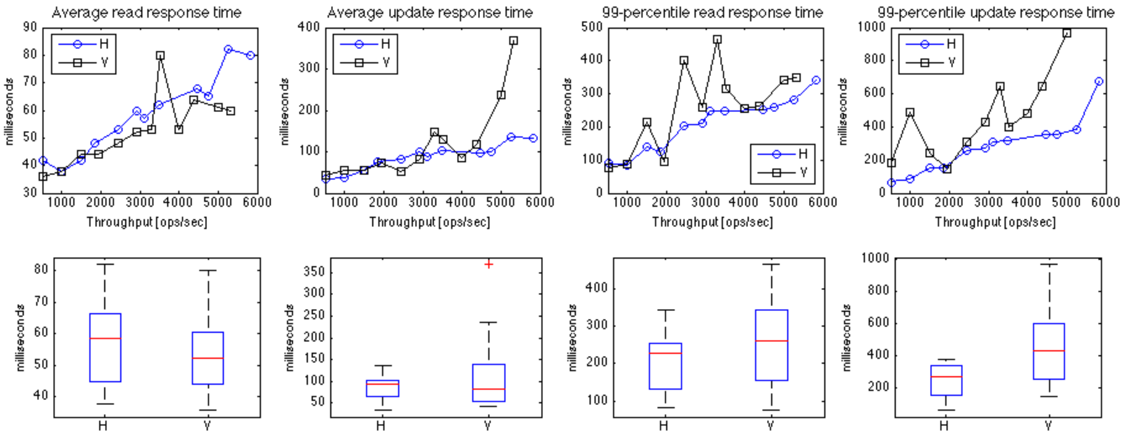
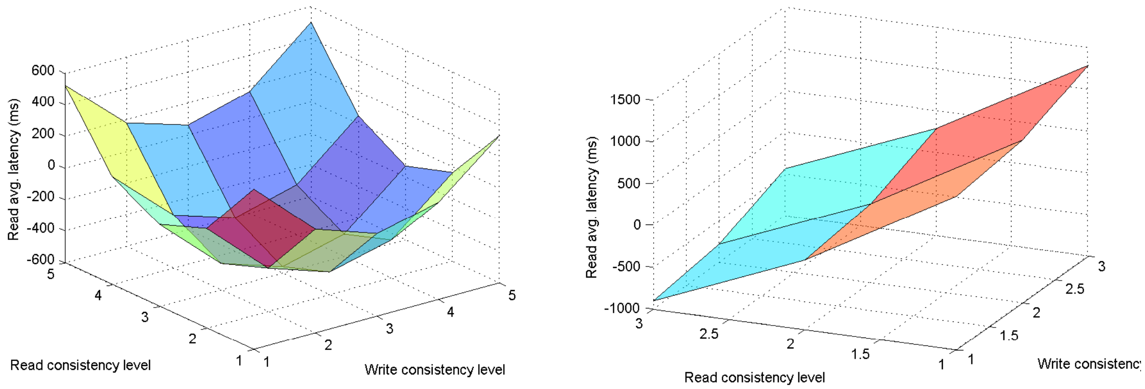


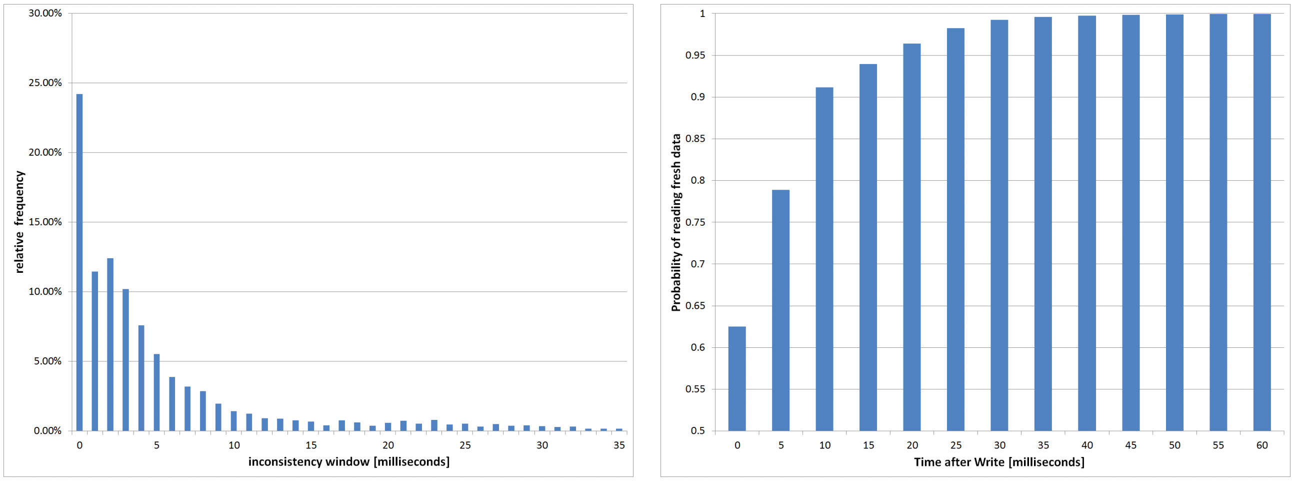
Fig. 2. Horizontally (H) vs. vertically (V) scaled Cassandra 1.0 cluster.



(a) n=16, N=5

(b) n=16, N=3

Fig. 3. Average Read latencies subject to consistency level and replication factor. N is the Cassandra replication factor and n is the number of m1.small EC2 instances. The Write and Read consistency level numbers refer to the number of replicas that must acknowledge a client request synchronously.



(a) Relative Frequencies of Inconsistency Windows

(b) Probability of Reading non-Stale Data over Time

Fig. 4. Distribution of Cassandra's Inconsistency Windows (without Outliers).

scaling strategies with a workload with a zipfian request distribution where 80% are read and 20% are update requests. The initial cluster with four m1.large instances can process up to 1,200 ops/sec. Figure 2 shows that the cluster can be vertically scaled to ca. 4,000 ops/sec after which the update response time dramatically increases whereas it can be horizontally scaled to more than 5,000 ops/sec. The horizontal scale-up strategy amounts to a 4× increase in throughput processing capacity, i.e. roughly $T(8, \mathrm{large})/T(4, \mathrm{large}) = 5,000/1,200$. The vertical scale-up, on the other side, achieves only a 3.3× increase of the original throughput, i.e. roughly $T(4, \mathrm{xlarge})/T(4, \mathrm{large}) = 4,000/1,200$.

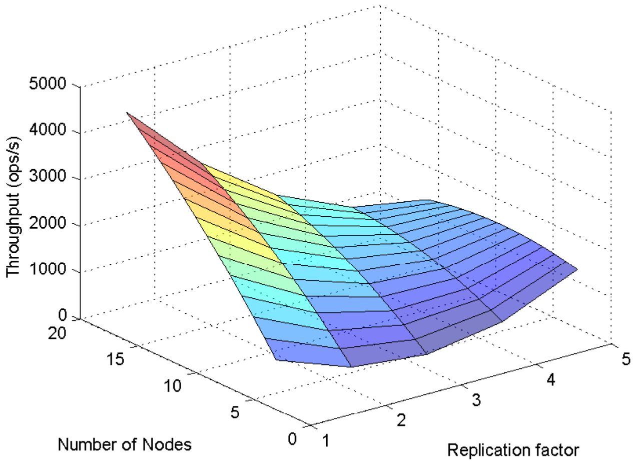


Fig. 5. Response surface of maximum throughput of a Cassandra 0.7 cluster depending on replication factor N and #nodes (m1.small EC2 instances), and consistency level ONE, given a YCSB workload with 80% read and 20% update requests with a zipfian request distribution.

The boxplots in figure 2 show that the response time variance of the vertically scaled cluster tends to be much higher than of the horizontally scaled cluster (except for average read response time). We suspect that this effect stems from more imbalanced load distribution in the vertically scaled cluster, however, we have not collected monitoring data to evaluate this hypothesis, yet.

2) Cassandra Replication Settings: With another experiment design (table IV), we measure average throughput as well as average and upper percentile latency of a Cassandra cluster, depending on replication factor, consistency levels, and cluster size. The replication factor N specifies the number of data copies and thus increases service availability in the event of server failures or network partitions. Increasing the replication factor, however, can affect performance.

Experiments with a workload model that specifies 80% read and 20% update requests shows that this is even the case for moderately write-intense workloads with weak consistency guarantees. The data in figure 5 shows that, if the Cassandra cluster is small, increasing the replication factor from 1 to 3 affects throughput negatively, however, increasing it from 3 to 5, has a slight positive effect. However, if we scale the cluster to larger sizes, increasing the replication factor has a significant negative effect on throughput.

Figure 3 shows that, in the case of a very large replication factor of $N = 5$, the Read latency is optimal for a Read and Write QUORUM consistency level configuration. In the case of $N = 3$, the optimal configuration is ALL. $N = 5$ with QUORUM level and $N = 3$ ALL level, respectively, both refer to 3 replicas that must synchronously acknowledge a client request.

3) Consistency Benchmarking: Using the approach described in [14], we deployed 3 large EC2 instances distributed in the region eu-west (one per availability zone) running Cassandra 0.6.5, additionally we evenly distributed 12 readers and one writer over the same region. All requests were issued at consistency level ONE over a period of 24 hours. Each reader issued a request in 1ms intervals, every 10 seconds the writer issued an update. Our results show a geometric distribution of inconsistency windows with more than 98% of all writes showing less than 40ms of inconsistency. This result is close to the accuracy limitations of the Network Time Protocol which is an unresolved issue of our benchmarking approach. Monotonic read consistency was violated in about 0.00057% of all test runs. Figure 4 shows the geometric distribution in the left part and the probability of reading fresh, i.e., non-stale, data on the right. It can easily be seen that actual inconsistency windows are very small in a situation without failures. In contrast to experiments with Amazon S3 [14], though, our experiments were run on an isolated cluster running our software stack only. But when creating additional load on the system, the behavior considerable worsens (though still less than S3). Currently, we are analyzing the relation between load and inconsistency window as our original Cassandra benchmark with additional load overloaded the cluster.

B. Analysis of Amazon Database Services

Since both DynamoDB and Cassandra implement aspects of the original Dynamo [2] design, we are interested in comparing quality between the managed database service DynamoDB and our EC2-based Cassandra setups. We additionally benchmark SimpleDB in comparison to DynamoDB. For all benchmarks, we use the same benchmarking setup and workload as for the Cassandra experiments.

1) SimpleDB: We measure a bad maximum throughput of approximately 400 ops/s which is less than the throughput achieved with a single Cassandra node. The average SimepleDB write latency rockets to over 4700 ms if we try to achieve higher throughput numbers and only the average read latency comes close to Cassandra's values.

Consequently, we reduce the throughput target to 100 ops/s. We then compare the two consistency levels in SimpleDB, Consistent Read (C) and Eventually Consistent Read (EC), shown in table VI. The read latencies are not affected by the consistency level and have a very low spread between the average and 95th percentile values. To our surprise, the write latencies increase by 15-20% when choosing the Consistent Read option.

Similar experiments with SimpleDB, described in [19], report lower latency values for a workload with maximum throughput of up to 1,200 ops/s which could be an indication that our benchmarking client might be a bottleneck in the setup. On the other side, our data does not show request failures whereas the SimpleDB experiments in [19] report up to 50% service unavailability failures (HTTP return code 503) for the high-throughput workloads. The authors observe no significant differences between the latencies for Consistent Read and Eventual Consistent Read requests and suggest not to use the Eventual Consistent option because there were no performance improvements that would reward for the 30% chance to read stale data with a staleness of up to 500 ms.

2) DynamoDB: DynamoDB offers a configuration parameter "Provisioned Throughput" per table. The configuration parameter allows customers to specify the read and write operation throughput they currently need and DynamoDB accordingly allocates or deallocates resource capacity. This online capacity management is executed in a way so that low-latency performance can be guaranteed even for upper percentiles of the request distribution. The capacity is calculated based on two metrics: the item size and, for read operations, the read consistency level. One write capacity unit is needed to provide one write operation per second. One read capacity unit is needed to provide one Consistent Read operation, or two Eventually Consistent Read operations, respectively. In other words, Eventually Consistent Reads are "half the price" of Consistent Reads.

Our experiments show a similar throughput limit as in SimpleDB – even if "Provisioned Throughput" is set to large values of 1,000 write and 1,000 read capacity units. Maximum throughput is again low (430 ops/s), write latencies reach 1,300 ms and read latencies 600 ms which would be a 2,000% and 1,200% increase in comparison to Cassandra, respectively.

Again, we limit throughput to 100 ops/s and compare the two consistency level options (table VII). Unlike SimpleDB, DynamoDB balances read and write latency very well. This results look reasonable, considering that DynamoDB uses solid state drive hardware.

Consistent Reads turn out to be expensive: average latency increases by roughly 200% compared to Eventually Consistent Reads. We also observe that Eventually Consistent Reads have similar values for 95th percentile and average latency.

VI. Discussion

A. Interpretation of Experimental Results

Interestingly, in our experiments with both SimpleDB and DynamoDB we are not able to scale throughput beyond three-digit numbers – which is considerably lower than the throughput that we achieved with our Cassandra clusters. The related work [19] suggests that this might in fact be a limitation of the current SimpleDB offering. Moreover, at the time of running the experiments, DynamoDB still is in a public beta test phase which might explain why we cannot scale DynamoDB to the provisioned throughput. Therefore, a meaningful comparison between Cassandra and Amazon's database services is not possible since the EC2-based Cassandra cluster easily outperforms both SimpleDB and DynamoDB.

B. Lessons Learned

During our research work, we learned that automating the setup and configuration of the system under test requires a reliable configuration management solution. The configuration management solution used in our research prototype implementation still needs further improvement. Moreover, we acknowledge that the implementation of the benchmarking clients has a significant impact on the test methodology and the test results and must therefore be carefully reviewed.

We are working on improvements of our setup to more efficiently gather measurement data for a broader variety of systems. We are also extending the benchmarking framework to measure additional objective metrics, such as availability and reliability in the face of server failures.

C. Potential Use of Experimental Results

1) Graceful degradation: A common fault-tolerance strategy of Web applications is the degradation of features or quality in the event of a failure [20]. There are two basic types of stop-failures that can harm cloud service availability: server failures and network partitions. A temporary server failure renders data unavailable during the time to recovery. Even worse, a permanent server failure leads to data loss. Fortunately, availability and durability can be increased by replicating data across multiple failure-isolated servers. During the failure of one or more servers, the remaining replica servers pitch in – given that they are not overwhelmed by the additional workload. Unfortunately, replication for availability can harm performance and elastic scalability. The impact of replication as well as the impact of recovery mechanisms is therefore addressed by our quality measurement framework.

Data migration is another data management mechanism that can lead to degraded performance and availability. Online data migration is necessary when data needs to be re-distributed in a running database system. This is usually the case when new servers join or leave a database system and data must be migrated to the joining, or from the leaving servers, respectively. Thus, the basic operations of scaling a system temporarily affect performance. If data cannot be re-distributed without taking the system down, availability is reduced by scheduled downtime. Another case of data migration is data load balancing to resolve imbalanced data distribution in a database system.

We are working to extend our framework in support of the analysis of the performance and availability impact of failures along with an analysis of strategies to reduce these negative side-effects. Moreover, the framework should provide more advanced features to analyze the elastic scalability of a database system with focus on two metrics: the speed and the performance impact of online data migration.

2) Quality optimization tradeoffs: We argue that fine-grained metrics are better for controlling the quality of a system than coarse-grained metrics. For example, the binary

QoS levels {consistent, inconsistent} are less expressive than QoS levels that specify the maximum allowed average t-visibility (i.e., staleness in terms of time [15]) on a continuous scale of milliseconds. The consistency of a service can be optimized by adapting consistency control mechanisms, such as Quorum-based consistency protocols.

If qualities are in conflict with one another, tradeoff decisions must be made during optimization of one of the qualities. Fine-grained quality metrics allow the definition of more expressive tradeoff decisions between conflicting qualities. For example, a quality control system could trade marginal units of read-after-write inconsistency for marginal units of harvest.

VII. Conclusion

We present a runtime quality measurement and analysis framework for cloud database service systems. The framework allows the automated distributed setup of cloud database software on elastic compute cloud infrastructure along with the automated setup of an experiment framework to efficiently conduct performance benchmarks. The framework can be used for the analysis of scaling strategies, such as vertical vs. horizontal scaling, for the analysis of conflicting objectives, such as consistency vs. performance vs. availability.

We compare our results with performance benchmarks of the managed database services Amazon SimpleDB and DynamoDB. Surprisingly, in our experiments, the managed services do not scale beyond low throughput numbers of 400 ops/sec whereas even small Cassandra clusters can process multiple thousand ops/sec. We are working on extensions of our framework for the analysis of a broader spectrum of NoSQL database systems, including HBase and MongoDB. Moreover, we are working on a framework extension for the analysis of fault tolerance mechanisms of cloud-based NoSQL database systems.

We envision our framework as a basis for advanced database management and optimization decision support mechanisms, including graceful quality degradation decisions and quality tradeoff decisions.

VIII. Acknowledgements

We thank our student Holger Trittenbach and our colleague Michael Menzel for their support in conducting the benchmarking experiments. We furthermore thank Amazon Web Services for supporting our work with a generous research grant. The work presented in this paper was performed in the context of the Software-Cluster projects InDiNet and EMERGENT (software-cluster.org). It was partially funded by the German Federal Ministry of Education and Research (BMBF) under grant numbers “01IC10S04” and “01IC10S01”. The authors assume responsibility for the content.

References

[1] P. Mell and T. Grance, “The NIST Definition of Cloud Computing.” [Online]. Available: http://csrc.nist.gov/publications/nistpubs/800 - 145/SP800-145.pdf

[2] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, “Dynamo: Amazon’s Highly Available Key-Value Store,” in *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles*, ser. SOSP ’07. New York, NY, USA: ACM, 2007, pp. 205–220.

[3] J. Elson and J. Howell, “Handling flash crowds from your garage,” in *USENIX 2008 Annual Technical Conference on Annual Technical Conference*, ser. ATC’08. Berkeley, CA, USA: USENIX Association, 2008, pp. 171–184. [Online]. Available: http://dl.acm.org/citation.cfm?id=1404014.1404027

[4] D. Bitton, D. J. DeWitt, and C. Turbyfill, “Benchmarking Database Systems - A Systematic Approach,” in *VLDB*, 1983, pp. 8–19.

[5] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, “Benchmarking Cloud Serving Systems with YCSB,” in *Proceedings of the 1st ACM symposium on Cloud computing*, ser. SoCC ’10, New York, NY, USA, 2010, pp. 143–154.

[6] C. Binnig, D. Kossmann, T. Kraska, and S. Loesing, “How is the Weather Tomorrow? Towards a Benchmark for the Cloud,” in *Proceedings of the Second International Workshop on Testing Database Systems*, ser. DBTest ’09. New York, NY, USA: ACM, 2009.

[7] T. Dory, B. Mej, and P. V. Roy, “Measuring Elasticity for Cloud Databases,” *Computing*, no. c, pp. 154–160, 2011.

[8] S. Gilbert and N. Lynch, “Brewer’s Conjecture and the Feasibility of Consistent, Available, Partition-tolerant Web Services,” *SIGACT News*, vol. 33, pp. 51–59, June 2002.

[9] E. Brewer, “Pushing the CAP: Strategies for Consistency and Availability,” *Computer*, vol. PP, no. 99, p. 1, 2012.

[10] A. Fox and E. A. Brewer, “Harvest, Yield, and Scalable Tolerant Systems,” in *Proceedings of the The Seventh Workshop on Hot Topics in Operating Systems*, ser. HOTOS ’99. Washington, DC, USA: IEEE Computer Society, 1999.

[11] M. Klems, M. Menzel, and R. Fischer, in *Proceedings of the 8th International Conference on Service Oriented Computing (ICSOC)*, 8th International Conference, ICSOC. Berlin Heidelberg: Springer, Dezember 2010, Inproceedings, pp. 627–634.

[12] A. S. Tanenbaum and M. V. Steen, *Distributed Systems - Principles and Paradigms*, 2nd ed. Upper Saddle River, NJ: Pearson Education, 2007.

[13] W. Vogels, “Eventually Consistent,” *Queue*, vol. 6, pp. 14–19, October 2008.

[14] D. Bermbach and S. Tai, “Eventual Consistency: How Soon is Eventual? An Evaluation of Amazon S3’s Consistency Behavior,” in *Proceedings of the 6th Workshop on Middleware for Service Oriented Computing*, ser. MW4SOC ’11. New York, NY, USA: ACM, 2011.

[15] P. Bailis, S. Venkataram, J. Hellerstein, M. Franklin, and I. Stoica, “Probabilistically Bounded Staleness for Practical Partial Quorums,” in *Technical Report No. UCB-EECS-2012-04*. University of California at Berkeley, 2012.

[16] S. Patil, M. Polte, K. Ren, W. Tantisiriroj, L. Xiao, J. López, G. Gibson, A. Fuchs, and B. Rinaldi, “YCSB++: Benchmarking and Performance Debugging Advanced Features in Scalable Table Stores,” in *Proceedings of the 2nd ACM Symposium on Cloud Computing*, ser. SoCC ’11, New York, NY, USA, 2011.

[17] A. Lakshman and P. Malik, “Cassandra: Structured Storage System on a P2P Network,” in *Proceedings of the 28th ACM symposium on Principles of distributed computing*, ser. PODC ’09, New York, NY, USA, 2009.

[18] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, “Bigtable: A Distributed Storage System for Structured Data,” in *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, ser. OSDI ’06. Berkeley, CA, USA: USENIX Association, 2006, pp. 15–15.

[19] H. Wada, A. Fekete, L. Zhao, K. Lee, and A. Liu, “Data Consistency Properties and the Trade-offs in Commercial Cloud Storages: the Consumers Perspective,” in *Conference on Innovative Data Systems Research*, 2011, pp. 134–143.

[20] J. Allspaw, Ed., *Web Operations : Keeping the Data on Time*, 1st ed. Beijing: O’Reilly, 2010.