Transactions with Consistency Choices on Geo-Replicated Cloud Storage

Douglas Terry, Vijayan Prabhakaran, Ramakrishna Kotla,
Mahesh Balakrishnan, Marcos K. Aguilera

Microsoft Research Silicon Valley

ABSTRACT

Pileus is a replicated and scalable key-value storage system that features geo-replicated transactions with varying degrees of consistency chosen by applications. Each transaction reads from a snapshot selected based on its requested consistency, from strong to eventual consistency or intermediate guarantees such as read-my-writes, monotonic, bounded, and causal.

1. INTRODUCTION

Cloud storage systems need to support the needs of a broad class of applications with differing performance, consistency, fault-tolerance, and scalability requirements. Our Pileus system was designed to support four desirable features:

Geo-replication. To achieve high availability and provide low latency access, data is stored in multiple servers not only within a datacenter but also across datacenters in different parts of the world. Replicas are updated lazily, thereby avoiding multiple rounds of high latency communication for write operations.

Data sharding. For scalability, large data sets are partitioned into fragments of manageable size that have their own primary and secondary replicas. The data partitioning scheme can change over time; for example, a growing fragment can be split dynamically into two smaller fragments unbeknownst to clients.

Consistency choices. Clients can read data with different consistency guarantees ranging from strong consistency to eventual consistency. Strong consistency ensures that clients see the latest data but offers the worse performance, whereas eventual consistency allows client to read stale data with optimal performance. Other intermediate consistency choices include monotonic, read-my-writes, causal, and bounded staleness. The chosen consistency can vary for each transaction and for different clients that are sharing the same data.

Transactions. Sequences of reads and writes are grouped into transactions; the read operations within a transaction access a consistent snapshot while write operations are performed atomically upon commit. The snapshot being read is determined by the choice of consistency, with strongly consistency transactions accessing the latest data while transactions with relaxed consistency accessing potentially stale snapshots.

While these features individually have been widely used and proven effective in other cloud storage services, as we discuss in more detail in the next section, we are not aware of any system that offers this combination. The challenges we faced in providing these features include defining consistency choices for transactions, selecting suitable snapshots for a given consistency while taking into account the staleness of nearby replicas, and committing transactions that write data in multiple partitions. Our key insight is that the chosen consistency determines a minimum read timestamp that constrains but does not dictate the replicas from which data can be read and that decouples consistency guarantees from transaction execution.

A large e-commerce Web application is an example of a system that could benefit from Pileus. Geo-replication provides low latency access from around the world, while data sharding enables the required scalability. Transactions simplify the application logic for handling shopping carts, user accounts, and orders. Consistency choices improve efficiency: the application could utilize read-my-write transactions to manage a user's shopping cart, strongly-consistent transactions to finalize orders, and eventually-consistent read-only transactions for data analytics of past purchases.

This paper is organized as follows. Section 2 discusses prior related work. Section 3 presents the system model and API that is visible to Pileus applications. Sections 4, 5, and 6 describe the implementation and technical challenges, starting with the system architecture then discussing consistency guarantees and finally transaction commit. Section 7 touches on additional implementation issues. Section 8 presents some performance experiments, and Section 9 concludes.

2. RELATED WORK

The design of the Pileus system borrows features from many other systems.

Key-value stores. Key-value stores with a simple Get/Put interface are quite popular as the underlying storage systems of many cloud services. These include Cassandra [31], BigTable [16], SimpleDB [3], Dynamo [24], Windows Azure Storage [14], Oracle NoSQL Database [37], and PNUTS [17][41]. Often these system provide key-range partitioning for scalability [15], where a large table can be sharded into one or more tablets as in Pileus.

Geo-distributed storage. Geo-replication is a common technique for surviving complete datacenter outages. Windows Azure Storage replicates data within a primary datacenter and lazily propagates updates to a secondary datacenter in a nearby region [14]. Pileus uses a similar architecture but allows any number of secondary datacenters in all parts of the world, and it permits clients to read from secondary servers with selectable consistency guarantees.

Megastore synchronously replicates data in a few datacenters within a region, incurring the cost of running Paxos for every write but allowing strongly consistent reads from any replica [5]. Spanner [19] similarly replicates data across regionally close datacenters.

Relaxed consistency. Relaxed consistency models for replicated data have been explored for many years [39] and have been used in a wide variety of applications including social networking, e-mail, news, recommendations, and analytics [8][44][50][28][40][46]. These include eventual consistency [45][50], bounded inconsistency [2][7], causal consistency [34][35], continuous consistency [54], and probabilistic quorums [6]. Some work has been done on expressing relaxed consistency guarantees in SQL [28]. Within the cloud, early commercial storage services, like Amazon's S3, offered only eventual consistency [10][50][4][52]. More recently, systems have added strong consistency as an option. DynamoDB and SimpleDB from Amazon [51] as well as systems from Google [27], Oracle [37], and Yahoo! [17] provide a choice of strongly or eventually consistent reads.

Influenced by prior work on session guarantees [44], Pileus broadens the set of choices to include intermediate guarantees, like monotonic and read-my-writes consistency. A companion publication describes how Pileus supports consistency-based service level agreements which permit applications to specify their consistency-performance needs in a declarative manner [47]. In this paper, we describe how Pileus defines a range of consistency choices for distributed transactions and focus on techniques for implementing transactional consistency guarantees on geo-replicated data.

Distributed Transactions. Transactions, which are widely used in the database community [12][43], have also found recent adoption in cloud storage systems, though often with limitations. Many systems, like ElasTraS [21], BigTable [16], SimpleDB [3], PNUTS [17], and Windows Azure [14], allow transactions only on individual rows of a table or on data objects that reside in the same partition on a single server. Some, like G-Store [22], limit the scope of a transaction to explicit groups. Deuteronomy [33] provides better modularity and scalability by separating transactional and data storage components, but does not allow transactions to span multiple transactional components. Some systems, like Sinfonia [1] and Granola [20], support only one-shot minitransactions. Some systems just support read-only transactions or write-only transactions. Others have proposed new transactional semantics for geo-replicated systems, such as *parallel snapshot isolation* in Walter [42].

Distributed read-write transactions on partitioned data with the traditional semantics of serializability [12] or snapshot isolation [9] are now supported in a number of cloud systems besides Pileus. Megastore uses a conventional two-phase commit protocol for transactions that span entity groups [5]. Percolator, like Megastore, stores its data in BigTable, but adds multi-row transactions with snapshot isolation using primary update and locking [38]. Calvin supports distributed transactions on replicated data using a deterministic locking mechanism along with a decentralized sequencer and transaction shipping [48]. MDCC uses a protocol based on variants of Paxos for optimistic transactions on geo-replicated partitioned data; MDCC offers only strong consistency, and it incurs a large network latency across data centers to commit each transaction [30]. In contrast, Pileus transactions provide a way to trade-off consistency for latency.

CloudTPS [53] is a distributed key-value store with distributed transactions built over a cloud storage system. CloudTPS provides two types of transactions: strongly consistent static transactions and weakly consistent non-static read-only transactions. Compared to Pileus, the consistency choices are limited in number and scope (e.g., weak consistency is available only for read-only transactions).

Snapshot isolation. Snapshot isolation [9] has been shown to fit nicely with lazy replication. Elnikety *et al.* introduced the notion of *prefix-consistent snapshot isolation* [25]. This is one of the options supported by Pileus when clients choose read-my-writes consistency, but Pileus offers broader choices of generalized

snapshots. Daudjee and Salem defined *strong session snapshot isolation* which avoids transaction inversions for a sequence of transactions within a session [23]. Similar guarantees are provided in Pileus by choosing both monotonic and read-my-writes consistency.

Timestamps and multiversion systems. Pileus uses timestamp-based and multi-version concurrency control techniques that were first explored in the 1970s. SDD-1 was the first distributed database that assigned timestamps to transactions and used these to achieve lock-free synchronization [11]. As in Pileus, SDD-1 allowed concurrent read and write transactions without requiring locking but, unlike Pileus, it preanalyzed transactions and grouped them into conflict classes.

Relaxed currency serializability has been previously proposed and evaluated as a model that allows applications to read stale data controlled by freshness constraints [13]. The target setting has a single master database and asynchronously updated caches. Although the ideal system suggested in that paper uses multi-version concurrency control, the described implementation uses traditional locking. Pileus adopts this serializability model and shows how it can be implemented in a cloud setting with geo-replication, multiple partitions, and broad consistency choices while using multi-version concurrency control.

Granola [20] uses timestamps to order its *independent distributed transactions* and assigns commit timestamps using a distributed protocol similar to that used in Pileus. Granola executes transactions in timestamp order, and a transaction may need to wait for an earlier transaction to complete, which is problematic when participants fail. Moreover, Granola's use of timestamps does not guarantee external consistency.

Recently, multiversioning has been used in Megastore [5] and Percolator [38], which exploit Bigtable's ability to store multiple rows with the same key but different timestamps. This idea is also used in Spanner, which supports read-write, read-only, and single-read transactions on old snapshots [19]. As in Pileus, Spanner's read-write transactions are assigned read times and commit timestamps make use of multiversion concurrency control, but Spanner only offers serializability with external consistency and relies on synchronized clocks.

In summary, Pileus differs from previous systems in that it supports traditional read-write transactions in a geo-replicated environment with either snapshot isolation or serializability while offering a choice of consistency guarantees ranging from eventual consistency to strong external consistency. Snapshots specified by read timestamps are selected automatically to meet the desired consistency. Pileus does not need synchronized clocks; it does not force clients to read from a single server; and it does not require synchronous replication.

3. APPLICATION MODEL

Application developers see Pileus as a traditional key-value store with high availability (through geo-replication), selectable consistency, and atomic transactions. Conceptually, a *snapshot* is a copy of the store comprising all of the data versions that existed at some point in time. Read operations (*Gets*) and write operations (*Puts*) are performed within *transactions*. Transactions, in turn, are associated with *sessions* which provide the context for certain consistency guarantees. For each transaction, the application developer chooses the desired consistency from a fixed set of options. The chosen consistency indirectly selects the snapshot that is read throughout the transaction.

3.1 Operations

The API provided to applications includes the following operations.

Put (key, value): Creates or updates a data object. Each data object consists of a unique string-valued key and a value that is an opaque byte-sequence.

Get (key): Returns the data object with the given key.

BeginSession (): Defines the start of a session. Each session can contain one or more transactions. A session denotes the scope for consistency guarantees such as read-my-writes and monotonic reads.

EndSession (): Terminates a session.

BeginTx (consistency, key-set): Starts a new transaction. All subsequent Get and Put operations that occur before the end of the transaction are included. The requested consistency (or set of consistency choices) governs the snapshot used for each Get operation. The caller also hints at the set of keys that will be read within the transaction; providing an empty or inaccurate key-set does not invalidate the consistency guarantees but could have performance consequences.

EndTx (): Ends the current transaction and attempts to commit its Puts. It returns an indication of whether the transaction committed or aborted due to conflicts with concurrent transactions. Note that transactions cannot be nested.

3.2 Transaction semantics

Get and Put methods are grouped into transactions with the following key properties:

Read isolation. All Gets within a transaction access the same snapshot of the data store (with the exception that Gets always read the results of any previous Puts within the same transaction). The freshness of this snapshot depends on the requested consistency guarantee. For example, if the transaction asks for strong consistency, then its snapshot will include all Puts that were committed before the start of the transaction, whereas an eventually consistent transaction may get a snapshot that is arbitrarily out-of-date.

Atomicity. Either all Puts within a transaction succeed at updating their data objects or none of them do. Clients never see the results of partially executed transactions; that is, if a client Gets a version produced by a Put in a previous transaction then it will also observe other Puts in that transaction.

Strict order. Each transaction is committed at a specific point in time. If one transaction commits successfully before another starts, then the results of these transactions are observed in that order.

Write isolation. If concurrent transactions update the same data object(s), then at most one of them will succeed and others will abort. Application developers must deal with aborted transactions, such as by restarting them.

3.3 Consistency guarantees

Each Get operation sees the results of some, but not necessarily all, of the prior transactions that Put the key being read. Consistency choices guarantee that certain previously committed transactions are included in the snapshot that is accessed by the current transaction. Pileus offers the following consistencies:

- *strong*: The snapshot contains the results of all transactions that were successfully committed before the start of the current transaction. This ensures that a Get returns the value of the latest committed Put to the same key.
- *eventual*: The snapshot is produced by an arbitrary prefix of the sequence of previously committed transactions. Thus, a Get can return the value written by any previously committed Put. Pileus actually provides a stronger form of eventual consistency than many systems, a guarantee that has been called *prefix consistency* [45].
- *read-my-writes*: The snapshot reflects all previous write transactions in the same session, as well as previous Puts in the same transaction.
- *monotonic*: The prefix of transactions defining the current snapshot is a superset of those in snapshots that were accessed by earlier transactions in the same session. Simply stated, the current snapshot is as up-to-date as any previously read snapshots in this session.
- *bounded(t)*: The snapshot includes the results of all transactions that were committed more than *t* seconds before the start of the current transaction. From the application's perspective, a transaction is committed at the time that the EndTx call returns with a success indication.
- *causal*: The snapshot includes the results of all causally preceding transactions. Specifically, consider a graph in which the nodes are transactions and a directed edge from T1 to T2 indicates that either (a) transaction T2 read a value written by transaction T1 or (b) transaction T2 was executed after transaction T1 in the same session or (c) transactions T2 and T1 performed a Put on the same key and T2 was committed after T1. The causally consistent snapshot reflects all transactions for which this graph contains a directed path to the current transaction.

Different consistency choices will likely, but not always, result in different performance for transactions. In particular, since consistency guarantees place restrictions on the acceptable snapshots that are accessed by a transaction, they limit the set of servers that can process a Get operation. Limiting the set of suitable servers indirectly increases the expected read latency since nearby servers may need to be bypassed in favor of more distant, but more up-to-date replicas.

Choosing weaker consistency guarantees can improve the performance not only of read-only transactions but also of read-write transactions. But, an interesting interplay exists between a read-write transaction's consistency and its ability to commit. Applications that tolerate staleness can read from snapshots that are farther in the past, thereby improving the availability and performance of Gets but also extending the time interval over which transactions can conflict. That is, older snapshots increase the likelihood of concurrent Puts and subsequent aborts, though this effect can be offsetby reduced execution times due to faster read operations (as shown in Section 8). When data objects are infrequently updated, aborts may be rare even for transactions that choose eventual consistency. Application developers should choose a consistency based on both the correct operation of their application and the cost of aborted transactions.

4. SYSTEM ARCHITECTURE

To provide consistency-aware transactions on data that is partitioned and geo-replicated, Pileus uses a client-server architecture. Servers manage multiversioned data, implement Get and Put operations, participate in transactions, and replicate data among themselves. Clients maintain state for active sessions and transactions, they track the staleness of various serv-

ers, they decide where to send each Get operation, and they measure and record the roundtrip latencies between clients and servers.

4.1 Replication and partitioning

Each data object is replicated on multiple servers within a datacenter and across geographically distributed datacenters. To ensure elastic scalability, no server stores the complete set of data objects. Objects are horizontally partitioned according to key ranges. Objects within the same key-range are replicated on the same sets of servers.

Applications access servers through a client library that routes Get and Put operations to appropriate servers based on the chosen consistency. The client library is aware of which servers store which partitions, but this information is hidden from applications so that data can be moved or repartitioned transparently. For example, clients may get information on how keys are partitioned from a cloud directory server.

4.2 Primary update

One datacenter is chosen as the *primary* site for each data partition. For simplicity, the rest of this paper refers to a single primary server for each data object/partition, though in practice a set of primary servers could form a highly-available primary cluster using a protocol like chain replication [49]. A primary server participates in transactions that perform Put operations for keys in its partition's key-range.

After a write transaction has been committed, each primary involved in the transaction asynchronously propagates new data object versions to *secondary* replicas. The actual details of the replication protocol are unimportant. The only assumption is that updates are transmitted and received in timestamp order. Secondary servers *eventually* receive all updated objects along with their commit timestamps (described in Section 6). No assumptions are made about the time required to fully propagate an update to all replicas, though more rapid dissemination increases a client's chance of being able to read from a nearby server.

4.3 Multiversion storage

Servers, whether primary or secondary replicas, manage multiple timestamped versions of data objects. The Put operation creates an immutable version with a given timestamp, while the Get operation returns the latest version before a given time. As discussed later, this allows snapshot isolation using multiversion concurrency control without requiring locking [36].

Each server stores the following basic information:

- *key-range* = the range of keys managed by the server.
- *store* = set of <key, value, commit timestamp> tuples for all keys in the range.
- *high timestamp* = the commit timestamp of the latest transaction that has been received and processed by this server.
- *low timestamp* = the time of the server's most recent pruning operation.

Additionally, primary servers store:

- *clock* = a logical clock used to assign timestamps to transactions when they commit.
- *pending* = a set of <put-set, proposed timestamp> pairs for transactions that are in the process of being committed.
- *propagating* = a queue of <put-set, commit timestamp> tuples for recently committed transactions that need to be sent to secondary replicas.

Since primary servers assign increasing commit timestamps to transactions and the replication protocol transfers updated objects in timestamp order, a single high timestamp per server is sufficient to record the set of updates that it has received. When a server receives and stores new versions of the data objects that were updated by a transaction, it updates its high timestamp to the transaction's commit timestamp and records this same timestamp with each object. If no write transactions have committed recently, the primary server sends out periodic "null" transactions with its current time causing secondary servers to advance their high timestamps.

Servers are allowed to discard old versions after some time to free up storage space. Each server periodically, and independently, prunes its data store by deleting any versions that were not the latest version as of a selected time (as discussed in more detail in Section 7.2). The last pruning time is recorded as the server's low timestamp.

The client-server protocol includes a Get operation that takes a read timestamp in addition to a key. Upon receiving a Get(key, timestamp) request, a server checks that it manages the given key and that the requested timestamp is between its high timestamp and low timestamp. Essentially, the interval from low timestamp to high timestamp indicates the set of snapshots that can be accessed at this server. If the requested read timestamp does not fall in this interval, then the Get request is rejected by a secondary server; a primary server always accepts Get operations as long as the timestamp exceeds its low timestamp (though the request may need to wait for in-progress transactions to be assigned commit timestamps). If the check succeeds, the server replies with the most recent version of the object with the requested key whose timestamp is no later than the requested time.

Included in the response is the object's timestamp, the timestamp of the next highest version of this object (if known to the responding server), and the server's high timestamp.

5. READING SNAPSHOTS

To ensure that each Get operation within a transaction accesses data from the same snapshot, the client selects a *read timestamp* for the transaction. All Gets are then performed using this timestamp. The *minimum acceptable read timestamp* is determined by the transaction's requested consistency guarantee, the previous object versions that have been read or written in the client's current session, and the set of keys used in the transaction's Gets (as passed to the `BeginTx` method).

Although we have defined and implemented a set of consistency guarantees that have been proven to be useful in practice, this set could easily be extended. To add a new consistency guarantee, one simply needs to write a method that returns the minimum acceptable read timestamp. This minimum timestamp serves as a narrow interface between the components of the Pileus system that understand consistency choices and the components that implement transactions.

The following sections describe how the minimum acceptable read timestamp is determined for each of our consistency guarantees and then how clients select a particular read timestamp for a transaction. For any consistency guarantee (except strong), there's a trade-off in selecting a read timestamp: choosing more recent times produces more accurate data, which applications desire, but choosing older times results in a broader set of servers that can potentially answer each Get, thereby reducing average response times.

5.1 Strong consistency

To ensure strong consistency, the minimum acceptable read timestamp is the maximum timestamp of all versions stored at primary servers for the set of keys that are expected to be read in the current transaction. This timestamp guarantees that each Get accesses the latest version of the object that existed at the start of the transaction. Although an initial round of messages is generally required to determine the minimum acceptable read timestamp for a strongly consistent transaction, optimizations are possible in some cases.

If the transaction involves a single key, as hinted in the key-set that was passed to BeginTx, then the read timestamp can be set to a special value indicating that the primary server should simply return its latest version. This case efficiently handles single-Get transactions as well as simple read-modify-write transactions executed at the primary. Similarly, if all of the keys involved in the transaction belong to the same partition, and hence have the same primary, then the first Get operation can be sent to the primary requesting its latest version. The primary will respond with its current high timestamp (i.e. the timestamp of its last committed transaction), and this is used as the read timestamp for subsequent Gets.

Once a client establishes the minimum acceptable read timestamp for a strongly consistent transaction, the transaction's Gets might be performed at servers other than the primary. For example, suppose that a data object is updated once per day. At the start of a strong consistency transaction, the client contacts the primary server for this object and determines that the object was last updated 14 hours ago. The actual Get operation could then be sent to any secondary server that is no more than 14 hours out-of-date, such as one that's nearby, is more lightly loaded, or has a higher bandwidth connection.

Computing the minimum acceptable timestamp for strong consistency requires knowing, at the start of the transaction, the set of Gets that it will perform. In some cases, predicting the set of keys being accessed may be impractical, such as when the key for a Get depends on user input or data retrieved by some prior Get within the same transaction. If a key-set is not provided, then the client assumes pessimistically that all primaries are involved in the transaction and retrieves high timestamps from each primary. Another option, however, is to assume optimistically that the transaction involves a single partition (the one accessed by the first Get) and then abort the transaction if this assumption turns out to be false. The client can easily restart the aborted transaction with the newly discovered key-set.

What if the provided key-set is wrong? If the key-set contains keys that are not actually read, that will not violate correctness; it will perhaps unnecessarily reduce the choice of servers. But if the key-set is missing some key used in a Get operation, this could be a problem. For strong consistency, it is only a problem if the key has a different primary than any of the keys in the presented key-set. In this case, the minimum acceptable read timestamp may have been set lower than the latest committed transactions for this key, and the transaction must be aborted.

5.2 Relaxed consistency

Other consistency guarantees have the advantage over strong consistency that the minimum acceptable read timestamp can be determined based solely on information maintained by the client. No client-server communication is required at the start of a transaction.

The read-my-writes, monotonic reads, and causal consistency guarantees requires the client to maintain per-session state that records the commit timestamp of previous Puts in the session as well as the timestamps of versions returned by previous Gets. For the read-my-writes guarantee, the minimum acceptable read timestamp is the maximum timestamp of any previously committed Puts in the current session that updated objects being accessed in the current transaction. For monotonic reads, the read timestamp must be at least as large as the maximum recorded timestamp for previous Gets in this session.

If the key-set is unknown at the start of the transaction, then, for monotonic reads, every previously read version is considered when choosing the minimum acceptable read timestamp. For read-my-writes, every previously written version is relevant. Clearly, this has performance and availability consequences since it may set the read timestamp higher than needed, and hence rule out servers that might otherwise be accessed; but there are no correctness concerns.

Even if the key-set is missing keys that are used in Get operations, the transaction may be allowed to continue in some cases. For monotonic reads, the transaction need not abort as long as the read timestamp exceeds previously read versions for the missing key. Similarly, for read-my-writes, the chosen read timestamp must be later than the previously written version of the missing key.

Pileus ensures that if one version causally precedes another then the first version will be assigned a lower commit timestamp (as explained in Section 6.2). Thus, causal consistency can be guaranteed conservatively by ensuring that the read timestamp is at least as large as the timestamp of *any* object that was previously read or written in the session.

If an application chooses bounded staleness, the minimum acceptable read timestamp is selected at the start of the transaction (or when it issues its first Get). Gets that are performed later in the transaction may return data that is older than the staleness bound since they need to read from the same snapshot. In theory, implementing bounded staleness is easy; the smallest acceptable read timestamp is simply the current time minus the desired time bound (plus the maximum possible clock error), assuming that clients and servers have approximately synchronized clocks, as in the Spanner system [19].

In Pileus, providing staleness bounds is a bit more challenging because the timestamps assigned to transactions, and hence to object versions, are obtained from logical clocks. Our solution is to have clients maintain a mapping from real time to each primary

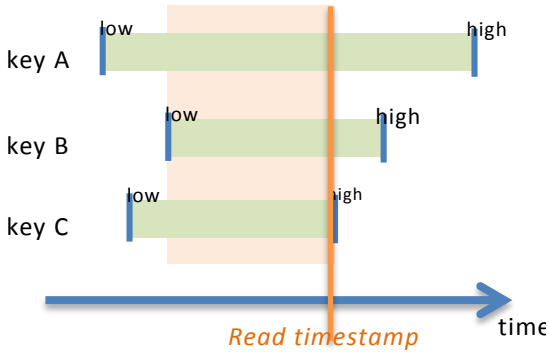

Figure 1. Selecting read timestamp for transaction that accesses keys A, B, and C residing in different partitions on different servers.

server's logical clock. Whenever a client accesses a primary server to perform a Get or Put operation, the server returns its current high timestamp. The client records its current time (taken from the local real-time clock) along with the server's high timestamp (which indicates the server's current logical clock). This mapping table need not be large since it does not need to contain a record for every interaction with the primary. One entry per minute, for instance, is probably sufficient. To determine the minimum acceptable read timestamp for a given staleness bound, the client simply subtracts that bound from its current clock and then uses its mapping table to find the closest logical clock value. A key advantage of this scheme is that it does not require any form of clock synchronization.

For eventual consistency, any read timestamp is acceptable. This also holds for the first transaction in a session when monotonic, read-my-writes, or causal consistency is requested. However, reading from time zero, while technically not a violation of the consistency guarantee, would be ineffective for applications and their users. Choosing the current time also would not be wise since it has the same effect as choosing strong consistency with the same limitations on performance and availability. The next section discusses how clients pick read timestamps that cause low response times, when possible, while reading data that is not overly stale.

5.3 Selecting the read timestamp

From the target consistency guarantee, the client obtains the minimum acceptable read timestamp. Any read timestamp that exceeds this minimum could be selected. In our current implementation, the client selects the largest read timestamp that allows it to read from the closest servers that are sufficiently up-to-date. This provides the freshest data that can be

obtained without compromising performance or consistency.

For each key in the transaction's key-set, the client selects the closest server (according to its own round-trip latency measurements) whose last known high timestamp is greater than the minimum acceptable read timestamp. It then sets the read timestamp to the lowest known high timestamp for any of these servers. This read timestamp should both fall within the interval of snapshots that are accessible at this set of servers and also comply with the minimum acceptable timestamp for the desired consistency. Figure 1 depicts an example in which a transaction performs Gets on keys A, B, and C, and each of these keys belongs to different partitions with different closest servers.

Note that the client's cached information about the high timestamps of various servers is almost certainly outdated since it was obtained from previous Get and Put operations. However, a server's high timestamp is strictly increasing, and so even stale information can be used to make conservative decisions about the server's suitability.

If servers aggressively discard old versions of data objects (as discussed in Section 7.2), one server's low timestamp may exceed another server's high timestamp. Thus, the set of nearby servers for a transaction's key-set may have no suitable read timestamp in common. In this case, the client must try more distant servers. One strategy is to select the set of servers with overlapping high-to-low timestamp ranges that minimizes the average round-trip latency. An acceptable read timestamp can always be found since, at the very least, clients can choose to read from primary servers using the current time as the read timestamp.

During execution of the transaction, the chosen read timestamp is used for all Get operations, and such operations are sent to servers based on the client's current information about the state of the system. Several relevant events can happen during the lifetime of a transaction: secondary servers can receive new updates from the primary, servers can prune old versions, servers may fail and become unavailable, network loads may change causing variations in the client-server latencies, and clients can update their local state based on information obtained through normal operations or periodic pings. All of these may cause the client to direct Gets to servers other than those used to select the transaction's read timestamp. For example, a nearby server that initially was unable to satisfy a bounded staleness guarantee may have recently replicated data from the primary, and hence can now be used for future Gets.

The selected read timestamp can be adjusted during the course of a transaction while still preserving snapshot isolation. This timestamp could be set as low as the maximum timestamp of any object that has been read (as long as it remains above the minimum acceptable read timestamp) or set just below the minimum next version timestamp returned by all prior Gets. Moving the read timestamp within this interval would not affect the Gets that have already been performed in the transaction but could possibly alter the selection of servers for future Gets.

If the BeginTx call has an empty or incomplete key-set, then the client can still select a read timestamp, but may make a less desirable selection. As an example, suppose that the transaction's key-set includes key A but not key B, and that the application requested eventual consistency. The closest server for key A has a recent high timestamp, and the client selects this as the transaction's read timestamp. The client Gets the latest value for the object with key A. Now suppose a Get for key B is performed. For this key, there may be few servers that store the selected snapshot, perhaps just the primary, although any of key B's servers could meet the eventual consistency guarantee.

It is possible that the selection of the read timestamp for the current transaction affects later transactions in the same session. For example, suppose that the current transaction only performs one Get for key A and that the closest server for this key's partition is the primary. The client will choose to read the latest version from the primary. Now, suppose the next transaction performs Gets on keys A and B and requests monotonic consistency. This transaction is forced to use a read timestamp at least as great as that of the previous transaction. But the only server for key B that is sufficiently up-to-date may be far away. If the first transaction knew that about the second, it could have made a more globally optimal selection of its read timestamp. To avoid this problem, applications can include keys read by later transactions in the key-set presented for the current transaction.

6. COMMITTING TRANSACTIONS

Read-only transactions automatically commit with no extra processing, while read-write and write-only transactions require extra processing when EndTx is called. *Commit coordinators* assign timestamps to transactions, validate transactions at commit time, and manage the participation of all of the primary servers holding data being updated. In our current system, the commit coordinator is one of the primary servers involved in the transaction, though any machine, including the client, could play this role.

6.1 Sending buffered Puts

While the Gets within a transaction are performed as they are issued and may be sent to different servers, each Put operation is simply buffered at the client. Puts, therefore, involve no communication with primary servers until the end of the transaction. The new objects are visible to Gets within the transaction, but not to others until the transaction commits. If a transaction aborts, its Puts are simply discarded; no updates need to be undone at any servers.

When a transaction begins, neither clients nor coordinators need to know whether the transaction is read-only, write-only, or read-write. The client makes this determination when EndTx is called and then invokes a commit coordinator if any Puts were buffered.

At the end of a transaction, the client sends the set of Puts as an atomic batch to a commit coordinator along with the read timestamp that was used by the transaction. The commit coordinator splits the transaction's Put-set into multiple disjoint sets of Puts for each partition and sends each subset to its designated primary. Only primaries who store data objects being written are participants in the transaction commit process.

6.2 Assigning commit timestamps

The commit coordinator assigns a commit timestamp that orders the transaction relative to other transactions. The coordinator starts by asking primaries to propose a timestamp; this request is included with the Put-set that is sent to each participant. Upon receiving such a request, a server proposes its current time (taken from its local logical clock) and increments its logical clock. The server adds the Put-set and its proposed timestamp to its *pending transactions* list.

From the set of proposed timestamps, including its own time, the coordinator chooses the maximum timestamp (for reasons described below) as the commit timestamp for this transaction. Note that if the primary servers had perfectly synchronized clocks, then the coordinator could merely use its own time and avoid the round of proposals. But, relying on tight synchronization of clocks among servers in different data centers perhaps run by different organizations can be problematic; instead, servers maintain logical clocks from which all timestamps are obtained [32].

When a primary server learns of the chosen commit timestamp, it advances its logical clock to exceed this time, if necessary, so that future transactions will be assigned higher commit timestamps. The commit coordinator might also advance its logical clock upon receipt of a commit request from a client. In particular, clients keep track of the largest commit timestamp for transactions that they have performed or observed. This largest timestamp is included in the commit request, and the coordinator increases its clock, if necessary, to guarantee that the commit timestamp for the current transaction exceeds those of causally preceding transactions.

One subtlety is that different coordinators may assign the same commit timestamp to concurrent transactions that involve overlapping primary servers. Servers deterministically order such transactions using the identities of their commit coordinators.

6.3 Validating transactions

Given both the read and commit timestamps for the transaction, the coordinator must *validate* the transaction to check whether it can commit or must be aborted based on other concurrent transactions. Basically, the read timestamp serves as the start time for the transaction and the commit timestamp as its ending time. To ensure snapshot isolation, the transaction must not have updated any objects that were also updated by other transactions that overlap in time [9].

Specifically, the coordinator asks each primary server whether it holds any versions of objects in the transaction's Put-set that have timestamps between the transaction's read timestamp and commit timestamp. Optionally, to enforce serializability, this check could include the transaction's Get-set as well. Only if these checks succeed at all participating primaries is the transaction allowed to commit. Write-only transactions, consisting solely of blind Puts, never abort.

6.4 Writing new versions

After a transaction has been validated, the commit coordinator writes a commit record to stable storage that includes the transaction's commit timestamp and set of Puts. To safeguard the durability of the transaction, the coordinator also writes the commit record to one or more other servers who can inform participants of the transaction's outcome if the coordinator fails. At this point, the coordinator replies to the client that the transaction has been committed.

The coordinator informs the other participating primary servers of the transaction's assigned commit timestamp (or that the transaction aborted). This can be done lazily. Each primary server performs the Puts for a committed transaction without coordinating with other servers or waiting for pending transactions (which might receive earlier commit timestamps). In particular, new versions are created for each Put and marked with the transaction's commit timestamp.

These Puts are placed on the primary server's propagating queue and eventually propagate to secondary servers. The primary can send Puts to a secondary when it has no pending transactions that might be as-

signed earlier commit timestamps. Secondary servers immediately apply any Puts that they receive and advance their high timestamps to the commit timestamp.

Each primary server sends the coordinator an acknowledgement when its work is completed and removes the transaction from its pending list. To ensure that the transaction is fully performed, the coordinator periodically retransmits it to servers that have not responded.

Observe that this commit protocol does not require any locks to be held. For the most part, Gets and Puts do not interfere with each other. That's the beauty of multiversion concurrency control: each transaction writes its own timestamped versions and reads from a snapshot.

6.5 Gets and pending transactions

If a primary server receives a Get request for a timestamp that exceeds its current logical clock and there are no pending transactions, the server bumps its clock to the requested time and proceeds to answer the Get knowing that future transactions will be assigned higher commit timestamps. However, there is one situation in which Get operations (and validation checks) may need to wait for a pending transaction. This occurs if a primary server receives a request to read an object with a given key and (a) some pending transaction includes a Put for that object and (b) the read timestamp is greater than the time that was proposed for the pending transaction. In this case, there's a chance that the pending transaction may be assigned a commit timestamp that is earlier than the read timestamp and hence should be visible to the Get. Note that pending transactions with later proposed times can be ignored since the commit time for such transactions must be at least as large at the proposal (which is why coordinators choose the maximum proposal).

As an example, suppose a committed transaction involving two or more partitions has been completed at one primary, e.g. the coordinator, but not the others. The client is informed that the transaction was committed, and this client immediately issues a strong consistency Get request. This Get will be assigned a read timestamp that is greater than the previous transaction's commit timestamp, as desired to ensure external consistency. If the client attempts to read from a server that has not yet processed the previous transaction, it must wait. This should not happen often, and, when it does, the delay should be small. The system could avoid such situations by informing a client that its transaction has committed only after all of the participating primary servers have performed their updates. But that would require another round of communication between primary servers for each transaction and increase the time-to-commit for all clients. Instead, we opted to accept an occasionally delayed Get operation.

7. OTHER IMPLEMENTATION ISSUES

7.1 Client caching

Clients can freely cache any version of any data object since versions are immutable. A client can cache and discard versions without the servers' knowledge. It can even cache older versions of objects without caching the later versions, or vice versa.

For each Get operation performed at a server, the client receives the data along with the version's timestamp. Additionally, the server returns the timestamp of the next later version (if any). This precisely tells the client the valid time interval for the given version. If the version returned is the latest version, the server returns its current high timestamp. The client knows that the returned version is valid until at least this time (and maybe longer). The client caches the retrieved data along with its valid timestamp interval.

When performing a Get, the client needs to determine whether a cached version's timestamp is less than or equal to the read timestamp and whether it is the latest such version for the given key. In short, the client simply checks whether the read timestamp is in the valid interval for one of its cached versions of the desired key.

A client could consult its cached data when selecting the read timestamp for a transaction. For example, when reading a single object (or even multiple keys in a key-set), the client might as well choose a read timestamp that allows it to read from its cache (if this meets the consistency guarantee). Essentially, the local cache is treated as another (partial) replica.

7.2 Pruning old snapshots

Servers must prune old versions of data objects to reduce their storage usage. But choosing the best pruning policy is difficult. If servers are overly aggressive about discarding versions, then a transaction may be prevented from reading local data since the nearby servers may no longer hold snapshots for the desired read timestamp or two servers may have no overlapping times that could be used as a common read timestamp. If servers are overly conservative about pruning, then they may store old versions that are never read, wasting storage space.

Servers discard snapshots in the following manner. When a server increases its low timestamp, it discards any versions that are not the most recent version as of

the current low timestamp; that is, the server retains exactly one version with an older timestamp. These versions comprise the snapshot corresponding to the server's low timestamp.

In our current implementation, servers keep their low timestamp a fixed distance behind their high timestamp. This simple scheme requires no coordination between servers and, given knowledge of the update rates, it bounds the amount of storage used. Also, different servers advance their low timestamps at about the same rate, and so it should be possible to find common read timestamps. However, it may still retain versions unnecessarily.

Observe that a read timestamp will never be chosen that is less than the high timestamp of the most out-of-date server (using the strategy presented in Section 5.3). So, an alternative scheme is for all servers to exchange their high timestamps. Each server could then independently prune versions by advancing its low timestamp to the oldest high timestamp. We decided against this approach since it requires servers to know about each other and to exchange messages, even servers for different keys and different partitions.

One concern is that servers might prune a snapshot that is currently in use by some long-running transaction. After a read timestamp has been selected, all of the servers for a key could advance their low timestamp beyond the read timestamp so that no future reads can be performed, causing the transaction to be aborted.

To prevent this problem, servers could advance their low timestamp based on the read timestamps they receive in Get requests. For example, they could choose the lowest read timestamp in the past five minutes. This requires no coordination among servers and avoids pruning snapshots in use by transactions that are actively reading data. Moreover, keys that are regularly accessed in the same transaction will have their servers naturally coordinate their pruning, while the servers for keys that are accessed independently will prune independently.

8. PERFORMANCE

For our experiments, we ran Pileus on a cluster of machines in a geo-distributed system. Our setup consisted of multiple sites separated by long-distance links, with a Pileus primary server running on one site in China and secondary servers running on other sites in California. The measured California-to-China round-trip latency averages 164 ms.

The key trade-off in Pileus involves transaction latency vs. commit rate: reading from an older snapshot corresponding to a weaker consistency guarantee reduces the expected transaction execution time but increases the probability of the transaction aborting. Effectively, choosing an earlier read timestamp results in a longer span between the transaction's start and commit, which increases the chance of conflicting with other concurrent transactions.

Figure 2 shows the commit rate of Pileus transactions for different consistency guarantees as we reduce contention by increasing the total number of keys in the system. Each transaction runs at a secondary; it reads three randomly selected keys and then writes them back. As shown in the graph, choosing a weaker consistency guarantee has a negligible effect on the commit rate for moderate to low levels of contention, i.e. if the system has 10K or more keys. In this experiment, the secondary receives updates from the primary every 500 ms.

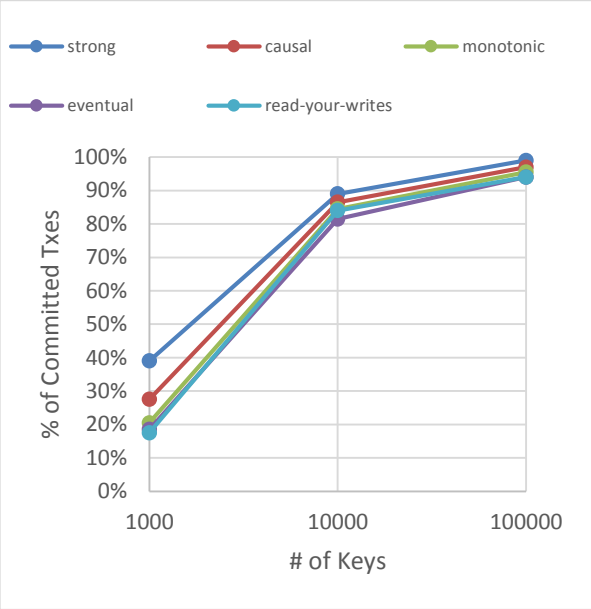

Figure 2. Effect of contention on probability of commit for different consistency choices.

For read-only transactions, choosing a relaxed consistency model can result in a dramatic improvement in performance. Strong consistency provides an upper bound on transaction latency since it requires the client to always read from the primary. Eventual consistency provides a lower bound since the client can always read from the secondary. Intermediate choices, such as causal and read-my-writes, permit a client to read from the secondary if it synchronizes frequently with the primary, but default to reading from the primary if the secondary is not sufficiently up-to-date. Prior work has shown that strong consistency transactions can be as much as seven times the cost of eventual consistency within a datacenter [29]. Here, we

explore the transaction cost with geo-replication and observe that the difference can be two orders of magnitude.

Figure 3 shows the execution time of read-only transactions for two different workloads. In both workloads, a client at the secondary site issues strongly consistent read-write transactions interspersed with read-only transactions requesting various consistency guarantees, but we only measure and report the execution times of the read-only transactions. As before, a client at the primary site issues read-write transactions in parallel. The workloads differ in the nature of their read-write transactions. Workload #1 is a high-contention workload: the client at the secondary site reads and writes the same three keys in each transaction, while the primary site accesses 1K keys randomly. Workload #2 is low-contention: both the primary and the secondary site clients read and write randomly selected 3-key sets from a total of 1K keys.

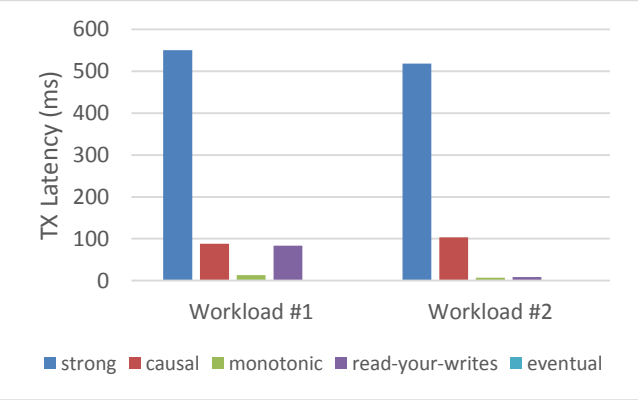

Figure 3. Latency for read-only transactions for a high-contention workload (#1) and a low-contention workload (#2).

Note in Figure 3 that all of the relaxed consistency choices result in substantially faster read-only transactions compared to strong consistency. The bars for eventually consistent transactions are barely visible in the figure since they execute in a few milliseconds. For Workload #1, read-your-writes provides equal latency to causal consistency since the keys read are always recently modified, and hence must often be read from the primary. For Workload #2, read-your-writes is equivalent to eventual consistency, since the keys read by a read-only transaction are rarely modified in the recent past. In both workloads, monotonic consistency is slightly slower than eventual consistency since the interspersed strongly consistent transactions continuously raise the last read timestamp.

9. CONCLUSIONS

Transactions consisting of a sequence of read and write operations have proven to be valuable to application programmers. Pileus supports transactions that not only are performed atomically, even on partitioned data, but also are governed by the relaxed consistency models common in geo-replicated cloud storage systems. Its design shows how to combine transactions with consistency choices to provide snapshot isolation and relaxed currency serialization through multiversion concurrency control, consistency-driven timestamp selection, and cross-primary commit.

10. ACKNOWLEDGEMENTS

Hussam Abu Libdeh participated in early meetings leading to the design of the Pileus system. We thank Phil Bernstein for his careful reading of an earlier draft of this paper and his advice on commit protocols. Discussions with Alan Fekete and David Lomet were instrumental in prompting us to write this paper.

11. REFERENCES

[1] Marcos K. Aguilera, Arif Merchant, Mehul Shah, Alistair Veitch, and Christos Karamanolis. Sinfonia: A new paradigm for building scalable distributed systems. *ACM Trans. Comput. Syst.* 27 (3), November 2009.

[2] R. Alonso, D. Barbara, and H. Garcia-Molina. Data caching issues in an information retrieval system. *ACM Transactions on Database Systems* 15(3):359-384, September 1990.

[3] Amazon SimpleDB. http://aws.amazon.com/simpledb/

[4] E. Anderson, X. Li, M. Shah, J. Tucek, and J. Wylie. What Consistency Does Your Key-value Store Actually Provide? *Proceedings Usenix Workshop on Hot Topics in Systems Dependability*, 2010.

[5] Jason Baker, Chris Bond, James C. Corbett, JJ Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, Vadim Yushprakh. Megastore: providing scalable, highly available storage for interactive services. *5th Biennial Conference on Innovative Data Systems Research (CIDR '11)*, January 2011.

[6] Peter Bailis, Shivaram Venkataraman, Michael J. Franklin, Joseph M. Hellerstein, and Ion Stoica. Probabilistically bounded staleness for practical partial quorums. *Proceedings VLDB*, August 2012.

[7] D. Barbara-Milla and H. Garcia-Molina. The demarcation protocol: A technique for maintaining constraints in distributed database systems. *VLDB Journal* 3(3):325-353, 1994.

[8] N. Belaramani, M. Dahlin, L. Gao, A. Nayate, A. Venkataramani, P. Yalagandula and J. Zheng. PRACTI replication. *Proceedings USENIX Sym-*

posium on Networked Systems Design and Implementation (NSDI), May 2006.

[9] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A critique of ANSI SQL isolation levels. In *Proceedings ACM SIGMOD International Conference on Management of Data* (SIGMOD '95), 1995.

[10] David Bermbach , Stefan Tai, Eventual consistency: How soon is eventual? An evaluation of Amazon S3's consistency behavior, *Proceedings of the 6th Workshop on Middleware for Service Oriented Computing*, p.1-6, December 12-12, 2011, Lisbon, Portugal.

[11] Philip A. Bernstein, David W. Shipman, and James B. Rothnie, Jr. Concurrency Control in a System for Distributed Databases (SDD-1). *ACM Transactions on Database Systems* 5(1): 18-51, March 1980.

[12] Philip A. Bernstein and Nathan Goodman. Concurrency control in distributed database systems. *Computing Surveys* 13(2): 185-221, June 1981.

[13] Philip A. Bernstein, Alan Fekete, Hongfei Guo, Raghu Ramakrishnan, Pradeep Tamma. Relaxed currency serializability for middle-tier caching and replication. *Proceedings SIGMOD*, June 2006.

[14] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, Jaidev Haridas, Chakravarthy Uddaraju, Hemal Khatri, Andrew Edwards, Vaman Bedekar, Shane Mainali, Rafay Abbasi, Arpit Agarwal, Mian Fahim ul Haq, Muhammad Ikram ul Haq, Deepali Bhardwaj, Sowmya Dayanand, Anitha Adusumilli, Marvin McNett, Sriram Sankaran, Kavitha Manivannan, and Leonidas Rigas. Windows Azure Storage: a highly available cloud storage service with strong consistency. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (SOSP), October 2011.

[15] R. Cattell, Scalable SQL and NoSQL data stores, *ACM SIGMOD Record*, v.39 n.4, December 2010

[16] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A Distributed Storage System for Structured Data. *ACM Trans. Comput. Syst.* 26, 2, Article 4 (June 2008).

[17] B. 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. *Proceedings International Conference on Very Large Data Bases*, August 2008.

[18] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC)*, June 2010.

[19] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, Dale Woodford. Spanner: Google's globally distributed database. *Proceedings Symposium on Operating System Design and Implementation (OSDI)*, October 2012.

[20] James Cowling and Barbara Liskov. Granola: Low-overhead distributed transaction coordination. *Proceedings USENIX Annual Technical Conference*, 2012.

[21] Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. ElasTraS: An elastic transactional data store in the cloud. Proceedings *USENIX Workshop on Hot Topics in Cloud Computing*, June 2009.

[22] Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. G-Store: a scalable data store for transactional multi key access in the cloud. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (SoCC '10), June 2010.

[23] K. Daudjee, K. Salem: Lazy database replication with snapshot isolation. *VLDB* 2006

[24] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. In *Proceedings of twenty-first ACM SIGOPS Symposium on Operating systems Principles* (SOSP '07), 2007.

[25] S. Elnikety, W. Zwaenepoel, and F. Pedone. Database replication using generalized snapshot isolation. *SRDS* 2005.

[26] Wojciech Golab , Xiaozhou Li , Mehul A. Shah, Analyzing consistency properties for fun and profit, *Proceedings of the 30th annual ACM SIGACT-SIGOPS symposium on Principles of dis-*

tributed computing, June 06-08, 2011, San Jose, California, USA.

[27] Google. Read consistency & deadlines: more control of your datastore. *Google App Engine Blog*, March 29, 2010. http://googleappengine.blogspot.com/2010/03/read-consistency-deadlines-more-control.html

[28] Hongfei Guo, Per-Åke Larson, Raghu Ramakrishnan, Jonathan Goldstein. Relaxed currency and consistency: How to say "good enough" in SQL. *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, 2004.

[29] T. Kraska, M. Hentschel, G. Alonso, and D. Kossmann. Consistency rationing in the cloud: pay only when it matters. *Proceedings International Conference on Very Large Data Bases*, August 2009.

[30] Tim Kraska, Gene Pang, Michael J. Franklin, and Samuel Madden. MDCC: Multi-data center consistency. *Proceedings EuroSys*, April 2013.

[31] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *SIGOPS Operating Systems Review* 44, 2 (April 2010).

[32] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM* 21, 7 (July 1978).

[33] Justin J. Levandoski, David B. Lomet, Mohamed F. Mokbel, Kevin K. Zhao. Deuteronomy: Transaction Support for Cloud Data. *Proceedings CIDR*, January 2011.

[34] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Don't settle for eventual: scalable causal consistency for wide-area storage with COPS. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP '11)*, October 2011.

[35] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Stronger semantics for low-latency geo-replicated storage. *Proceedings Symposium on Networked System Design and Implementation (NSDI)*, April 2013.

[36] David Lomet, Alan Fekete, Rui Wang, and Peter Ward. Multi-version concurrency via timestamp range conflict management. *Proceedings IEEE International Conference on Data Engineering (ICDE)*, April 2012.

[37] Oracle. Oracle NoSQL Database. An Oracle White Paper, September 2011. http://www.oracle.com/technetwork/database/nosqldb/learnmore/nosql-database-498041.pdf

[38] Daniel Peng and Frank Dabek, Large-scale incremental processing using distributed transactions and notifications. *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation*, 2010.

[39] Y. Saito and M. Shapiro. Optimistic Replication. *ACM Computing Surveys*, March 2005.

[40] Marco Serafini and Flavio Junqueira. Weak consistency as a last resort. *LADIS*, 2010.

[41] Adam E. Silberstein , Russell Sears , Wenchao Zhou , Brian Frank Cooper, A batch of PNUTS: experiences connecting cloud batch and serving systems, *Proceedings of the 2011 international conference on Management of data*, June 12-16, 2011, Athens, Greece.

[42] Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. Transactional storage for geo-replicated systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP '11)*, October 2011.

[43] Michael Stonebraker. Concurrency control and consistency of multiple copies of data in distributed INGRES. *IEEE Transaction of Software Engineering* SE-5(3): 203-215, May 1979.

[44] D. Terry, A. Demers, K. Petersen, M. Spreitzer, M. Theimer, and B. Welch. Session guarantees for weakly consistent replicated data. *Proceedings IEEE International Conference on Parallel and Distributed Information Systems*, 1994.

[45] D. B. Terry, M. M. Theimer, Karin Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles* (December 1995).

[46] Doug Terry, Replicated data consistency explained through baseball, *Microsoft Technical Report MSR-TR-2011-137*, October 2011. To appear in *Communications of the ACM*.

[47] Douglas Terry, Vijayan Prabhakaran, Rama Kotla, Mahesh Balakrishnan, Marcos K. Aguilera, and Hussam Abu-Libdeh. Consistency-based service level agreements for cloud storage. To appear in *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, November 2013.

[48] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. Calvin: Fast distributed transactions for partitioned database systems. *Proceedings ACM*

SIGMOD International Conference on Management of Data, May 2012.

[49] Robbert van Renesse and Fred B. Schneider. Chain replication for supporting high throughput and availability. *Proceedings Symposium on Operating System Design and Implementation (OSDI)*, December 2004.

[50] W. Vogels. Eventually consistent. *Communications of the ACM*, January 2009.

[51] W. Vogels. Choosing consistency. *All Things Distributed*, February 24, 2010. http://www.allthingsdistributed.com/2010/02/strong_consistency_simpledb.html

[52] 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. *Proceedings CIDR*, January 2011.

[53] Zhou Wei, Guillaume Pierre, and Chi-Hung Chi. CloudTPS: scalable transactions for web applications in the cloud. *IEEE Transactions on Services Computing*, 2011.

[54] Haifeng Yu and Amin Vahdat. Design and evaluation of a conit-based continuous consistency model for replicated services. *ACM Transactions on Computer Systems* 20(3):239-282, August 2002.