Distributed Indexing and Locking: In Search of Scalable Consistency

Mahdi Tayarani Najaran
Computer Science
Department
University of British Columbia
tayarani@cs.ubc.ca

Primal Wijesekera
Computer Science
Department
University of British Columbia
primal@cs.ubc.ca

Andrew Warfield
Computer Science
Department
University of British Columbia
andy@cs.ubc.ca

Norman C. Hutchinson
Computer Science
Department
University of British Columbia
norm@cs.ubc.ca

ABSTRACT

Indexing and locking are two important components of a Relational Database Management System (RDBMS), which pose as potential bottlenecks when scaling. We present novel distributed alternatives to both of these components which are based on a spatial partitioning tree designed for scalable low-latency information storage and retrieval. The proposed solutions provide strong consistency, fault-tolerance and low latency. Our initial evaluations using the TPC-C workload show they are capable of utilizing more processing power for scalability. We believe our contributions to be an important first step toward our goal of a scalable, cloud aware and full-featured RDBMS architecture.

1. INTRODUCTION

Relational Database Management Systems serve as a fundamental component for data storage in almost all industrial and commercial environments. They provide a high degree of expressiveness, a sufficiently simple data model, a sound mathematical foundation, and widely accepted standards. This combination of features has lead to their widespread usage in systems ranging from small businesses to large corporations. However, with the emergence of cloud computing and the need to handle ever-larger amounts of data, the scalability of full-featured RDBMSs has been drawn into question. This has become an active area of research, but faces multiple major challenges.

Traditional non-distributed RDBMS follow a monolithic and shared-everything architecture with a tight coupling between the components, two of the most important of which are the index and locking mechanism. An item (or tuple) stored in a database may have multiple attributes (or fields), some of which are indexed. The indexing system is expected to be able to answer complex queries over the items it stores, such as finding items with specific values on selected attributes. The locking mechanism is required to lock both individual items or all items having some attribute values in specific ranges. Locking is the only way of providing isolation between transactions.

An RDBMS relies heavily on the strong consistency of both components. As a consequence, this requirement limits the ability to easily scale an RDBMS. Considerable work has been done on working around this issue in trying to achieve scalability. However, either consistency is traded in favor of scalability by migrating to eventual consistency [16], or generality and expressibility are sacrificed by limiting the query model.

Key/value storage systems [9, 3] are a good example of systems that tolerate looser consistency of the eventual consistency model. They avoid distributed indexing and locking by simplifying the query model, which is often too simplistic for general RDBMS users.

Partitioning is a widely employed technique used for distributed indexing and locking [4, 11]. Data is partitioned into smaller portions where indexing and locking are handled independently and locally to the partition. The performance of partitioning systems suffers in the face of transactions touching multiple partitions. Executing these transactions requires them to be serialized across partitions, creating a potential bottleneck. Smart partitioning tries to reduce the probability of such partition violations by optimizing the partitioning mechanism [8], but requires a-priori knowledge of all queries that may ever be executed; this is not always practical and limits expressibility. Additionally, partitioning is subject to hotspots, i.e., partitions with high demand, which leads back to the original problem of scalable distributed locking and indexing.

Specialized locking systems, such as Chubby [7] and ZooKeeper [10], provide scalable locking along with strong consistency guarantees. Due to the way locking is implemented in these systems, efficiently implementing complex multi-attribute range locking (which is a common requirement of an RDBMS) is fundamentally not feasible. Google’s
Percolator [12] also implements a scalable locking mechanism on top of BigTable, but only works on snapshot isolation and can not support serialization isolation, as required by strict ACID. Thus these specialized systems lack the generality and expressibility needed by an RDBMS. Finally, the distributed B-Tree [6] (which is more of a direct distributed version of the current single node indexing) provides features needed for distributed indexing (except multi-attribute range queries), but its performance suffers in workloads with moderately high write operations.

In this paper we aim to implement distributed, scalable and strongly consistent indexing and locking mechanisms which could ultimately address the need for scaling an RDBMS. To this end, we have implemented these two services using SinExTree [14], a distributed spatial partitioning tree designed for scalable low-latency information storage and retrieval. SinExTree is built over Sinfonia [5] which provides atomic access to distributed memory suitable for a cloud environment. We believe our contributions serve as the first step toward our goal of scaling an off-the-shelf RDBMS engine without compromising its expressiveness or consistency.

2. THE BIG PICTURE

While our ultimate goal is to implement a highly scalable truly elastic RDBMS, the most important question we seek to answer is whether we can build a fully SQL-compliant RDBMS without having to make compromises? How can an RDBMS simultaneously provide scalability, consistency and generality?

In current RDBMS architectures, from the query planning module to the locking module, all the components are tightly coupled with each other in a single node setup. Commercial RDBMS engines are highly sophisticated and optimized with a solid mathematical foundation. Re-designing database concepts in a distributed fashion is not a trivial task by any means. Instead, we take a different approach and consider how multiple RDBMS engines can operate on the same data set simultaneously, without having to know about each other.

Figure 1 presents our target solution, where we take a commodity-off-the-shelf RDBMS engine and scale it out horizontally by substituting its single node storage with a distributed storage layer, leaving the RDBMS engines intact. Each engine is independently given the illusion that it’s the only engine running in the system, and is granted access to all the data which is not partitioned. This allows multiple RDBMS engines to run simultaneously, providing scalability, while not changing the consistency nor the generality of the provided service.

The interface between an RDBMS engine and the storage layer masks out the underlying distributed nature and correctly isolates the engines from each other. Data storage, indexing and locking are the core functionalities of the storage layer, where each component can be scaled independently from the others. With such a loose coupling between query processing and storage, the entire system can scale in two dimensions. In the first dimension, the storage can transparently scale to provide more storage capacity and operation throughput. In the second dimension, the number of RDBMS engines interacting with the storage may scale to accommodate more query processing power and handle higher transaction rates.

Figure 1: Target RDBMS Architecture.

As a first step of implementing such an architecture, we foresee distributed indexing to be the major challenge. The index is an integral part of an RDBMS, making data retrieval efficient by answering complex queries about stored data. Since almost every operation needs to either read from or write to the index, the index is an important potential point of contention. Hence, any scalable RDBMS should efficiently address indexing. The need for transactional consistency and to be able to cope with high contention in a scalable fashion makes implementing distributed indexing a difficult task. To this end, we propose a solution to distributed indexing which can provide strong consistency guarantees with the ability to serve write-intensive workloads without having to partition the data or limit the query model.

Additionally, in an RDBMS, locking is the only mechanism used to provide isolation between transactions. Locking is used for hiding the transient state of one transaction from another. Based on a transaction’s isolation requirements, different items and attribute ranges are locked as the transaction executes; these locks are released when the transaction aborts or commits. Either way, the locking mechanism should ensure isolation semantics are never violated, requiring a scalable but fully consistent architecture. The need for efficiently handling complex range locking renders most available scalable locking systems inefficient for our target RDBMS. To address distributed locking, we also implement a distributed lock manager with the ability of handling complex range queries.

The last component needed for our distributed RDBMS is scalable storage that should support standard data storing and retrieval operations. Since a myriad of solutions currently exist for scalable storage, intuitively we expect scalable locking and indexing to be the most significant challenges en-route to our goal. We expect these three subsystems (i.e. indexing, locking, storage) to sufficiently provide each RDBMS engine with the illusion of isolation. In the next section we provide implementation details of the distributed index and the lock manager.

3. IMPLEMENTATION

We implemented both systems over SinfoniaEx [13], a Sinfonia-like memory service [5]. Sinfonia is a proprietary distributed transactional memory service that provides atomic access to a shared memory via light-weight mini-transactions (not to be confused with traditional database transactions). Minitransactions have been shown to serve
as a powerful primitive for large-scale applications requiring ACID properties. SinfoniaEx is our open-source implementation of Sinfonia here at UBC, with minor modifications and extensions to the interface, including a built-in memory allocator.

Our decision to use a Sinfonia-like service was motivated by the wide range of features and benefits it offers to applications built over it, most of which are necessary for distributed systems. Sinfonia has a simple programming interface, and seamlessly provides durability and fault-tolerance using logging. It achieves high performance by serving data from memory rather than disk and scales because of the short time required to complete minitransactions. It also provides a means for users of a distributed application concurrently accessing shared data to isolate their operations from each other by validating read data before making any modifications. This provides the users with strong consistency, i.e. ACID guarantees.

Both the index and lock manager are based on a modified SinExTree [14]. SinExTree extends the scalability of tree data structures by protecting both the data it stores and its structure from corruption and race conditions by using SinfoniaEx. SinExTree maintains the features offered by SinfoniaEx, while providing the ability to efficiently execute storage operations. We first elaborate on the structure of SinExTree, then provide details of how we use SinExTree for distributed indexing and locking.

### 3.1 SinExTree

SinExTree is a scalable distributed spatial partitioning tree (SPT) built over SinfoniaEx which provides a key/value storage interface. A key may have multiple attributes, and each attribute may have any data type as long as a strict ordering can be defined over the type. Being an SPT, the spatial layout of SinExTree allows for efficient implementation of various types of complex queries. It allows an application to define its own queries over multiple attributes, such as range queries. Operation latencies of SinExTree are in the order of milli-seconds, even in a virtualized cloud environment, while it can scale to hundreds of machines with a steady increase in throughput even under write-intense workloads. The collection of these features make SinExTree an ideal solution for distributed indexing and locking.

In an SPT data is only stored in leaf nodes. Branch nodes hold pointers to their children which may be other branch nodes or leaves. In an n-dimensional SPT, each node encloses a specific subspace of n dimensions, storing all items with keys falling inside its region. If the number of items stored in a leaf node exceeds its capacity, the node is split into smaller regions and data is moved to the new leaves, transforming the original node into a branch. Empty leaves are eventually merged back into a larger region. Hence, an SPT avoids contention in a region by increasing the granularity of spatial partitioning.

SinExTree provides a key/value interface and uses SinfoniaEx minitransactions to perform all of its operations. Using minitransactions is vital to protect the tree structure from race conditions and corruption. For each SinExTree instance, a subset of tree nodes are stored in each machine. An application using SinExTree links to a local library and only interacts with its own library. To carry out an operation, the library internally traverses the tree, communicating with other machines that host other parts of the tree, and reads nodes required to complete the operation via separate minitransactions.

While several applications may concurrently be using SinExTree, they have no knowledge about each other and should be properly isolated. Since parts of the data may need to be fetched asynchronously over the network, race conditions may cause some of the already read data to change while other parts are still being read. To address this, each operation must ensure that none of the nodes it read from have been modified by a conflicting operation while the current operation was in progress. For example, a lookup operation should ensure the key/value pair it references has not been removed nor modified by another operation since it was read.

SinExTree handles this by using a special minitransaction called the operation’s uber transaction. By versioning tree nodes, the uber transaction validates the version of data it has read in order to complete the operation before committing any changes. An operation completes when its uber transaction is committed successfully. If two or more operations have conflicting results, only one uber transaction will succeed and the rest will fail. An operation with a failed uber transaction has to be retried until it succeeds. This allows SinExTree to provide strong consistency on operations it performs, which means the result of an operation executed by one application is immediately visible to other applications using the same tree.

Multi-attribute application-defined queries is the key feature of SinExTree that presents it as a suitable candidate for distributed indexing in our RDBMS. The fact that operations are performed atomically strongly recommends SinExTree for locking, which is why we built both our systems on top of SinExTree, described next.

### 3.2 Index

Our distributed index inherits some key features from SinExTree. It is scalable, fault-tolerant and provides low latency for operations it performs. It is capable of handling any kind of query an RDBMS engine can possibly issue, while providing all the users with full access to the entire indexed data.

Figure 2 illustrates the architecture of the index. As with any system built over SinfoniaEx, the system consists of two levels of processing nodes. In the back-end memnodes store the index data and service SinfoniaEx minitransactions. In the front-end, index hosts serve users of the index. Each user is assigned to one index host which will handle its requests. An index host uses SinExTree to carry out requests on behalf of its users. Note that both the index hosts and the memnodes are part of the distributed index.

The index can dynamically scale by increasing the number of memnodes and by increasing the number of index hosts without interrupting the service. Index hosts do not store state, but read (and possibly cache) data they require from the memnodes. This allows the index to spawn more index hosts to accommodate more users, or kill some of them once their users have been handed-off to other hosts to reduce allocated resources. The memnodes store all the data and are therefore protected from failures by SinfoniaEx, through means of logging, replication or both depending on the configuration. Scaling the memnodes requires the assistance of a migration application, which uses SinfoniaEx minitransactions to transparently move data to newly joined memnodes.
or away from memnodes about to leave.

In an RDBMS, each table is associated with an index. While an item stored in a table may have multiple attributes, only a subset of the attributes are indexed. In our system, to index an item, an n-dimensional SinExTree is used which is created alongside the table. To index an item, the index host translates the selected attributes of the item into a key, and inserts the key accompanied with a pointer to the item inside SinExTree. The same key should be provided later on to delete the item from the index. To retrieve data, the key is generated from the requests coming from the RDBMS engine by the index host. It also provides SinExTree with the set of handlers it needs to complete the query and returns the result upon completion. SinExTree’s strong consistency ensures that the result will include all the latest indexed items, but not removed ones.

The original target environment of SinExTree was 3-D spatial partitioning for multi-player games. A key in this environment is simply a fixed-length vector of floating-point numbers representing the spatial coordinates of an item in a virtual world. We had to modify SinExTree to provide more flexibility in how a key is created from multiple attributes and to accept other data types. Our modifications allow the application to provide a set of handlers to SinExTree which dynamically alter its behavior, such as defining the number of dimensions and customizing the policy that chooses how to divide a set of keys into multiple sub-ranges when a node must be split. This enables our index to support any operation requested by an RDBMS engine.

3.3 Lock Manager

Range locking mandates locks to be granted on attribute ranges. The spatial layout of SinExTree provides an insight on how to address range locking in a scalable fashion. We observe a lock represents a region in the n-dimensional space of attributes, where different regions may not intersect. Thus it may be possible to use SinExTree for locking, where locking a range means inserting a region inside SinExTree and unlocking translates to removing it. The important invariant is that inserting (i.e., locking) will only succeed if no conflicting regions currently exist in SinExTree.

The lock manager is built over a more extensively modified SinExTree but still with an architecture similar to the index and uses lock manager hosts in the front-end instead of index hosts. SinfoniaEx even allows the index and lock manager to share the same memnodes to better utilize resources.

In SinExTree, a key is a point in the n-dimensional space and all operations operate on keys. To support range locking, we modified SinExTree to operate on ranges. The modified interface only supports two operations: insert and remove, which translate to lock and unlock in the lock manager, respectively. When a lock manager host receives a lock request, it transforms the requested range into a region and tries to insert it inside SinExTree. Unlocking involves removing the region.

Inserting a region in SinExTree requires finding all the leaf nodes that intersect the given region. The leaves should store their intersection with the region, which will later be used to reject conflicting regions. If a conflict is found in any of the leaves, inserting has failed and it is up to the RDBMS engine to decide when to retry.

Being an SPT, when a leaf stores more regions than its capacity, it has to be split into smaller regions. The regions it currently stores have to be further split and forwarded to the new children that intersect them. Hence, compared to an unmodified SinExTree, operating on ranges is more expensive that operating on keys. While we cannot quantify performance degradation of SinExTree at this point, we note the lock manager only stores locks, not items. Even if a database stores millions of items, the number of locks stored in the lock manager is still proportional to the number of active transactions in the system, which is expected to be much smaller than the number of items. Hence, intuitively, we expect the overhead of our modifications to be acceptable for the lock manager.

Using a similar architecture for both the index and lock manager creates a harmony in the system and simplifies the task of resource allocation to each component and recovery from failures. Considering they both provide strong consistency, they can be seamlessly be plugged into existing RDBMS engines.

3.4 Correctness

To verify correctness of the overall system, we classify correctness into two categories: data integrity and data correctness. Data correctness implies that all operations should comply to semantics required by an RDBMS. For example, each transaction should be provided with the isolation level it requested, and rolling back a transaction should not create a transient inconsistent state, no matter how brief. Data correctness is affected by the distributed nature of the system. Since multiple operations may be simultaneously operating on the shared data, it should not be corrupted due to race conditions between (possibly) conflicting operations.

We use SinfoniaEx to provide data correctness and to isolate operations. Each operation essentially runs a read-modify-write cycle where it has to validate what was read before committing a write. This ensure data correctness is never violated. Higher level RDBMS logic, such as executing transactions, is conducted by commercial RDBMS engines on top of our components. Since we are not modifying the engine or changing any database fundamentals, we expect data integrity to be a given once we have the whole system running.

Finally, any distributed system has to explicitly deal with failures, where a failure could be an application crash, a node crash, or even multiple simultaneous node crashes. SinfoniaEx’s recovery manager deals with all three types of failures. In case of failures, parts of the data may become unavailable while recovery is in progress, but it is guaranteed that data
never ends up in an inconsistent state which could affect correctness. The only part not protected by SinfoniaEx is the RDBMS engine. If an engine crashes, some transactions may be left uncompleted. We foresee either using well known logging techniques or protecting RDBMS engine state with SinfoniaEx to suffice. In the latter scenario, each engine stores its current state, e.g., list of locks it holds, transaction items it wrote or modified before committing, inside SinfoniaEx. When an engine crashes, an engine recovery manager will then deal with cleaning-up state or assigning a new engine to resume where the previous left off.

4. EVALUATION

In this section we present a preliminary evaluation of our prototype implementation of the index using TPC-C. The TPC-C benchmark is a popular benchmark that extensively simulates a real world transactionally consistent workload, and we believe it to present a challenging workload on an RDBMS. Our goal is to see how the index scales under the same workload but when more processing power is available.

4.1 Benchmark

In an RDBMS, the execution of a transaction involves sending requests to the many different components of the system, including the lock manager, the storage layer, and the index. These requests must be performed sequentially. For example, first locks have to be granted, then the index is queried, and finally items are retrieved and/or stored. Because in this paper we are interested in the performance of the index, we assume that except for the index all other components of an RDBMS have infinite throughput and zero response times. The results obtained by making these assumptions have two properties. First, the benchmark will put the maximum stress on the index, as requests to the index are not delayed as they normally would be when a transaction is busy with other components. Second, throughput results will be an optimistic upper bound on what the RDBMS can actually deliver. Nonetheless, we wish to see how our distributed index performs under maximum stress, helping us identify potential limitations.

To obtain the benchmark, we ran TPC-C with one warehouse with multiple users on a single node RDBMS and logged all the index operations performed by the users. Each user executes a chain of 1000 transactions in sequential order, where the completion of one transaction initiates the start of the next. We then use the logs to replay the benchmark on our prototype distributed index. Note that at this stage we're more concerned about the trend of throughput scalability of the index rather than absolute performance numbers.

4.2 Scalability

We use Amazon's EC2 as our evaluation environment with 72 High-CPU Extra Large instances, each having 8 virtual cores with 2.5 EC2 compute units of processing power and 7GB of memory [1]. We run a set of experiments, each emulating between 4 and 64 different users. Each user replays a log captured from the single node RDBMS. While the processing power of any single machine is limited, the objective of our evaluation is to observe how the index can utilize more processing power as the system scales. Hence, we vary the number of index hosts from 1 to 64 to scale the front-end and measure the overall throughput of the index. In all our experiments, SinfoniaEx was set to log data synchronously to disk to provide full fault-tolerance. Figure 3 depicts the results when using 8 memnodes in the back-end.

The system starts to thrash as the number of users increases to 64. However, the thrashing point can be shifted by increasing the number of hosts. For example, with 4 hosts the maximum throughput occurs at 16 users, while by using 16 hosts the maximum is shifted to 32 users. The best performance we achieve in our evaluation is with 64 hosts (top right) with 23554 Ops. The results confirm our hypothesis that the index can scale with more processing power. As a rough estimate, multiplying measured Ops by 1.7 translates to TPC-C transactions per minute (TpM). Thus, if the overall RDBMS were to have the same performance, it would reach approximately 40000 TpM.

This choice of 8 memnodes represents the knee in performance as the number of memnodes changes. With fewer than 8 memnodes, the memnodes become the bottleneck; with 4 memnodes the peak throughput of the system with 64 users drops by 19%. Additional memnodes beyond 8 provide very little performance gain; with 16 memnodes the peak is only increased by less than 3%.

Amazon's Relational Database Service (RDS) [2] is their current DB-as-a-Service for cloud-based applications. RDS provides an intriguing price model where users are charged by the hour based on the selected instance type. Such a feature allows commercial service providers that use RDS as part of their system to optimize their operation cost based on their current demand. Unfortunately, RDS’s throughput is limited to the throughput that the largest EC2 instance can provide. For example, RDS can sustain at most 5000 transactions per minute with 64 concurrent users [15].

Our distributed index extends RDS’s price model to allow system operators to buy higher throughput for their system, beyond the capabilities of the largest EC2 instance, as required. Table 1 presents the operation costs of the distributed index at each setting of the evaluation. Operation cost is equal the cost of running the required number of hosts plus 8 memnodes at the nominal price of $0.68/hour for the selected instance type. The last row presents the cost of operating the system normalized to its maximum throughput, in terms of dollars per 1 million operations per hour. Hence, an operator can balance price and throughput based on current demand by dynamically scaling the system. For
example, with 16 users using 4 or 8 hosts has a 4% difference in throughput, but operating with 8 hosts costs 33% more than operating with 4. This provides an analogy to utility services, where users are provided with a high maximum possible throughput, but are charged by their actual usage.

Finally, we emphasize our code is research code currently in an early stage of development, abundant with assertions, running in virtual machines on commodity hardware. We expect simple optimizations and dedicated hardware to improve our performance results significantly. We do not evaluate the lock manager, as the performance of a locking system heavily depends on contention created by conflicting locks. This is tightly coupled with the timing of concurrent events which may not be accurately emulated.

5. FUTURE WORK

With the distributed index and lock manager in place we’re one step closer to our goal, a scalable cloud-aware RDBMS. Our current focus is to fully implement the rest of the components required for functionality of the RDBMS. If the final system’s scalability follows the same trend as the distributed index, it will serve as a suitable replacement for existing DB-as-a-Service solutions.

The new utility service price model provides more incentive for corporations to migrate over to our elastic service as the infrastructure弹性ally adapts to their instantaneous demands. On the other hand, service providers better utilize their resources by sharing them between their customers. However, new challenges such as providing security and privacy for stored data by means of access control could be an interesting venue to pursue.

6. CONCLUSION

In this paper we propose scalable distributed indexing and locking services which are two of the vital components of a distributed RDBMS. Our proposed index is scalable, consistent and fault-tolerant, and compared to other indexing techniques, such as data partitioning, provides the most flexibility as it does not require any knowledge about nor does it enforce any restrictions on the queries that can be executed. The lock manager is also consistent and fault-tolerant, and provides range locking. Scalability results of the index obtained using the TPC-C workload show our solution to be promising. While the absolute throughput values will change when used in a fully functional RDBMS, we expect the trend in capacity growth to stay more or less the same.

7. ACKNOWLEDGEMENT

We thank Kenneth Salem for their insightful guidance and feedback through the course of this work. We also thank Rachel Pottinger for the valuable feedback.

8. REFERENCES


