Adapting Microsoft SQL Server for Cloud Computing

Philip A. Bernstein, Istvan Cseri, Nishant Dani, Nigel Ellis, Ajay Kalhan, Gopal Kakivaya,
David B. Lomet, Ramesh Manne, Lev Novik, Tomas Talius

Microsoft Corporation
One Microsoft Way, Redmond, WA, 98052-6399, U.S.A.

{philbe, istvanc}@microsoft.com
nishant_dani@hotmail.com
{nigele, ajayk, gopalk, lomet, rmanne, levn, tomtal}@microsoft.com

Abstract— **Cloud SQL Server is a relational database system designed to scale-out to cloud computing workloads. It uses Microsoft SQL Server as its core. To scale out, it uses a partitioned database on a shared-nothing system architecture. Transactions are constrained to execute on one partition, to avoid the need for two-phase commit. The database is replicated for high availability using a custom primary-copy replication scheme. It currently serves as the storage engine for Microsoft's Exchange Hosted Archive and SQL Azure.**

I. INTRODUCTION

Large-scale web applications generate classical database workloads consisting of transactions and queries. Although the required functionality is classical, the system-level requirements can be quite daunting. To handle heavy workloads, they need to scale out to thousands of servers. For low-cost data center operation, they need to run on inexpensive unreliable commodity hardware, yet they still need to be highly available. Since labor is a major cost, they need to automate system management functions, such as machine and disk failover, load balancing, and dynamic repartitioning of data. Moreover, they need to offer predictable, fast response time.

These requirements on web applications imply the same set of requirements on the database system used by the applications. To meet these requirements, many providers of large-scale web services have developed custom record-oriented storage systems to support their applications [6][7][8]. Given this research literature, one may be left with the impression that a custom record-oriented storage system is the only practical way to satisfy these requirements.

Microsoft has taken a different approach to these requirements, namely to build its distributed storage system using its relational database system product, Microsoft SQL Server, as its core. It is the first commercial system we know of that takes this approach. The resulting system, which we call Cloud SQL Server, exposes much of SQL Server's functionality. This includes aggregation, full-text queries, and referential constraints, views, and stored procedures—most of which are not supported by custom record stores. Thus, users of SQL Server can learn to use the system with only modest effort. Depending on how it is used, there are some functionality restrictions to satisfy the cloud requirements, which are described in Section II-A.

One important capability of Cloud SQL Server is its support of ACID transactions. Although this is a standard and popular feature of relational database system products, it is not universally supported by record managers for web applications, such as Amazon's Dynamo and Yahoo's PNUTS. It is often argued that web-application developers do not need transactions and that a system that supports transactions will not scale, primarily due to the need for two-phase commit. Our experience is the opposite. Our users asked for transactions. To get transactions, they were willing to accept a constraint on schema structure that ensures transactions are non-distributed and hence do not need two-phase commit. We explain the constraint in the next section.

To attain high availability on unreliable commodity hardware, the system replicates data. The transaction commitment protocol requires that only a quorum of the replicas be up. A Paxos-like consensus algorithm is used to maintain a set of replicas to deal with replica failures and recoveries. Dynamic quorums are used to improve availability in the face of multiple failures. Replication protocols are described in Section IV.

We designed Cloud SQL Server in early 2006, and it was first deployed in mid-2009. It is currently used as the storage system for two large-scale web services: Exchange Hosted Archive [15], an e-mail and instant messaging repository that helps organizations manage retention, compliance, e-discovery, and regulatory requirements; and SQL Azure [14], a relational database system offered as a service, as part of the Windows Azure™ computing platform.

A high level view of Cloud SQL Server appeared in [5]. This paper extends [5] with details of the data model, replication and recovery protocols, applications, and performance measurements. To the best of our knowledge, it is the first description of a cloud-oriented scaleout version of a widely-used relational database product that supports ACID transactions.

The paper is organized as follows. Section II describes the data model, including the constraints that ensure transactions are not distributed. Section III presents the overall system architecture. Section IV explains the approach to replication. Section V discusses applications. Section VI presents some

performance measurements. Section VII covers related work, and Section VIII is the conclusion.

II. Data Model

A. Logical Data Model

Cloud SQL Server needs to offer relational access to very large datasets. This requires scaling out to at least hundreds of machines. To get fast and predictable performance, we want each transaction to be non-distributed. That is, we want all of the data that each transaction directly reads and writes (as opposed to replicated writes) to be stored on a single server. To ensure this holds, we restrict the database schema and transaction behavior as follows.

In Cloud SQL Server, a logical database is called *a table group*. A table group may be *keyless* or *keyed*. A keyless table group is an ordinary SQL Server database. In particular, there are no restrictions on the choice of keys for its tables. By contrast, if a table group is keyed, then all of its tables must have a common column called the *partitioning key*. For example, Figure 1 shows a keyed table group with two tables, Customers and Orders. The partitioning key is (Customer) Id.

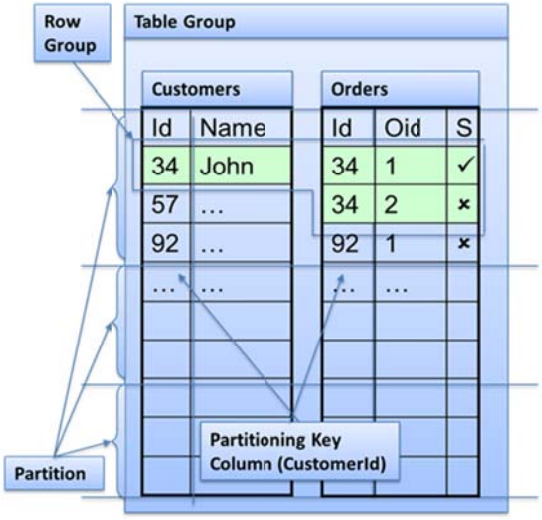

Figure 1 Cloud SQL Server's data model

The partitioning key need not be a unique key of each relation. For example, Id happens to be a unique key of the Customers table, but it is not a unique key of the Orders table. Similarly, the partitioning key need not be the key of the clustered index that is used to store rows of the table. For example, the cluster key of the Orders table is a composite key [Id, Oid].

A *row group* is the set of all rows in a table group that have the same partition key value. For example, in Figure 1, the first row of Customers plus the first two rows of Orders comprise a row group. *Cloud SQL Server requires that each transaction executes on one table group. If the table group is keyed, then the transaction can read and write rows of only one row group.*

Given these requirements, there are two ways one can build a transactional application that scales out. One way is to have the application store its data in multiple table groups, where each table group can fit comfortably on a single machine. In effect, the application takes responsibility for scaling out by partitioning data into separate table groups and explicitly referring to the table groups in the application code. The second way is to design the database as a keyed table group, so that Cloud SQL Server performs the scaleout automatically.

We say that the *consistency unit* of an object is the set of data that can be read and written by an ACID transaction. Given the above requirement, the consistency unit of a keyed table group is the row group, while that of a keyless table group is the whole table group. Each copy of a consistency unit is always fully contained in a single instance of SQL Server running on one machine. Hence, there is never a need for two-phase commit.

Our decision to create an execution model that avoids two-phase commit was driven by two factors, blocking and performance. First, a participant in two-phase becomes blocked if the coordinator fails during the transaction's uncertainty period, that is, after the instance has acknowledged having prepared the transaction and before it is notified that the transaction committed or aborted. To unblock the transaction, the system may need to resort to a heuristic decision (i.e., a guess) or operator intervention. Both choices are unattractive, especially in a system that is intended to grow to a large scale and to host a large number of independent applications. Second, two-phase commit introduces a large number of messages that follow a relatively random distribution pattern and hence are not easily batched. This overhead can be significant, making it hard to offer predictable performance.

A query can execute on multiple partitions of a keyed table group with an isolation level of read-committed. Thus, data that the query reads from different partitions may reflect the execution of different transactions. Transactionally consistent reads beyond a consistency unit are not supported.

During the initial design of Cloud SQL Server, we gathered the requirements of numerous Microsoft groups that were supporting or developing web-based applications. All of them found keyed table groups to be acceptable. The main problem with this restriction is many-to-many relationships. For example, a movie review application relates users to their reviews, which is a many-to-many relationship. A relational schema could represent this by MovieReview(UserID, MovieID, review) with UserID-MovieID as the compound key. If UserID is the partitioning key of the table, then one can write a transaction that accesses all of a user's reviews, but not one that updates all of the reviews of a movie (because a transaction cannot span different UserID's). In such cases, we found transactional access to the relationship to be required in only one direction (e.g., from user to the user's movie reviews). In the other direction (i.e., from movie to the movie's reviews), only queries need to be supported, and they need not execute as ACID transactions. That is, a query can execute with loose consistency guarantees. For example, a query could be decomposed to execute independently against many different row groups, or a materialized view of the

relationship could be created that is keyed on MovieID and is refreshed only periodically.

B. Physical Data Model

At the physical level, a keyed table group is split into *partitions* based on ranges of its partitioning key. The ranges must cover all values of the partitioning key and must not overlap. This ensures that each row group resides in exactly one partition and hence that each row of a table has a well-defined home partition.

In the rest of this paper, unless otherwise noted, we use the term *partition* to refer either to a partition of a keyed table group or to an entire keyless table group.

Partitions are replicated for high availability. We therefore say that a partition is the *failover unit*. Each replica is stored on one server. Since a row group is wholly contained in one partition, this implies that the size of a row group cannot exceed the maximum allowable partition size, which is at most the capacity of a single server.

Replicas of each partition are scattered across servers such that no two copies reside in the same "failure domain," e.g., under the same network switch or in the same rack. Replicas of each partition are assigned to servers independently of the assignment of other partitions to servers, even if the partitions are from the same table group. That is, the fact that replicas of two partitions are stored on the same server does not imply that other replicas of those partitions are co-located on other servers.

For each partition, at each point in time one replica is designated to be the *primary*. A transaction executes using the primary replica of the partition (or simply, the *primary partition*) that contains its row group and thus is non-distributed. The primary replica processes all query, update, and data definition language operations. It ships its updates and data definition language operations to the secondaries using the replication mechanisms described in Section IV. The system currently does not allow (potentially stale) reads of secondary replicas, though it would be a simple change if additional read bandwidth is required.

Since a transaction executes all of its reads and writes using the primary partition that contains its row group, the server that directly accesses the primary partition does all the work against the data. It sends update records to the partition's secondary replicas, each of which applies the updates. Since secondaries do not process reads, each primary has more work to do than its secondaries. To balance the load, each server hosts a mix of primary and secondary partitions.

On average, with *n*-way replication, each server hosts one primary partition for each *n*-1 secondary partitions. Obviously, two replicas of a partition are never co-located. Figure 2 illustrates a possible deployment of four logical partitions PA, PB, PC, and PD of table P, each of which has a primary, e.g. PA_P, and two secondaries, e.g., PA_{s1}, and PA_{s2}. Each server hosts one primary and two different secondary partitions.

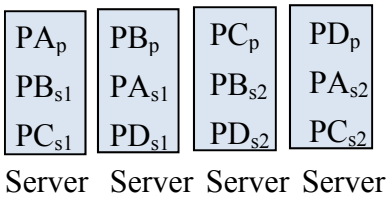

Figure 2 Load balancing primary and secondary replicas

Another benefit of having each server host a mix of primary and secondary partitions is that it allows the system to spread the load of a failed server across many live servers. For example, suppose a server S hosts three primary partitions PE, PF, and PG. If S fails and secondaries for PE, PF, and PG are spread across different servers, then the new primary partition for PE, PF, and PG can be assigned to three different servers.

Since some partitions may experience higher load than others, the simple technique of balancing the number of primary and secondary partitions per node might not balance the load. The system can rebalance dynamically using the failover mechanism to tell a secondary on a lightly loaded server to become primary and either demoting the former primary to secondary or moving the former primary to another server.

A keyed table group can be partitioned dynamically. If a partition exceeds the maximum allowable partition size (either in bytes or the amount of operational load it receives), it is split into two partitions. To do this quickly, the design allows for the possibility of splitting partitions dynamically using existing replicas. For example, suppose partition A's primary replica is on server X, and its secondaries are on servers Y and Z. Then to split A into two partitions A1 and A2, each replica is split. For good load balancing, the replicas can take on different roles. For example, A1 on X may be designated the primary with Y and Z holding secondaries, and A2 on Y is designated the primary with X and Z holding secondaries, all without moving any data between servers.

Given that update rates may vary between partitions, dynamic reassignment of the primary and secondary roles may be needed. Like partition splitting described above, the design allows for this to be done without any data movement. One simply tells the global partition manager that a primary replica at the busy server is now a secondary, while a secondary for this partition at another server is now the primary. This transition needs to be made "between transactions," which means that transactions executing on the primary in the original deployment need to commit or abort before one of the secondaries takes on the role of primary. Additional replicas of partitions can be built at less-busy servers at leisure to aid in this process.

III. System Architecture

A *SQL Server instance* is a process running SQL Server. A SQL Server instance can manage many independent databases. However, in Cloud SQL Server, all of the user databases managed by a SQL Server instance are stored in one database. Each user database U is a *sub-database* of the instance's database and is isolated from all other user databases. The

sub-database contains the partition replicas of U that are stored at that instance, plus U's schema information. Supporting multiple virtual databases in a single SQL Server instance saves on memory for internal database structures in the server and enables the databases to share a common transaction log, which improves performance.

To attain high availability, a large-scale distributed system needs a highly-reliable system management layer that, among other things, maintains up/down status of servers, detects server failures and recoveries, and performs leadership election for various roles in a cluster. The layer used by Cloud SQL Server, called the *distributed fabric*, implements these capabilities using a distributed hash table (see Figure 3). The distributed fabric runs as a process on every server that runs a SQL Server instance. It is a distinct component that is currently being used for other services at Microsoft. It is a major system in its own right, many of whose details are outside the scope of this paper.

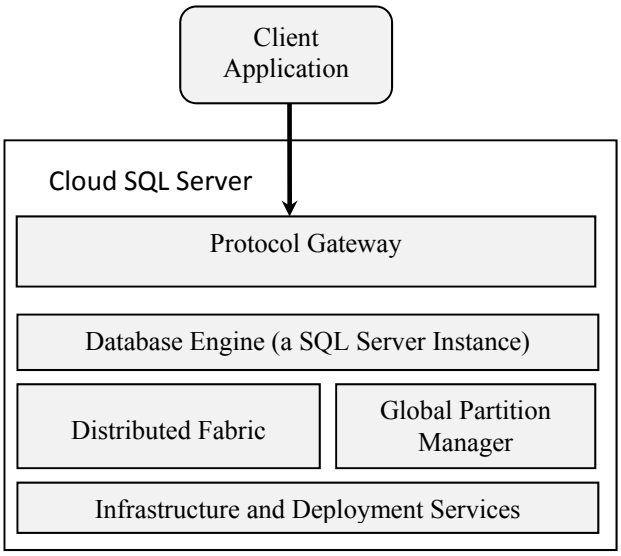

Figure 3 Cloud SQL Server layers

There is a highly-available *global partition manager* that maintains a directory of information about partitions. For each partition P, it knows the key range that defines P. And for each replica R of P, it knows R's location and its state, such as whether it is the primary, a secondary, a former primary or secondary (in a past life), becoming primary, being copied, or being caught up. The set of operational replicas of a partition is called a *configuration*.

When a server fails, the distributed fabric reliably detects the server failure and notifies the global partition manager. The global partition manager reconfigures the assignment of primary and secondary partitions that were present on the failed server. When a server recovers, it announces its presence to the global partition manager along with a summary of the state of its replicas. Using this information, the global partition manager can decide whether to refresh, replace, or discard them. Section IV explains the handling of replica failures and recoveries, including those of the global partition manager.

Some failures are planned, notably upgrades of Cloud SQL Server or an application that uses it. The *infrastructure and deployment services* layer is responsible for upgrades and other activities of the physical machines within a cluster. It loads the initial software image onto a machine based upon its specific role in the cluster. The roles include: front end machines, which handle protocol activity; database nodes, which manage the cluster data; and cluster infrastructure roles, which perform global state management, operational data gathering and processing, and other cluster-specific tasks. The infrastructure and deployment services also run various workflows in response to requests from the distributed fabric such as node restart, shutdown, and software reimage.

Upgrades are performed on an *upgrade domain*, which is a subset of the servers in a failure domain. The upgrade process begins by sending a request to the global partition manager to determine whether taking down each server in the domain will make a partition unavailable, because the partition will have too few replicas. For example, suppose a partition normally has three replicas but currently only two are active. Since taking down one of the two replicas would cause a quorum loss, an upgrade of the replica's server is delayed. If the server does not contain a partition whose loss would degrade the partition's availability, then the global partition manager allows the server to be upgraded. For all upgradable servers in the upgrade domain, the system management layer is invoked to take down the server, install the upgraded software, and activate it.

A well-designed upgrade should, of course, allow existing applications to run without modification. That is, it should be backward compatible. In addition, the upgraded software must be able to interoperate with its previous version, so it can run correctly during the upgrade process, when the system has a mix of upgraded and non-upgraded servers.

Typically, users are prevented from using the new features of the upgraded version until all servers have been upgraded. Thus, upgrades have two-phases: the first phase rolls out the code that understands the new protocols and features and the second phase actually enables them.

An instance of Cloud SQL Server typically manages multiple disks, each of which is private to that instance. If one of the instance's disks fails, the instance is regarded as having failed. Currently, each server runs one instance, but the system is designed to allow multiple instances per server and multiple databases per instance.

The system is accessed using a *protocol gateway* that enables client applications to connect to Cloud SQL Server. The protocol gateway supports the native wire protocol of SQL Server. It is responsible for accepting inbound database connection requests from clients and binding to the node that currently manages the primary replica. It coordinates with the distributed fabric to locate the primary and renegotiates that choice in case a failure or system-initiated reconfiguration causes the election of a new primary. It also masks some failure and reconfiguration events from external clients by renegotiating internal sessions while maintaining the protocol session state between the client and protocol gateway.

IV. Logging and Replication

A. Replica Algorithm

In this section, we describe the processing of updates to replicas during normal operation. Section IV.B covers some design details, and Section IV.C covers failure handling.

The propagation of updates from primary to secondary is shown in Figure 4. A transaction T's primary partition generates a record containing the after-image of each update by T. Such update records serve as logical redo records, identified by table key but not by page ID [13]. These update records are streamed to the secondaries as they occur. If T aborts, the primary sends an ABORT message to each secondary, which deletes the updates it received for T. If T issues a COMMIT operation, then the primary assigns to T the next *commit sequence number* (CSN), which tags the COMMIT message that is sent to secondaries. Each secondary applies T's updates to its database in commit-sequence-number order within the context of an independent local transaction that corresponds to T and sends an acknowledgment (ACK) back to the primary. After the primary receives an ACK from a quorum of replicas (including itself), it writes a persistent COMMIT record locally and returns "success" to T's COMMIT operation.

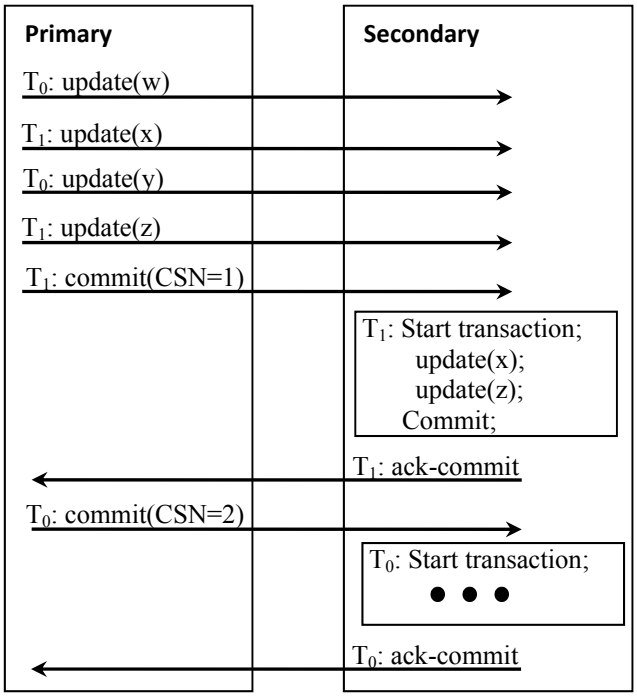

Figure 4 Primary-to-secondary replication

A secondary can send an ACK in response to a transaction T's COMMIT message immediately, before T's corresponding commit record and update records that precede it are forced to the log. Thus, before T commits, a quorum of servers has a copy of the commit. If servers are unlikely to experience correlated failures (e.g., because they are in different data centers), then this provides a satisfactory degree of fault tolerance. If tolerance of correlated server failures is needed, then each secondary could also be required to flush T's commit record to disk before sending an ACK. Currently, SQL Azure takes the latter approach.

Updated records are eventually flushed to disk by primaries and secondaries. Their purpose is to reduce the amount of catching up that a server needs to do should it fail and recover.

Updates for committed transactions that are lost by a secondary (e.g., due to a crash) can be acquired from the primary replica. The recovering replica sends to the primary the commit sequence number of the last transaction it committed. The primary replies by either sending the queue of updates that the recovering replica needs or telling the recovering replica that it is too far behind to be caught up. In the latter case, the recovering replica can ask the primary to transfer a fresh copy.

A secondary promptly applies updates it receives from the primary server, so it is always nearly up-to-date. Thus, if it needs to become the primary due to a configuration change (e.g., due to load balancing or a primary failure), such re-assignment is almost instantaneous. That is, secondaries are hot standbys and can provide very high availability. Secondaries can also be used as read-only copies (i.e., not within update transactions). Although their isolation level is only read-committed, the schema information for sub-databases on secondaries is transactionally consistent.

B. Replication Design Details

In the beginning of Section IV-A, we said that the primary sends after-images, rather than redo log records that refer to physical offsets on pages. The benefit of this approach is that partition replicas do not need to be physically identical. This avoids the need to align the disk allocation of replicas of a partition between servers. It also enables certain optimizations. For example, in the partition-splitting strategy of Section II-B, a partition can be logically split without being physically split. In this case, a page with keys at the boundary of the two partitions might have records from both partitions. Two different replicas might serve as primary for those two partitions. If updates referred to physical addresses on the page, then updates generated for different logical partitions might collide on the same physical address, which would corrupt the page; for example, the updates might both insert a record into the same empty page slot.

After-images of both clustered and non-clustered indices are logged and sent. This speeds up processing at the secondaries in two ways. First, it avoids having to push each update on a secondary through the upper layers of the relational engine, which determines which non-clustered indices are affected by an update to a clustered index. And second, it avoids having to perform a read at a secondary before applying each update.

Much effort was invested in identifying hardware failure modes and working around them. For example, after discovering faulty behavior of some network interface cards, it was decided to improve error detection by signing all messages. Similarly, after observing some bit-flips on disks, we enabled checksums. A third example is SATA drives that acknowledge a write-through as soon as the data is in the

drive's cache. Additional work was needed to guarantee that log handshakes are honored, such as flushing the cache at critical points in the execution.

In Section II-B we said the same replication model is used for both data manipulation and data definition operations. This simplifies the handling of schema changes. For example, it avoids the need for special logic to synchronize updates with schema changes that the updates depend on.

As another example, a job service is used to deploy schema changes to all partitions in a keyed table group. This service applies the schema change to each partition's primary. The replication mechanism ensures that the schema change is eventually applied to all replicas. Since some primaries may be temporarily unreachable, the job service tracks which partitions have processed the schema change and periodically retries changes to partitions that have not yet been updated. Each partition stores its current schema version, so the schema change script is made idempotent by checking the schema version before applying the changes.

C. Coping with Replica Failures

We describe the basic approach to failure handling by walking through some of the major scenarios. The protocol details are quite complicated and are not covered here.

Since only a quorum of replicas needs to send an ACK in response to a commit request, availability is unaffected by a single secondary failure. In the background, the system simply creates a new replica to replace the failed one. With a large enough replica set, multiple secondaries can fail concurrently without affecting availability.

Cloud SQL Server is designed for a deployment that has enough bandwidth and computing power to accommodate the complete rebuild of a replica. The global partition manager chooses a lightly-loaded server that will host the new replica, tells it to copy the primary replica to that server, and updates its directory to reflect this change.

If a replica fails for only a short time and then recovers, it can be caught up. Its server asks an operational replica to send it the tail of the update queue that the replica missed while it was down. This saves on both bandwidth and computation, and usually shortens the replica's recovery time.

If a primary replica fails, then a secondary must be designated as the new primary and all of the operational replicas must be reconfigured according to that decision. The first step in this process relies on the global partition manager described in Section III. It chooses a leader to rebuild the partition's configuration (i.e., operational replica set). The leader attempts to contact the members of the entire replica set. If the leader cannot contact a quorum of the pre-failure replica set, the system declares a "permanent quorum loss" and requires an intervention to proceed. Otherwise, since it can contact a quorum of them, the recovery protocol can ensure that there are no lost updates. The leader determines which secondary has the latest state. That most up-to-date secondary propagates updates that are required by the secondaries that are less up-to-date.

Suppose that before the primary failed, a transaction's updates reached at least one secondary but less than a quorum. Therefore, the transaction did not commit before the failure and the client did not receive a notification of the transaction's outcome. Suppose a secondary that received those updates survives in the new configuration. Then as described above, during recovery the transaction's updates are propagated to all the secondaries in the new configuration. Thus, the transaction is committed before the service is open to the public, albeit with a different quorum than the one that existed while it was executing. However, the client still does not receive a notification of completion, because the client is not communicating with the new primary. This is a normal behavior that occurs in transaction mechanisms when a communication failure is encountered during the processing of a commit request. The problem can be avoided by adding a persistent, transactional queuing system on top, which guarantees the delivery of transaction outcomes ([4], Chapter 4).

Suppose a configuration has N replicas. After a replica failure is detected, and before another replica is ready to replace it, the global partition manager "downshifts" the replica set for the partition to be N-1. This improves the fault tolerance of the partition while it is operating with fewer replicas than normal. For example, suppose N=3 and a replica fails. Then the global partition manager downshifts to N=2, with a write-quorum of 2 and read-quorum of 1. Thus, if another replica fails before the third replica recovers or is replaced, then the system knows the remaining replica is consistent and up-to-date (something it would not have known if N were still 3). It can therefore rebuild replicas from the remaining replica without declaring a quorum loss.

The preceding discussion assumes there is a unique "most up-to-date" secondary. The global partition manager ensures it can identify such a secondary by totally ordering the quorums of each partition, P. It assigns an *epoch number* to each configuration of P that is one greater than the epoch number of the previous configuration of P. Commit records in the log are identified by both a commit sequence number and an epoch number. Among the replicas in a replica set with the highest epoch number, the one with the highest commit sequence number has the latest state. This is the replica chosen as most-up-to-date after a replica set is reconfigured.

The global partition manager's database replicas are treated just like other replicated partitions, with one exception. If the primary replica of the global partition manager fails, the distributed fabric chooses a new primary from the set of global partition manager replicas. It uses an implementation of Paxos [11], thereby ensuring the instances of the global partition manager itself are totally ordered into epochs. Current deployments have seven replicas of the global partition manager's partitions, five replicas of some critical application metadata, and three replicas of user table-group partitions.

V. Applications

Cloud SQL Server is currently used as the database system for Exchange Hosted Archive (EHA) and SQL Azure.

A typical customer of EHA is an organization, which uses EHA for archiving messages and ensuring compliance, for example, by implementing retention policies. EHA takes an

email stream from any messaging system as input. Specifically, the customer's message transfer agent is configured to forward a copy of each incoming message (i.e., email, instant message, voicemail, etc.) to a customer-specific Microsoft email address. EHA stores the messages in Cloud SQL Server and offers extensive query functionality on top. It currently stores hundreds of terabytes on more than a thousand servers. The system supports each customer on its own cluster of machines.

To attain this degree of scale-out, EHA uses table-group partitioning. Some customer databases have over a thousand partitions. The partitioning key includes tenant, time, and a content hash of the email message. So given a message, EHA knows in which partition it will land. Therefore, it can sort the message stream by partition for faster bulk loading.

EHA makes heavy use of SQL Server functionality, for document discovery during legal proceedings (many large companies are sued daily) and for emergency email service when the corporate email server is down. This highlights the value of building web-scale database storage on a SQL engine. For example, it creates many non-clustered indexes. It uses selection, aggregation, full-text queries, and referential constraints. It also makes extensive use of stored procedures. If EHA were built on a key-value store, all of these functions would have to be implemented by the application.

SQL Azure is a multi-tenant SQL database service available on the Windows Azure platform. It too uses Cloud SQL Server for storage. In its first release, it uses keyless table groups, so each user database must fit in one partition. This enables the system to offer richer SQL functionality than if keyed table groups were used, since a partition behaves like an ordinary set of SQL tables. It also simplifies schema upgrades, since data definition operations can be applied to the one-and-only primary replica of a table group without using the job service. To support multiple keyed table groups, the implementation of many SQL operations will need to be extended to make partitions transparent.

VI. PERFORMANCE

The Cloud SQL Server engine is a modified version of the Microsoft SQL Server packaged product. Since the core engines are nearly the same, we would expect their performance to be very similar too.

To test this, we used a variant of TPC-C, a benchmark defined by the Transaction Processing Council to measure OLTP performance [17]. The benchmark was modified in our tests to require much less memory and disk I/O than the standard benchmark, to more closely match the configurations run in data centers for cloud computing. It is very different than configurations used for official TPC-C results tuned for optimal performance or price-performance, which typically have ten times as much RAM and 100-500 disk drives per CPU. Our results shown here provide meaningful relative performance comparisons, but are not comparable to published TPC-C results.

We compared the performance of the two software systems running on the same hardware: a server machine with two dual-core 2.33 GHz processors, 8GB of RAM, and 4 hard drives, running Windows Server 2008. This configuration is typical of the kinds of machines that are run in data centers for cloud computing.

In all tests that we report here, one hard drive stores the database and one stores the log. In each test, steady state throughput was reached in 5-10 minutes, after which the test was run for about two hours. We report on behavior based on average performance during the steady-state period.

We ran a TPC-C workload with a database that fits in RAM, to factor out the effect of the small disk configuration. In this case, the throughput of Cloud SQL Server was 8.4% lower than that of the SQL Server product (see Figure 5). This performance difference comes from two main sources. One source is the activities performed by Cloud SQL Server outside of the database engine itself. These activities include monitors, watchdogs, and traces, plus a backup every five minutes, all of which Cloud SQL Server runs as part of its normal service, and none of which run on the SQL Server product during the benchmark. The second source is low-level code optimizations that have been applied to the SQL Server engine but not yet to the Cloud SQL Server code.

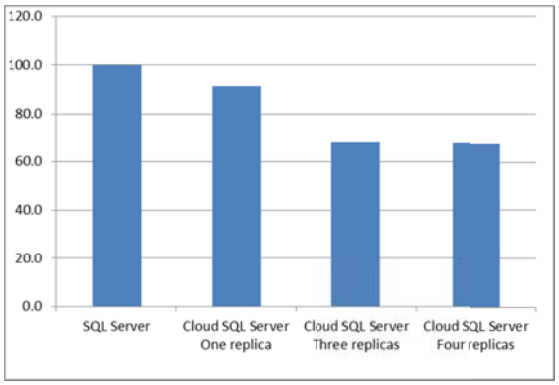

Figure 5 Relative throughput of SQL Server and Cloud SQL Server

User connections are a scarce resource that can limit throughput. When too many connections are present, the wait time for a connection to acquire a worker thread can become the main bottleneck. Cloud SQL Server uses many worker threads for its background processing activities. As a consequence, the SQL Server product was able to support nearly double the number of user connections than Cloud SQL Server, with each system executing at its peak throughput.

We ran a second test to quantify the impact of replicas on overall throughput. We used the same hardware configuration as above, first with just a primary, and then with secondaries, where each replica ran on a separate machine. The machines were on different racks, a configuration that isolates them from the failure of a rack (whose machines share common power and networking). Thus, a message between a primary and secondary had to pass through three network switches—two top-of-rack switches and a switch that connects the racks. We again ran TPC-C using a small database size that fits in

RAM, to eliminate the disks as a source of delay except for handshakes with the log.

With two secondaries (i.e., three replicas in Figure 5), throughput dropped by 25.8% compared to a system with a primary and no secondaries (i.e., one replica in Figure 5). This drop in throughput was due to delays introduced by the network, by waits for worker threads, and by handshaking with the logs on the secondaries. The addition of a third secondary had a negligible impact; it reduced throughput by only an additional 0.4% beyond the reduction using two secondaries.

We ran a third experiment to show the effect of balancing the load of primary and secondary partitions. In the first configuration (Configuration 1 in Figure 6), we ran three primary partitions on one machine. Two other machines ran secondaries for each of the partitions. In the second configuration, each machine ran one of the primaries and two of the secondaries (Configuration 2 in Figure 6). Each machine had two 4-core 2.1 GHz processors and 30GB of RAM running Windows Server 2008, with separate 10,000 rpm 300GB SATA drives to store the database and log.

Configuration 1 - Concentrated		
Machine 1	Machine 2	Machine 3
P1	S1	S1
P2	S2	S2
P3	S3	S3

Configuration 2 - Balanced		
Machine 1	Machine 2	Machine 3
P1	P2	P3
S2	S1	S1
S3	S3	S2

Figure 6 Test configurations

To compare these configurations, we ran TPC-C with a 5 GB database per partition. The three partitions comfortably fit in RAM, so the vast majority of I/O on the database disks was for writes (about 97%). In this case, the throughput of configuration 2 was 7.2% higher than that of configuration 1. As expected, resource utilization in configuration 2 is more balanced than in configuration 1. Per-node processor utilization in configuration 2 is 29%, whereas in configuration 1, machine 1's utilization is 35% vs. 20% for machines 2 and 3. Configuration 2 shows a small improvement in disk I/O balance as well.

In a final set of tests, we manually killed a primary copy and measured the elapsed time until the first transaction was able to execute on the newly elected primary. In 95% of the runs, failover time was less than 30 seconds. In the remaining 5% of the runs, the failover time was less than 60 seconds.

A performance study of transaction processing services in the cloud was published in [11]. The paper reports measurements of the TPC-W benchmark [18] for a variety of cloud-based services, including SQL Azure, in February 2010. SQL Azure was tied for the highest throughput reported, and had the lowest cost for medium to large workloads.

VII. Related Work

At its core, Cloud SQL Server is a parallel database system that uses data partitioning on a shared-nothing architecture. The benefits of this architecture were demonstrated in the 1980s in many such systems, such as Bubba, Gamma, Tandem, and Teradata. DeWitt and Gray [10] provide an excellent summary of that work.

We focus here on recent papers describing commercial systems with similar goals to Cloud SQL Server: Bigtable/Megastore, Dynamo, and PNUTS. All of these systems are intended to scale out to a large number of servers with very high availability.

Google's Bigtable offers atomic read and rite operations on rows of a single table [6]. A table is partitioned into tablets, which is the unit of distribution and load balancing. Rows, tables and tablets in Bigtable are analogous to row groups, table groups, and partitions in Cloud SQL Server. In Bigtable, tablets and update logs are stored in the Google File System and hence are replicated. It uses the Chubby lock manager for a similar role to the global partition manager in Cloud SQL Server, although the underlying mechanisms of Chubby and the global partition manager are rather different. Column values in Bigtable are timestamped, to capture version history. Although SQL Server, and hence Cloud SQL Server, does not currently offer multiversion databases, it does support snapshot isolation. Hence, a query within a partition can return a consistent result and be unaffected by concurrent updates.

Google's Megastore adds transactions to Bigtable [1][2]. In Megastore, a transaction can read and write entities in an entity group, which is a set of records, possibly in different Bigtable instances, that have a common prefix of their primary key. An entity group corresponds to a row group in Cloud SQL Server. Megastore uses optimistic concurrency control, whereas Cloud SQL Server uses locking. Megastore uses a per-entity-group replicated transaction log to ensure atomicity and durability and a Paxos protocol to handle recovery from failures. By contrast, Cloud SQL Server's log is shared by all table groups managed by a server instance. Megastore also offers declarative schema and non-clustered indices.

Amazon's Dynamo offers atomic read and write operations on key-value pairs [9]. The value is an unstructured payload. So there is no schema or table abstraction. It uses a multi-master replication algorithm based on vector clocks, rather than log-based replication as in Cloud SQL Server. This leads to a weaker consistency model, where an application can read different values from different copies. This choice offers high write-availability by allowing a write to update less than the whole replica set and hence requires a read quorum greater than one. But there is a cost to application programmers, who need to cope with value skew between replicas.

Like Bigtable, Yahoo!'s PNUTS offers atomic read and write operations on rows of a single table [7]. For update durability, it uses the Yahoo! Message Broker to send update log records to a replicated log. It partitions the table by key value. For each key value, it uses a master row to totally order updates to that row. This is similar to the storage system for Microsoft Live services [3], but is unlike Bigtable or Cloud SQL Server. Although PNUTS does not support multi-record transactions, it has an option where a write operation can depend on a particular earlier version of the row. This enables the implementation of an ACID transaction that reads and later writes the same row.

A new research project at Stanford, called RAMClouds, proposes a partitioned main memory database for large-scale web services [16]. Although the design is a work-in-progress, the paper describes mechanisms that are similar to Cloud SQL Server's, such as replicating the update log and waiting till replicas respond before reporting completion of the update.

VIII. CONCLUSION

We have described Cloud SQL Server, a distributed storage system for web-scale applications based on Microsoft SQL Server. It requires that a table group (i.e., a user database) is either keyless, meaning that its tables are co-located, or it is keyed, meaning that its tables have a common partitioning key and that every update transaction reads and writes records with a single value of that partitioning key. This ensures that every transaction can be executed on one server. A keyed table group is partitioned by value ranges of the partitioning key and each partition is made highly available by primary-copy replication. The system is currently being used as the storage engine for Microsoft's Exchange Hosted Archive and SQL Azure.

Previous approaches have built custom storage systems for web applications and have gained some advantages from non-ACID execution models. Cloud SQL Server demonstrates that a scalable and highly-available storage system for web applications can be built by extending a classical relational database system, thereby offering users a familiar programming model and the functionality of a powerful SQL engine. It is the first such commercial system that we know of.

There is much room for future research in this area. The design of Cloud SQL Server was heavily influenced by that of SQL Server, on which it is built. Other mechanisms might be preferable when building on different database infrastructure, including storage systems that do not support transactions [13]. Another direction is to develop techniques to analyze applications to (semi-)automatically partition them, such as the one proposed in [8]. Even better would be an architecture that drops the partitioning requirement yet still obtains satisfactory performance with a sufficiently low probability of blocking due to two-phase commit. With or without partitioning, the system needs to be self-monitoring and self-managing. For example, it should be able to monitor fragmentation and opportunistically defragment. Support for distributed queries is yet another challenge, especially, balancing the load so that queries and updates meet a given service level agreement. In a multi-tenant environment, users need to get the resources they paid for, while the system needs to load balance users across servers. Part of the solution may be defining the parameters of a service level agreement in a way that can be supported with high probability. And of course, energy efficiency is a growing concern. No doubt, as more web-scale distributed storage systems are deployed, many more research problems will emerge.

ACKNOWLEDGMENTS

We gratefully acknowledge the hard work of the entire Cloud SQL Server, EHA, and SQL Azure teams who built these systems and continue to improve upon them. In particular, we thank Erik Cai and Henry Zhang for the performance measurements included in Section VI.

REFERENCES

[1] R. Barrett, "Migration to a Better Datastore," http://googleappengine.blogspot.com/search?q=megastore

[2] P.A. Bernstein, Guest posting on James Hamilton's blog, *Perspectives*, about Google's Megastore http://perspectives.mvdirona.com/2008/07/10/GoogleMegastore.aspx

[3] P.A. Bernstein, N. Dani, B. Khessib, R. Manne, D. Shutt, "Data Management Issues in Supporting Large-Scale Web Services," *IEEE Data Eng. Bull.* 29(4): 3-9 (2006)

[4] P.A. Bernstein, E. Newcomer, *Principles of Transaction Processing, Second Edition*, Morgan Kaufmann, 2009.

[5] D. G. Campbell, G. Kakivaya, N. Ellis, "Extreme Scale with Full SQL Language Support in Microsoft SQL Azure," *Proc. SIGMOD 2010*, pp. 1021-1023.

[6] F. Chang, J. Dean, S. Ghemawat, W.C. Hsieh, D.A. Wallach, M. Burrows, T. Chandra, A. Fikes, R.E. Gruber, "Bigtable: A Distributed Storage System for Structured Data", *ACM Transactions on Computer Systems*, 26(2): 4:1 - 4:26, 2008.

[7] B.F. Cooper, R. Ramakrishnan, U. Srivastava, A.Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, R. Yerneni, "PNUTS: Yahoo!'s Hosted Data Serving Platform," *PVLDB* 1(2): 1277-1288 (2008)

[8] C. Curino, Y. Zhang, E.P.C. Jones, S. Madden: Schism, "A Workload-Driven Approach to Database Replication and Partitioning," *PVLDB* 3(1): 48-57 (2010)

[9] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati , A. Lakshman , A. Pilchin , S. Sivasubramanian , P. Vosshall , W. Vogels, "Dynamo: Amazon's Highly Available Key-value Store", Proc. 21st SOSP, October 2007, pp. 205-220.

[10] D.J. DeWitt and J. Gray: Parallel Database Systems, "The Future of High Performance Database Systems," *CACM* 35(6): 85-98 (1992)

[11] D. Kossmann, T. Kraska, S. Loesing, "An Evaluation of Alternative Architectures for Transaction Processing in the Cloud," *Proc. SIGMOD* 2010, pp 579-590.

[12] L. Lamport, "The Part-time Parliament," *ACM Transactions on Computer Systems*, 16(2):133–169, 1998.

[13] D. B. Lomet, A. Fekete, G. Weikum, and M.J. Zwilling, "Unbundling Transaction Services in the Cloud," CIDR 2009.

[14] Microsoft Corp.: Microsoft SQL Azure. http://www.microsoft.com/windowsazure/sqlazure/, and http://msdn.microsoft.com/en-us/library/ee336279.aspx

[15] Microsoft Corp.: Microsoft Exchange Hosted Archive. http://www.microsoft.com/online/exchange-hosted-services.mspx

[16] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, G. Parulkar, M. Rosenblum, S.M. Rumble, E. Stratmann, and R. Stutsman.:" The Case for RAMClouds: Scalable High-Performance Storage Entirely in DRAM," Stanford tech report. http://www.stanford.edu/~ouster/cgi-bin/papers/ ramcloud.pdf

[17] TPC-C Benchmark, Transaction Processing Performance Council, www.tpc.org.

[18] TPC-W 1.8 Benchmark, Transaction Processing Performance Council, www.tpc.org, 2002