Scaling Distributed SQL Databases: Deuteronomy and Aurora

Zining Yin School of Electrical and
Computer Engineering
University of Toronto
Toronto, Ontario
Email: zi.yin@mail.utoronto.ca

***Abstract*—This paper analyzes Microsoft's Deuteronomy and Amazon's Aurora's architecture in great detail. With a deep understanding of their architecture, this paper compares them to Google's Spanner and several other (claims to be) scalable distributed databases. These references provide a direction to scale Deuteronomy and Aurora. Several obstacles prevent the efficient scaling of these databases. Although this survey was not able to solve them, it provides a direction of inquiry for future research.**

I. Introduction

There is a strong need for a scalable and distributed SQL database [1]. However, most SQL databases have hard limits on their scalability. To move beyond these limits, we have managed DB which requires manual partitioning, data migration, and load balancing. This is not feasible for many small and medium-sized companies. Even with a single database system, it requires a DB admin to manage it effectively, this adds significant overhead costs. Another reason to move a DB into the cloud is rapid scaling. Often the peak workload is many times greater than the average workload. Having excess hardware to support off-peak performance is a waste of resources. A scalable and cloud-based transactional database would provide great value to the cloud infrastructure.

Currently, Amazon's Aurora is one of the most successful distributed SQL databases. It achieved performance a magnitude faster than a distributed MySQL database through innovative architecture. It also had the added benefit of strong resilience (with a goal towards scalability). The database computation and data storage components exist on different machines. Although the concept of separating computation and storage of a SQL database has been around for decades, it was Microsoft's Deuteronomy team that made it viable. The Deuteronomy team made several innovations in redesigning the relational database. Deuteronomy integrated several core features of the NoSQL database, like entry based locking. Other the other hand, Google's Spanner database is a semi-relational database redesigned from Google's Bigtable [10]. Spanner imposes external read and write consistency and supports SQL queries. These were implemented using hierarchy, timestamp management, versioning, partitioning and so on. Spanner is not an SQL database, but it has many similarities with Deuteronomy and Aurora.

One key difference between Spanner and Aurora is scalability. Aurora cannot support 2 write instances without conflict and can only scale to 64TB of storage [11]. Despite being part of a cloud offering, Aurora is still a single instance SQL database. We will examine the details of the Deuteronomy Architecture to better understand Aurora. From there, we will propose the next steps for integrating features from Spanner into Aurora to achieve scaling. Scaling will come at the cost of much higher latency, complexity, or cost. These proposals are made with these costs and various other unsuccessful attempts [8, 9] in mind.

II. Cloud Based Transaction Services

DBMS decomposition is hard to achieve because 4 interrelated services must be on the same machine for the DB to function correctly [2]. Lock manager, log manager, I/O buffer, disk data organization methods. These protocols communicate with each other heavily and will suffer serious performance degradation when separated by a network. This paper reconsiders the DB architecture and reads and writes a protocol to separate the database into a transactional component (TC) and a data component (DC). Earlier papers could not achieve a meaningful separation of the database because their designs require the concurrency control and logging service to know which page each data entry is stored in.

This paper removes this dependency between the lock and log managers and the data management layer and

creates two much more balanced components. The log and lock managers can approve transactions to the DC (without knowledge of the page). This means the DC can be replaced by any other DC with the same table structure. This separation virtualized the DC fleet and successfully implemented data as a service. The DCs in this database can be located anywhere. However, this will constrain the number of operations DCs' can receive. Because the TC and DC communications are cross-network, the operations sent to the DCs must be idempotent, so they can be resent. They should also be atomic to minimize conflicts and provide some consistency guarantees.

Multithreading the TC and DC are mandatory for high-performance software architecture. The previous implementations of concurrency management are architecturally dependent. They are ineffective when the lock manager does not know which page the entity is on. Thus, this paper dedicates much of its effort to concurrency control and creating new recovery protocols. For concurrency control of record accesses, the paper proposes two possible approaches - The fetch ahead protocol and the range locks. In the fetch ahead protocol, the TC messages the DC for a set of entries (which are on the same page as the entry being accessed/modified) that should be locked and locks those entities. The range locks implementation requires explicit partitioning of the database table into larger sections (more lock contentions) but requires less network traffic than the fetch ahead protocol. In the later papers by the author, Lomet, the TC's concurrency control used neither of these approaches.

A. TC Design

The goal of this design is to have a template for the TC and DC designs such that the DC and TCs are not completely dependent on each other and can be mixed and matched to flexibly create and grow DBMS. This will also allow for space independent DCs while minimizing the negative effects of network latency on performance. Transactional locking to provide non-conflicting, and serializable transactions to the DC. This must be done without any knowledge of physical page configurations in DC.

Each transaction sent to DC must be atomic. A transaction should only commit (be considered successful) after DC completed the transaction and returned. Therefore, the write and read transactions on the same data entity are blocking, this is not the case in Aurora. If a transaction in progress fails, the intermittent transactions must be reversed by the TC and DC. For the DC this means reversing B-Tree page splits and possibly reversing several other transactions (if that data structure is used).

Transaction undoing and redoing can be provided by transaction logging. By logging the operations for pending user transactions, the TC will be able to submit the inverse operations to the DC to undo a partial transaction. The TC can redo these transactions in case DC fails during that time. However, this requires idempotence in DC because the TC does not know which instructions have been processed, but not completed and which instructions have not been processed. The TC assists DC with providing idempotence by stamping a unique log sequence number (LSN) on each user operation and will resend a user operation with the same LSN. The DC can track the LSN it has processed and ignored the incoming operations that have already been processed. Fortunately, this is easy to achieve from the TC side by having unique log sequence numbers for each operation sent to DC.

B. DC Deisgn

The DC design is more difficult and leaves more open questions. During standard operation, the DC is designed to accept none conflicting atomic transactions and process them in parallel. Parallel atomic operations in DC has two obstacles: page access conflicts, and non-deterministic data structure transformation. Page access conflicts can be resolved with latching and deadlock prevention logic. These are standard in single server SQL databases. Non-deterministic data structure transformations on B-Trees are caused by two operations that can complete out of order. For B-Tree databases, the order of operation completion (atomicity) affects how the tree splits its nodes. Thus, redo operations require multi-level recovery to guarantee a reversion to the previous state. Without this, a replay of the operations may result in a different tree structure. However, the integrity of the data would be preserved by the idempotence of the DC operations (this is ensured by), and the requirement of non-conflicting parallel operations from the TC.

Even with the LSN provided by the TC, guaranteeing idempotence in DC is not trivial. In standard SQL, if the LSN of the newly received operation is less than the last processed LSN, the operation is ignored. However, across the network, out of order operations becomes a possibility. Thus, it is possible for a newly arrived operation that has never been processed to have a lower LSN than the last processed LSN. The naïve solution is to record the LSN of every operation that was processed. This will consume growing amounts of memory and is not practical. The solution proposed is very simple and

used in many derivative architectures like Deuteronomy and Aurora (and Spanner?).

The naïve solution can be improved by adding a "low watermark" LSN (LSNlw)which guarantees that any LSN less than this has been computed. With a single DC, every operation and LSN is received by the DC, the DC can determine the LSNlw by ensuring that all LSN before LSNlw has been processed. With multiple DCs, the LSNs will not be sequential, this means the TC will have to maintain all pending operations and periodically publish the LSNlw to all the DCs. However, DC will still have to track all LSNs that have been processed and are greater than the LSNlw, to avoid reprocessing a duplicate operation. This feature will allow for asynchronous reads and writes in other variants of the split database architecture.

C. Partial Failure

Partial failures are a new concept that arises from this separation of the database. This should not be framed as a new problem, but an improvement on the previous problem of complete database failure. Partial failure has significantly less impact on the performance of the database than complete failure and is easier to recover from. With some redundancy and careful allocation of DC nodes and TC nodes, complete database failure may become a thing of the past. In fact, in large databases like Spanner, partial failure is expected and could have minimal (if any) performance impact.

DC failure recovery resembles the standard SQL recovery. The DC should load its latest state from the stable storage. Any operations that were not persisted in stable storage should be replayed by the TC.

TC Failure is more difficult to resolve. When the TC fails, the tail end of its logs is lost, so the TC does not know which operations have been sent to the DC. The TC assumes it did not send any of the operations it does not remember sending to the DC and start sending those operations. DC has a more difficult role. It needs to reset the DC state to a state that agrees with the TC's logs. This is done by dropping all the page changes caused by operations the TC does not remember. The DC also needs to provide a guarantee called the causality principle, where it maintains the pages affected by an operation in the cache until the TC persists the log of that operation into stable memory. How causality can be implemented was omitted.

D. Additional Notes

A lot of the other implementation details restricted to the DC like caching, data structure, page latching, and multi-level recovery have been omitted because they are not strongly dependent on the interaction between the TC and DC. Similarly, many of the TC implementation details, like concurrency control, locking implementation, and log manager implementation has also been omitted. Although these details are vital to a correct and complete implementation of a DB, the focus of this survey has been on the specifications of these components, their high-level functionality, and the interaction between them.

III. Deuteronomy

Deuteronomy is an implementation of the Unbundling Transaction Services in the Cloud paper by Lomet [3]. However, the gap between better theory and practice resulted in several changes to the design in the original paper. These designs are refined further in a later paper on Deuteronomy.

The flexible implementation allows the TC to be portable across storage providers. This remains true for DC. However, this also makes the DCs difficult to implement because it must support several of TC's requirements for correct operation. The original paper includes control operations, recovery, communication guarantees. However, practical challenges across the network include unavailability events, network delays, data corruption.

Each DC is not thread-aware and does not have to manage replication. This means they can operate as independent entities (without communicating with another DC). This design allows the DC to have almost unlimited linear scaling and solves one of the biggest challenges for SQL servers while providing ACID transactions.

A. TC Redesign

The implementation of TC in Deuteronomy involves 5 components: session manager, record manager, table manager, lock manager, and log manager. The details of the TC design were glossed over in the previous paper but is the focus on this and the following paper.

1) Record Manager: Originally, the TC is supposed to send any number of non-conflicting operations and log them. When the DC returns, the transaction is committed. The new implementation is to lock a subset of the database without determining if the blocked transactions are conflicting, and only write to the log after the DC completes the operation and returns a response. Also, read operations will no longer be logged. Consequently, the LSN will only be generated for write operations, not read operations.

The record manager manages both read and write operations. It determines when each operation is sent

to the DC component. For read operations, the record manager requests a lock form the lock manager and the lock manager lock a range of keys without knowledge of the physical pages of the keys. For write operations, there is the additional step of logging, again without knowledge of the physical pages of the data. Logging, which would occur on requests sent to DC, only occurs after DC completes the request because the DC should not need to respond to all requests. If the DC gets a repeated request or a failed request, it can save network bandwidth and not send a response. By only logging responses, the TC saves cache space and reduces network communication. The other reason is for recovery or undo operations. To undo operation, the TC requires the initial state of the operation. For example, the initial state of an insert is null, while the initial state for update or delete is the original row. Because querying the DC before sending a delete operation will increase the operation latency, it makes sense to have the response from the DC contain the original row. This means the insert, update, and delete does not need an initial log, it makes sense to the only log after a response from the DC. This also simplifies the response to partial failure for the DC. It can assume that the TC does not have a log of any requests it did not respond to. Reads do not need to have an attached LSN, because reads are already idempotent and are not necessary for partial failure recovery of the TC or DC.

One detail that was neglected previously is the implementation of causality. This can be done by having the TC send control operations called EOSL ("end of stable log") which indicates the LSN of the last stable log. In DC, every change after this request cannot be written to stable storage or it will violate causality. However, the LSN of the logs written to the log manager will not be in order, and this will be addressed in the log manager.

2) Table Manager: The table manager chooses which table to access for data entry, and manages the equal-sized partitions in each table, where each partition is a lockable resource. This partition concept is also supported in DC, by providing a guarantee that the two partitions cannot share the same page.

3) Lock Manager: The lock manager provides and maintains 3 levels of locks to ensure concurrency. The assumption is that all database reads, and writes are page reads and write. The TC does not know which page it is reading or writing to, so it also does not know if another operation is using the same page, even if they are using different entries. One way to find out which physical page to lock is to query the DC, but that would introduce unwanted latency to every operation. Therefore, the table manager supports the logical partitioning of tables, a crude form of page locks. The table and partition locks are read locks while the entry-level lock is a write lock. Any write requires a partition and entry-level lock, while reads only require the partition level lock. When reading across multiple entries, the record manager will query the table manager for the corresponding partition ids to lock and request those locks from the lock manager.

4) Log Manager: The log manager is a conventional Windows Common Log File with a few exceptions. As mentioned previously, the logs may not arrive in the order of LSNs, and the logs in the TC must be coordinated with the cache of the DC to provide causality guarantees. This causality is provided through 2 low watermark LSNs that are published periodically by the TC. This low watermark implementation is an extension of the concept from the previous paper. The first one is the End of a Stable Log (EOSL). This is the last stable log in the TC where every LSN before it has been registered. After receiving this, the DC will write all the write operation results with LSN less than or equal to the EOSL LSN into persistent memory. The other low watermark LSN is the Redo Scan Start Point (RSSP). This indicates the last stable storage of DC. When DC receives the RSSP LSN, it will only respond to the operation after it has persisted all the operation results with LSN less than or equal to this. During a partial failure scenario where the DC fails, the TC attempts to redo by resending all the operations after the RSSP LSN but may not send anything before it. However, these two low watermarks are calculated has been omitted because it does not have an impact on the DB architecture.

The client session manager's detailed implementation has also been omitted. The client session manager has a similar implementation to the standard SQL and web session managers and is not at all affected by the internal architecture of the DB.

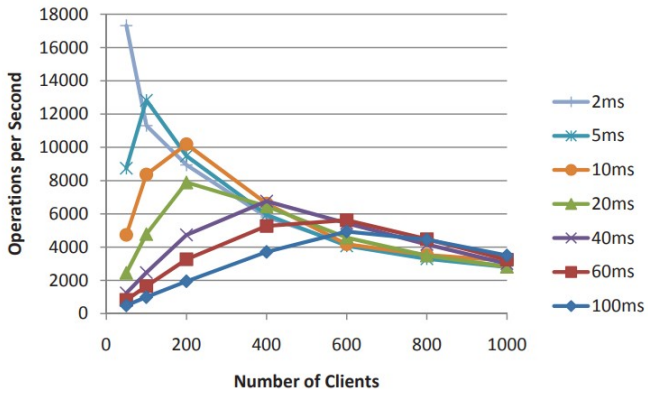
B. Partial Failures

1) TC Failure: Due to the refinement of the TC implementation, the recovery protocol is also clearer. The DB should always include a backup TC, which has access to all the logs and client operations of the primary TC. When the working TC fails, the backup TC can immediately take over, and begin redoing the operations from the RSSP. This arrangement will reduce the failure recovery time to almost 0. Naturally, the primary TC and its backup should be in different data centers. The messages to the backup TC should be on a separate thread and have minimal impact on the primary TC's performance.

2) DC Failure: The DC failure involves replaying the operations from the RSSP. It is more expensive because

the DC must recompute all the lost operations. There is little deviation from the previous failure protocols.

C. Performance Analysis



[3]

This performance analysis shows two sources of performance issues. First, when the number of clients exceeds 600, there is a degradation in the performance regardless of network latency. This resource contention still exists when the TC's locking and logging are disabled. The paper suggests that this can be relieved using SQL Server threads (called fibers). The second issue is the latency dependent throughput of the TC when the number of clients is low. This is a major obstacle for cloud implementation because high latency clients will have no way to increase their throughput. However, the paper fails to propose a solution to this issue or acknowledge that it should be tackled.

IV. Bw-Tree

The Bw-Tree paper focuses on the performance optimization of the DC component for multicore CPUs and flash drives [4]. This survey generally omits pure performance improvement changes and focuses on the architectural design of the database. However, the Bw-Tree paper offers a significant increase in the concurrency guarantees of the DC which positively affects the design and performance of the TC. From a high level, the Bw-Tree paper describes a DC which supports locking by entity rather than locking by physical pages. This relieves the TC from complex and inefficient lock management and table partitioning. Details will be provided in Section ??. The Bw-Tree paper has a sibling paper describing the new log structure storage manager, LLAMA. Both papers are crucial to the implementation of the new DC and the two orders of magnitude improvements to the TC (described in a third sibling paper).

A. Introduction

The key concept in the Bw-Tree paper is the implementation of an Atomic Record Store (ARS) using a SQL B-Tree. The ARS supports reading and writing to individual records (independent of physical pages), provides stable storage access protocols, and includes recovery operations in case of storage failure. This synergy allows the implementation of performance optimizations like the latch free technique and caching optimizations (which will be omitted). The Bw-Tree will be used together with the log-structured storage manager, LLAMA, when redesigning the new DC of Deuteronomy. One of the key drivers behind this improvement is that the CPUs are not being used efficiently by current SQL database processes. There are a growing number of cores per CPU and Moore's law for single-core performance is reaching its limits. With multiple cores, page latching will block more often, limiting the effectiveness of multi-core server CPUs. Updating memory in place (modifying the existing memory) causes cache invalidations. Without proper update manager, higher-level caches will experience misses more often as the number of underlying threads increases. The Bw-Tree is latch-free and creates update deltas that avoid overwriting existing entries (avoids update in place). A secondary improvement of this data structure is designing around SSDs' slow random writes. These concurrency and caching concepts are valuable to all DB.

B. Architecture

Without providing background into the B-tree and B+-tree, the key differentiator between the Bw-Tree and the other B-tree variants is the (mostly) non-blocking threads. This is achieved using the CPU level atomic operation called compare and swap (CAS). This increases instruction cache hits, reduces idle time, and reduces context switches.

It performs node updates using delta updates, which involves attaching the update to an existing page instead of update in place which modifies the page. This significantly reduces CPU cache invalidation and increases the cache hit ratios.

The log structure storage layer (LLAMA) [6] supports large write buffers which permit multi-page writes. This, in turn, supports variable sized pages. This property compliments the logical page identifier (PID) introduced by the mapping table.

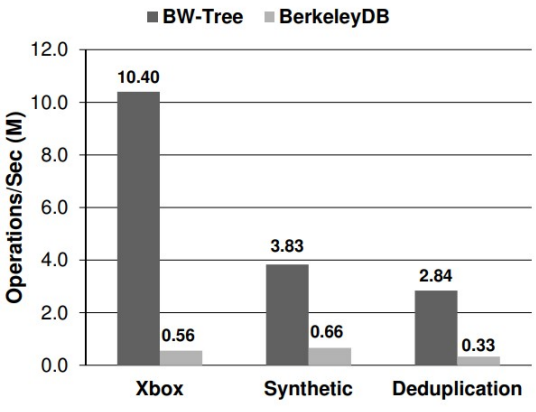
1) The Mapping Table: The mapping table translates the logical (virtual) page identifier (PID) into a physical address or memory location. This is mapping is also used to link the nodes of the Bw-tree. A parent

node's children are identified using PIDs. This allows the physical address to change without having to update the parent node references. Because each node, and its corresponding page, is virtual, its size can be elastic. With the log structure storage layer, this flexibility avoids the cost of filler in the irregular sized pages. However, each virtual page has a rule for managing its contents. For example, this page only allows entries within the specified key range.

2) Delta Updating: By taking advantage of the mapping table and flexible page sizes, a technique called delta-updating can be applied. Delta updating is creating a record describing the change to the existing page and prepending it to the front of the page. This is followed by an atomic compare and swap (CAS) update to the mapping table. This technique avoids page latching using a hardware atomic operation and does not invalidate (only increase) the contents of the memory cache. This technique requires an occasional consolidation of the deltas to reduce the in-memory size of the page and improve access time. This is done by applying all the deltas to the page, writing the new page with a CAS, and updating the mapping table.

3) Page Consolidation: Consolidation of deltas to improve search performance and reduce cache footprint is critical. This is triggered whenever a read operation detects a delta chain that exceeds the threshold. The consolidation takes two steps. First, it dynamically allocates a block of memory for the new page and writes the new page to that address. Second, it uses a CAS to write the new page address to the mapping table. Third, if this CAS succeeds, it will flag the original page for garbage collection. If the CAS fails, it will deallocate the new page's memory.

C. Performance



[4] In addition to its execution guarantees, the performance of the Bw-tree is phenomenal. The performance of the Bw-tree is measured using the Xbox workload, a deduplication workload, and a synthetic workload. Both the deduplication and synthetic workloads are geared towards the Bw-tree's weakness of splits and consolidation failures. When compared against Oracle's high-performance BerkeleyDB, it performed 5.8x to 18.7x better.

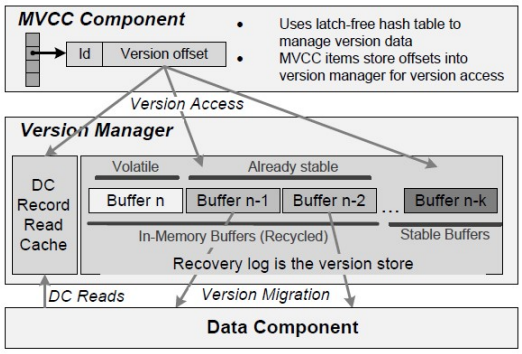
D. Conclusion

The Bw-tree had an immense impact on the design of the Deuteronomy database. It greatly improved the performance and capabilities of the traditional B-tree structure. It also improved many other file systems like the SQL Server Hekaton and the Azure DocumentDB. Its success was a large motivator in the redesign of the TC in Deuteronomy.

V. Improving the Performance of Deuteronomy

Thanks to the Bw-tree, the new DC is performing several million operations per second [5]. This made the TC an extremely obvious bottleneck. This paper meets the looming challenge of redesigning the TC to meet the same performance standards of DC. Using many techniques from Bw-tree as well as many of the convenient properties of the Bw-tree and LLAMA, the performance of the TC was improved by more than 100x.

A. TC Redesign



[5] The previous TC design contained a complex arrangement of the session, log, lock, record, and table managers, while the new TC only contains a Modern multi-version concurrency control (MVCC) component, a version manager, and a TC proxy. The MVCC manages the concurrency control logic. The version manager uses a redo log to manage the versions. The TC Proxy is a module (on the DC server but not part of the DC) that manages interactions between the TC and DC.

One important change is the log writing procedure. In the initial implementation [2] the TC writes to the log for every transaction and updates the log once the transaction

completes. A later version [3] improves on this by only updating the log with reading/write commit records only when the transaction completes. In this version of the TC, there is no undo operation, so the update should only be published to the version manager and the DC once the transaction is committed. Because the update to the DC is no longer part of the commit path of a write transaction, it can be parallelized effectively and the throughput's dependency on latency is significantly reduced.

Aside from the addition of the TC Proxy, the update handling on the DC side remains unchanged. The DC will only write the record changes to persistent storage once the LSN for the change is below the low water mark LSN.

B. Timestamp order MVCC

To eliminate the previously blocking interaction between reads and writes, the MVCC uses an old concept called time order concurrency control. This allows a read operation to access a previous version based on the timestamp. It allows a write operation, to the same entity, to run concurrently with the read operation because they are accessing a different version of the entity. One exception is when the read is accessing the latest version of the entity, and a write operation which began earlier is updating that entity. In this case, the read is blocked until the write completes to ensure that the later read operation gets a consistent and correct result. The write operation would not affect what is read. In the rare cases where the timestamp order cannot be maintained, the transactions are aborted. This works well for the TC because its transactions are all short-lived. Each transaction is tracked based on its timestamp. The oldest active transaction (OAT) is tracked and used to determine which cached versions are ready for garbage collection. The entity versions have respective read and write version caches which are used to minimize version lookup time. A read cache is necessary because some entities might not get updated often but is read frequently.

Same as the previous implementation, the TC still maintains 2 LSNs to track the low water mark of the stable logs. One is used to track the latest stable transaction in the DC, while the other tracks the latest stable logs in the TC.

C. Fast Commit

Another new optimization and arises from the redesign are the fast commits. The commit process is the same for any read or write operation that creates a commit record. The transaction is first processed, if successful, it will write a commit record to the log in the version manager. Later, the commit will be persistence into stable storage. A commit has two stages of committed, a system-level committed, and a user-level committed. The system-level committed is when the system treats the record as fully committed but does not inform the user. This committed level occurs when the transaction has been processed and is being written to the log. The user-level committed is the response to the user. This is sent once the update is stable – the log is written to stable storage.

The read-only commit is more complex because reads rarely require a commit record and can return a stage two committed to the user when it is ready to write a commit to the log. However, this read will return the incorrect entity if the record returned was not stable and the TC or DC crashes before it becomes stable (the update to the record is lost). To avoid this, the read-only transaction should also write a commit, and follow the two-stage committed protocol. This caused significant bottlenecks in reading on workloads. As an optimization, the read transaction will only write a commit and wait for committed, if the version of the entity being read is not stable at the time of writing the commit.

D. TC Proxy

The TC is designed to be a middleman that passes the stable logs from the TC to the DC. It enforces the contract between the TC and DC, so the DC can run in any location. It should rest on the same server as the DC. The stable recovery log buffer is sent to the TC Proxy as a batch of commands which are unbundled by the proxy. As mention earlier, the contact protocol between the TC and DC had significant changes in this iteration. Previously, every transaction was first sent to the DC and only committed (to the stable log) after the DC processes the transaction. After that, the TC would send the DC an end-of-stable log message, which indicates which entries the DC should persist into stable storage. Now, all transactions sent to DC are already stable and can be persisted into stable storage immediately. The new protocol eliminates the round-trip communication between the TC and DC. This effectively makes writes latency independent.

One unexpected optimization of the TC Proxy's update protocol is the upsert. Upsert is an update operation which "has the same effect regardless of the prior state of the record" it is updating. This operation required a slight modification to the Bw-tree described earlier. There are two important performance benefits. First, this eliminates the round-trip communication for standard DB updates where the TC Proxy reads and confirms the entity before

sending the update. Second, this may eliminate the need for the (variable size) page to be in the cache. Only a reference to the page and its key is needed to apply the update. Naturally, one problem is the existence of multiple deltas updating the same entity in the same page. When that occurs, lazy evaluation of the deltas can be used to determine the latest update and use that as the correct version.

E. Performance

Throughout this survey paper, there has been little focus on the performance because the previous designs were proofs of concept. However, this iteration of Deuteronomy has been fully optimized and warrants a careful performance report.

Due to the architectural redesign as well as incremental improvements to the TC, Bw-tree, and LLAMA, Deuteronomy became significantly more effective. Deuteronomy is competitive with the best available databases (on realistic workloads), as well as main memory databases (on memory only workloads). With a small dataset and a random Zipfian distribution where 20 percent of the entities are accessed 80 percent of the time, the single Deuteronomy server can reach 400k write operations per second or 2.3 million read operations per second. This is, in part, thanks to the high hit rates (92 percent) of reads due to the small dataset.

The experimentation in this paper was, unfortunately, all done with the TC and DC on a single server. This result does not provide a meaningful point of comparison with the distributed database setups in Spanner and Aurora. It does not meaningfully demonstrate the intended usage of this distributed database. One significantly more meaningful test would have been measuring the performance of Deuteronomy (with the TC and DC on separate servers) with varying latencies between the TC and DC. Another would be to measure its performance on very large workloads that necessitate multiple DCs. This would demonstrate its importance as a flexible and scalable distributed database.

VI. Amazon Aurora

A difference between the paper on Aurora [10] and Deuteronomy is the maturity of the underlying product. Aurora has been in use for several years (under various workloads) before the Aurora paper was published, while Deuteronomy's paper was written shortly after the database's creation. Aurora was also designed, in part, off of Deuteronomy. This survey will attempt to smooth over the difference of refinement between the two papers and present a comparable view of their designs. For example, the terms TC and DC are not used in the Aurora paper, but this survey will continue to use them for the sake of consistency.

Aurora contributes several important improvements to the Deuteronomy configuration and develops it towards the original idea of a scalable, mobile, resilient, and distributed database for the cloud. First, Aurora provides resilience for the cross-network communications through duplication of the DC nodes. This reduces the server downtime for a single DC failure from a short interval to no downtime at all. This is also true for database updates. Aurora's replication setup eliminates the data loss from server level and datacenter level failures. Second, Aurora offloaded the redo, durability, crash recovery, and backup management from the TC onto DC. This reduces the network communication between the two components (a previous bottleneck for Aurora) and improves the throughput of the TC (A bottleneck in Deuteronomy). Third, Aurora's design was focused on cross-network performance and the paper offers extensive performance tests against a comparable configuration of a distributed MySQL setup.

What is lacking in the paper is the analysis of a multiple DC setup where there are 2 DCs with different datasets (not just duplicate DCs). From the web documentation, it appears Aurora is still not completely scalable (one of the original goals of the TC and DC separation). Aurora can only support datasets of up to 64 Tebibytes. [10] It also cannot support multiple TCs without serious write concurrency issues. This means it is still several steps from being a globally distributed SQL database.

A. Durability Across Netwroks

When transitioning to the cloud, customers expect constant availability and durability. Across networks and with a large number of servers, there will be a constant stream of temporary or permanent failure of the disk, server, rack, network, or datacenter. It can also be a very temporary event like a latency spike due to a long queue on a network router. All of the failures are considered availability events and are handled in very similar ways. Aurora developed resilience against them using DC replications and TC backups. With replication comes the challenge of data integrity when two or more DCs have conflicting data. The standard quorum-voting protocol to resolve conflicts is used. For a read operation to be accepted, at least Vr nodes must be in consensus. For a write operation to be accepted, at least Vw nodes must be in consensus. There are two rules for preserving data integrity when using replication. First, Vr plus Vw

must exceed the total number of nodes. This means at one of the nodes needed to confirm the read, also has the latest written value. Second, Vw must exceed half of the total nodes. This means at least one node that approves the new write knew about the previous write. This prevents conflicting writes.

The minimum setup for replication requires 3 DCs, where Vw is 2 and Vr is 2. Aurora opted for 6 node replications instead. This is due to the threat of concurrent failure events. Although the 3 DCs are placed in 3 different Availability Zones (AZ) without a shared point of failure (this includes fires, power outages, software deployments, and networks). If there is a complete failure or "a long-lasting availability event" at one datacenter and a failure occurs at for another DC, the database would be unable to read or write. Therefore, Aurora used six replicate DCs, with 2 DCs in each AZ. The write quorum requires four out of six DCs, and the read requires 3 out of 6. This means if a single AZ (two DCs) plus one DC went down, the database can still read even if it can no longer write. This replication arrangement combined with the recovery protocols of the DC and the fast network connection between the AZs provides incredible database resilience. Amazon claims that it is unlikely for any of their customers' databases to experience a read quorum failure.

B. Log as the Database

Surprisingly (or perhaps it was carefully considered), the latest DC design of the Deuteronomy is well suited for the data replication and cross-network communication that Aurora needs. Aurora made slight modifications to the DC to support a log only database as well as redo, quorum, and recovery functionalities. However, the underlying communication between the TC and DC (the asynchronous delta updates messages), the concurrency optimizations for modern hardware, and the designs of the LLAMA log structure were heavily reused. (How is redo done in the last paper??) For example, the Aurora TC only writes the redo logs records to the database, not the full records, and no database response is needed to read or write operations. This means the underlying principles (TC delays updates to DC until the operation is committed, and DC uses a mapping table to virtualize the page locations) is the same. The logs and physical page management are all done on the DC (just like in the latest DC in Deuteronomy). Once the number of logs (similar to delta updates from before) for a page gets too large, the page will be added to the cache to avoid recreating it from scratch every time. The difference is that this page will not be written to the database because Aurora considers the log to be the database.

VII. Google Spanner

Spanner by Google [7] is a globally distributed database based on the Bigtable [11] infrastructure. It is scalable and offers read and write consistency. However, this comes at the cost of data partitioning as well as higher latency. A Spanner universe (database) is composed of multiple zones, and each zone contains multiple Spanservers. The Spanner database can be conceptualized as a fleet of Spanserver databases. Each Spanserver database is composed of a Spanserver leader and its replicas. A Spanserver contains a shard of the full database. A Spanserver leader manages the distributed writes for any data shard (and its replicates). It is enforced by a single Spanserver leader using the 2-part commit protocol. The setup of the Spanserver replicates closely resembles the mirrored MySQL.

A. Concurrency

Each partition is placed on a group of Paxos state machines. Each Paxos can be on a separate server. A write operation on a single data partition is managed by a Paxos group leader. The write is done using the standard 2-phase write protocol with locking and unlocking. However, reads are completely lock-free because it references the past versions. Spanner's read is similar to Deuteronomy and Aurora.

To execute a write, the Paxos leader acquires the locks for all the affected entries. Then it acquires a timestamp from Spanner's TrueTime service. TrueTime is a service in Spanner which provides a global timestamp (described in the next section). This timestamp is guaranteed to be monotonically increasing for all writes, even across different Paxos groups. Finally, the write is committed and the locks are released. This timestamp serves the same purpose as Deuteronomy and Aurora's LSN. It allows Spanner to have a consistent database image at all times. When a read or write interacts with multiple Paxos groups, the group leaders select a new leader among themselves to manage that query.

B. TrueTime

Spanner has an API TrueTime which can provide a strict timing guarantee across all its instances around the globe. TrueTime is an API which guarantees that the timestamp of a committed transaction will always be greater than the timestamp of a transaction that occurred after it. By using atomic clocks and GPS clocks distributed across all its data centers, TrueTime can provide a time interval when the transaction is guaranteed to have occurred. If transaction 1 commits before transaction 2 begins, then transaction 1's timestamp will be less

than transaction 2's. This allows Spanner to guarantee a variety of consistency-based actions without interrupting other ongoing transactions. This includes externally consistent reads and writes consistent global MapReduce and atomic schema updates.

TrueTime "allows the implementation of external consistency, non-blocking reads in the past and lock-free snapshot transactions, and atomic schema changes" [7]. Each of these properties provides significant benefits. Lock-free snapshot transactions, for example, amortizes the cost of creating snapshots across the system without causing availability issues and providing much stricter latency guarantees. While taking regular snapshots is crucial to crash recovery, a distributed MySQL database needs to pause all writes and wait for current writes to complete before creating a snapshot. While it makes sense to create a cutoff point in time and add all commits up to that point to the snapshot. A transaction by another server may have a timestamp that is smaller than the cutoff, but the timestamp could be very inaccurate due to a lagging server clock. This need to compare timestamps created by different servers drove the creation of TrueTime.

TrueTime was extremely expensive for Google to implement and maintain. It requires a time master server with a GPS clock or atomic clock in every datacenter and a time slave daemon on each machine. Each time slave polls multiple time masters and determines the current time as well as its margin of error. It also filters out outliers using Marzullo's algorithm. Finally, it continuously increases the margin of the error until it resets it during a periodic realignment.

VIII. Scaling Aurora and Deuteronomy

	Spanner	Aurora
Multi-versioned	Yes	Yes
Distributed	Yes	Yes
Consistent Read and Writes	Yes	Yes
Fault Tolerant	3+ replication	6+ replication
Versioning	Time range/Timestamp	Log Seq Num
Partitioning	Yes	No
Relational	Semi-Relational + Key Value	Yes
Write Operation	2-phase commit	asynchronous
Scalable	Yes	No
Global Reference Point	TrueTime	No
Data Management	Auto sharding	No

Comparing Spanner and Aurora

Although Spanner and Aurora have many similarities because they both distributed SQL databases. There are key advantages they have over each other. Spanner is partitioned and scalable. This comes at the cost of being semi-relational, longer read and write latencies (due to 2-phase commit as well as network latencies) and requires a global timestamp. If an Aurora DB's data can be sharded into many instances, each managed by an independent writer node, the number of writer nodes can increase without interfering with each other. Of course, this would introduce new problems like what Spanner faced and can be solved with an API like TrueTime. Sharding is difficult to implement to a SQL DB because sharding requires a parameter, like a key or partition or table, to dictate how to shard the database. Sharding by key, splitting a table based on the key will make querying the table more time-consuming. Sharding by the table, the size of tables can vary greatly, and the user also varies greatly; this will have very uneven performance. Sharding by a partition, this makes the most sense because it obeys space locality and allows the table to scale, unfortunately, SQL tables are not split by partitions. This means a pure SQL database that is backward compatible with MySQL is difficult to shard. Even if Aurora can be sharded, TrueTime is expensive to implement.

Without sharding, TrueTime intervals will not allow parallel writers. The TrueTime is not able to give an exact time, but a time interval described by the error in measurement. This means only transactions that do not have overlapping time intervals can be sequential. This allows us to decide if an interval overlaps with the cutoff time, but it is impossible to compare two overlapping time intervals. This means each writer cannot assign a consistent Log Sequence Number to each log entity. Without a monotonically increasing log sequence to manage the files, the storage nodes cannot maintain correctness in the face of duplicate log numbers. Therefore, Aurora currently rejects the writes when a duplicate log number is seen. [12] The implementation by Yesquel allowed for multiple writer instances while maintaining consistency and resolving contingency. However, the write throughput may have worse performance than the single writer node. The overhead of managing locks and synchronization may prevent the effective utilization of resources. Moreover, write demand and contention for data is not randomly uniform across time or space. There are often rouge events where a piece of data is accessed far more than it has ever been. On the other hand, there are naturally data that is written to more often while others are never used at all. These events create contention between the resources [8].

A. Detailed Analysis

In the following sections, we will examine a few questions and obstacles to a scalable Aurora DB. To define a few terms, an Aurora instance is a single TC

plus six DC (one copy replicated five times) as described in the Aurora section. This instance is comparable to a Spanner Paxos group. The LSN will be used as an instance-specific value which is meaningless globally. A global timestamp management process is assumed to exist, and the timestamp will map to a local LSN for each instance. We discuss the timestamp as if it is a single value when it is a range. This is a reasonable simplification.

1) Aurora with Timestamp: First, we will take the smallest possible step towards making Aurora more scalable. Let us make Aurora function with a timestamp instead of a Log Sequence Number (LSN). Originally, each DC's log manager determines which log updates are missing based on gaps in the LSN. This change prevents the log managers from determining the missing log deltas independently. Here is one scenario where this would be a serious problem. If two writes are sent to the TC for the same entity. The first write is lost on the TC side, and the second write updates the entity to the incorrect value. All the DCs will confirm the write, and the second write would be committed, but the first write will be lost. How can this be resolved without making writes synchronous?

2) Reading Between Two Instances: Can we avoid this problem by avoiding timestamps within the instance? The answer is yes. If the LSN is internal to the instance, and each instance's TC provides a timestamp for each LSN when it is committed. This raises another question. Why do we need a timestamp? Suppose there are two Aurora instances, and they must coordinate to answer a read query (Assuming we manually sharded the partitions, and no table's size the size of one instance). Which write deltas should be published in this read operation? To answer this, the two instances will need to agree on the snapshot of the database at the time of the read. If each instance has their internal LSN and a timestamp when each LSN becomes stable, a read with the same timestamp will always return the same result because any log delta which becomes stable afterward will have a later timestamp (assume monotonically increasing timestamps like Spanner).

3) Writing Between Two Instances: Does writing to two Aurora instances require locking? Assume we have one write to instance one followed by a write to both instances one (write to the same entity as the first write) and two, the two writes can be non-interfering. The two writes to the entity on instance one can easily be resolved by instance one (existing behavior). If the second write to instance two finishes before the two writes on instance one there will be a read operation inconsistency. If the user reads at this time, he/she will only see half of the second operation applied, but he/she expected one of three things: no operations applied, the first operation applied, or both applied. Spanner's protocol of joining two Paxos groups together and electing a new leader will not work because the TC does not have the same level of control over the delta updates. To support this the TC design would be taking several steps back and processing and managing more of the transactions.

We do not have an easy solution to this problem. We can attempt a solution by designing a multi-instance write protocol. First, one instance receives the cross instance write operation. It retrieves a starting timestamp for this operation and sends the write to all relevant instances. These instances will prepare their portion of the write and inform the write issuer instance. Once all the instances are ready, the write operation issuer will request a commit timestamp and notify all other nodes to commit.

There are two problems with this approach. First, any unavailability event at a single instance will directly affect the latency of the multi-instance query. This was one of the driving reasons behind the duplicate DC nodes. There would be a need for an additional failover step or greater redundancy for each instance. Second, this does not entirely solve the read inconsistency (race condition). If a read occurs after the commit timestamp is issued, but before all instances can stabilize their commits the snapshot will not contain all expected transactions. If the read is blocked until all the writes are completed, then the read throughput will be delayed by the worst-case network latency. Plus this additional memory and processing will greatly increase the TC's workload and reduce throughput.

4) Redo Between Two Instances: Given the scenario above, how should we handle a failure of the first write to the first instance after the second write to the second instance is completed? This is an open question. Without solving the write operations, it is rather pointless to discuss redo implementations. However, it is an important consideration none the less.

IX. Conclusion

We spent a lot of words analyzing what exists and relatively few words describing what could be. Despite the few words, no practical conclusion or clear direction was provided for improving Aurora. However, this is an expected outcome for this paper. The database architecture is very complex and highly dependent on the implementation details. It is only with a careful analysis that we can make a sensible suggestion for improvement. However, if it was easy to scale the Aurora database, then Amazon would have done so after seeing its great success. The same argument can be made for

the derivative databases being proposed by many other researchers. It appears that there is no escape from a 2-phase blocking commit when there are multiple instances.

The good news is that we have identified some key questions about writes and redo operations. Also, we imposed a few strong constraints upon our design. We reject any two-phase commits for writes, and we want to support a fully relational database. However, we are open to the partitioning of data for sharding purposes. With these goals and constraints, we will avoid many of the pitfalls of past distributed database designs. Despite these challenges, a scalable, fault-tolerant, and distributed SQL database is crucial to many applications and workloads.

REFERENCES

[1] C. Curino, E. Jones, Y. Zhang, E. Wu, and S. Madden. Relational Cloud: The Case for a Database Service. Technical Report 2010-14, CSAIL, MIT, 2010. http://hdl.handle.net/1721.1/52606.

[2] D. Lomet, A. Fekete, G. Weikum, M. Zwilling. Unbundling Transaction Services in the Cloud. In CIDR, 2009: 123–133.

[3] J. Levandoski, D. Lomet, M. Mokbel, and K. Zhao. Deuteronomy: Transaction Support for Cloud Data. In CIDR, 2011, pp. 123–133.

[4] J. Levandoski, D. Lomet, and S. Sengupta. The Bw-Tree: A B-tree for New Hardware Platforms. In ICDE 2013: 302-313.

[5] J. Levandoski, D. Lomet, S. Sengupta, R. Stutsman, and R. Wang. High performance transactions in deuteronomy. In CIDR 2015.

[6] J. Levandoski, D. Lomet, S. Sengupta. LLAMA: A Cache/Storage Subsystem for Modern Hardware. PVLDB 6(10): 877-888, 2013.

[7] J. C. Corbett, J. Dean, et al. Spanner: Google's globally distributed database. In OSDI 2012.

[8] M. Aguilera, J. Leners, and M. Walfish. Yesquel: scalable SQL storage for web applications. In SOSP 2015.

[9] M. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis. Sinfonia: A new paradigm for building scalable distributed systems. ACM Trans. Comput. Syst. 27(3): 2009.

[10] Verbitski, A. Gupta, D. Saha, M. Brahmadesam, K. Gupta, R. Mittal, S. Krishnamurthy, S. Maurice, T. Kharatishvili, X. Bao. Amazon Aurora: Design considerations for high throughput cloud native relational databases. ACM SIGMOD, 1041 – 1052, 2017.

[11] F. Chang, J. Dean, S. Ghemawat, W.C. Hsieh, D.A. Wallach, M. Burrows, T. Chandra, A. Fikes and R.E. Gruber. Bigtable: A distributed storage system for structured data. ACM OSDI, 205 – 218, 2006.

[12] https://docs.aws.amazon.com/AmazonRDS/latest/ AuroraUserGuide/Aurora.Managing.Performance.html