Towards Elastic Transactional Cloud Storage with Range Query Support

Hoang Tam Vo [#1], Chun Chen [§2], Beng Chin Ooi [#3]
[#] *School of Computing, National University of Singapore, Singapore*
[1,3] {voht,ooibc}@comp.nus.edu.sg
[§] *College of Computer Science, Zhejiang University, China*
[2] chenc@cs.zju.edu.cn

ABSTRACT

Cloud storage is an emerging infrastructure that offers Platforms as a Service (PaaS). On such platforms, storage and compute power are adjusted dynamically, and therefore it is important to build a highly scalable and reliable storage that can elastically scale on-demand with minimal startup cost.

In this paper, we propose ecStore – an elastic cloud storage system that supports automated data partitioning and replication, load balancing, efficient range query, and transactional access. In ecStore, data objects are distributed and replicated in a cluster of commodity computer nodes located in the cloud. Users can access data via transactions which bundle read and write operations on multiple data items stored on possibly different cluster nodes.

The architecture of ecStore follows a stratum design that leverages an underlying distributed index with a replication layer in the middle and a transaction management layer on top. ecStore provides adaptive read consistency on replicated data. We also enhance the system with an effective load balancing scheme using a self-tuning replication technique that is specially designed for large-scale data. Furthermore, a multi-version optimistic concurrency control scheme matches well with the characteristics of data in cloud storages. To validate the performance of the system, we have conducted extensive experiments on various platforms including a commercial cloud (Amazon's EC2), an in-house cluster, and PlanetLab.

1. INTRODUCTION

Cloud computing is a step towards the notion that all aspects of computation can be organized as a utility, and it embraces paradigms ranging from Platform as a Service (PaaS) to Software as a Service (SaaS). As industry transits from in-house data management to cloud-hosted management, cloud storage has become one of the most widely acceptable infrastructure services [7].

Unfortunately, current cloud storage services are not adequate to support applications that require guarantees on consistency especially in the presence of data updates. For example, while Amazon's Dynamo[14] supports key-value insert and lookup operations, it does not offer range query support, and more importantly, transactional semantics for operations spanning multiple keys. More recently, Google MegaStore [3] has started to provide a certain level of transactional semantics for cloud storages.

In this paper, we deal with the consistency issue of replicated data and load balancing problem in range-partitioned systems. While this problem has a broad application domain ranging from traditional parallel and distributed databases to peer-to-peer range index systems to the emerging cloud storage, we choose cloud storage as our targeted application. Some example applications that might embrace the use of cloud range storages are: a webshop which wants to store listings in a sorted order based on their timestamps so that users can query the latest ones and also needs to support transactions such as updating user information or submitting user orders; or a Web 2.0 photo sharing application which stores the metadata of photos uploaded by users in date order and supports range scan queries like "finding the highly ranked photos uploaded within the last month".

We propose **ecStore**, a highly elastic distributed storage system with efficient range-query and transactional support that can be dynamically deployed in the cloud cluster. The design of ecStore is motivated by the fact that current closed-source cloud data serving systems (such as Dynamo and Pnuts) and open-source systems (such as HBase and Cassandra) do not support transactional semantics across multiple keys. In addition, most of these systems employ data migration to balance the storage load of the servers in the system. However, under skewed query distributions, we also need to balance the query execution load, which drives the design of the load-adaptive replication technique in ecStore. This work is part of our cloud-based data management system, named epiC (elastic power-aware data-intensive Cloud) [2], which is designed to support both analytical and OLTP workloads.

ecStore is designed as a stratum architecture. At the lowest level, it employs a distributed data structure to decluster data objects across the storage nodes and facilitates parallelism to improve system performance in terms of both throughput and response time. While any DHT-based structure can be used, in this work, we employ BATON (BAlance Tree Overlay Network) [19] which supports efficient range query processing. In addition, BATON could automatically repartition and redistribute the data when storage nodes are added into or removed from the system. This function is desirable since a cloud storage should allow users to scale up and down on the fly based on load and need.

In the middle tier, referred to as the replication layer, we leverage on the underlying distributed index structure to support replication. Here, we extend BATON to effectively support load-adaptive replication for large-scale data. The idea of tuning the replication process based on data popularity is common; however, most of the previous work maintain the query access statistics on a per data ob-

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. Articles from this volume were presented at The 36th International Conference on Very Large Data Bases, September 13-17, 2010, Singapore.
Proceedings of the VLDB Endowment, Vol. 3, No. 1
Copyright 2010 VLDB Endowment 2150-8097/10/09... $ 10.00.

ject basis. This approach is inefficient when the amount of data in the system is large, especially for cloud-scale databases. By the use of self-tuning range histogram, ecStore can efficiently deal with skewed access patterns while creating only a small number of replicas and keeping the histogram maintenance cost minimal.

Finally, in the highest tier, referred to as the transaction layer, we deploy a multi-version optimistic concurrency control scheme over the cloud storage. While multi-versioning enhances the performance of read-dominant applications, the use of optimistic concurrency control takes advantage of emerging applications where users typically access mutually exclusive data. In addition, the recovery control technique in ecStore guarantees the data durability requirement when building a transactional cloud storage on virtual infrastructures. While many aspects of this work have been studied in isolation in the past, we believe that the fusion and adaptation of these techniques in a real system has not been reported or evaluated.

In summary, the major contributions of this paper are as follows:

- We design ecStore – a highly scalable storage for elastic computing environments which processes range queries efficiently. We also support transactional access which bundles multiple read and write operations in ecStore.

- We provide high resilience capability with replication and recovery control. We also enhance the load balancing function in ecStore with a two-tier partial replication strategy, which is adaptive with the database workload.

- We validate the prototype of ecStore on various platforms including a commercial cloud (Amazon's EC2), an in-house cluster, and PlanetLab.

The paper is organized as follows. In the next section, we review existing research work that are related to our proposal. We present the system architecture of ecStore in Section 3. Then, in Section 4, we propose a load-adaptive replication technique for large-scale data. In Section 5, we present the transaction management technique in ecStore. In Section 6, we study various aspects of system performance in a real environment. We conclude in Section 7.

2. RELATED WORK

2.1 Replication in distributed and peer-to-peer systems

A large-scale storage system with built-in replication was proposed in [24] as the back-end for a commercial internet service. In this system, the primary copy of data is responsible to handle both read and write requests from clients. In addition, the system only supports operations on a single data item, and the problem of managing arbitrary transactions on multiple data items in replicated environment has not been addressed.

In [9], a ring overlay structure named PRing is leveraged with replication in cluster computing environment. Data resided on a storage node is replicated on the successor nodes. The system employs pessimistic replication technique where an update needs to be reflected on all replicas before coming to effect.

2.2 Distributed and parallel databases

In [8], a transactional distributed B$^+$-tree is built for range query processing in cluster environment. This distributed data structure uses optimistic scheme to speed up the concurrency control, while the underlying platform uses the two-phase commit protocol to ensure the correctness of transactions.

In [15], DeWitt and Gray present a thorough review on the techniques used by various research and commercial parallel database systems. Other related work include online load balancing in range-partitioned systems using data migration [16] and self-tuning approach to re-organize the data in a shared-nothing system [22].

However, while traditional parallel database technologies form the basis for the design of our system, they are not 100% fit for a scalable storage which needs to elastically scale on-demand with minimal overheads.

2.3 Cloud data and transaction service

Recently, data management facilities have been introduced into cloud storage services. In [10], Brantner et al. proposed a data management system on top of the Amazon S3 based on the client-server model. The authors also proposed a solution for transaction management in cloud databases in [20], which categorizes the application data into three types and provides a different consistency treatment for each category. However, consistency rationing at data level instead at transaction level may incur additional meta-data management overhead when the database size is large.

Lomet and Mokbel [23] put forward that a modern transactional storage can be designed as a system consisting of a transactional component and a data component, which are not tightly coupled together as in the traditional storage. G-Store, a scalable data store that provides transaction consistency for key groups, has been proposed in [13].

Amazon has developed a highly available key-value store called Dynamo [14] in cluster environment for its internal use. In Dynamo, storage nodes are organized on a ring-based distributed hash table (DHT) and each data item is asynchronously replicated on the successor storage nodes. The inconsistency between replicas of a data object is nevertheless reconciled after a period of time, thereby ensuring eventual consistency.

Compared to Dynamo, the Pnuts [12] cloud data platform from Yahoo! provides per-record timeline consistency and supports more expressive queries. As for the long term vision, the paper has identified the need for an extension of the consistency model to bundle updates, which aims to provide atomic updates to multiple records.

3. SYSTEM DESIGN

Following the principle of pay-per-use model or the notion of computing services being organized as a utility, a cloud storage system should be able to provide dynamic scalability and allow users to scale-out and scale-back on the fly based on the load characteristics. This desideratum can only be achieved when storage nodes could be easily added into or removed from the system without manually partitioning, replicating and redistributing the data.

We do not follow the client-server model that builds a database on top of an existing cloud storage (e.g. Amazon S3), in which clients retrieve data pages from S3, buffer and update the pages locally, and finally write them back to S3. Instead, we propose to construct a scalable storage system within the cloud cluster to achieve higher performance.

Figure 1 depicts the proposed architecture of ecStore. The storage system consists of three main stratums: a distributed storage layer, a replication layer, and a transaction management layer. The key challenge is how to design techniques for each layer component and make these components work together in a coherent system. Here, we will briefly describe the design of each component. The way these components work together is presented in Section 5.

At the bottom stratum, we use a tree-based structure, BATON[19], as the underlying overlay to realize a scalable range-partitioned system. In particular, ecStore organizes the storage nodes (e.g. vir-

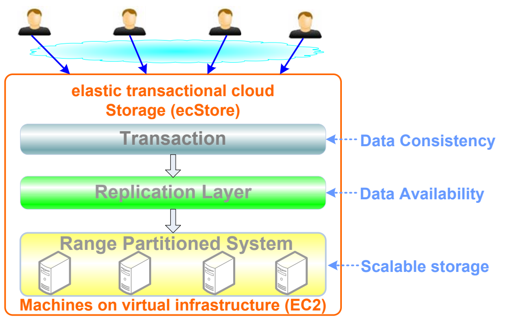

Figure 1: Stratum architecture of a transactional cloud storage

tual machines leased from EC2) as a balanced tree structured overlay and assigns a data range for each storage node. BATON has been reported to be robust and adaptive in terms of handling node joining and leaving, and offers efficient range query processing.

Note that only when we add (remove) storage nodes into (from) ecStore, due to the pay-as-you-go principle, do we need to maintain the BATON structure. In other cases, ecStore acts as a range-partitioned system. For example, to process an insert operation, ecStore looks up which storage node is responsible for the data range that covers the inserted data, and forwards the request directly to that storage node to insert the data into its local storage. Each storage node in ecStore caches enough routing information so that it can route a request directly to the appropriate node which is responsible to serve the request.

BATON, unfortunately, does not provide replication and transaction support, which is required for data management over the cloud platform to provide the required reliability and correctness, while improving efficiency. In the middle layer, we extend BATON with a two-tier partial replication strategy to provide both data availability and load balancing function in ecStore. The replication process is designed to adapt with the database workload. The detailed description of the replication layer is provided in Section 4.

The top stratum of this architecture is a transaction management module which implements a combination of multi-version and optimistic concurrency control scheme. In addition, we believe that data durability is important in cloud storages. ecStore provides this function with the recovery control technique designed to handle different types of node failures. We shall elaborate on the transaction management layer in Section 5. A summary of the techniques used in ecStore and their advantages are presented in Appendix A.

4. LOAD-ADAPTIVE REPLICATION

In this section, we design an adaptive replication strategy in ecStore that is driven by the query load and update load in the system.

4.1 Replication in BATON

Existing work on replication in peer-based data management system is not applicable to BATON directly as BATON is range based, instead of hash-based. Here, we examine three possible approaches to replicating data in BATON. A straightforward approach is to replicate data on the surrounding nodes of a storage node in the BATON tree including: parent node, children nodes, left-adjacent node, right-adjacent node, and the nodes in the inverted routing table. However, in this approach, the locations of replicas of a data item are implicitly identified by the location of the primary copy. To create or search a specific secondary copy, we first use the routing protocol to identify the node which stores the primary copy, then follow the appropriate link from that node. Nevertheless, it is complicated to identify the surrounding links of a failure node.

The second approach to replicating data in BATON is replicating based on data range. In particular, if the key of a data item belongs to a certain range we hash the range value and use the output of this hash function to determine the identity of the storage node where we can store the replica of that data item. However, BATON protocol uses the key value to start and route a search rather than identity. Furthermore, hashing breaks the order of replicated data. Although we can reduce the effect of hashing by replicating on range-basis, not per data item basis, the order of replicated data is not preserved on the BATON structure. Thus, the range query performance will be not as effective as the basic BATON scheme.

Since the above two approaches have their own short-comings, we propose the third approach called the *shift key value* scheme, which is based on key shifting. Assume the replication level of a data object is K. We define the shift key distance (δ) as the size of the key's domain (*Key_Range*) divided by the replication level. Given the initial key of a data object, we generate $K-1$ virtual keys for that data object as follow:

$$VirtualKey_i = (initialKey + i * \delta)\%MAX\_KEY \qquad (1)$$

where $i = 1 ... (K-1)$ and $\delta = KEY\_RANGE/K$,

In this way, different replicas of a data object will be stored in the same BATON structure of the primary copy but associated with their virtual keys. Hence, there are totally K copies of each certain data object replicated in the cluster. Moreover, with the above setting of the shift key distance, the replicas of a specific data object are well distributed across the storage nodes in the cluster.

It is important to note that the initial key should also be stored together with the replicas for verification during query processing. In addition, if two virtual keys fall into the same range managed by one physical storage node, the replication scheme just forwards the later replica to the right-adjacent of that node. In summary, by shifting the initial key to multiple virtual keys, this approach preserves the order of replicated data. Furthermore, this technique is efficient and elegant, and yet simple to implement.

4.2 Two-tier Partial Replication in ecStore

In the above section, we have described *where* to replicate a certain data object. The next key question is *which* data should be replicated. A straightforward approach is to replicate all data objects in the system with the same replication level, K. However, this may not be necessarily good. If K is large, the system storage and the overhead to keep them consistent can be considerably high.

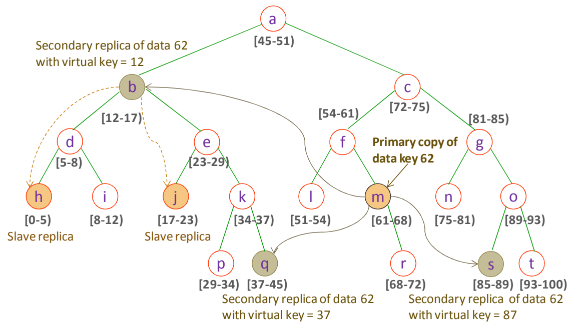

Figure 2: Two-tier partial replication

Moreover, Gray et al. [18] showed that traditional replication schemes do not scale well. The reconciliation rate grows as the squares of the number of replicas and the dead lock rate increases as the cube. Additionally, in distributed and web applications databases, the access pattern is often skewed and changes over time. Data migration is often used in range-partitioned systems to deal with skewed data distributions [16, 22]. However, under the skewed query execution load, migrating hot data from one overloaded node to another node only shuffles the hot spot throughout the system.

Therefore, we propose a two-tier replication mechanism to provide both data availability and load balancing function for ecStore. In this scheme, each data object (e.g., the data object with key 62 as depicted in Figure 2) is associated with *two kinds of replicas* - secondary and slave replicas - in addition to its primary copy. The first tier of replication is essentially a level K replication for all data objects, where K is typically a small number. The objective is to maintain a minimal number of replicas, named secondary replica, together with the primary copy for data reliability requirement.

At the second tier, popular data objects are associated with additional replicas, called slave replicas. The purpose is to facilitate load balancing for frequently accessed objects. When a primary copy or secondary replica faces a flash crowd (sudden increase in query requests), it will create slave replicas (which become associated with it) to help resolve the sudden change in the workload. In this way, the costs of replication, including replica storage cost and replica consistency maintenance cost, are always kept minimal.

4.3 Load-Adaptive Replication with Self-tuning Histogram

Figure 3 shows the process to selectively replicate data ranges that is beneficial for relieving the hot spot. While the idea of tuning replication process based on data popularity is common, most of the previous work propose to maintain the query access statistics on a per data object basis. This approach is inefficient when the amount of data in the system is large, especially for the case of cloud storage. In this paper, we propose a new approach to effectively support load-adaptive replication for large-scale data with low cost of access statistics maintenance.

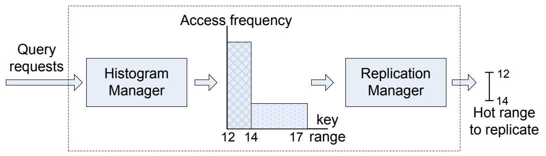

Figure 3: Replication process at a storage node

In particular, we use histogram to approximately estimate the access frequency of a data range. The boundary of a bucket forms the two ends of a data range. When the storage node serves a range query (an exact query can be considered as a range query whose two ends' value are equal), it will increase the access frequency of all the buckets whose boundaries overlap with the query range.

Consider a storage node S which manages a whole data range R. Suppose there are n buckets in the histogram and r_i is the range of bucket i, we have $\bigcup_{i=1}^{n} r_i = R$. Let Q_i be the *access frequency* of the data range corresponding to bucket range r_i. Then we define the workload $Load(S)$ of node S as the total access frequency of all bucket range r_i.

$$Load(S) = \sum_{i=1}^{n} Q_i \tag{2}$$

Since the replication process for load balancing incurs additional overhead to the system, it should not be activated in an ad-hoc manner. Therefore, we consider an approach in which a storage node will trigger the load balancing process whenever its $Load(S)$ increases by a *threshold factor* λ. In particular, we can model the values of $Load(S)$ of a storage node during its operation time as a geometric series of $L_i = c\lambda^i$ where c is a constant representing a unit amount of workload. Thus, when the $Load(S)$ of a storage node increases from L_i to L_{i+1}, we initiate the replication process for load balancing.

When the load balancing process is triggered, the storage node will choose m most popular data ranges to replicate to other lighter-loaded nodes to relieve the amount of its overloaded load. The replication speed, which determines how many replicas should be populated for the chosen data range, is load-dependent. The more load the data range observes, the more replicas created for this data range. Note that a storage node can gather the load information of other nodes in the system via tablet controller servers as in Pnuts [12], or via gossip-based protocol as in Dynamo [14]. In ecStore, we piggy-back the load information on the query processing messages and heart-beat messages sent between the storage nodes in the system. The convergence rate of the load statistics information is studied in detail in the technical report [1]. Based on the load information of other nodes, the overloaded node will choose the lightest-loaded node for replication to shed its work load.

Now, we describe how to determine the *suitable range for each bucket* in the histogram. A simple method is using equi-width histogram. Each bucket is assigned a range approximately to the ratio of the key range R managed by the storage node divided by the number of bucket n. However, this method is not flexible. If we assign a large key range for buckets, the benefit is low cost in histogram maintenance; but the access frequency estimation provided by this histogram is not accurate enough, which results in high cost in replication due to replicating a large data range containing non-popular data objects inside. On the contrary, a small key range for buckets guarantees the accuracy of access frequency statistics, thus the replication process is more effective since we only need to replicate the beneficial data ranges. However, the cost of histogram maintenance is high in this case.

In this paper, we use a self-tuning histogram to get a higher accuracy of the access frequency estimation while keeping the histogram maintenance cost minimal. The key idea of self-tuning histogram is dynamically restructuring the histogram (splitting/merging the buckets) so that the total number of buckets in the histogram is kept constant. In particular, all the buckets in the histogram are initially assigned equal bucket ranges. However, during the runtime, the buckets will diverge in the value of access frequencies maintained by them: some buckets will have much higher access frequencies than the others due to skewed access patterns.

In this case, we merge the consecutive buckets with similar frequency into a bucket with a larger data range and split the bucket with high access frequency into buckets with smaller data range. During the replication process, we only choose to replicate the data ranges maintained by small buckets because they provide more accurate access frequency estimation and the cost of replicating small data ranges is also cheaper than replicating large data ranges. The technique for self-tuning histograms was first used in [5] to maintain an estimation of the data distribution in a relational table.

However, it is common that data access patterns change over time. The slave copies of the used-to-be popular data object may no longer serve its purpose and become redundant after a period of time. Hence, we need to reduce the cost of maintaining unnecessary replicas. When a slave replica of a data range does not provide benefit to load balancing anymore, we discard it from the system. For each data range r_i managed by the bucket i in the histogram, we also maintain the data *update frequency* U_i on this data range.

When the update frequency U_i of a range is larger than the access frequency Q_i, maintaining the replicas of such a data range only incurs high cost of update propagations. In this case, we remove the information of this data range from the replica list and notify the nodes storing these replicas to discard them from the storage.

4.4 Replica Consistency Management

In cloud storages, we have to provide 24x7 data availability. Therefore, updating all copies synchronously is not suitable due to longer response time, especially when there are storage node failures or when storage nodes are located in distributed clouds [6]. Unlike the proposal in [9] which uses the pessimistic replication technique (an update needs to be reflected on all replicas before coming to effect), we employ the optimistic replication method in ecStore. In particular, the primary copy is always updated immediately, while updates to secondary (and slave) replicas can be deferred. In this optimistic replication method, the single primary copy is the data object indexed with the original key. Secondary copies of a data object form a set of replicas indexed with the virtual keys generated from the primary copy's key.

The use of optimistic replication allows us to increase the responsiveness and availability of ecStore. However, by CAP theorem [17], any distributed system faces the trade-off between availability and consistency. In this case, there is a possibility that the modification to primary copy gets lost when this operation has not been propagated to other secondary copies before the primary copy crashes suddenly. In ecStore, we adopt the write-ahead logging scheme and devise a recovery technique (cf. Section 5) to deal with the problem of "lost updates" due to different types of node failures. Hence, ecStore guarantees that updates to the primary copy are durable and eventually propagated to the secondary copies.

Note that ecStore provides *adaptive read consistency* by using the quorum model for read operations. A read request is successful only when it collects sufficient votes for a read quorum. If users require strict consistency (desire to access the latest version of a data item), then they might want to configure the value of read quorum to be equal to K, the total number of copies of that data object. In the other extreme, users can set read quorum value to 1 to speed up the read process at the cost of weak consistency (accessing older versions of data).

In our replication scheme, K is normally the number of replicas, including the primary copy and secondary replicas, for data reliability requirements. A read request will collect read votes from these copies. A write request will update primary copy first and asynchronously propagate the effect to secondary replicas. Although there could be an increasing number of replicas created by the self-tuning replication process for load balancing, to process a read request the system still collects votes from above K copies. However, if any copy (among the primary copy and secondary replicas) is overloaded, that copy will redirect the read request to one of the slave replicas attached to it. A write request is performed similarly as in the initial case with one more step: a secondary replica is responsible to update its slave replicas asynchronously. Nevertheless, this step could be executed periodically and less frequently than the initial case, for example, sending update messages to slave replicas when there is spare network bandwidth. Therefore, ecStore does not need to track the exact number of replicas corresponding to each data object.

Another notable point is that ecStore guarantees the *order of modification* done by different users to be the same on each replica in spite of the asynchronous update propagation process. As we will discuss in Section 5, ecStore is designed as a version-based storage system: each data object is attached with a transaction commit number, which is monotonic increasing in the system. Based on this version number, the replica of a data object can order the updates propagated to it correctly.

In summary, ecStore adopts the notion of BASE (BAsically available, Soft state, Eventually consistency) [25] to deal with the replica consistency issue. In this way, it does not need to implement the two-phase commit for the refresh transactions in order to bring replicas up-to-date as in the case of strong replica consistency.

5. TRANSACTION MANAGEMENT

Different parts of the system can choose different points in the spectrum between BASE and ACID. Since we desire to provide transactional semantics bundled with read/write operations in ecStore, we address the transaction management issue in this section.

5.1 Data in The Cloud

In general, data in cloud storages possesses two typical characteristics. First, it is usually sufficient to perform operations on a recent snapshot of data rather than on up-to-second most recent data [4]. Second, the locality of data accessed by transactions: this data tends to be independent between concurrent transactions of different users. It is because of the fact that in web applications, users are more likely to operate on their own data, which forms an entity group or a key group as characterized in [3, 13].

The above characteristics of cloud data drive the design of the concurrency control technique in ecStore. A hybrid scheme of multi-version and optimistic concurrency control becomes a good candidate to implement isolation and consistency for cloud-scale databases. The essence of this approach is that multiple versions of data can benefit the read-only transactions, while the optimistic method protects the system from the locking overhead of update transactions. We present more details on the rationale of combining the multi-version and optimistic scheme in Appendix B.1.

5.2 Multi-version Optimistic Concurrency Scheme

In this hybrid scheme, each transaction has a startup timestamp, which is assigned when the transaction starts, and commit timestamp, which is set up during the commit process. In addition, each data object also maintains the commit timestamp of its most recent update transaction. When a transaction accesses a data object, the most recent version of the data with a timestamp less than transaction's startup timestamp is returned. Thus, no locking overhead is incurred by the read requests.

Likewise the case of traditional optimistic control methods, at commit time each transaction is to be validated against other transactions that have committed successfully during its execution time. However, there are two main differences when we combine with the multi-version method. First, read-only transactions run against a consistent snapshot of the database, hence they can commit without the validation phase. Second, the validation phase of update transactions uses the version number of data objects to check for write-write and write-read conflicts among concurrent transactions.

In particular, an update transaction is allowed to commit only if the version of any data object observed by this transaction during the read phase is still the same when the transaction is validated, meaning that these data objects have not been updated by other concurrent transactions. By using version-based validation, there is no need to store old write-sets of committed transactions just for the purpose of validating read-set/write-set intersections. The version-based validation algorithm is presented in Appendix B.2.

With this protocol, ecStore provides Snapshot Isolation property, which is a widely accepted correctness criterion and adopted by many commercial and open-source database systems. Note that snapshot isolation is known to be not serializable in all executions [11] since it does not check the read-before-write conflicts . We discuss modifications to our protocol in Appendix B.3 to provide stricter guarantees beyond snapshot isolation. In addition, ecStore maintains a commit-number generator to ensure the global order of all committed update transactions. We present the implementation of this commit-number generator in Appendix B.4.

Now, we describe how the transaction and replication layer work together in ecStore. Read-only transactions will access the replicas for the load balancing purpose, so that the primary copy will not be the bottleneck under skewed workloads. In addition, the consistency of the replicated data observed by the read-only transactions is tunable with the quorum model. Users can set the quorum parameter to appropriate values based on their application consistency requirements. However, the update transactions are always required to access the primary copy of data, both in the read phase and write phase, to ensure that the updates in ecStore are well-behaved. Additionally, ecStore uses *mastership failover* to handle unsuccessful updates on the primary copy; if the primary copy fails during the processing of an update transaction, one of the secondary copies will be promoted to take over the mastership.

5.3 Commit Protocol

In addition to the concurrency and isolation issue as addressed above, two other desiderata are atomicity and durability, which require that all or none of the updates of a transaction come into effect and the modifications which have been confirmed with users should be persistent in the storage. These two issues are handled by the commit protocol and recovery control.

In ecStore, read-only transactions access the consistent snapshots of the database, and hence they do not need to perform the commit process. For update transactions, during the commit process, the log and commit records are stored on a local dedicated disk of the transaction coordinator and also replicated over the storage nodes in the system for durability. This information is used for the system recovery from different types of node failures. The details of the commit protocol are presented in Appendix B.5.

5.4 Recovery Control

In ecStore, a storage node can leave the system in two manners. In the case of safe departures, a storage node will notify appropriate nodes in the cluster, transfer any of it roles and data and safely leave the system. No recovery process is needed for this case. However, we need to take the case of unsafe departures into account. We divide the unsafe departure into two types of failures with different recovery treatment for each: short-term failure (due to software bugs or communication failure) and permanent failure (mainly due to hardware crashes or the virtual machine is terminated).

When a storage node rejoins the system after a short-term failure, it will check its local log store to see whether there is any log record of committed transactions coordinated by itself that has not been sent to other transaction participants. These log records will be forwarded to the involving storage nodes to finish the commit process. In this way, transactions in ecStore are durable. The effect of committed transactions are persistent even when the transaction coordinator fails before sending the commit commands to other transaction participants. Another important point is that since ecStore uses mastership failover, we also need to get the primary copy of data on the failure storage node up-to-date. Particularly, the secondary copy that previously is promoted to mastership will periodically ping to check whether the primary copy has recovered and send back the updated value to the primary copy when possible.

Now, we describe the recovery control in the case of long-term failures. When a storage node suffers from a long-term crash, another healthy node will be chosen to take care of the range index that previously is managed by the failure node. Then the recovery process proceeds in two main steps. First, the new responsible node will recover the data in that range by copying the corresponding replicated data from other nodes in the cluster. Note that we copy back the latest version of data among the secondary copies.

Second, the new responsible node will check the transaction logs replicated in the cluster to see whether there is any log record of committed transactions coordinated by failure node that has not been executed at the transaction participants. The new responsible node will perform redo operations by forwarding the log records to the involving storage nodes to materialize all the effects of the committed transactions. Hence, the update transactions in ecStore are durable even in the case of long-term crashes of storage nodes. Note that redo operations are sufficient for the long-term failure recovery process since ecStore follows optimistic concurrency control scheme, which defers all updates until commit time.

6. PERFORMANCE STUDY

In this section, we present experimental results when testing ecStore on the commercial cloud EC2. In Appendix C, we provide additional results of the range scan latency, the effect of self-tuning histogram, TPC-W benchmark, performance comparisons with other systems in an in-house cluster, and the performance of ecStore on PlanetLab. The PlanetLab environment simulates distributed clouds [6] where compute nodes are not physically close.

6.1 Experimental Setup

We deploy a prototype of ecStore and conduct experiments on a cluster of commodity machines on Amazon EC2. Each storage node in our system runs on a small instance of EC2. This instance is a virtual machine with a 1.7 GHz Xeon processor, 1.7 GB memory and 160 GB disk capacity. We use the Berkeley Database Java Edition, which implements persistent transactional B$^+$-tree, as the physical data store of each storage node.

Experimental data is synthetically generated based on a social application. A data record has a key, which is the user identity, and contains a string representing this user's friend list (a list of other user ids). A write operation will update the friend list in the record while a read operation returns this information to the users. When two users accept as friend of each other, we execute a transaction bundling four operations: two read operations to retrieve information of these two users and two write operations to update their buddy lists. The identity of a user is randomly chosen from a space of 10^9 users. The system is initially bulk loaded with $10000 * N$ records where N is the number of storage nodes in the system.

The default system size for the experiments is 18 storage nodes. Each data object is stored with replication level of 3. The threshold factor (cf. Section 4.3) to trigger the replication process for load balancing is set to 2. A workload of 1000 operations is continually submitted to each storage node in the system. A completed operation will be immediately followed up by another operation.

6.2 Scalability

In this section, we experiment the elastic scaling property of ecStore in terms of system throughput and response time when testing the system with different system sizes.

6.2.1 *System throughput*

Figure 4 shows the read throughput of ecStore with different levels of read consistency. When users require strict consistency for a read operation, the system needs to collect all replicas of a

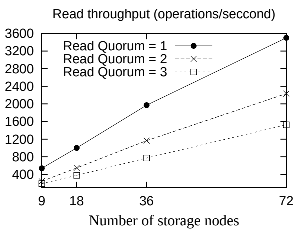

Figure 4: Read throughput with different consistency levels

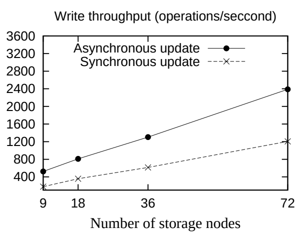

Figure 5: Write throughput with replication level = 3

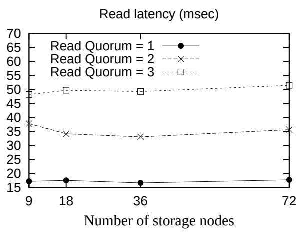

Figure 6: Read latency with different read consistency levels

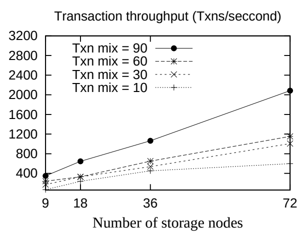

Figure 7: Transaction throughput with different read/write ratio

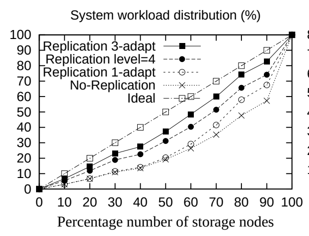

Figure 8: Distribution of load under skewed query distribution

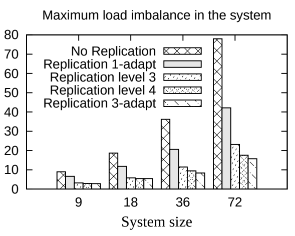

Figure 9: Load imbalance under skewed query distribution

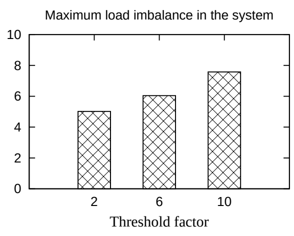

Figure 10: Effect of threshold factor to activate replication process

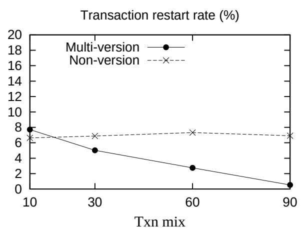

Figure 11: Transaction restart probability under skewed workload

data record and return the most recent copy to the user. The trade-off of high level of read consistency is the decrease in throughput. This figure also shows that ecStore can scale well: as the number of storage nodes increases, the aggregated read throughput also increases. The system read throughput scales almost linearly when read consistency is relaxed (by requesting a small read quorum).

In addition, the write throughput of ecStore with replication level 3 is shown in Figure 5. As expected, the pessimistic replication method is outperformed by the optimistic replication technique adopted in ecStore.

6.2.2 System response time

Figure 6 demonstrates a good elastic scaling property of ecStore, where more load can be handled by adding more storage nodes to the system. In our experiment setting, the workload submitted into the system is proportional to the system size. However, with a larger number of nodes, the system has more capacity as well. Therefore, the system response time for a read request with respect to a specific read consistency is maintained nearly unchanged with different number of storage nodes. In addition, we can observe from Figure 6 that a query which requires better read consistency, by requesting a larger read quorum, suffers from higher latency.

6.2.3 Transaction throughput

Figure 7 shows the transaction throughput of ecStore when the percentage of read-only transactions (Txn mix) varies from 10% to 90%. In this experiment, each update transaction bundles two read operations and two write operations while a read-only transaction performs only two read operations.

The multi-version concurrency control scheme guarantees that read-only transactions will always commit successfully without spending time to check data conflicts with other concurrent update transactions. Hence, the transaction throughput regarding to each system size increases together with the percentage of read-only transactions in the workload. In addition, the transaction throughput scales well under heavier read workload (Txn mix = 60% and 90%).

6.3 Dealing with Skewed Query Distribution

In this experiment, we examine the effect of replication on the system load distribution and maximum load imbalance when the query distribution is skewed. Zipfian factor is set to 1 in this test. We also study the effect of varying replication threshold factor and transaction restart probability under the skewed access pattern.

6.3.1 System load distribution

The load of a node is measured by the number of queries that has been served by this node. Ideally, a certain percentage of the number of nodes in the system is expected to serve the corresponding percentage of the total system workload. However, as we can see from Figure 8, this is not the case when the system employs no replication. Under the skewed query distribution, the only one copy of data will soon become the bottleneck, leading to the imbalanced system load distribution.

A higher replication level will balance the system load distribution since the additional replicas could help to shed the workload on the overloaded primary copy. However, the system cannot afford to replicate all data records at a high replication level due to storage cost and replica consistency maintenance cost.

This is the case where the two-tier partial replication takes its effect. As we can see from the curve labeled '3-adapt' in Figure 8, when a replication level of 3 is augmented with load-adaptive replication, the system load is well distributed, even better than using replication level 4. It is because the proposed load-adaptive replication method selectively replicates more copies for the hot data ranges to shed the workload of the overloaded node to other under-loaded nodes. In this way, we can achieve a balanced system load distribution while keeping the cost of replication minimal.

6.3.2 Maximum load imbalance

The maximum load imbalance is defined as the ratio between the load of the heaviest-loaded node divided by the load of the lightest-loaded node in the system. Figure 9 shows the maximum load imbalance of different system sizes under the skewed access pattern.

With our experiment set up, a larger number of nodes in the system will also result in a higher query workload input. The situation becomes worse when the query distribution is skewed: an increasing number of queries will be directed to one hot spot. If no replication scheme is used, the system will end up with high load imbalance. On the contrary, the load-adaptive replication implemented in ecStore quickly helps to reduce more than half of the maximum workload imbalance without the need of replicating all data in the system at high replication level.

6.3.3 *Effect of varying replication threshold*

Recall that the threshold factor determines the rate at which the system can react to changes in access patterns. Figure 10 shows the effect of varying the threshold factor on the maximum load imbalance in a system of 18 nodes with replication level 3-adaptive.

We can observe from the figure that the system has less load imbalance when this threshold is set to small values. It is because with a small threshold, a storage node can recognize its overloaded state and hot query ranges faster. Consequently, it can activate the replication process for load balancing at the right time when the system faces a flash crowd query. However, setting a small value for the threshold factor benefits the system only when the query access pattern is often skewed and changes overtime. Otherwise, constantly checking the system overloaded state and determining which data ranges to replicate could consume CPU time and affect the overall performance of the system.

6.3.4 *Transaction restart rate*

In this experiment, each transaction submits a query with range size 100 (the start value of query range is selected with Zipfian distribution), updates ten values among them and writes back to the system. This setting of large read-set and write-set together with the skewed query distribution increase the probability of transaction restart as shown in Figure 11.

However, we can observe the advantage of multi-version concurrency control. Since read-only transactions do not need to check data conflicts with other concurrent update transactions in the system, the transaction restart probability reduces when the transaction mix (the ratio of read transactions over total number of transactions in the system) increases. On the contrary, under non-versioning scheme, each transaction needs to validate against other concurrent transactions at the commit time. Therefore, in the case of non-versioning scheme the transactions almost have the same restart probability with different transaction mixes.

7. CONCLUSIONS

In this paper, we introduce ecStore – an elastic transactional cloud storage that can be dynamically deployed on the virtual infrastructures. ecStore is designed as a stratum architecture which leverages the underlying BATON[19] distributed index with two extension functions: load-adaptive replication and multi-version optimistic concurrency control. We have conducted extensive experiments in different environments, including the commercial cloud EC2, an in-house cluster, and PlanetLab. Experimental results show that the proposed load-adaptive replication method can effectively balance the system load distribution under skewed workloads.

8. ACKNOWLEDGEMENTS

The work of Hoang Tam Vo and Beng Chin Ooi were in part funded by ASTAR SERC Grant 072 101 0017. We thank the anonymous reviewers for their useful comments. We would also like to thank Kian-Lee Tan and Divyakant Agrawal for their insightful discussions and detailed comments on this work.

9. REFERENCES

[1] http://www.comp.nus.edu.sg/∼voht/TechRepVLDB10.pdf.
[2] *epiC project*. http://www.comp.nus.edu.sg/∼epic.
[3] *Google MegaStore's Presentation at SIGMOD 2008.* http://perspectives.mvdirona.com/2008/07/10/GoogleMegastore.aspx.
[4] D. Abadi. *Data management in the cloud: Limitation and Opportunities*. http://sites.computer.org/debull/A09mar/abadi.pdf, 2009.
[5] A. Aboulnaga and S. Chaudhuri. Self-tuning histograms: building histograms without looking at data. In *SIGMOD'99*.
[6] S. Agarwal, J. Dunagan, N. Jain, S. Saroiu, A. Wolman, and H. Bhogan. Volley: Automated data placement for geo-distributed cloud services. In *NSDI*, 2010.
[7] R. Agrawal et al. *The Claremont Report*. http://db.cs.berkeley.edu/claremont/claremontreport08.pdf.
[8] M. K. Aguilera, W. Golab, and M. A. Shah. A practical scalable distributed b-tree. In *VLDB*, 2008.
[9] S. Antony, D. Agrawal, and A. E. Abbadi. P2p systems with transactional semantics. In *EDBT*, 2008.
[10] M. Brantner, D. Florescu, D. Graf, D. Kossmann, and T. Kraska. Building a database on s3. In *SIGMOD*, 2008.
[11] M. J. Cahill, U. Röhm, and A. D. Fekete. Serializable isolation for snapshot databases. In *SIGMOD*, 2008.
[12] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. Pnuts: Yahoo!'s hosted data serving platform. In *VLDB*, 2008.
[13] S. Das, D. Agrawal, and A. El Abbadi. G-store: a scalable data store for transactional multi key access in the cloud. In *SOCC*, 2010.
[14] 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 *SOSP*, 2007.
[15] D. J. DeWitt and J. Gray. Parallel database systems: The future of database processing or a passing fad? *SIGMOD RECORD*, 19:104–112, 1991.
[16] P. Ganesan, M. Bawa, and H. Garcia-molina. Online balancing of range-partitioned data with applications to peer-to-peer systems. In *VLDB*, 2004.
[17] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, 2002.
[18] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. In *SIGMOD*, 1996.
[19] H. V. Jagadish, B. C. Ooi, and Q. H. Vu. Baton: a balanced tree structure for peer-to-peer networks. In *VLDB*, 2005.
[20] T. Kraska, M. Hentschel, G. Alonso, and D. Kossmann. Consistency rationing in the cloud: Pay only when it matters. *PVLDB*, 2(1):253–264, 2009.
[21] L. Lamport. Paxos made simple. *SIGACT News*, 2001.
[22] M. L. Lee, M. Kitsuregawa, B. C. Ooi, K.-L. Tan, and A. Mondal. Towards self-tuning data placement in parallel database systems. In *SIGMOD*, 2000.
[23] D. B. Lomet and M. F. Mokbel. Locking key ranges with unbundled transaction services. In *VLDB*, 2009.
[24] J. Maccormick, C. A. Thekkath, M. Jager, K. Roomp, L. Zhou, and R. Peterson. Niobe: A practical replication protocol. *Trans. Storage*, 3(4):1–43, 2008.
[25] D. Pritchett. Base: An acid alternative. *Queue*, 6(3):48–55, 2008.

APPENDIX

A. TECHNIQUES USED IN ECSTORE

Table 1: Summary of techniques used in ecStore

Problem	Technique	Advantages of the technique
Partitioning	Range partitioning	Efficient range query processing
Routing	P2P with routing cache	- No central router needed - Zero-hop routing cost
Load balancing	Data migration and load-adaptive replication	- Data migration balances the storage load - Replicating popular data ranges balances the query execution load
Replication	- Shift key approach - Two-tier partial replication - The replication process adapts with the database workload - Self-tuning range histogram for access frequency statistics	- The order of replicated data is preserved - Provide both data reliability and load balancing function - Low replica storage cost and replica consistency maintenance cost - Low cost of access statistics maintenance
Replica consistency management	Asynchronous write + quorum read following CAP/BASE principle	- Low write latency - Adaptive read consistency
Transaction management	Multi-version optimistic concurrency control for consistency across multiple keys	- Favor read-only transactions - No locking overhead
Recovery control	WAL with treatments for recovery from short-term and long-term node failures	Updates to primary copies are durable and eventually propagated to secondary copies

B. TRANSACTION MANAGEMENT DETAILS

B.1 The Rationale of Multi-version Optimistic Concurrency Scheme

It has been a consensus that locking approach may suffer from problems such as the lock maintenance overhead and the lack of general-purpose deadlock-free locking protocol. In addition, locking may be necessary only in the worst case. In environments with little resource contention, transactions could be optimistically allowed to execute and the possible conflicts among concurrent transactions will later be validated when these transactions enter their commit phase. With this optimistic scheme, there is no blocking caused by the locks, and thus the system performance is improved in query-dominant environments.

The main shortcoming of the optimistic concurrency control scheme is that transactions may be restarted unnecessarily and even a read-only transaction may have to abort due to data conflicts with other transactions committed during its execution time. Generally, there are two potential ways to reduce the data contention among the concurrent transactions in the system. One possible way is to compromise the data consistency by running queries at non-repeatable read or dirty-read isolation level. This approach, nevertheless, suffers from a certain level of serializability violation. Another promising way is to compromise the timeliness of the data by the use of versioning to avoid conflicts between the read-only and update transactions. In this method, multiple versions of data are maintained to allow queries to run against consistent snapshots of the database. Hence, read-only transactions are serializable before all concurrent update transactions and more importantly, there is no concurrency control overhead with the read-only transactions.

B.2 Version-based Validation Algorithm

Algorithm 1 : Validation process at a participant node

```
Input: read-set ($RS$) of the validating transaction $T$
Input: node $S$ where $T$ is validated
Output: $valid\_state$ of the validation process
1. $valid\_state := true$
2. for all data object $O$ in $RS(T, S)$ do
3.    if $read\_version(T, O) < latest\_version(O)$ then
4.       $valid\_state := false$
5. if $valid\_state = true$ then
6.    Send vote message to the transaction coordinator
7. else
8.    Send abort message to the transaction coordinator
9. return $valid\_state$
```

B.3 Serializable Snapshot Isolation in Distributed Environments

In [11], it has been showed that database systems with snapshot isolation can be enhanced to guarantee serializability by detecting the cyclic read-write dependency in the serialization graph in runtime and restarting one of the involving transactions. However, the implementation of this approach in distributed environments is a challenging task. Here, we discuss modifications to the concurrency control protocol in ecStore to deal with this problem.

In particular, after executing the read phase of an update transaction, the transaction coordinator will request the write locks at the time of transaction validation to prepare for the write phase. If all the locks can be obtained and the validation succeeds, then the transaction can comfortably execute the write phase, and finally release the locks. Nevertheless, if the transaction cannot acquire all the necessary write locks, it will re-execute the read phase and request for the locks that it could not get in the first time. That is, the transaction keeps pre-claiming the locks until it gets all the necessary locks, so that it can enter the validation phase and write phase safely. Hence, the global serializability is guaranteed. Note that if the database is well partitioned based on the workload so that transactions spanning multiple partitions do not occur frequently, then the distributed locks are only necessary in the worst case.

B.4 Commit-Number Generator

The benefit of multi-version optimistic concurrency control scheme does not come for free. The challenging task when implementing this hybrid scheme in ecStore is how to establish a global counter to ensure a global order of all committed update transactions. In ecStore, a certain storage node is chosen as the commit-number generator. Typically, the first storage node in the cluster will assume this role. The commit-number generator also chooses other two storage nodes in the cluster as its standby successor. In our implementation, we randomly select two nodes in the cluster that have just sent some messages (e.g. the query processing messages) to the commit-number generator. In case the commit-generator fails, one of its two successors can take over the role. Moreover, the contact information of the commit-number generator and its standby successors can be easily maintained at each storage node in the cluster. Piggy-backing this information on the periodical heartbeat messages sent between storage nodes in the cluster is sufficient.

When an update transaction successfully validates against other update transactions which have committed during its execution time, it will get a commit-number from the commit-number generator. The commit-number generator guarantees to generate monotonic values over the sequence of requests from the update transactions by increasing the value of latest generated commit-number before returning the new commit-number. Note that only update transactions need to contact with the commit-number generator after successful validation phase; hence, the commit-number generator is not the critical point of failure. In addition, the latest generated commit-number is replicated on storage nodes in the cluster and also piggy-backed on the query processing messages sent among storage nodes. In this way, each storage node can cache the recent commit-number and use this as the start timestamp for the coming transactions.

B.5 Commit Protocol

By adopting multi-version optimistic concurrency control, all read-only transactions would always succeed because they only access data in a consistent snapshot of the database. The timestamp to identify a snapshot is the commit numbers of committed update transactions in the system. Since optimistic concurrency control defers update effect until commit time, we can piggy-back its concurrency control information for validation phase on the messages of the commit protocol as illustrated in Algorithm 2.

Algorithm 2 : Commit protocol at transaction coordinator

```
1. Send validation requests to participant nodes
2. Collect vote messages from the involved nodes
3. if all validation successful then
4.     Generate commit number (timestamp) for the transaction
5.     Store and replicate the log records and commit record
6.     Send COMMIT message to the participant nodes
7. else
8.     Send ABORT message to the participant nodes
```

A notable point in the commit algorithm is that when all the transaction participants have positively voted, the transaction coordinator will store the log records and commit records to its local disk and also replicate these records over the storage nodes in the cluster for durability. This information is useful for the recovery process. When all the updates of a committed transaction have been successfully propagated to other replicas, the storage node can safely delete the log records and commit record for this transaction. Thus, the size of the log store is not large.

Note that when deploying ecStore on virtual infrastructures such as Amazon EC2, the storage nodes (virtual machines) do not have dedicated disks to store the transaction log records. However, when ecStore is set up to run directly on physical hardware (for example an in-house cluster), installing dedicated disks for storage nodes can help to increase the IO performance since ecStore can write data records and log records to separate disks.

Moreover, in web applications users tend to operate on their own data which forms an entity group and a key group as characterized in [3, 13]. By clever designing the key of data so that all data related to a user have the same key prefix which is the user's identity. Hence, data accessed by a transaction is usually clustered on a physical machine. In this case, the commit protocol is not expensive. Furthermore, there are available solutions in the literature for improving the performance of the two-phase commit such as the non-blocking Paxos algorithm [21].

B.6 Version Collection

We have proposed to integrate both replication and multi-version technique into ecStore. In fact, each technique has its own purpose. Replication helps to increase data availability of the system while multi-version scheme supports higher transaction concurrency. Therefore, a large amount of the total system storage might be merely used for replication and multi-version purpose, which reduces the storage utilization. Consequently, we need some mechanisms to prune old versions of data in the system.

A practical method to trim obsolete versions of data is the use of a version threshold. We only prune the versions whose version timestamps are more obsolete than the threshold. The value of the threshold affects the system in two ways. If we set the version threshold to be too large, then the storage is not effectively utilized. In contrast, small version thresholds might make more transactions to be aborted because they can not access the data versions which are in the snapshot before their start timestamps (all these obsolete versions have been thrown away by the pruning process). Knobs can be provided for users to tune the version threshold value.

C. ADDITIONAL EXPERIMENTAL RESULTS

C.1 Varying Size of Range Scans

In this experiment, we study the impact of varying the size of range scans on the request latency. Consider a Web 2.0 photo sharing application, e.g. Flickr, where users upload and share photos. Examples of range scan queries in this application are: finding the photos having the top ranking by users within the last 7 days, the last month, the last 4 months, etc. We generate a data set representing the metadata records for 9 million photos ordered by the date when photos are uploaded. The average record size is 200 bytes. We distribute the data set on 18 storage nodes where each node maintains the metadata of photos uploaded in 1000 days (there are 500 photos uploaded each day). Thus, the query "finding the top ranking photos within the last 4 months" requires scanning about 0.7% of the sample data set.

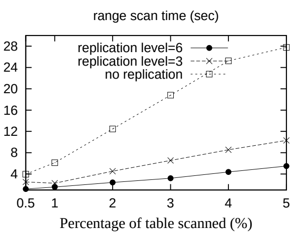

Figure 12: Range scan test on EC2

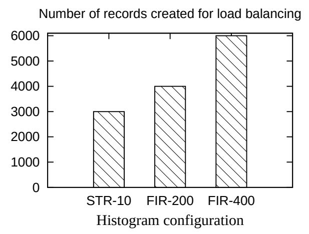

Figure 13: Number of created replicas

As shown in Figure 12, while sequentially scanning the range could be inefficient, we can improve the request latency by utilizing the existing replicas of the data and perform *parallel range scan*. In particular, we divide the range request into smaller chunks and scan these chunks at the same time but on different replicas. Since these replicas are distributed on different nodes, the completion time for scanning the whole range is improved considerably.

C.2 Effect of Self-tuning Range Histogram

We now study the effect of self-tuning range histogram in handling access patterns with flash crowd queries. In the above photo sharing application, for instance, there are more queries like "finding the highly ranked photos uploaded today" where today has some special event like the eclipse happening. In this experiment, we test the system by continuously submitting 200 queries, 60% of which are the flash crowd requests, to each storage node in a system of 18 nodes. Under this access pattern, the system load distribution is highly skewed as shown in Figure 14 when no replication-based load balancing technique is employed. Note that the data migration technique would not help in this case because it only migrates the hot data from one node to another.

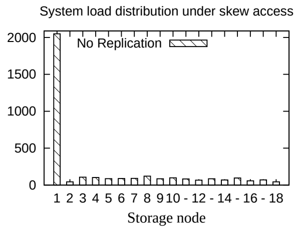

Figure 14: Load distribution without load balancing under skewed access

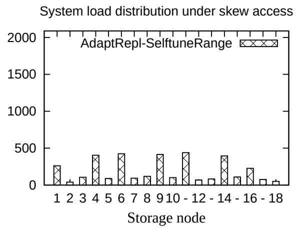

Figure 15: Load distribution with self-tune range replication

We now examine the effect of the load balancing technique with three histogram configurations: STR-10 stands for self-tuning range histogram with 10 buckets while FIR-200 and FIR-400 stands for fixed range histogram with 200 and 400 buckets respectively. Although the memory cost for maintaining FIR-200 and FIR-400 histogram is much higher than STR-10 histogram, they still could not capture the access frequency of popular queries precisely.

As a result, FIR-200 and FIR-400 populated many more records during the replication process for load balancing than STR-10 as depicted in Figure 13. Unfortunately, many of these records are "false positive", populated but do not really help much for load balancing purpose. Note that FIR-200 could not estimate the access frequency as accurate as FIR-400, thus it cannot afford to replicate data at high speed as FIR-400; otherwise leading to high storage cost and replica update cost. Therefore, FIR-200 populated less number of records than FIR-400 in the end.

On the contrary, STR-10 can capture the hot query even when its memory cost is much less than the other two histograms. Hence, STR-10 can comfortably replicate the right small number of hot queried data at high replication speed (creating more replicas each time) to quickly balance the system load. Consequently, the query execution load when using STR-10 is well distributed across the system as shown in Figure 15.

C.3 TPC-W Benchmark

We now describe the results when testing ecStore on EC2 with TPC-W benchmark, which models the on-line book store application workload. The browsing mix, shopping mix and ordering mix have 5%, 20% and 50% update transactions respectively. Shopping mix is the most representative workload. Since we only focus on storage system performance, we do not implement the application server or measure the web-interaction throughput and web-interaction response time.

Instead, we stress test the system by using a client thread at each storage node to continuously submit transactions to the system and then benchmark the transaction throughput and response time. Read-only transactions perform one read operation to query the details of a product. We implements two kinds of update transactions: adding an item to a user's shopping cart (this transaction includes 1 write operation) and performing the order request (this transaction bundles 1 read operation to retrieve the user's shopping cart and 1 write operation to the orders table). Each storage node is bulk loaded with 10000 items and customers before the experiment.

Figure 16 illustrates that under browsing mix and shopping mix, ecStore scales well with nearly flat transaction latency when the system size increases. It is because the multi-version optimistic concurrency control scheme favors read-dominant workload. As a result, the transaction throughput shown in Figure 17 scales linearly under these two workloads. In contrast, there is a decline in the transaction throughput when the ratio of update transactions increases as in the case of ordering mix.

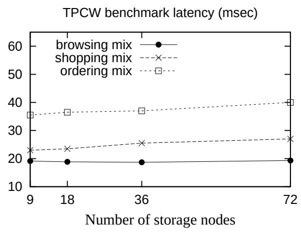

Figure 16: TPCW transaction latency

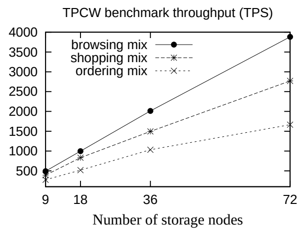

Figure 17: TPCW system throughput

Note that the transaction throughput when we test with TPC-W benchmark is higher than that of the social application in Section 6.2.3 because the transactions in this TPC-W benchmark setting bundle less number of operations than in the other experiment. Moreover, the transactions in TPC-W benchmark have more data locality when users update their own shopping carts and orders information, which are usually located on one storage node. Hence, the transactions do not spread over different storage nodes in the system.

C.4 Comparisons with Other Systems

Compared to other closed-source cloud data serving systems (such as Dynamo and Pnuts) and open-source systems (such as HBase and Cassandra), ecStore provides two *additional features*, transactional semantics across multiple keys and load-adaptive replication. The performance of the transaction management and replication technique in ecStore are studied in previous sections.

We note that these systems have been implemented to achieve **different** degrees of transaction consistency and fault tolerance. Therefore, it is not straight forward to compare these systems just on the performance of a single read or write operation. However, we shall attempt to compare ecStore and Cassandra, an open-source cloud storage that combines the idea of Bigtable and Dynamo, based on common features such as system scalability and range query processing. We also note that both ecStore and Cassandra (we use version 0.6.2 in this test) are on-going projects and the results here are based on the snapshot of the systems.

We test the two systems on an in-house cluster including 18 machines with Intel X3430 2.4 GHz processor, 8 GB of memory, 500 GB of disk capacity and gigabit ethernet. The memory buffer for the persistent B^+-tree used in ecStore and for the memtable used in Cassandra are set to 64MB, which is the default setting in Cassandra package. To support range query, Cassandra is configured to use *OrderedPartitioner*. The systems are initially bulk loaded with 144 GB of data (144 million 1KB records). Each storage node thus maintains an average of 9 GB on disk. A workload of 1000 operations is continuously submitted to each node in the system, a completed operation will be immediately followed up with another operation. The record selection for each operation follows uniform distribution.

Note that the way Cassandra and ecStore physically store data on each storage node are *different*: Cassandra uses SSTable while ecStore uses persistent B^+-tree. In particular, Cassandra buffers the write operations in a memtable and periodically flushes the data to the SSTable on disk with sequential IOs. ecStore also buffers the write operations in an in-memory B^+-tree and merges out these data to the persistent B^+-tree backing store (Berkeley Database Java Edition) after a period of time or when the buffer for the in-memory B^+-tree is full.

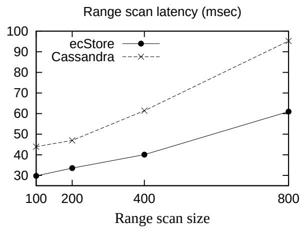

Figure 18: Range scan latency

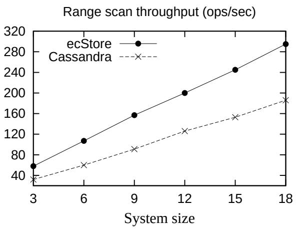

Figure 19: Range scan throughput

Range query. Figure 18 shows the performance of range can query in ecStore and Cassandra when varying the size of query range in a system of 18 nodes. We can observe that the response time of range query in both systems increases together with the query range size. In addition, ecStore has lower latency with range query because the B^+-tree in ecStore supports range query efficiently, while in Cassandra the range query processing might need to check multiple SSTable.

As a result, ecStore also has better range query throughput when testing the systems with different system sizes. Figure 19 shows this result with range query size set to 800.

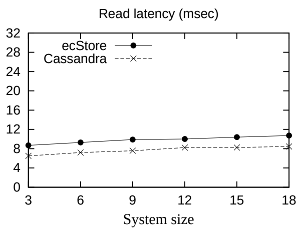

Figure 20: Read response time

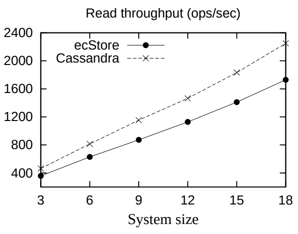

Figure 21: Read throughput

Read operation. In contrast to the results of range query performance, Figure 20 illustrates that Cassandra has better performance than ecStore in the case of read operations. It is because of the fact that Cassandra uses bloom filters to speed up read operations. The bloom filters help Cassandra efficiently identify which SSTable contains the queried record rather than traversing a long chain of intermediate nodes as in the B^+-tree used in ecStore. Thus, there is a *trade-off* on the performance of range query and exact query in Cassandra and ecStore, because of their different implementations of the physical storage at each node.

Additionally, the *elastic scaling property* of Cassandra and ecStore are well demonstrated in Figure 20. The results show that the read latency in these two systems only increases a small amount when we increase the system size. This means that more load can be handled by adding more storage nodes into the system.

Write operation. Both Cassandra and ecStore buffer the write operations in the memory and immediately return the success code to the users' requests. After a period of time or when the reserved memory is full, the effect of these operations will be materialized into the on-disk data structures (SSTable for Cassandra and persistent B^+-tree for ecStore) with a background process. Therefore, the write operations in Cassandra and ecStore have low latency, about 1 msec in the experiment.

We also note that the design of ecStore does not fix to use the persistent B^+-tree as its local storage at each node. Users can use other pluggable local data stores for different application workloads when necessary, e.g. using MySQL to handle large data objects.

C.5 Experiments on PlanetLab

In this paper, we deal with the consistency issue of replicated data and load balancing problem in range-partitioned systems, which can be applicable to storage nodes that are located across the wide-area network as in distributed clouds [6]. Therefore, we also deploy ecStore and conduct experiments on PlanetLab. The system size includes 18 nodes in the US region. In this experiment, the query workload is generated according to Zipfian distribution with the skew factor set to 1.

Percentage of ill-queries. In this test, we measure the number of failed queries with different queries rate ranging from 50 to 500 queries submitted to each node per second. We set the capacity of each storage node to 100 messages in the message-processing queue. This means that incoming messages will be dropped if the message processing queue is currently full with 100 messages already. A query request will fail if its messages are dropped during the query processing.

Figure 22 depicts the percentage of failed-queries with different query rates under the skewed query distribution. It can be observed from the figure that there is a high percentage of failure queries when no replication is exploited. Especially, the system suffers from the highest percentage (up to 27%) of failure queries when there are 500 queries submitted to each node per second.

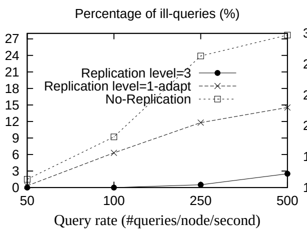

Figure 22: Percentage of ill-queries under skewed work load

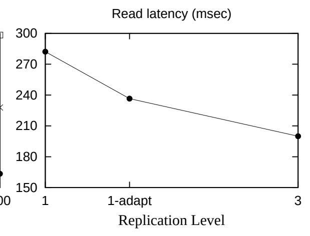

Figure 23: Latency of read operation under skewed workload

On the contrary, the system can perform well when the replication-based load balancing technique takes its effect. By maintaining a replication level of three, we can reduce the percentage of ill-queries significantly. Furthermore, the curve with key '1-adapt' in Figure 22 also shows that the load-adaptive replication reacts effectively to the skewed access pattern. The system starts with no replication, then gradually creates more replicas of popular data ranges to shed the skewed query execution to others node, thus reducing the percentage of failed-queries.

Improved query response time. In this test, we measure the latency of read operations in three settings: replication level 1, '1-adapt' and 3. As depicted in Figure 23, the workload of the skewed access pattern is dispersed to other replicas, which prevents the primary copy of a data object from becoming the bottleneck. Hence, the latency of read operations is decreased when the replication technique is employed in the system. In particular, the average query latency is significantly improved when we increase the replication level from 1 (no-replication) to 3 (there are totally three copies of data in the system). Especially, with the replication level 1 augmented with the load-adaptive technique, ecStore gradually populates more replicas of the hot query ranges and improves the query response time when compared to the case of no-replication.