Uniqorn: Scalable and Consistent Global Secondary Indexes for Sharded Data Stores

Paper # 222

Abstract

Sharding a logical table by primary keys has been a scalable way to meet the enormous volume growth for large-scale business applications. It provides fast access to data records by primary keys. However, many applications might benefit from fast access to data records by one or more business-meaningful secondary keys with global secondary indexes (GSI). Nevertheless, system unreliability and operation concurrency complicate the task of building and maintaining GSI. If not handled appropriately, anomalies may occur. Due to this, many large-scale data systems either sacrifice GSI, only support local secondary indexes (LSI), or support GSI with reduced consistency in favor of scalability or performance. The lack of full support of GSI limits the usability and efficiency of application development.

We built Uniqorn, a sharded data store that supports linearizable create, read, update, and delete (CRUD) operations of individual data records with either primary or secondary keys via GSI. Using global optimistic indexing, Uniqorn tames system unreliability and operation concurrency to prevent anomalies. It is lightweight, non-blocking, scalable, applicable to a few out-of-box relational or key-value/document data stores. Our experiments and production deployments demonstrate that large-scale sharded data stores need not forgo the benefits of GSI even with strict consistency requirements.

1 Introduction

Our organization has an ever-growing, enormous amount of business data. We have sharded (e.g., by hashing the primary key into specific servers/clusters [4]) our logical data table to meet the sheer needs of large-scale business applications. In such a sharded data store, applications can rapidly create, read, update, and delete (CRUD) data records by primary keys. However, unlike [26] where most services only need primary-key access, many business-critical applications in our organization also need to rapidly access data records by one or multiple secondary keys. For example, a customer account application may need to access customer data records by their (secondary key) email addresses or phone numbers instead of (primary key) user ids. A third-party market application may need to find all vendors of a given item by a given item id. Enabling data record accesses by secondary keys greatly enhances the usability of a sharded data store while still retaining its high scalability and availability.

Moreover, some of our applications optionally need strong consistencies upon secondary keys. First, CRUD operations on a data record (including its index records), in the presence of current CRUD operations and system failures, appear to be instantaneous (linearizability [10]). For example, a user must be able to log in immediately using her/his email address after creating a account. Second, some applications need the uniqueness guarantee that no two data records have the same secondary key, we designate such a secondary key with *global uniqueness constraint* (or just uniqueness constraint) as an *alternate key*. For example, the customer account application guarantees that no two customer accounts share the same email address. Third, some applications need the validity guarantee that if no data record has an alternate key in data store, then this secondary key can be immediately owned by a new or existing data record. For example, a user canceled her/his account should be allowed to immediately create another account with the same email address if nobody has taken it in between.

To read a data record by one of its secondary keys, an existing solution is to scatter/gather across all shards with the aid of a local secondary index (LSI) on each shard (i.e., a secondary index of only local data records on the same shard). As an example of two shards with data records (sharded and sorted by ids): [id:1, email:alice@x.com, name:Alice] and [id:3, email:carl@x.com, name:Carl] in shard1 and [id:2, email:bob@x.com, name:Bob] and [id:4, email:dave@x.com, name:Dave] in shard2, respectively. A LSI on secondary key emails are built and maintained locally on each shard with index records (sorted by emails): [email:alice@x.com, id:1] and [email:carl@x.com, id:3] on shard1 and [email:bob@x.com, id:2] and [email:dave@x.com, id:4] on shard2. Note that in LSI, a data record and all of its index records are located on the same shard. Looking for a data record by email alice@x.com will request each shard to first look up the corresponding primary key of secondary key alice@x.com in its LSI and then read the data record corresponding to the found, if any, primary key (id:1 in this case). Scattering/gathering is not scalable in terms of throughput, latency, and resource consumption as the number of shards increases. With even one shard unavailable, the read may fail. In case that a LSI is not provided in each shard, frequent and costly shard scans will render the whole shard unusable.

To avoid costly scattering/gathering, global secondary indexes (GSI) was utilized to facilitate secondary key lookups, where both data records (by their primary keys) and their index records (by their secondary keys) are sharded, respectively. A read by a secondary key is performed in a two-step lookup. First, we lookup the alternate key to yield the shard key. Second, we access the data record using the shard key. For instance of the same two shards and the same distribution of data records among shards as before. A GSI on secondary key emails is built and maintained globally with index records (sharded and sorted by emails): [email:bob@x.com, id:2] and [email:carl@x.com, id:3] on shard1 and [email:alice@x.com, id:1] and [email:dave@x.com, id:4] on shard2, respectively. Different from LSI, data record [id:1, email:alice@x.com, name:Alice] and index record [email:alice@x.com, id:1] are now located on different shards. Looking for a data record with email alice@x.com will first look up the corresponding primary key of secondary key alice@x.com directly on shard2 and then read the data record corresponding to the found primary key (id:1 in this case) directly on shard1.

However, unlike LSI, data records and their index records in GSI are distributed across different shards, leading to potential inconsistency between data records and their index records in the presence of system failures and operation concurrency. If not handled appropriately, consistency anomalies may occur. As an example, suppose that both Alice and Bob try to change their email to ab@x.com by first updating their index records from [email:alice@x.com, id:1] and [email:bob@x.com, id:2] to [email:ab@x.com, id:1] and [email:ab@x.com, id:2], respectively, and then updating their data record to [id:1, name:Alice, email:ab@x.com] and [id:2, name:Bob, email:ab@x.com], respectively. It is possible with interleaving of actions that index record [email:ab@x.com, id:1] is overwritten by [email:ab@x.com, id:2], yet both data records [id:1, name:Alice, email:ab@x.com] and [id:2, name:Bob, email:ab@x.com] are persisted in shard1 and shard2, violating the uniqueness constraints. Our applications treats such a violation as a disaster as if the customer account Alice has lost in our system.

Due to those consistency anomalies, many large-scale data systems choose to sacrifice secondary indexes [5], only support LSI only [7, 14, 12, 22], support GSI with eventual consistency in favor of scalability or performance [11, 17, 24], or support GSI with the supports of general distributed transactions [25]. The lack of full support of GSI limits the ease and efficiency of application development on such systems. The extra complexity of general distributed transactions brings unnecessary burden to data store.

Uniqorn was designed to enhance a sharded data store with GSI with the following requirements/goals:

- **Consistency**. The system must be able to provide linearizability among CRUD operations, uniqueness and validity of secondary keys. It is an intolerable burden for applications to reason about these consistencies. The uniqueness constraint can be optionally dropped.
- **Scalability**. Scatter/gather is prohibited for all CRUD operations. As the number of shards increases, the system should sustain linearly increased CRUD throughput via secondary keys and each CRUD operation via secondary key should have nearly constant latency. We aims to scale to hundreds of shards and deem unacceptable for a customer-facing application with prolonged CRUD latency.
- **Applicability**. The system should work with several popular storage engines in our organization with the following common characteristics:
  - *Single-Record Read Consistency*. A read returns the last persisted data/index record of a given shard key. A shard key will be the primary key of a data record or the secondary key of an index record.
  - *Single-Record Conditional Write*. Write of a data/index record in a compare-and-swap manner, conditional on a given column/attribute value of the data/index record (i.e., an insert succeeds only if a data/index record of the given shard key has not been persisted, an update/delete succeeds only if the specific column/attribute of the data/index record of a given shard key is equal to a given value). This characteristic can be used to guarantee if a data/index record in the database has been changed be-

tween the time it was read and the time it is to be written, the write will fail.

Relational databases (e.g., MySQL and PostgreSQL) and key-value/document stores (e.g., SimpleDB [23], and Couchbase [11]) satisfy these two requirements.

- **Availability**. Most of our applications are read-dominating and secondary keys are not frequently updated. Without sacrificing consistency, we want high read-availability and high write availability for create and update operations without secondary key changes.

Uniqorn uses an approach of optimistic indexing to achieve aforementioned goals with following characteristics:

- Unlike general distributed transactions, it is lightweight without the heavy-lifting (e.g., pre-writes, rollbacks) of two-phase commits.
- It is non-blocking comparing to pessimistic two-phase-locking based concurrency controls.
- It does not rely on the timestamp oracle and clock synchronization facilities like timestamp-ordered concurrency control.
- It does not rely on the multiple versions of a data/index record of mutli-version concurrency control.

Our main contributions lies in that the demonstration and evaluation of a shard data store with GSI that augments the consistent access patterns of such a data store from one dimension (i.e., by primary key only) to two-dimensions (i.e., by either primary keys or secondary keys). Our work also contradicts a common belief that GSI cannot be kept transactionally consistent with data records when designing for almost-infinite scaling [9].

The remainder of this paper is organized as follows. Sections 2 describes the system architecture, anomaly analysis, and conflict resolution, Section 3 briefly discusses some implementation concerns we have in our implementation. Section 4 presents performance evaluations. Sections 5 presents related work and Section 6 concludes.

2 Design and Implementation

A Uniqorn system consists of three components (Figure 1): an *index store*, a *data store*, and stateless *Uniqorn clients* (or just *clients*) for applications to use the system.

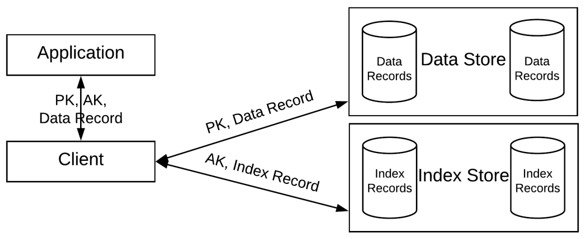

Figure 1: High-level view of Uniqorn

The storage engines of both index store and data store are required to meet the two requirements aforementioned (Section 1). Data records are horizontally partitioned by their primary keys into data store partitions. A data record is logically denoted as a tuple [pk, aks, val], where pk serves as both the primary key and the shard key of a data record, aks is the set of alternate keys of the data record, and val represents all other attributes/columns as a single field val for the convenience of presentation. Depending on specific type of data or index stores, a val can be normalized and physically stored into multiple tables (e.g., in relational databases) or denormalized to physically store as a single row (e.g., in key-value or document stores). Likewise, index records are similarly partitioned in index store partitions by their alternate keys. For each ak in aks of the data record, there is an index record denoted as [ak, pk].

Though index store is logically separated from data store, they can co-locate with each other. Index store and data store can be of different types (e.g., one is relational and the other non-relational) and form a hybrid system. Clients access both data store and index store (via either specific drivers or remote procedure calls) and implements the CRUD operations for applications. There are no direct interactions among clients. Clients are implemented in the form of libraries and embedded inside applications in our implementation. There is no direct interaction or dependency between index store and data store. Their indirect interactions are bridged and managed by clients. To make our system model applicable to various data store engines, we intentionally leave all functionality on clients and do not ask for or install special features or software on data stores, though they may imply better performance.

2.1 Anomaly Analysis

An index record may be in one of the following 5 potential states at a given time:

- **valid**: the data record referred by the index record is persisted in data store and has the alternate key of the index record.
- **missing**: the index record that can be derived from a persisted data record is not persisted in index store.
- **orphaned**: the data record referred by the index record is not persisted in index store.
- **disowned**: the data record referred by the index record is persisted in data store but does not have the alternate key of the index record.

All index record states except valid are treated together as **anomalies**. We also denote both orphaned and disowned index records collectively as **garbage** index records, which may be cleaned up at any time. Note that since alternate keys serve as shard keys of index records, no two index records of the same alternate key will be persisted in index store at any given time, therefore duplicated index records never occur.

In a snapshot of the system with data records ([id:1, email:alice@x.com, name:Alice], [id:2, email:bob@x.com, name:Bob], [id:3, email:bob@x.com, name:Carl]) and index records ([email:alice@x.com, id:1], [email:bob@x.com, id:3], [email:carl@x.com, id:3], [email:dave@x.com, id:4]). Index record [email:alice@x.com, id:1] and [email:carl@x.com, id:3] are valid. However [email:dave@x.com, id:4] is orphaned, [email:bob@x.com, id:3] is disowned, while [email:bob@x.com, id:2] is missing.

A record may have multiple alternate keys and their index records may be distributed across multiple index store shards. Reading or writing alternate keys across multiple shards without transactions can not be performed consistently. Therefore, index records can not be taken as the source-of-truth of records. Instead, Uniqorn treats *data records as the **source-of-truth** for alternate keys, version, and attributes*.

The source-of-truth nature of data records implies that Uniqorn can

- **avoid deleting valid index records**: Every operation first reads the data record referred by the index record and then validating if it has the alternate key of the index record. The garbage cleanup operation will abort if the index record is validated as valid.
- **avoid the time window of missing index records**: If a create/update operation first persists a data record and then persists its index records. Immediately after the data record is persisted and before the derived index records persist, these derived index records are missing in index store. During this time window, a read operation for one of the alternate keys of the data record will not find any persisted index record that has this alternate key and hence return empty to application, though the data record that has the alternate key is persisted. For this purpose, Uniqorn always persist the index records into index store before persisting the data record into data store. However, it has a side effect of leaving garbage index records in index store if server/client/network fails after persisting the index records (but before persisting the data records).
- **reduce all anomalies to a single anomaly of missing index records and exclude read and delete operations as anomaly sources**: If we could assume missing index records never occur, we can solve for all other anomalies:
  - **mask garbage index records by client for read**: If no missing index records ever occur, Uniqorn client can mask garbage index records for read operations as follows: if the index record of an alternate key is not persisted, no data record that has that alternate key is persisted; if the data record of a primary key is persisted and has that alternate key, then that index record is valid and the data record can be returned to applications, otherwise, that index record is garbage.
  - **tolerate garbage index records by client for delete**: a delete operation can delete a data record directly without deleting any of its derived index records, which does leave them orphaned, but it does not result in any missing index records.

  Therefore, we just need to consider create, update, and internal garbage cleanup (**GC**) operations.

To simplify discussion, we will assume in the following that only GCs will delete index records. If a create/update operation needs to persist an index record whose alternate key has been used by an existing (be garbage or valid) index record, it must first invoke an internal GC to safely delete the index record and then insert the index record. GCs can also be invoked at background to clean up accumulated garbage index records as well. We also collectively denote both create and update as write, and will distinguish them only if necessary.

If (1) every write of a data record dr is carried out by first reading the persisted data record pdr, then inserting an index record for each alternate key $ak \in dr.aks \setminus$

$pdr.aks$ into index store that the read data record does not have and next persisting dr in data store, and (2) no GCs will ever be performed, then no missing index record would ever happen. That is, concurrent writes (without involving GCs) do not cause the anomaly of missing index records. However, it is not possible to bypass deleting garbage index records since either accumulated garbage index records take significant amount of space or the alternate keys of garbage index records have to be reused by a write (otherwise, validity is violated). Essentially, *the anomaly of missing index records is caused by write/GC conflicts*.

Suppose that an index record has been persisted in index store by an ongoing create or update that has not persisted the data record referred by the index record into data store. On one side, if such an ongoing create or update later persists the data record successfully, this persisted index record will transition from garbage to valid. If such a transitional index record is unsafely deleted by a garbage cleanup operation, the index record is missing. On the other side, if the ongoing create or update fails (e.g., due to system failures) to persist the data record permanently, the prior persisted index record becomes garbage (which can be safely deleted). However, it is ambiguous whether a garbage index record was persisted by an ongoing or failed write.

In the following, we will first introduce versions and augment data records and index records with versions, and then show how versions are used optimistically to resolve this ambiguity (therefore resolve write/GC conflicts).

2.2 Conflicts Resolution

We augment each data record and index record with a "version" [g, v] to form [pk, g, v, aks, val] and [ak, pk, g, v] respectively, where g is the generation identifier of the data record among all data records that have ever associated with pk, v is increased upon every update of the data record (starting from 0) within a generation. We embed the versions within data records and index records mainly for efficiency and consistency between the versions and other columns of the same data/index record. In our implementation, client generates a generation number in the form (ts, $clientId$) for each newly created data record, where ts is a local timestamp reading from a logically forward-moving clock at the client and $clientId$ is the unique identifier of the client among all clients. We use a simple mechanism to avoid clock backward-tick at normal runtime and in the presence of crash-restart. At normal runtime, each client keeps the last allocated ts in memory and allocates next ts = max($ts + 1$, wall-time).

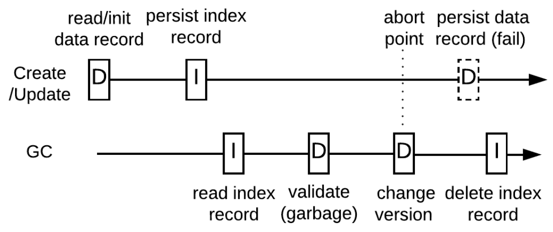

Figure 2: The safe sequence of four phases for a GC to delete an index record: the index record must not be changed by a create/update from the first to the last phase; the data record must not be changed by a create/update from the second action to the third phase. Both are realized by a read phase of index/data record and a following write phase of index/data record conditional on the previously read versions. After a GC changes the version of the data record, the ongoing create/update is effectively aborted. A box with D or I inside means an access (with specific action noted nearby) to data store or index store, respectively. Phases occur from left to right in time.

To avoid backward-ticks after restart, a client persists locally a $maxTs$ that it can allocate up to. When ts reaches $maxTs$, $maxTs$ is extended (e.g., 10 seconds) and persisted again before the client can allocate more timestamps. When the client restarts, it initializes ts with the persisted $maxTs$.

On one hand, Uniqorn could have used a single global timestamps ts to order those operations. However, Uniqorn avoids it due to three reasons: (1) Uniqorn avoids an external time oracle to provide increasing timestamps. (2) Lack of accurate time synchronization facility, clock drift between Uniqorn clients is usually larger than or on par with expected operation latency. Waiting out of clock drifts to resolve concurrency conflicts can drastically increase operation latency. and (3) Uniqorn does not expect high concurrency of conflicting operations in the targeting workload. On the other hand, Uniqorn could use UUIDs as generation numbers. We did not use if since (1) even though the timestamp info in generation number from different clients are not comparable; they are comparable if they are generated from the same client, which helps reduce concurrency failure occurrences and (2) we utilize the timestamp info in each generation number to diagnose production issues at runtime.

Uniqorn equips a write operation with versions and restricts a GC to only delete an index record of certain versions, as shown in Figure 2. A write operation of data

	Phase	Data Store [pk, g, v, aks, val]	Index Store [ak, pk, g, v]
Alice	insert dummy data record	[1, g_a, 0, ∅, ∅]	
Alice	insert index record	[1, g_a, 0, ∅, ∅]	[ab@x.com, 1, g_a, 0]
Bob	insert dummy data record	[1, g_a, 0, ∅, ∅] [2, g_b, 0, ∅, ∅]	[ab@x.com, 1, g_a, 0]
Bob	insert index record (find ab@x.com in use and then call GC)	[1, g_a, 0, ∅, ∅] [2, g_b, 0, ∅, ∅]	[ab@x.com, 1, g_a, 0]
GC	read [1, g_a, 0, ∅, ∅] and validate [ab@x.com, 1, g_a, 0] as garbage	[1, g_a, 0, ∅, ∅] [2, g_b, 0, ∅, ∅]	[ab@x.com, 1, g_a, 0]
GC	delete Alice's dummy data record	[2, g_b, 0, ∅, ∅]	[ab@x.com, 1, g_a, 0]
GC	delete Alice's index record	[2, g_b, 0, ∅, ∅]	
Bob	insert index record (continue)	[2, g_b, 0, ∅, ∅]	[ab@x.com, 2, g_b, 0]
Bob	write data record, succeed	[2, g_b, 1, ab@x.com, "Bob"]	[ab@x.com, 2, g_b, 0]
Alice	write data record, fail	[2, g_b, 1, ab@x.com, "Bob"]	[ab@x.com, 2, g_b, 0]

Figure 3: Alice and Bob attempt to create their data record with the same alternate key email ab@x.com, respectively. Alice inserts a dummy data record [1, g_a, 0, ∅, ∅] first, then inserts index record [ab@x.com, 1, g_a, 0] and slows down thereafter. Then Bob inserts a dummy data record [2, g_b, 0, ∅, ∅], Bob finds the alternate key ab@x.com has been used. It calls a GC to delete it if safe. The GC validates index record [ab@x.com, 1, g_a, 0] as garbage (since this dummy data record [1, g_a, 0, ∅, ∅] does not has any alternate keys), deletes the dummy record [1, g_a, 0, ∅, ∅], and deletes index record [ab@x.com, 1, g_a, 0] successfully since no other has modified this index record. The GC succeeds and then Bob continues to insert its own index record [ab@x.com, 2, g_b, 0], and last successfully updates the dummy data record it inserted previously to [2, g_b, 1, ab@x.com, "Bob"] since no other has modified its dummy data record. Alice now continues to update the dummy data record it inserted previously to [1, g_a, 1, ab@x.com, "Alice"], but fails since its dummy data record has been deleted by the GC.

record dr is performed in 3 phases:

1. **Read/Initialize Data Record**. It reads the data record pdr (with an embedded version) from data record. A create may not read an existing data record pdr (pdr = ∅). In this case, the create makes a new version [($timestamp$, $clientId$), 0] in memory, persists in data store a dummy data record pdr = [$dr.pk$, ($timestamp$, $clientId$), 0, ∅, ∅] first.

2. **Persisting Index Records**. It persists an index record ir for each alternate key $ak \in dr.aks \setminus pdr.aks$ in the form ir = [ak, $dr.pk$ $pdr.g$, $pdr.v$] into index store in parallel. If $dr.aks \setminus pdr.aks$ is empty, this phase can be skipped. No existing index records for each alternate key in $pdr.aks \setminus dr.aks$ will be deleted in this step since they are still valid until the data record dr is persisted. When a write persists ir, an index record pir with the same alternate key ($ir.ak$ = $pir.ak$) may exist. In this case, the write needs to invoke an internal GC to delete it and then insert ir thereafter. If the GC fails to delete pir, the write fails due to pir is valid or another concurrent write is reusing the alternate key $pir.ak$.

3. **Persist Data Record**. It writes dr with [$dr.g$, $dr.v$] = [$pdr.g$, $pdr.v$ + 1] into data store conditional on [$pdr.g$, $pdr.v$] (no other operation has changed the data record since its previous read)

A GC of an index record is performed in 4 phases:

1. **Read Index Record**. It read an index record ir.

2. **Validate Index Record**. ir is validated by reading the referred data record pdr (where $pdr.pk$ = $ir.ak$) and checking whether it has the alternate key of the index record. If the index record is valid ($ir.ak \in pdr.aks$) instead of garbage, the GC fails.

3. **Change Version**. If the data record pdr exists and is not a dummy data record, the GC changes the version of the data record ($pdr.v$ = $pdr.v$ + 1) conditional on [$pdr.g$, $pdr.v$] (no other operation has changed the data record since it is previously read for validation), which in effect fails any ongoing update operation. If pdr is a dummy data record, it can directly delete pdr instead of increasing $pdr.v$ by 1 since the pdr was inserted by an ongoing create. Deleting a dummy data record has the effect to fail a ongoing create. If pdr does not exist, ir is orphaned since the referred data record has been deleted from data record. In this case, the GC directly proceeds to next phase. Failing to update the version or deleting a dummy data record will fail the GC.

4. **Delete Index Record**. It deletes the index record at last conditional on $[ir.g, ir.v]$ (i.e., no other operation has attempted to reuse it since last read by the GC), which in effect prevents the GC from deleting a reused index record.

All CRUD operations, are implemented without timeouts, backoffs, and retries, but return specific exceptions to applications. Web applications usually have tight operation deadlines and are more suitable to handle exceptions to achieve best customer experience. A read/write call to either data store or index store is a simple get() or put() for key-value store or single-record query/insert/update/delete transaction (with auto commit) for relational store. Uniqorn clients currently do not support transactional bulk/range operations. Applications usually build their own non-transactional bulk/range operations by iterating the API calls. Figure 3 shows an example of CRUD operation with optimistic indexing. We provide a pseudocode description of this implementation, an informal proof of the uniqueness guarantee, and a formal TLA+ [18] specification (for model checking) in [28].

3 Discussions

3.1 Garbage

Garbage index records may be persisted in index store due to failures and concurrency conflicts. On one side, accumulated garbage index records may bloat disk storage. Usually the number of business-meaningful alternate keys (each alternate key has at most one index record in index store) are limited and saturated at a certain time. Uniqorn concerns disk usage less since index store is sharded to sustain large disk usage. On the other side, accumulated garbage index records may slow down CRUD operations. A create or update operation that reuses a garbage index record may have to additionally change the version and then delete the garbage index record before inserting its own index record. A read or delete operation with a garbage index record will incur an additional trip to get the referred data record for validation. However, our experiments and deployment shows that even excessive amount of garbage has only negligible impact on application-sensible performance (Section 4.4). Therefore, Uniqorn currently does not run a periodical background cleanup process to actively delete garbage index records. Instead, Uniqorn internally queues garbage-suspected records that are not immediately reused by a create/update operation and clean them up asynchronously with a limited number of threads and connections.

Dummy data records may be persisted in data store caused by an ongoing create that inserts it in data store, writes to index store, then fails to write to data store. Dummy data records can be directly deleted without breaching consistency. If an ongoing create fails to write to index store, it will delete the created dummy data record in best-effort. Frequent failures in between the short window of two consecutive accesses of data store is not expected, leading to negligible garbage dummy data records. Moreover, Uniqorn will directly fail a create operation without even creating a dummy data record in the first place if the operation has to add index records into an unavailable index store. Uniqorn currently does not run background cleanup operations to clean up garbage dummy data records.

3.2 Availability and Recovery

Uniqorn itself does not replicate index records or data records for improved durability and availability. It relies on index and data stores to provide redundancy and high availability. In our deployment, each index store shard or data store shard usually comprises a cluster of physical data store servers that replicate to one another. Since data store is the source-of-truth of records, the unavailability of relevant data store shards will fail all CRUD operations that access them. The unavailability of index store shards affects the availability of different operations differently. We optionally enhance Uniqorn clients to maintain high availability for all operations by either sacrificing scalability or consistency temporarily.

Upon the unavailability of relevant index store shards, Uniqorn maintains high availability, without sacrificing consistency, for read and delete operations and for create and update operations that do not add new alternate keys or change existing alternate keys. A read (by an alternate key) relies on the availability of the index store shard where the index record of the alternate key is persisted. If the index store shard is unavailable, the read operation will fail. In this case, we enhance Uniqorn clients to fall back to looking up the alternate key by scattering/gathering across all data store shards if they provide the LSI for alternate keys. Similarly, a delete (by an alternate key) can fall back to looking up the alternate key by scattering/-gathering across all data store shards. However, scattering/gathering across all data store shards incurs large network traffic, prolonged operation latency, extra resource consumption at both server and client sides. It is effective for a small-scale data system, but may not scale to a large-scale data system. A create operation for a record does not need to access index store if the record does not have alternate key. An update operation for a record does not need to access index store if it does not add new alternate keys to the record or change any existing alternate key. An update operation for a record that removes an existing alternate key from the record or keeps the existing alternate keys intact does not need to access index store. Most of our applications do not change alternate keys of data records frequently. Once a record is created, its alternate keys are only occasionally changed.

Upon the unavailability of relevant index store shards, Uniqorn optionally maintains high availability, while sacrificing consistency temporarily, for create and update operations that add new alternate keys or change existing alternate keys. For such a create or update operation for a data record, we enhance Uniqorn to directly persist the data record into data store and marks it for "repairing" later after the relevant index store shards recover. A repair is a special update operation where a data record is forced to update itself with no actual changes just to populate its index records to index store. Each uniqueness violation that was undetected during the downtime of index store shards will be found and reported by repairs. Human intervention is needed for resolution. After the completion of all repairs and resolutions, the consistency of the system is restored.

At the client side, Uniqorn clients are stateless since the system state (i.e., data records, index records, and their embedded versions) is persisted in index store and data store. The crash and slowdown of a client does not block or slow down other clients, yet the client may be more susceptible to concurrency conflicts. At the server side, Uniqorn leverages the existing recovery mechanisms of the key-value store or relational data stores to recover. In Uniqorn, the only additional metadata, versions, are embedded into index and data records, which does not impose additional constraints on the split and merge of shards for load balancing and system maintenance.

3.3 Optimization

In our implementation, we also conservatively exploit the special or stronger supports from specific data store and index store. For example, relational databases support SQL updates with a rich where-clause. We exploit it to combine two consecutive delete and insert operations into one replace operation. We also totally bypass all of the phases for a create/update operation if a data record and all of its index records co-locate in the same shard, using a single native relational transaction to update both index record table and data record table. We also combine consecutive phases of an operation into one whenever possible. For example, delete and insert an index record can be combined into a replace if the underlying storage engine supports it.

We also exploit the opportunities using the partial ordering of versions $[g, v]$ to resolve concurrency conflicts faster. Most of our applications rarely create and delete data records in frequent succession. They usually create a data record and update it for many times before delete it. Since the generation number g of a data record is intact since its creation, we can just compare the v field to establish a priority among concurrent operations. If an operation detects a higher-priority is ongoing, it can abort itself and retry. For example, on one hand, if a create/update detects the version of an existing index record has a higher priority than the version of the index record it attempts to insert, it can abort itself immediately without invoking a GC. On the other hand, the create/update can directly replace the existing index record if it has a higher priority than the existing index record without resorting to a GC.

4 Evaluations

Uniqorn has been in production for about 2 years and has a broad adoption within our organization. We think it is mainly due to (1) Uniqorn allows applications to access a sharded data store at the same consistency and performance level by both primary keys and secondary keys. Applications can easily change their data models by adding or remove secondary keys into their data scheme, which was hard for a purely key-value store after deployment. (2) Uniqorn maintains the scalability of a purely key-value or document store yet with added functionality, and (3) Uniqorn sacrifices availability only in a tiny portion of most application workloads, which is deems acceptable. The performances of our customer application resemble our test-bed experiments (shown below), except some of them has a larger scale to above 100 servers.

Uniqorn lies in the performance space between a single monolithic data store with secondary indexes and a sharded data store system with LSI on each shard. On one hand, Uniqorn incurs longer latency than a single monolithic data store for the benefit of scalability, though retaining the consistent properties, for create, read, and update (except delete) operations due to additional roundtrips to index store. Moreover, Uniqorn may leave garbage index records as a side effect of maintaining consistency, which may degrade data store system performance. On the other hand, Uniqorn has an additional dependency on index store than a sharded data store system with LSI for the benefit of consistency, scalability, and usability. Comparing to both, Uniqorn suffers reduced write availability in the case of addition or update of alternate keys during the downtime of index store. Though Uniqorn aims to be consistent and scalable, we question ourselves what the costs are. We conduct experiments to show (1) how much more latency Uniqorn incurs to maintain consistency, (2) how Uniqorn performs with the increasing number of shards and alternate keys per record, (3) how Uniqorn performs in the presence of failures of index store, and (4) how garbage index records affect its performance.

All of the experiments in this section are run on a subset of servers in our private data center. The servers run the Linux operating system on x86 processors. All shards of index and data stores (clients, respectively) run on the servers of the same configuration and are located in the same data center. All of them are connected with high-speed networks. We have implemented and deployed Uniqorn in production with Couchbase, Oracle, MySQL, and MariaDB. We chose MySQL as the underlying data store technologies to run each index store and data store shards, each shard consists of a cluster of 5 servers replicated to each other. All MySQL data stores have the same settings.

We use a record whose size varies from 2Kbytes to 3Kbytes for our tests. A record can have 0 to 6 alternate keys. If not mentioned explicitly, every record in our experiments has 2 alternate keys. Each client runs 2x threads of the number of CPU cores. We maintain a global pool for primary keys and a global pool for each alternate key. The sizes of all the pools are the same. Each thread repeatedly chooses and then performs a random CRUD operation in a serial manner. For a create operation, a random primary key and a given number of random alternate keys are chosen from their respective pools to make a new record. For a update operation, a record of a primary key (randomly chosen from the primary key pool) is read out from data store directly, then all of its alternate keys are updated randomly from their respective pools as well as its value. Both read and delete (by alternate key) operations choose an alternate key from their respective alternate key pools. If an exception (e.g., uniqueness violation) occurs, it is ignored and a thread moves to next operation. We choose this work load specifically to emulate a potentially application workload worse than our actual production deployments.

During all of our experiments, we keep the ratio of the number of index store shards to the number of data store shards as 1:1. Each data store shard has the approximately same number of records during all experiments, so does each index store shard. As the number of data store shards increases, the total size of the data system increases linearly to emulate a scaling data system. To emulate a linearly scaling application, we keep the ratio of the number of data store shards to the number of clients as a constant factor during all experiments. As a consequence, the application scales linearly with the data system.

We implemented Uniqorn clients on a layered software architecture, which sacrifices efficiency (e.g., latency and resource consumption, etc.) for development velocity. We scale the system throughput by adding more hardware capacity, which falls in the targeting scope of the design of Uniqorn.

4.1 Performance Cost

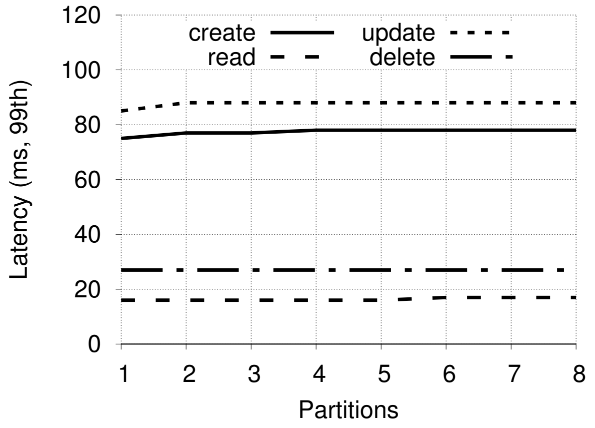

Figure 4: Latency of CRUD operations as data store, index store, and clients scale linearly.

We compare Uniqorn to a single-sharded (Monolithic) data system, where the sole data store has all records and a LSI for each alternate key. For Uniqorn, we co-locate index store and data store on the same server, where the data store does not have any LSI. That is to say, both Uniqorn and the monolithic data system have the same amount of data and resource, and take the same workload from application. We measure the latency of CRUD operations at 99 percentile (Table 1), counting in aborted operations (e.g., due to consistency violations). The latency of create and update operations that add new alternate keys or change existing alternate keys in Uniqorn is approximately 3 times of those in Monolithic due to (1) additional round trips to index record to persist alternate keys and (2) another additional round trip to validate and change the version of a garbage index record in the case of reusing it. However, if a create or update operation does not have alternate key addition/update, no index store access is needed and so the latency is on par with that in

Monolithic. Uniqorn incurs an additional read of an index record (though the size of an index record is usually much smaller than the data record that it refers to) than Monolithic for every read and delete (by alternate key) operation, resulting a constant additional latency.

Table 1: Latency (ms, 99th)

Operation	Monolithic	Uniqorn (1-shard)
create with 2 AKs	25	75
create with no AKs	16	16
read by AK	12	16
update with 2 changed AKs	30	85
update without AK changes	25	25
delete by AK	22	27

4.2 Scalability

To emulate a scaling system, we increase both index store and index store shards linearly (while keeping their ratios to 1:1) and increase the application load by increasing the number of clients linearly (while keeping the ratio of the number of data store shards to the number of clients as a constant factor). We measure the latency of CRUD operations in 99 percentile to observe how the system scales (Figure 4). The latency of all operations are approximately equal to their latency in Table 1 with a single shard. We observe the CPU, network, and disk usage at all data store and index store servers and the clients keep constant as the number of shards increases. The increased system load is evenly distributed across all clients and servers.

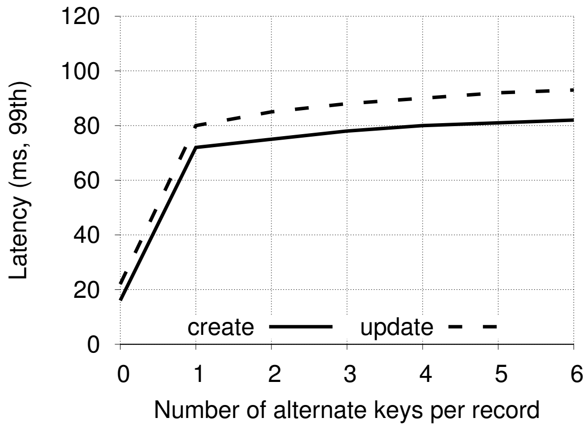

Figure 5: Latency of create/update operations as # of alternate keys per record increases.

We also measure the latency in 99 percentile when application increases the number of alternate keys per record (Figure 5) in the setting of 8 shards. We omit the latency of read and delete (by alternate key) since they are immune to the increase of alternate keys per record. When a record to be created does not have any alternate key or a record to be updated does not add new alternate keys or change existing alternate keys, access to either index store or LSI in data store is unneeded. The resulting latency is the shortest. As the number of alternate keys per record increases, Uniqorn performs indexing phases in parallel for each alternate key, resulting nearly constant latency. The slight increases of latency is contributed by the stragglers.

4.3 Availability

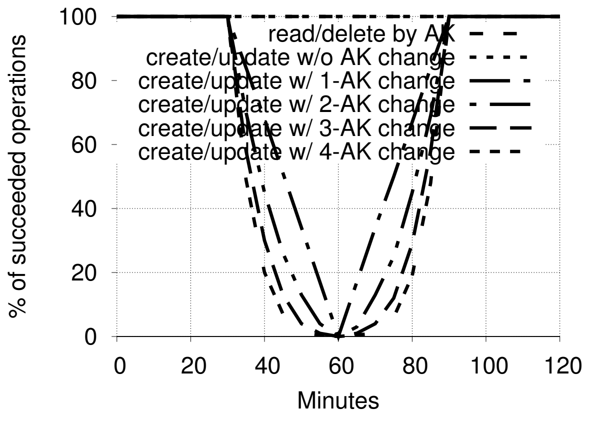

Figure 6: Operation availability during index store outage and recovery time window.

We run the workload in the setting of 6 shards while bring down one index store shard every 5 minutes to emulate an outage and then bring them back one by one every 5 minutes to emulate a rolling recovery. We do not enable "repairs" (Section 3.2) for the improved availability at the cost of temporarily reduced consistency. We measure the ratio of succeeded CRUD operations among all issued operations (Figure 6). As an index store shard goes down, all create operations with at least one of its alternate key on this shard will fail; all update operations with at least one of its changed or newly added alternate key on this shard will fail. However, if an update operation deletes an alternate key on this shard, the update operation will succeed while leaving the index record of the alternate key as garbage. Read and delete (by alternate key) operations still proceed since they scatter/gather across all data store shards to look up for a given alternate key when the corresponding index store shard is down. However it incurs prolonged latency and additional resource cost due to the scatter/gather. As more index store shards are down, more create/update operations will fail. As a create/update operation has more alternate key changes, it has more chance to hit a down index store shard and more chance to fail. In both cases, the total operation availability dips.

4.4 Effect of Garbage Index Records

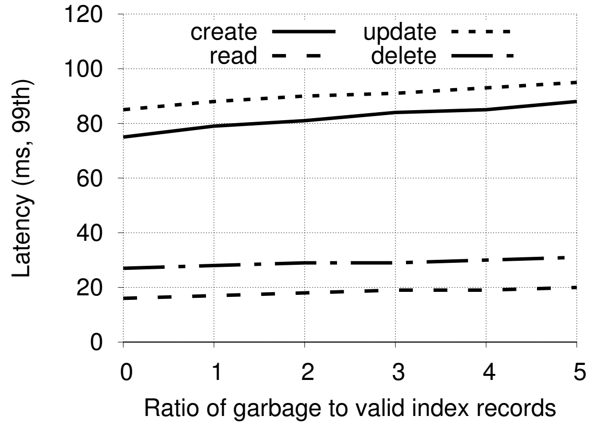

Figure 7: The effect of garbage index records on latency.

We run the workload in the setting of 8 shards while turning off the garbage cleanup of index record. We measure the ratio of garbage index records to valid index records in index store non-intrusively and periodically. Accumulated garbage index records will increase the probability of reusing them by a create/update operation. However, a create/update operation will clean them up in parallel (with additional round trips to data store to validate and change its version for clean up) if it needs to reuse them, avoiding accumulated latency. A read/delete (by alternate key) operation does not incur extra round trips to either index store and data store regardless of accumulated garbage index records. Due to the underlying data store technologies, the data system can tolerate high ratio of garbage index records with negligible latency increase (Figure 7).

5 Related Work

Concurrency control methods for generic distributed transactions in a distributed data store system [3, 6], such as two-phase indexing and timestamp-ordering, can be applied to manage GSI with uniqueness constraint. Unlike Uniqorn, both two-phase indexing and timestamp-ordering methods assume participating data managers (DMs) to support two-phase commit [20]. They employ a pre-write phase to each DM before finally committing a transaction. Due to the nature of two-phase commit, those systems are blocking and need complicated, error-prone recovery mechanisms upon failures [20]. Uniqorn, however, addresses a specific problem (i.e., GSI) and therefore does not necessarily need the full machinery of generic distributed transactions. It is non-blocking, does not support transaction rollbacks (yet may result in garbage records due to failures as a side-effect), recovers from failure simply by deleting garbage records if necessary. Unlike timestamp-ordering methods [3, 21] in general, Uniqorn does not depend on an external timestamp oracle to generate globally monotonic timestamps among all transactions. Unlike multi-version timestamp-ordering methods in particular [3], Uniqorn persists a single copy of a record (though it embeds a "version" mark into the record) at any time. Industry implementations (such as Java Enterprise Edition [19]) of general distributed transactions assume that participating resources (called X/Open XA resources) support pre-write operations and therefore implicitly use two-phase commit protocol.

Conceptually Uniqorn uses an optimistic concurrency control method to provide GSI transactionally, similar to the optimistic concurrency control method in [15, 27, 1]. However, unlike [15], Uniqorn's optimistic concurrency control method is specially designed for a sharded data store system, instead of solely for a single data store. In Uniqorn, the versions are partially ordered, yet the transaction timestamps are totally ordered in [15]. In [27], a distributed optimistic concurrency control method followed by indexing (indexing is an integral part of distributed validation and two-phase commit) is proposed to reduce data contention and increase throughput in a distributed data store. However, Uniqorn is purely an optimistic concurrency control method from application programmers' perspective. In [1], records are cached and manipulated at client machines while persistent storage and transactional support are provided by servers using loosely synchronized clocks to achieve global serialization. It stores only a single version of each record (like Uniqorn) and tracks recent invalidations on a per-client basis for concurrency control. Uniqorn also relies on servers for persistent storage and transactional support, but it does not resort to clock synchronization for linearibility and maintain per-record concurrency control information ("version") on an embedded per-record basis.

Traditional monolithic relational data store systems like Oracle, MySQL, SQL Server, and PostgreSQL, etc. perform secondary indexing on internal indexing data structures (e.g., B-tree) in the same transaction as the record modification (e.g., [8]). Uniqorn achieves the same semantics by building GSI on top of over-the-shelf data stores without using their support of LSI.

Key-value or document data store systems, like Cassandra [16], Couchbase [11, 17], and DynamoDB [26], etc. shard data across multiple servers/nodes and usually do not support distributed transactions. They either do not support GSI or support with relaxed consistencies. Cassandra [7] and MegaStore [2] store index data along with original data on the same shard (therefore only supports LSI). DynamoDB [24] and Couchbase [24] support GSI with eventual consistency, that is, indexes are updated in a best-effort manner. Uniqorn supports GSI transactionally and peculiarly enforces global uniqueness among all secondary keys.

Percolator [21] implements snapshot isolation by extending multi-version timestamp ordering [3] across a distributed system using two-phase commit. It builds on the single-row transactions of BigTable [5] to provide multi-row, distributed transactions. Uniqorn relies on conditional write support of the underlying data systems, weaker than the single-row transaction support from BigTable, to provide GSI with uniqueness constraint. Moreover, Uniqorn does not need multi-version supports from underlying data systems like BigTable.

Similar to SLIK [13], Uniqorn uses clients to mask temporary internal inconsistencies to provide applications with a consistent view of data and uses records as source-of-truth to determine validity of index data. However, the concurrency control in SLIK lies in each server where the record is persisted, a centralized “synchronization point” for all write operations pertaining to the record. In SLIK, a server pessimistically locks the record being written in a blocking manner, synchronously writes secondary keys , and then is persisted the record. In contrast, the concurrency control in Uniqorn lies in distributed Uniqorn clients in a non-blocking manner without any enhancement from server side and supports the constraint of global unique secondary key as well.

6 Conclusion and Future Works

We have built Uniqorn to show that large-scale sharded data store systems need not forgo the benefits of GSI, even with strict uniqueness constraint. The system achieved the goals we set for scaling the system with nearly constant read and write latency, nearly linearly-increasing throughput, and strong consistency supports of all CRUD operations of individual records, at the cost of either slightly reduced write availability or temporarily reduced consistency only upon alternate key additions or changes. We are interested in knowing how far we can go with the global optimistic indexing mechanism to tame concurrency anomalies to provide transactional multi-record CRUD operations or generic GSI, and what latency and garbage collection overhead will incur. We are also keen to know the performance and adaptations after porting our current implementation to other data store technologies.

References

[1] A. Adya, R. Gruber, B. Liskov, and U. Maheshwari. Efficient optimistic concurrency control using loosely synchronized clocks. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, SIGMOD ’95, pages 23–34, New York, NY, USA, 1995. ACM.

[2] J. Baker, C. Bond, J. C. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *Proceedings of the Conference on Innovative Data system Research (CIDR)*, pages 223–234, 2011.

[3] P. A. Bernstein and N. Goodman. Concurrency control in distributed database systems. *ACM Comput. Surv.*, 13(2):185–221, June 1981.

[4] S. Ceri, M. Negri, and G. Pelagatti. Horizontal data partitioning in database design. In *Proceedings of the 1982 ACM SIGMOD International Conference on Management of Data*, SIGMOD ’82, pages 128–136, New York, NY, USA, 1982. ACM.

[5] 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. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, OSDI ’06, pages 15–15, Berkeley, CA, USA, 2006. USENIX Association.

[6] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google’s globally-distributed database. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI’12,

pages 251–264, Berkeley, CA, USA, 2012. USENIX Association.

[7] D. Enterprise. Cassandra native secondary index. https://www.datastax.com/dev/blog/cassandra-native-secondary-index-deep-dive, 2018.

[8] G. Graefe. Modern b-tree techniques. *Found. Trends databases*, 3(4):203–402, Apr. 2011.

[9] P. Helland. Life beyond distributed transactions. *Queue*, 14(5):70:69–70:98, Oct. 2016.

[10] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990.

[11] C. Inc. Global secondary indexes. https://developer.couchbase.com/documentation/server/current/architecture/global-secondary-indexes.html, 2018.

[12] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. C. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. J. Abadi. H-store: A high-performance, distributed main memory transaction processing system. *Proc. VLDB Endow.*, 1(2):1496–1499, Aug. 2008.

[13] A. Kejriwal, A. Gopalan, A. Gupta, Z. Jia, S. Yang, and J. Ousterhout. SLIK: Scalable low-latency indexes for a key-value store. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 57–70, Denver, CO, 2016. USENIX Association.

[14] D. Kuhn, S. Alapati, and B. Padfield. *Expert Indexing in Oracle Database 11G: Maximum Performance for Your Database*. Apress, Berkely, CA, USA, 1st edition, 2011.

[15] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Trans. Database Syst.*, 6(2):213–226, June 1981.

[16] A. Lakshman and P. Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, Apr. 2010.

[17] S. Lakshman, S. Melkote, J. Liang, and R. Mayuram. Nitro: A fast, scalable in-memory storage engine for nosql global secondary index. *Proc. VLDB Endow.*, 9(13):1413–1424, Sept. 2016.

[18] L. Lamport. Tla+ toolkit. https://lamport.azurewebsites.net/tla/tla.html, 2018.

[19] I. S. Microsystems. Java transaction api, version 1.0.1 (jta specification). http://java.sun.com/products/jts, 02 2016.

[20] C. Mohan, B. Lindsay, and R. Obermarck. Transaction management in the r* distributed database management system. *ACM Trans. Database Syst.*, 11(4):378–396, Dec. 1986.

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

[22] L. Qiao, K. Surlaker, S. Das, T. Quiggle, B. Schulman, B. Ghosh, A. Curtis, O. Seeliger, Z. Zhang, A. Auradar, C. Beaver, G. Brandt, M. Gandhi, K. Gopalakrishna, W. Ip, S. Jgadish, S. Lu, A. Pachev, A. Ramesh, A. Sebastian, R. Shanbhag, S. Subramaniam, Y. Sun, S. Topiwala, C. Tran, J. Westerman, and D. Zhang. On brewing fresh espresso: Linkedin's distributed data serving platform. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 1135–1146, New York, NY, USA, 2013. ACM.

[23] A. W. Service. Aws documentation – amazon simpledb developer guide. https://docs.aws.amazon.com/AmazonSimpleDB/latest/DeveloperGuide/ConditionalPut.html, 2018.

[24] A. W. Service. Global secondary indexes, amazon dynamodb developer guide. https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/GSI.html, 2018.

[25] J. Shute, R. Vingralek, B. Samwel, B. Handy, C. Whipkey, E. Rollins, M. Oancea, K. Littleeld, D. Menestrina, S. Ellner, J. Cieslewicz, I. Rae, T. Stancescu, and H. Apte. F1: A distributed sql database that scales. In *VLDB*, 2013.

[26] S. Sivasubramanian. Amazon dynamodb: A seamlessly scalable non-relational database service. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 729–730, New York, NY, USA, 2012. ACM.

[27] A. Thomasian. Distributed optimistic concurrency control methods for high-performance transaction processing. *IEEE Transactions on Knowledge and Data Engineering*, 10(1):173–189, Jan 1998.

[28] J. Yi. The specification and model checking of uniqorn's global secondary index uniquness guarantee in tla+. https://github.com/uniqorn-tla, 2018.