Deuteronomy: Transaction Support for Cloud Data

Justin J. Levandoski[1§]   David Lomet[2]   Mohamed F. Mokbel[1§]   Kevin Keliang Zhao[3§]

[1]University of Minnesota, Minneapolis, MN, {justin,mokbel}@cs.umn.edu
[2]Microsoft Research, Redmond, WA, lomet@microsoft.com
[3]University of California San Diego, La Jolla, CA, kezhao@cs.ucsd.edu

ABSTRACT

The Deuteronomy system supports efficient and scalable ACID transactions in the cloud by decomposing functions of a database storage engine kernel into: (a) a transactional component (TC) that manages transactions and their "logical" concurrency control and undo/redo recovery, but knows nothing about physical data location and (b) a data component (DC) that maintains a data cache and uses access methods to support a record-oriented interface with atomic operations, but knows nothing about transactions. The Deuteronomy TC can be applied to data *anywhere* (in the cloud, local, etc.) with a variety of deployments for both the TC and DC. In this paper, we describe the architecture of our TC, and the considerations that led to it. Preliminary experiments using an adapted TPC-W workload show good performance supporting ACID transactions for a wide range of DC latencies.

1. INTRODUCTION

The Brewer CAP theorem [8], formalized by Gilbert and Lynch [20], states that "A distributed computer system can simultaneously provide only two of three desirable properties: *Consistency*, *Availability*, and *Partition tolerance*". This suggests that it is difficult to support ACID transactions in the cloud environment where distributed data and high availability are essential elements. As testimony to the influence of the CAP theorem, many cloud providers have pretty much abandoned transactional support for data spanning multiple nodes. Dynamo [17], BigTable [10], Facebook Cassandra [24], Windows Azure [29], and PNUTS [11] all stop short of providing general purpose transactions. Instead, these systems opt for *availability* by relaxing *consistency*, supporting either (a) atomicity over only a single data item or a collection of items, or (b) eventual consistency [34], i.e., data updates become visible to everyone after a finite time. Weak consistency is sometimes acceptable. Examples of applications that can tolerate weak consistency include keyword search, inventory search, and setting user preferences or recommendations (e.g., Facebook "Like" options, or movie ratings). On the other hand, many applications, new and old, would like to use cloud storage, yet find that weak consistency makes life very difficult. Examples of new applications include social networking and Web 2.0 applications, online auctions, and collaborative editing. Old applications include traditional database services such as credit card transactions and flight reservations.

To date, three primary approaches have explored providing ACID transactions in a cloud environment: (a) relying on application developers to implement their own consistency checks [7], which is both burdensome and inefficient, (b) making databases and data storage elastic to scale up and down with the current workload [12, 13, 14, 33], which still has either limited scalability or limited transaction support, and (c) extending single-key transactional support to multi-key [9, 15, 19], which is still limited in terms of not supporting transactions over keys in different groups.

This paper describes the architecture and functionality of the Deuteronomy system that provides efficient ACID transactions for data anywhere, including the cloud. Deuteronomy distinguishes itself from previous efforts by its radical approach of factoring the functions of a database storage engine kernel into: (a) a transactional component (TC) that provides transactions via "logical" concurrency control and undo/redo recovery but does not know physical data location and (b) a data component (DC) that caches data and knows about the physical organization, e.g. access methods, and supports a record-oriented interface with atomic operations, but knows nothing about transactions. Applications submit requests to the TC. The TC uses a lock manager and a log manager to logically enforce transactional concurrency control and recovery. The TC passes requests to the appropriate Data Component (DC). The DC, guaranteed by the TC to never receive conflicting concurrent operations, need only support atomic record operations, without concern for transaction properties that are already guaranteed by the TC.

A salient feature of Deuteronomy is the ability for transactions to span multiple DCs. For example, consider a transaction that updates two tables: an **order** table stored at a DC hosted on a local enterprise server, and a **payment** table stored at another DC hosted in the cloud. A client executes this transaction at a single TC, without specifying the physical location of the data. The TC performs all operations necessary for transactional support (e.g., logging/locking), and routes the data update operations to the correct DC ei-

---

ther at the enterprise server (i.e., an order update) or in the cloud (i.e., a payment update). Upon completion, the TC is responsible for committing the transaction and ensuring the updates are stable at both DCs.

The deuteronomy architecture is scalable in three dimensions. (1) From a user perspective, if more application execution capability is needed then more applications servers (i.e., TC clients) can be added that interact with a single TC. (2) If data volume grows, more DC servers can be added "underneath" a TC to handle the storage and data manipulation workload. (3) For the case that transaction rates reach a degree that saturates the computational resources of a single TC servicing multiple clients and interacting with multiple DCs, multiple TCs can be instantiated (on separate machines) supporting transactions on disjoint sets of data. Thus, workloads can be split between TCs as long as the data they update is disjoint. However, we believe a single TC is capable of handling large workloads, as its performance-intensive operations consist mainly of locking, logging, and communication overhead. For OLTP workloads (for which Deuteronomy is intended) that are update intensive and do not deal with large answer sets, communication bandwidth should not be a system bottleneck until transaction rates saturate the communication link. Similarly, the execution load (i.e., logging and locking) should not be substantial at the TC node until transaction rates are enormous.

A Deuteronomy TC can be seen as providing *transactions as a service*. Storage systems in the cloud can "outsource" their transaction services to a TC. Alternatively, a TC can be seen as a storage engine that "outsources" its physical data storage management to another component (DC) in the cloud. Careful separation of transaction and data services enables multiple deployment scenarios. A TC can be applied to data *anywhere*, e.g., in one cloud, in multiple clouds, local, or at an enterprise server. Further, a TC can allow a transaction to spread over multiple DCs. Multiple TC deployment scenarios are also possible. Both TC and DC can be at a client and access local data, a TC can be at the client while the DC is in the cloud, or both the TC and DC can be cloud-based.

Previously [26, 27, 28], we described how to separate transactional functionality from data management functionality. This motivated our current effort, where we view cloud storage as an enormous atomic key-value store where data accessed within a transaction need not be co-located on an individual node to enable ACID transactions. The contributions of the current work are (1) the architecture of our multi-threaded TC; (2) a new TC:DC protocol in which the DC executes operations prior to TC logging them; (3) a new implementation of log control operations to deal with this protocol; and (4) initial experiments using an adapted TPC-W workload [32] that demonstrate both good performance and the impact of cloud latency on performance.

The rest of this paper is organized as follows. Section 2 highlights related work. The Deuteronomy system architecture is presented in Section 3. Section 4 provides details of the TC internals. Section 5 discusses transaction optimizations, while Section 6 provides an end-to-end example of a transaction executing in Deuteronomy. Section 7 discusses DC deployment scenarios. Preliminary performance results are provided in Section 8. Section 9 discusses availability. Finally, Section 10 concludes the paper, and Section 13 discusses a demonstration of the Deuteronomy system.

2. RELATED WORK

Several approaches have been proposed to enable ACID transactions for the cloud. This has been motivated by: (a) the lack of transactional support in existing commercial and open-source cloud services (e.g., [3, 9, 10, 11, 17, 19, 21, 24, 29]), (b) the emergence of new applications that require transactional support in the cloud, e.g., Web 2.0 applications, social networks, and collaborative editing [4, 23], and (c) the desire for traditional database applications, e.g., credit card transactions and flight reservation, to make use of the cloud infrastructure. The approaches can be divided into the following three broad categories:

Application-provided consistency. This approach relies on application developers to be responsible for ensuring transactional consistency. It puts a huge burden on application developers. It may also incur a big performance hit due to the need to call the server multiple times to ensure consistency [7]. While operations that do not require strong consistency guarantees can be realized efficiently, for others needing strong consistency, this performance impact may be unavoidable. This approach makes sense only for applications that have a very low fraction of transactions that require strong consistency guarantees [23].

Localized transaction support. Google Megastore [6] and Microsoft SQL Azure [9] support transactions over multiple records, however, they require that these records be co-located in some way. Developers must cluster Megastore data items into hierarchical groups. For SQL Azure, database size is constrained to fit on a single node. For larger data sets, an application needs to partition the data among different database instances. ElasTras [14] is an elastic database design that scales up and down with the transaction workload, but does not provide transactions upon recovery and supports a restricted transaction semantics, termed minitransactions [2], that execute within one data partition.

Limited wider transaction support. G-Store [15] allows for dynamic group formation (a relatively costly operation), where transactions are not allowed across these formed groups. CloudTPS [35, 36] assumes that transactions are short-lived and only access well-identified items (a group). Then it employs a two-level hierarchy of transaction managers running a global two-phase commit protocol over a set of transactions that each access a single item. The ecStore [33] is an elastic distributed storage system with three layers: storage nodes, replication layer, and a transaction layer that provides a hybrid design of multi-version and optimistic concurrency control. Transactions need full knowledge of data layout in the storage nodes.

Deuteronomy distinguishes itself from these approaches in architecting a complete separation of transaction services from data services. Such modular design allows for porting the Deuteronomy TC over any local or cloud data storage. The Deuteronomy TC can provide transactions across data anywhere, in the cloud or locally, not limited to a specific node, nor requiring special setup costs. This transaction support is transparent from both application developers and DCs, making their life much easier when dealing with the "elastic storage" provided by the cloud, though including DC caching and log synchronization functionality on top of some cloud infrastructures does take some effort.

3. OVERALL SYSTEM ARCHITECTURE

Figure 1 gives the Deuteronomy architecture depicting one transaction component (TC) (the large gray rectangle) and multiple data components (DC) with their physical storage as either local or in the cloud. The TC has five main components, depicted as white rectangles: *session manager*, *record manager*, *table manager*, *lock manager*, and *log manager*. The TC-DC interaction contract [27] is enforced by a set of control operations (depicted by dark rectangles in both the TC and DC) that mainly ensure recovery after failure of TC and/or DC, e.g. the write-ahead log protocol. The DC needs to manage its own data and storage, and can do so any way it likes as long as it supports atomic record operations and the "other side" of the control operations. This is a strong feature of Deuteronomy, as it makes the TC portable to many data storage providers and describes precisely what is needed on the DC side, i.e., the atomic record and table operations, the control operations, and the interaction contract.

Implementing a DC is non-trivial, as the DC provides both cache management and access method support, and in addition, it must fulfill the TC-DC contract, which includes control operation support, guaranteeing idempotence of operations, and recovery. A record-oriented cloud infrastructure is used by a cloud DC as if it were record-oriented disks. The important thing here from the transactional viewpoint is that there can be multiple DCs, the DCs can be located anywhere, the data managed by a DC can itself be "scattered across" the cloud or be local, and yet a single TC can effectively provide transactions for applications under any of these circumstances.

Applications submit their requests directly to the TC. These requests are handled by a multi-threaded *session manager*. The first request initiates the session, and the session manager establishes an authenticated session for the user. Subsequent requests flow through the session manager and are dispatched to the other components. Based on whether the request is for a record operation (e.g., read/write record) or table operation (e.g., create/delete table), the TC "session" invokes either its *record manager* or *table manager*, respectively. In both cases, the record/table manager calls both the lock and log manager to perform "logical" concurrency control and recovery.

We limit TC knowledge of threading to (1) the session manager, which does all thread management, (2) the lock manager which has to arbitrate and protect its lock data from race conditions and occasionally needs to block a thread when transactions have conflicting accesses, and (3) the log manager, which needs similar protection for its data structures, and occasionally needs to block a thread while log records are forced. Importantly, both table manager and record manager are thread safe without needing to be aware of threading issues (i.e. they are "thread oblivious").

Resources within the TC are all treated as logical data items in that their identification does not include physical location information. Locks are taken without knowledge of the physical layout of the stored data. Similarly, the log manager posts log records with resources described logically and without physical location information. These logical resources are mapped via metadata stored via the *table manager* to identify which DC owns the requested data. Metadata can be added throughout the lifetime of the TC in a similar way as done with traditional database catalogs.
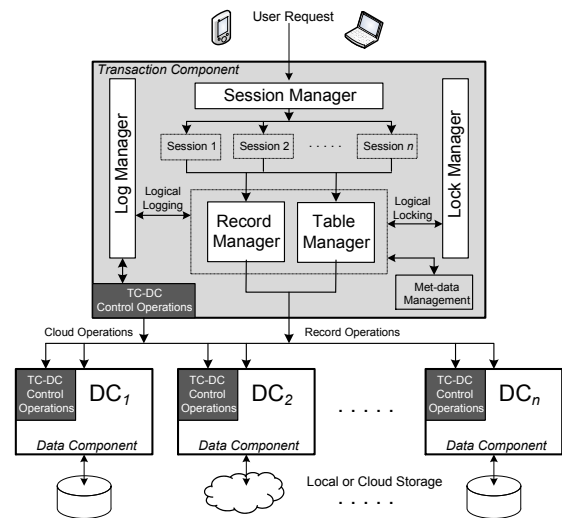

Figure 1: Deuteronomy System Architecture

Using its metadata, the TC sends table or record operations to the appropriate DC. The DC executes each of these operations as a stand-alone atomic operation, without worrying about any transactional conflicts among concurrent requests as TC locking will guarantee the absence of such conflicts. The TC also logs the operations as they are successfully completed (i.e., after the DC returns). This sequence of locking, forwarding an operation to a DC, and then logging, is quite different from our original thoughts [27], and will be elaborated in the record manager section.

There are three important points to emphasize:

1. Data can be stored *anywhere*, e.g., in a local disk, in a flash device, in the cloud, etc. TC functionality in no way depends on where the data is located.

2. The TC and DC can be deployed in a number of ways. Both can be located within the client, and that is helpful in providing fast transactional access to closely held data. The TC could be located with the client while the DC could be in the cloud, which is helpful in case a user would like to use its own subscription at a TC service or wants to perform transactions that involve data in multiple locations. Both TC and DC can be in the cloud, which is helpful if a cloud data storage provider would like to localize transaction services for some of its data to a TC component.

3. There can be multiple DCs serviced by one TC, where transactions spanning multiple DCs are naturally supported because a TC does not depend on where data items are stored. Also, there can be multiple TCs, yet, a transaction is serviced by one specific TC.

4. TRANSACTION COMPONENT (TC)

This section discusses the five major components of the TC: the *session manager*, *record manager*, *table manager*, *lock manager*, and *log manager*, the ones on the most performance sensitive execution path.

4.1 The Session Manager

The *session manager* is the application facing module of the TC, providing the interface to the application (henceforth referred to as user) and overseeing the execution of

its requests. The session manager maintains communication connections to users and provides multiplexed use of these connections. It authenticates a user when a session is initiated. Each session can support a stream of transactional requests, though within a session, there are no concurrent transactions. Hence it is sessions that are assigned to threads, with a session never using more than a single thread.

The session manager maintains a thread pool, a thread being dispatch for a session when a request from that session arrives and there is no thread assigned for this session. A dispatched session thread handles session requests in sequence. After serving a request, the session thread finds the next unprocessed queued request in this session and executes it. Queued requests are possible since a client can issue a batch of ordered requests. If there is no waiting request for the session, the thread is returned to the thread pool so that it can be used to handle requests from another session. Maintaining a thread pool permits fast request handling without the overhead of thread creation.

The session manager oversees the execution of user requests, calling the record manager and/or table manager as needed. User requests may be bracketed with explicit Begin/Commit transaction operations or use implicit transactions in which the session manager will provide these operations when a request from a client is not explicitly bracketed. In any case, a user (session) has only one open transaction at a time. User requests may result in multiple calls to table and record manager, e.g. a record update may require that the table manager be accessed to interrogate the catalog to identify the DC at which the data is managed; followed by an invocation of the record manager to perform the data manipulation operation. The session manager also marshals and de-marshals requests and replies between client and TC.

4.2 The Record Manager

The *record manager* supports operations that include modifying or reading records from DCs. Record operations "logically" coordinate with both the lock and log managers without knowledge of physical data placement and, in some cases, of the values of keys at the DC. In general, the *record manager* is responsible for two classes of operations: (a) reading operations that include reading a single record or a range of records, and (b) writing operations that include inserting, updating, and deleting a single record.

4.2.1 Read Operations

For read operations, the record manager first requests from the *lock manager* the appropriate lock(s) on the requested resource(s) (detailed in Section 4.4). Once the relevant locks are granted, the read request is forwarded to the DC either for a single record or for a range of records. Reads are not logged.

A user-provided key identifies a record for a singleton record operation. However, this is not the case for range reads, where the boundary keys that bracket the records of the range need not be real key values associated with existing records. Thus our "range" locking needs to be done without knowledge of the key values of records in the range. This is discussed in the lock manager subsection 4.4.

4.2.2 Write Operations

Write operations need to be both locked and logged. Both locking and logging are done with neither physical data placement information nor knowledge of surrounding keys. Both locking and logging requirements caused us to re-think the nature of the TC:DC interface, with the result that we changed the expected protocol specifics introduced in [27] to improve performance.

Locking: We do not exploit "next key" range locking, even of the type described in [28]. We made a strategic decision that even if we could afford to set such key value range locks, we could not afford to test them during inserts and deletes. These operations would need to know the "next key" value to test the "next key" lock protecting the gap (range) between keys for a range reader. To learn the "next key" requires that we read it from the DC, but this doubles the number of round trips to the DC needed for every insert and delete. While this may be reasonable when the DC is local, it is not when dealing with remote DCs with large latencies.

Logging: There is no problem with logging logical record identifiers for log records, and earlier work [26] demonstrated that doing this would produce at worst a modest reduction in speed of recovery, and little impact on normal execution. There are two problems with logging addressed in Deuteronomy that are of a different nature.

1. We had modeled our TC:DC interface [27] on the recovery guarantees framework developed for application recovery [5]. This posted information to the log prior to sending messages to ensure that the sender remembered a message it sent whenever a receiver remembered it (causality). But if we logged requests, we would need to also log replies in order to know when an operation succeeded. And not all operations (requests) succeed. For example, inserting a record when a record with the same key already exists or updating a record that does not exist are errors. We want to log operations only once. To do that, we need to know their outcome at the point when written to the log.

2. A second problem is that transactional logging requires both before and after state so that operations can be undone [31]. When an insert succeeds, the before state is null, so undo logging for inserts requires nothing new. For deletes and updates however, we did not want to be required to read the record before we changed it. By waiting for the DC to execute the operation and return to the TC prior to logging, we can have the response from the DC include the before image of the record which we then can include in our log records.

The preceding problems caused us to want a record manager protocol that orders locking and logging activities as follows:

1. Request appropriate locks, and generate a log sequence number (LSN), in monotonically increasing order, for a DC data modification request. The LSN is generated only after the appropriate write lock is granted. LSNs are not generated for read or intention locks. Our lock manager guarantees the order of these LSNs is consistent with the conflict order of the operations.

2. Send the operation to the DC for execution. The DC uses the LSN as it would in a conventional setting to identify the operation and provide recovery idempotence. For update and delete operations, the before

image is returned. For all operations, an indication of whether they succeeded is returned.

3. Log the operation after the DC has returned having executed the operation, again using the lock manager provided LSN to identify the operation. During recovery, the LSN for a successful operation is sent again to the DC along with the operation for the DC idempotence test.

This protocol is possible only because the TC:DC interface includes control operations which can be used to enforce causality and permit recovery management in this new setting. In particular, the TC can, via the control operations, specify when the DC can make information stable. It does this via an EOSL ("end of stable log") call. Operations with LSNs larger than the latest LSN provided to the DC via an EOSL call must be "forgettable".

Note here that when using this protocol, the LSNs on the log may be out-of-order as there is no guarantee about which requests will be granted/acknowledged first due to the multi-threaded nature of both TC and DC. However, what we are sure about is that conflict order will be preserved through all LSNs in the log file. This is guaranteed by the *lock manager*.

4.3   The Table Manager

The *table manager* is mainly concerned with data definition language (DDL) operations on a table that include creating and deleting tables, as well as creating, deleting, and modifying table columns. These operations sync with the lock and log managers and are passed to the appropriate DC in exactly the same way as the update operations described for the *record manager* in Section 4.2. In addition to executing DDL operations, the table manager has two other responsibilities: *meta-data management* and *creating and altering logical locking partitions*; we now describe these responsibilities in detail.

4.3.1   Metadata Management

The table manager is responsible for maintaining two primary metadata catalogs: (1) A *table catalog* stores an entry for each table containing its name, owner information, and the DC that stores the table. This catalog is primarily used to direct read/write requests to the appropriate DC. (2) A *column catalog* stores table column information including column name, constraints (e.g., primary/foreign key), and minimum and maximum values. Each TC stores its metadata catalogs at a *master* DC, a designated "default" DC whose address is given to each TC upon initialization and kept safe and persistent. When a TC subsequently restarts (e.g., due to a crash), it uses the stored master DC's address to retrieve its catalogs.

Like regular table and record operations, all operations that modify metadata catalogs synchronize with the lock and log managers. In fact, to ensure consistency between tables and their metadata, we wrap each table operation (e.g., create table) in a transaction with a symmetric operation that manipulates the appropriate metadata catalog (e.g., adding an entry to the table catalog).

4.3.2   Logical Partition Creation

As will be discussed in detail for the lock manager, there is a need for logical locking partitions in Deuteronomy. Essentially, the TC knows *only* about table and record lock resources, but not about the pages on which records or tables are stored. Pages have served as an intermediate lock resource in traditional database systems. But, in Deuteronomy, the TC cannot lock pages since they are not known to it. To compensate for this, the table manager defines *logical partitions* as an intermediate granularity lockable resource for each Deuteronomy table. A straightforward approach we currently use to create logical partitions is to divide the key range equally into a fixed number of partitions. However, more sophisticated techniques can be used to provide more uniform coverage by data volumes. New partitioning techniques can be employed without affecting the overall operations in Deuteronomy.

4.4   The Lock Manager

The *lock manager* is called from either the table or record managers to obtain the appropriate locks before user requests are forwarded to the DC. The lock manager must be thread aware as its lock tables can be accessed by many session threads concurrently. It uses a monitor to protect the lock table entries. It causes a session thread to block (sleep) if it encounters a conflicting lock. This part of the lock manager is quite traditional. Our lock manger also supports requests for multiple locks, with the lock manager returning after all lock requests are granted.

We employ multi-granularity locking with three levels of resources: table, partition, and record. Our multi-lock requests are used to request known locks down this hierarchy so that a single call is sufficient to acquire a record level lock. For table or record level locks, it is straightforward to provide table ID(name) and record ID(key), respectively, which are known to the TC as well as the DC. The TC knows these via the user request and its metadata.

Partitions are present in our multi-granularity hierarchy to facilitate reading ranges of records [28]. For example, consider the query that selects all employees with IDs from 10 to 40, and runs with serializable isolation. The lock manager needs to lock all the keys in the range [10,40]. However, since the TC knows nothing about the stored data, it has no way of locking these records without first reading them from the DC, which is very expensive in a cloud setting. Instead, the record manager will consult the table manager, which will return to it with a partition ID. The table manager is responsible for knowing about key domains and can partition them "logically" as appropriate.

The TC record manager will utilize these logical partitions to request locks that cover the requested range through locking all the logical partitions that overlap with the requested range. With logical partitions, reading/writing a single record requires an intent lock on the table and the partition resources that cover the requested record, and an explicit lock on the record itself. Reading a range of records requires an intent lock over the table, then, a set of explicit locks on all the logical partitions that overlap with the requested range. Individual records are never locked for a range read. Note that this is a different protocol than used in [28], where individual records were locked in "border" partitions. We felt it essential to avoid checking a next key lock during insert and delete operations, which is required for key range locks. Using only partition locks, which are analogous to page locks, is a cruder approximation to the range of records, but it avoids this extra "next key" access.

As discussed in Section 4.2, the lock manager is also re-

sponsible for generating LSNs to be used when logging writing operations. The main idea is that LSNs generated by the lock manager are guaranteed to be in conflict order. Thus, they are ideal for communicating with the DC and for providing idempotence for write operations at the DC. They do cause a complication at the log manager, however, which we describe next.

4.5 The Log Manager

The core of the log manager is conventional. Indeed, in our implementation, we used the Windows Common Log File (CLF) [30] as the TC transactional log. CLF natively supports multi-threading. Our code that wraps the CLF invocations also must deal with explicit threading, and appropriately synchronizes access to private data structures (e.g., thread-safe hash tables). However, we differ from conventional logging in that (1) we must deal with LSNs that are stored somewhat out-of-order on the log; and (2) we need to coordinate log management at the TC with cache management at the DC.

Recall that our protocol for dealing with a record operation first acquires a lock at the lock manager, with an LSN issued for it in strictly monotonic and hence conflict order. The request is then forwarded to the DC for execution, and we log the request only after the operation succeeds at the DC and returns control to the TC. Given the extensive use of multiple threads at all system levels, the LSNs, ordered when they leave the lock manager, can arrive out-of-order at the log manager. The log manager does not wait for LSNs of missing requests but rather writes the log record describing an operation immediately to the log.

Deuteronomy's recovery protocol has been described in some detail in an earlier paper [27]. Deuteronomy requires some changes to the more traditional ARIES [31] style algorithm. However, it shares the log management aspects with ARIES. That is, we need to enforce the write ahead log protocol, and we need to determine at what log position to begin our redo scan. These two aspects of recovery are normally done within a single database kernel, as part of an integrated algorithm. For Deuteronomy, however, managing the transaction recovery log is done at the TC, while cache management is done at each DC. Hence, log management requires that log and cache management be coordinated between TC and DC for successful recovery.

For the above reasons, the TC log manager will send two control operations to the DC: to enforce the write-ahead log protocol (causality); and to enable it to truncate the active part of the log via a checkpoint so that redo recovery time can be bounded and log space recovered and reused. Both operations are implemented as separate background threads, and do not interfere with the session-oriented record/table operations.

EOSL: The TC log manager periodically sends to each DC an LSN (denoted $eLSN$) indicating the *End Of Stable Log*. This operation permits a DC to write updates that it has cached back to stable storage. Before receiving an $eLSN \geq LSN$ of a cached update, the DC must not make that update stable. This permits it to "forget" the update ("forced amnesia") should the TC crash and lose the log tail containing that log record.

RSSP: At less frequent intervals, the TC log manager sends to each DC an LSN (denoted $rLSN$) indicating its desired *Redo Scan Start Point*. This operation requires that the DC write to stable storage all updates with LSNs earlier than $rLSN$ prior to returning from this operation. When control returns to the TC, the TC writes the $rLSN$ into its checkpoint information on its log, and uses the last written $rLSN$ as the start point for its redo scan should recovery be needed.

Given that LSNs are not ordered monotonically on our log, both $eLSN$ and $rLSN$ are "low water marks". That is, for EOSL, an $LSN \leq eLSN$ is stable. But an $LSN > eLSN$ may also be stable. For RSSP, the TC promises that every operation with an $LSN \geq rLSN$ will be replayed during recovery. Further, there is no guarantee that an operation with an $LSN < rLSN$ will be replayed during redo, so the DC must promise to make those operations stable before ACKing an RSSP call. This is the case even though some of these operations may, in fact, be re-sent to the DC because they appear later than the log position the TC uses for its redo scan start point.

4.5.1 EOSL

For EOSL, we need to ensure that we have received replies for all operations up to and including the operation with $LSN = eLSN$, and that log records for these operations are on the stable log. We may have received replies for some requests with an $LSNs > eLSN$, but we can ignore them. We keep a vector LSN-V (starting at the last value for $eLSN$) indexed by LSN. Each element contains the log position LP at which the operation with the given LSN is placed. Log positions are monotonic such that when a log record is posted to the log, it has a log position higher than all preceding log records. We maintain a current log position cLP to identify the log record slot in the log where next log record will be written.

When an operation identified with an LSN $oLSN$ returns from the DC and arrives at the log manager, its log record is placed in the current log position cLP. cLP is then stored at LSN-V[$oLSN$]. cLP is then incremented to reference the next position in the log buffer. We also track the log position of the end of the stable log, called sLP, which is updated whenever a flushed log buffer is stably written.

To determine a new $eLSN$, we scan LSN-V from the old $eLSN$ position until we reach an LSN-V[lsn] that is not set (operation has not yet returned and been logged) or LSN-V[lsn] $> sLP$. $lsn - 1$ becomes the new $eLSN$ as we now know that lsn is the lowest LSN not on the stable log.

4.5.2 RSSP

Implementing RSSP would seem to be easy since all we need is to direct the DC to make operations earlier than $rLSN$ stable. But the TC itself starts its redo scan at a log position rLP. For a pair of $rLSN$ and rLP, we require two things. (1) Any $LSN \leq rLSN$ must be stable on the log. This is the same condition as for $eLSN$, suggesting that we use an earlier $eLSN$ as our $rLSN$. (2) Since we are starting our redo scan at rLP, we need to make sure that we see for redo all operations with LSNs greater than $rLSN$. Thus we need to ensure that no operation with an $LSN > rLSN$ has a log position $LP \leq rLP$.

We extend our work for EOSL to help us with RSSP. We keep track of the maximum LSN that we have seen up to the time we scanned LSN-V for $eLSN$, called $maxLSN$,

and also remember the sLP (end of stable log) used in determining $eLSN$. We retain this information for the last several EOSL requests (since the prior RSSP) in a list of its own called EOSL-L. An element of EOSL-L consists of $\langle eLSN, sLP, maxLSN \rangle$, where the EOSL-L elements are ordered consistent with the time of an EOSL request, EOSL-L[i] being at an earlier time than EOSL-L[$i+1$].

When an RSSP operation is invoked to determine a new $rLSN$, we set $rLP = $ EOSL-L[$oldest$].sLP. We then scan EOSL-L[i] entries beginning with $i = oldest + 1$. Any $LSN \leq$ EOSL-L[i].$eLSN$ is stable on the log. When EOSL-L[i].$eLSN \geq$ EOSL-L[$oldest$].$maxLSN$, we know that if the DC makes stable all operations with $oLSN \leq$ EOSL-L[i].$eLSN$ then all operations on the log before EOSL-L[$oldest$].sLP will be made stable since none has an $LSN >$ EOSL-L[i].$eLSN$. Hence there will be no operations preceding $EOSL-L[oldest].sLP$ on the log that need redo when we set $rLSN = $ EOSL-L[i].$eLSN$.

Once we choose $rLSN$, we improve rLP by scanning EOSL-L entries EOSL-L[j] beginning with EOSL-L[$oldest$]. We stop at the largest j that satisfies $rLSN \geq$ EOSL-L[j].$maxLSN$. EOSL-L[j].sLP is then used as the final rLP. The correctness of this improved rLP can be shown in the same way as the previous one.

5. TRANSACTIONAL OPTIMIZATIONS

Deuteronomy incorporates classical and important transactional optimization techniques. In this section we discuss how fast commit and group commit [18] optimizations are provided in Deuteronomy to improve throughput.

5.1 Fast Commit

Fast commit optimization allows a transaction to release all its locks before waiting for its commit record to be flushed to stable storage. Deuteronomy adopts the fast commit optimization by arranging the key steps that a transaction takes during commit as follows.

1. Create the commit record and post it to log buffer.

2. Release transaction locks.

3. Wait until transaction commit log record is flushed.

4. Remove the transaction from the table of active transactions.

5. Send reply to the client indicating the transaction has committed.

This order does not affect the correctness of the system since a transaction that releases its locks is guaranteed to commit earlier than any other transaction waiting for the locks. The guarantee comes from the fact that the earlier transaction places its commit record in log buffer before it releases its locks, and the log records are written sequentially during flush. So if a later transaction using the unlocked resources commits, it will be done only if the earlier transaction also commits. Compared to the original scenario where locks are released after the commit record is flushed, fast commit reduces lock wait time for every transaction.

To make this work correctly, we need to synchronize read-only transactions with this optimization. This can be done by writing commit log records for all transactions, including read-only transactions.
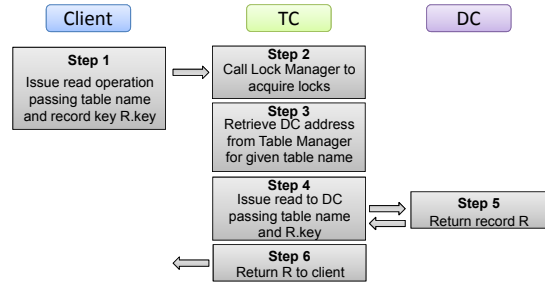

Figure 2: End-to-end steps for single record read

5.2 Group Commit

The group commit optimization delays a set of committing transactions for a small time period and commits them as a group by flushing the log buffer containing their commit records to the disk. For durability, a transaction is not committed until its commit log record has been flushed to the stable log. Without group commit, each commit triggers a system call to CLF to flush the log buffer to disk immediately. This requires CLF to flush the log buffer (write to the disk) for every transaction commit. Thus, every transaction incurs disk write latency, and log buffer storage utilization suffers.

Group commit amortizes the cost of a log force by grouping transactions that commit close in time and issuing a single log flush request for all transactions in the group. In Deuteronomy, this is achieved by having the thread of a committing transaction wait inside the log manager for a log flush. (This occurs at step 3 above.) A log flush writes all buffered log records to stable storage. This log buffer flush occurs either (1) when the buffer is full, or (2) after a small time delta, which is configurable, that enables the buffer to fill. In this way, we reduce the number of log I/O system calls from the number of commits to the number of commit groups by slightly holding back each committing transaction. All transactions of a group share the group commit write latency instead of each incurring the write latency.

6. AN END-TO-END EXAMPLE

To help demonstrate the technical details described so far, this section provides an end-to-end example of the steps necessary to execute a transaction in Deuteronomy. Our running example is the following transaction that reads and updates a single record R:

```
Begin Transaction
   Read record R
   Update record R with new value V
Commit Transaction
```

We describe the details of the four operations for this transaction in chronological order.

Begin transaction. To begin a transaction, the session manager first assigns a thread to the transaction (details in Section 4.1) and generates a unique transaction id, which is then used to initiate an entry in the transaction table that stores the log position of each active transaction's last operation (used to back-chain a transaction's log records).

Read record. The steps of a read operation for a single record are depicted in Figure 2. The client passes the TC
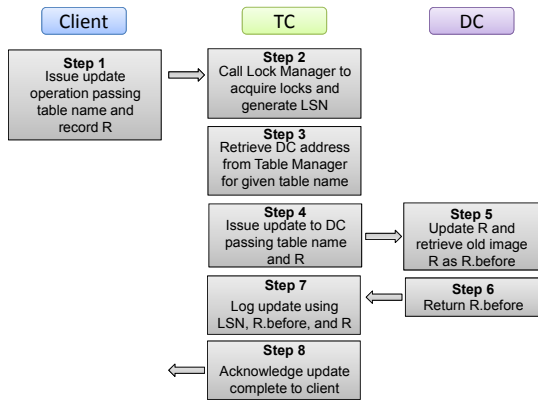
Figure 3: End-to-end steps for record update

Figure 4: TC Performance for Varying DC Latencies

the table name and record key R.key. Upon receiving the request, the TC calls the lock manager to acquire appropriate locks. The lock manager does not generate an LSN for these operations since reads are not logged in Deuteronomy. Next, using the given table name, the DC address is retrieved from the table manager, and the read request is sent to the DC passing the table name and record key. The DC then returns to the TC either record R if the read is successful, or an error if record R does not exist. The TC then passes back record R (or an error) to the client to complete the read operation.

Update record. The steps of an update operation are depicted in Figure 3. The client passes the TC a table name and updated record R. The TC then calls the lock manager to both acquire appropriate locks and retrieve an LSN for the operation. The DC address is then retrieved from the table manager, and the update is sent to the DC using the table name and record R. The DC first retrieves the before image of record R (abbr. R.before), and then performs the record update. The DC then returns R.before to the TC, and the TC logs the operation using the LSN, R.before as the before image, and R as the after image. Finally, the TC sends an acknowledgement of the update to the client, which completes the update operation. In the case of an error (e.g., record R does not exist at the DC), the TC will return an error to the client without logging the update.

End transaction. To end the transaction, the TC first writes a commit record to the log. Details of the Deuteronomy commit procedures are given in Section 5. Once the commit is logged, the transaction's entry is removed from the transaction table. Finally, the TC returns to the client and returns the transaction thread to the open thread pool.

7. DATA COMPONENTS (DCS)

We implemented and/or used a number of DCs: (1) a "stub DC" that merely returns immediately, used solely to test our TC code, (2) a "local DC" which keeps all data in main memory, also used in testing, (3) a "flash DC" that used a storage manager provided by the Communication and Collaboration Systems Group in Microsoft Research[1] that exploits flash memory, but is also usable with a disk for stability, and (4) a "cloud DC" written by a group in the

---

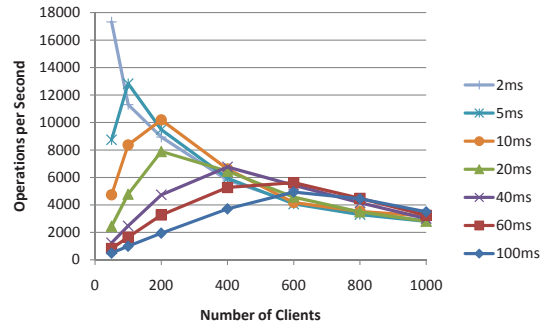[1]This group consisted of Sudipta Sengupta, Biplob Debnath, and Jin Li

Microsoft XTREME Computing Group[2] that uses Windows Azure storage to make data stable.

Our TC code is defined to permit us to use several DCs simultaneously, and the thread executing each TC operation simply invokes a DC operation as if it were local. Given our multi-threaded TC, this means that each DC needs to deal with multi-threading issues. When a DC is not local, there is a proxy DC behind the interface that forwards messages to the remote DC. In this case, it is the proxy that deals with the local (to the node of the TC) threading issues, such as commanding a requesting thread to sleep and waking it up when the reply comes back from the remote DC. A proxy DC is required for the cloud DC, which cannot be local.

Had we only intended using a TC with local DCs, we wouldn't have been so concerned about the need to read a record prior to locking records in a range or testing a next key lock when doing inserts and deletes. However, the cloud introduces large latencies (larger than a local disk). So it has been essential to tailor our TC:DC interface to minimize the number of times we incur cloud latency.

8. SOME PERFORMANCE RESULTS

8.1 Benchmarking

We evaluated performance via adapting a limited version of the TPC-W benchmarks [32] to work in the cloud environment. This is similar to the approach taken in [23, 22, 25]. Since we are most concerned about measuring TC performance, we show throughput under controlled changes in DC latency. The longer the latency to the DC, the higher the level of multi-threading we must employ to keep the processor busy. The more threads active at a time, the more collisions they will see, and perhaps lower cache hits as well. This results in higher throughput for lower latency deployments, as shown in Figure 4.

8.2 Improving Performance

We consider the performance reported for our benchmarking to be only respectable. We believe it is possible for performance to be substantially higher than we report above. Here we want to discuss the nature of the overhead introduced by the TC on the path to the data, and to understand what might be done to improve performance.

The first thing we did was to isolate TC performance from transactional aspects, in particular how much overhead was

---

[2]This group consisted of Roger Barga, Nelson Araujo, Brihadish Koushik, and Shailesh Nikam.

added by locking and logging. We ran tests with these capabilities disabled. Performance was only 20% better when locking and logging were disabled for the cloud latency case. As performance elsewhere improves, the cost of logging and locking will loom larger. But it is clear that performance gains will need to come from elsewhere.

We believe that there are two main impediments to higher performance. And these impediments are neither architectural nor are they intrinsic to the logic of operations as we have implemented operations. Rather, we believe there are two limitations that lie in the infrastructure that we used to build our prototype.

Threading: Operating system threads are much lighter weight than processes. However, whenever a system thread blocks or otherwise does a context switch, an OS thread crosses a protection boundary. This step adds substantial overhead to the cost of a thread switch. Because of this, SQL Server implements (in their SQL OS layer) user level threading called fibers. We did not use SQL OS in our implementation in the interest of rapid system prototyping. However, we believe that there is a substantial gain to be made by using fibers.

Implementation Language: We used C# as our implementation language because it reduced the programming effort. That was, we believe, a wise choice given the limited time we had to construct the system. But, for a "real" Deuteronomy deployment, we would have made a different choice. Building a system has a rough equivalence to working on the code in the inner loop of a large program. That is, system programming is much more performance sensitive than is application programming. Programming languages that are great for rapid prototyping and fine for application programming may be problematical for the inner core of system level programming. Deuteronomy (encompassing both TC and DCs) is part of that inner core. Indeed TC and DC together constitute the kernel of a database system. Thus, a language like C is the more appropriate language for a system to be widely deployed.

While we cannot quantify the performance that would result from using fibers and implementing in C, we expect substantial gains. At that point, we would need to revisit lock and log manager implementations. In the lock manager, we need to use more fine grained concurrency control for accessing the lock manager data structures, e.g. spin locks protecting smaller data extents. For the log manager, as with threading, performance would benefit from its code being entirely within user space to save the system protection boundary overhead of using CLF.

9. AVAILABILITY

Availability can be lost due to several forms of failure. Availability is maximized when the system can (1) minimize the extent of the availability loss when a failure occurs; and (2) reduce the time to recover from the failure causing the lost availability. In this section we describe a number of types of lost availability resulting from failures, and how the Deuteronomy architecture enables us to minimize the loss.

9.1 Data Unavailability

We focus first on data availability when the DC responsible for executing on the data has not failed. DC failures are considered below. A prime purpose of cloud infrastructures is to provide high availability. Data is typically replicated, with a consensus protocol used to ensure that a replica failure does not make data unavailable for update. So data unavailability should be a very rare event.

While there is nothing that Deuteronomy can do directly to make unavailable data available, accessible data need not become unavailable simply because it happened to be accessed in the same transaction as currently unavailable data. Such data can become transactionally consistent either by transaction commit or transaction abort. Both commit and abort are feasible depending on the specific state of each transaction. For transactions that are finished, commit releases locks on available data, while for abort, undo operations are first sent to the DC managing the data, and then locks are released. Neither outcome is a blocking outcome, and access to available data continues uninterrupted.

Further, a DC may be able to mask some data unavailability. This may enable some transactions to commit that would otherwise have aborted. So long as the data being accessed by transactions from the TC is in DC cache, availability continues. When the data becomes available again, the changes captured by the DC are written back to storage. Only when the DC needs to access data that is unavailable and not in its cache does it need to notify the TC. When this occurs, the TC aborts the transaction involved.

9.2 TC Failure

Availability in the presence of a TC failure depends on the robustness and accessibility of the TC transactional log. If the TC log is on a disk that is local to the TC, and is not itself replicated elsewhere, then the data handled via the TC is unavailable while the TC is down. However, if the log is available, e.g. cloud replication is used for the log, then the TC can failover to a standby TC that accesses the log. The standby TC initiates recovery using the log, and when recovery is complete, normal service resumes.

For a TC failure without an accompanying DC failure, recovery is very fast. Even with the TC log replayed from the RSSP, the DC has little recovery to perform as it has not crashed. The most substantive activity it needs to do is to reset its cache, removing updates that were not stable on the TC log at the time of the crash. Only data items modified by these updates, or ones reset along with them (e.g. if the cache was paginated and a reset affected other records on the page) need to be recovered.

9.3 DC Failure

Should a DC fail, a more expensive form of recovery is needed to bring the database back to the point where the failed DC can resume normal execution. We have outlined that recovery earlier [27]. The TC waits for the DC to come back up, and then initiates recovery with it. Because the DC cache has been lost, recovery now entails re-execution of all lost operations (i.e. redo recovery), and importantly, the DC cache needs to be re-populated with the active data as of the time of the crash to do this. While Deuteronomy recovery needs to be "logical", as shown in [26], recovery performance can be comparable to ARIES style recovery.

While it is conceptually possible to maintain a hot DC standby in the same way that a full-blown database systems maintains a standby, there is a negative to this. A database standby typically manages an independent database replica. Hence it can read and write to its replica and manage its cache independently of the primary database. To pursue

that strategy, we would need a hot standby DC to manage its own data replica. This can only be done by (1) replacing the normal cloud replication with our own replication so that a DC accesses a single replica or (2) having two replicated cloud data sets, each with a separate DC, thus adding another layer of replication on top of the normal cloud data replication. How best to maintain a hot standby using a single cloud replicated data set, where both primary and hot standby manage this data set is a topic for further work.

10. CONCLUSION

We have described our implementation of a TC that can provide transactional functionality over any storage infrastructure. To work effectively, we do need to provide a storage infrastructure with DC functionality in order to enable it to cache data and to post its results to stable storage lazily. The DC functionality also permits the TC to lazily force its log, and to truncate its log using normal database checkpointing methods.

Our TC provides transaction functionality, regardless of how the data may be distributed across the cloud. While accessing data in the cloud currently entails large latency, our TC nonetheless achieves decent performance. Most previous efforts have either tried to avoid cloud transactions, or have severely circumscribed their scope. Our TC implementation shows that this is not required, and that it is feasible to provide full ACID transactions and enable the applications that require them to be supported in the cloud.

11. ACKNOWLEDGMENTS

12. REFERENCES

[1] D. Agrawal, S. Das, and A. E. Abbadi. Big Data and Cloud Computing: New Wine or just New Bottles? In *VLDB*, 2010.

[2] M. K. Aguilera et al. Sinfonia: A New Paradigm for Building Scalable Distributed Systems. In *SOSP*, 2007.

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

[4] S. Amer-Yahia, V. Markl, A. Y. Halevy, A. Doan, G. Alonso, D. Kossmann, and G. Weikum. Databases and Web 2.0 panel at VLDB 2007. *SIGMOD Record*, 37(1):49–52, 2008.

[5] R. S. Barga, D. B. Lomet, G. Shegalov, and G. Weikum. Recovery Guarantees for Internet Applications. *ACM Transactions on Internet Technology, TOIT*, 4(3):289–328, 2004.

[6] Philip A. Bernstein. A blog entry about Google Megastore, published online on James Hamilton's Blog "Perspectives". http://perspectives.mvdirona.com/2008/07/10/GoogleMegastore.aspx.

[7] M. Brantner, D. Florescu, D. A. Graf, D. Kossmann, and T. Kraska. Building a Database on S3. In *SIGMOD*, 2008.

[8] E. A. Brewer. Towards Robust Distributed Systems (Keynote Talk). In *PODC*, 2000.

[9] D. G. Campbell, G. Kakivaya, and N. Ellis. Extreme Scale with Full SQL Language Support in Microsoft SQL Azure. In *SIGMOD*, 2010.

[10] F. Chang et al. Bigtable: A Distributed Storage System for Structured Data. In *OSDI*, 2006.

[11] B. F. Cooper et al. PNUTS: Yahoo!'s hosted Data Serving Platform. *PVLDB*, 1(2):1277–1288, Aug. 2008.

[12] C. Curino, E. Jones, Y. Zhang, E. Wu, and S. Madden. Relational Cloud: The Case for a Database Service. Technical Report MIT-CSAIL-TR-2010-014, Massachusetts Institute of Technology, MIT, Mar. 2010.

[13] S. Das, S. Agarwal, D. Agrawal, and A. E. Abbadi. ElasTraS: An Elastic, Scalable, and Self Managing Transactional Database for the Cloud. Technical Report 2010-4, University of California, Santa Barbara, UCSB, Mar. 2010.

[14] S. Das, D. Agrawal, and A. E. Abbadi. ElasTraS: An Elastic Transactional Data Store in the Cloud. In *USENIX HotCloud Workshop*, June 2009.

[15] S. Das, D. Agrawal, and A. E. Abbadi. G-Store: A Scalable Data Store for Transactional Multi-Key Access in the Cloud. In *SoCC*, 2010.

[16] B. Debnath, S. Sengupta, and J. Li. FlashStore: High Throughput Persistent Key-Value Store. In *PVLDB*, 2010

[17] G. DeCandia et al. Dynamo: Amazon's Highly Available Key-value Store. In *OSDI*, 2007.

[18] D. J. DeWitt et al. Implementation Techniques for Main Memory Database Systems. In *SIGMOD*, 1984.

[19] J. J. Furman et al. Megastore: A Scalable Data System for User Facing Applications. In *Invited Presentation in SIGMOD Products Day*, June 2008.

[20] S. Gilbert and N. A. Lynch. Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-tolerant Web Services. *SIGACT News*, 33(2):51–59, 2002.

[21] Hbase. http://hbase.apache.org/.

[22] D. Kossmann, T. Kraska, and S. Loesing. An Evaluation of Alternative Architectures for Transaction Processing in the Cloud. In *SIGMOD*, 2010.

[23] 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.

[24] A. Lakshman and P. Malik. Cassandra: structured storage system on a P2P network. In *PODC*, 2009.

[25] A. Li, X. Yang, S. Kandula, and M. Zhang. CloudCmp: Comparing Public Cloud Providers. In *IMC*, Nov. 2010.

[26] D. Lomet and K. Tzoumas. Implementing Performance Competitive Logical Recovery. In *Under Submission*, 2010.

[27] D. B. Lomet, A. Fekete, G. Weikum, and M. J. Zwilling. Unbundling Transaction Services in the Cloud. In *CIDR*, Jan. 2009.

[28] D. B. Lomet and M. F. Mokbel. Locking Key Ranges with Unbundled Transaction Services. *PVLDB*, 2(1):265–276, 2009.

[29] Microsoft Windows Azure. http://www.microsoft.com/windowsazure/windowsazure/.

[30] Microsoft Common Log File System: http://tinyurl.com/2fwlmux.

[31] C. Mohan, D. J. Haderle, B. G. Lindsay, H. Pirahesh, and P. M. Schwarz. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. *TODS*, 17(1):94–162, 1992.

[32] TPC-W Benchmarks. http://www.tpc.org/tpcw/.

[33] H. T. Vo, C. Chen, and B. C. Ooi. Towards Elastic Transactional Cloud Storage with Range Query Support. *PVLDB*, Sept. 2010.

[34] W. Vogels. Eventually Consistent. *Communications of ACM, CACM*, 52(1):40–44, 2009.

[35] Z. Wei, G. Pierre, and C.-H. Chi. Scalable Transactions for Web Applications in the Cloud. In *Proceedings of the Euro-Par Conference on Parallel Processing*, 2009.

[36] Z. Wei, G. Pierre, and C.-H. Chi. CloudTPS: Scalable Transactions for Web Applications in the Cloud. Technical Report IR-CS-053, Vrije Universiteit, Amsterdam, The Netherlands, Feb. 2010.
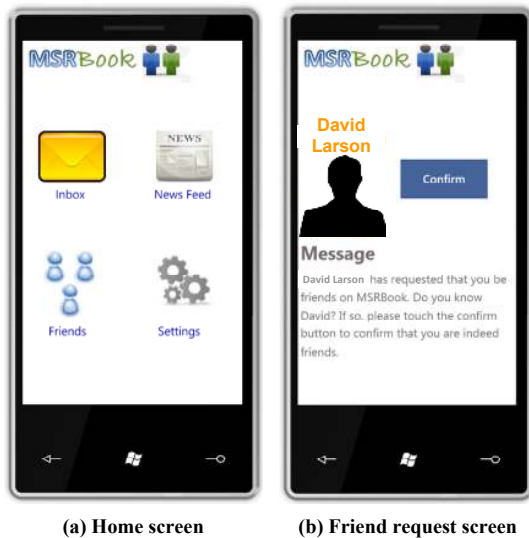
(a) Home screen   (b) Friend request screen

Figure 5: Demo app: MSRBook social network

13. DEMONSTRATION DESCRIPTION

This section provides a demonstration description of the Deuteronomy system. We cover the demo applications, data, and configuration necessary to support unrestricted transactions for a social networking application that stores its data anywhere in the cloud.

13.1 Application

The application we use to demonstrate Deuteronomy is *MSRBook*, a cloud-based social-networking application. *MSR-Book* comes in two versions: (1) *Mobile-based*, implemented as a Microsoft Windows Phone 7 application and depicted in Figure 5, and (2) *Web-based*, built for a standard web browser (screen-shot omitted due to space). Users perform actions in *MSRBook* similar to other well-known social networking applications, such as updating friend lists, posting items on friend feeds, and sending and receiving messages.

MSRBook uses Windows Azure cloud-based storage for managing its data. Deuteronomy provides transaction management for this data. *MSRBook* interfaces with the TC session manager and all transactions performed by the application use the TC interface methods covered in Section 4.2.

13.2 Data and System Configuration

For each user account, *MSRBook* stores two major pieces of data: (1) a friend list that stores who a particular user is connected with in the application, and (2) news feed items, containing news updates a user wishes to share with friends. *MSRBook* partitions its account data by user last name into three primary partitions: [a-f], [g-p], [q-z]. Each partition is managed by a separate DC, and is hosted on a different Windows Azure partition, meaning data for any two user accounts are *not* guaranteed to be co-located on the same Azure storage node. *MSRBook* interfaces with a single TC that provides transaction support for all three DCs as depicted in Figure 6. The TC, as well as all DCs, are cloud-based. The TC is implemented and hosted as a Windows Azure web role, while Each DC is implemented as a Windows Azure worker role (see [29] for details of web and worker roles).
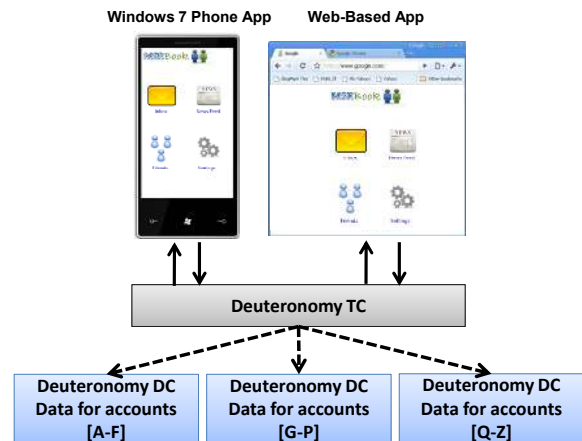

Figure 6: Demonstration scenario overview

This configuration is significant for two reasons. First, it mirrors a simple but realistic partitioning scheme typical for many cloud-based applications. Second, such a configuration does not guarantee that any two (or more) users are co-located on the same storage partition, meaning many existing cloud-based transaction approaches (e.g., Azure entity group transactions [29]) cannot provide support for a ACID transactions involving *any* two (or more) user accounts.

13.3 Demonstration Scenarios

Transactions in Deuteronomy. Our first demonstration focuses on the transactional details of users updating their friend lists in *MSRBook*. In this scenario, a user Larson notifies user Smith that he would like to connect as friends on *MSRBook*. Upon navigating to the friend notification screen (depicted in Figure 5 (b)), user Smith touches the "confirm" button in order to confirm his friendship with Larson. This action requires a transaction that updates both friend lists as well as both news feeds for the two users. Such a transaction requires only six lines of straightforward code in Deuteronomy as follows.

```
Begin Transaction
   Insert user Larson into friend list of Smith
   Insert new friend update into Larson's news feed
   Insert user Smith into friend list of Larson
   Insert new friend update into Smith's news feed
End Transaction
```

Given the configuration of the data (Larson and Smith exist in separate Azure storage partitions), such a simple and straightforward transaction is only possible in Deuteronomy. Under these partition constraints, all other cloud-based transactional support (see Section 2) require some form of eventual consistency that is orders of magnitude more complicated for application developers to implement [1].

Scalability. Our second demonstration provides a live showcase of scalability performance of Deuteronomy using the MSRBook application. Using a workload generator that simulates tens of thousands of simultaneous user friend update requests in MSRBook, we report live throughput numbers for Deuteronomy under such a workload. We also concurrently perform friend updates using the Windows Phone 7 application (Figure 5) while the simulated workload runs to show that response time is on the order of milliseconds.