# Uniqorn: Global Secondary Indexes with Uniqueness Constraint for Distributed Databases

Jun Yi, William Eschenbruecher, Jason Sardina, Nitin Chhabra, Annie Lu, Ying Zhang, Alexei Olkhovskii, Fred Goya, Bruce Woods, Scott Harvester, Bob Lowell
Walmart Labs
{jyi, weschenbruecher, jsardina, nchhabra, ylu, yzhang3, aolkhovskii, fred.goya, bwoods, sharvester, blowell}@walmartlabs.com

## ABSTRACT

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

We built Uniqorn, a system that supports (with global secondary indexes) transactional create, read, update, and delete operations of individual records with secondary keys. Using global optimistic locking, an extension of optimistic concurrency control to distributed databases, Uniqorn tames system unreliability and operation concurrency to prevent anomalies. It is lightweight, non-blocking, scalable, applicable to most out-of-box relational or key-value database systems. Our experiments and deployments demonstrate that large-scale partitioned database systems need not forgo the benefits of global secondary indexes even with strict uniqueness constraint.

## 1. INTRODUCTION

Due to the enormous growth in transaction volume of business databases, a single monolithic database server can not scale up to meet the sheer needs of large-scale business web applications. Partitioning a logical table across multiple physical databases by primary keys (e.g., by hashing the primary key into a specific database [3]) has been a scalable and effective way to meet the growing needs. In such a partitioned database, applications can rapidly create, read, update, and delete (CRUD) records by primary keys, however, many business-critical applications also need to rapidly access records by one or multiple secondary keys. For example, a customer account application may need to access customer records by their (secondary key) email addresses or phone numbers instead of (primary key) user ids. Some applications may need to guarantee that no two 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.

Global secondary indexes can provide fast access to records in a partitioned database by alternate keys. However, unlike a single monolithic database, records and their indexes are distributed across different partitions, leading to potential and inherent inconsistency between records and their indexes. System unreliability in a distributed database system and operation concurrency from a distributed application complicate the building and maintenance of global secondary indexes. If not handled appropriately, consistency anomalies, like missing or mismatched secondary keys in global secondary indexes, may occur. Due to those consistency anomalies, many large-scale data systems sacrifice secondary indexes [4], only support local secondary indexes [7, 11, 9, 19], or support global secondary indexes with reduced consistency in favor of scalability or performance [8, 14, 21]. The lack of full support of global secondary indexes limits the ease and efficiency of application development on such systems. To our best knowledge, we are not aware that any existing work supports global secondary indexing transactionally with uniqueness constraint.

Applications could build their own solutions to prevent or avoid anomalies. However, this will limit the productivity of individual application development and will not scale well across a medium-sized or large-sized organization. In this paper, we built Uniqorn, a system that supports (with global secondary indexes) applications with transactional CRUD operations for individual records with secondary keys. Using global optimistic locking, an extension of optimistic concurrency control to distributed databases, Uniqorn tames system unreliability and operation concurrency to prevent anomalies. It is lightweight, non-blocking, scalable, and applicable to most out-of-box relational or key-value database systems. It has nearly constant latency and linearly increased throughput for CRUD operations with secondary

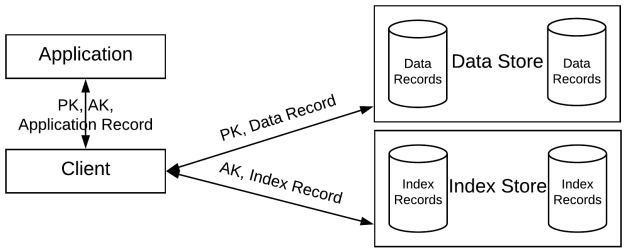



Figure 1: High-level view of Uniqorn

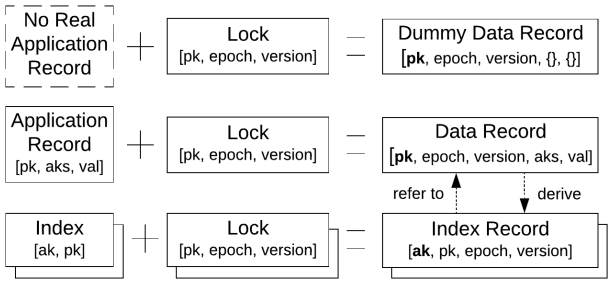



Figure 2: A record and its internal representation in Uniqorn

keys as the system scales out. It is highly available, without sacrificing consistency, for read and delete operations by secondary keys and for create and update operations without secondary key changes. It maintains high availability during the time of failures, while temporarily sacrificing consistency, for create and update operations with secondary key changes. Upon recovery, it restores consistency automatically. Our experiments and production deployment demonstrate that large-scale partitioned database systems need not forgo the benefits of global secondary indexes even with strict uniqueness constraint.

The remainder of this paper is organized as follows. Sections 2 describes the system architecture, data model, APIs, the concurrency control, the operations, and discussions of implementation issues. Section 3 presents performance evaluations. Sections 4 presents related work and Section 5 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. We require both data store and index store to provide:

- up-to-date read of a single record by a given primary key (i.e., a read returns the last persisted record of the given primary key) and
- optimistic write of a record in a compare-and-swap manner, conditional on a given column/attribute value of the record (i.e., an insert succeeds only if a record of the given primary key has not been persisted, an update/delete succeeds only if the specific column/attribute is equal to a given value).

Key-value stores, e.g., Cassandra [13], SimpleDB [20], and Couchbase [8], and all relational databases satisfy these two requirements. Both index store and data store can be centralized or partitioned respectively by alternate keys or primary keys. 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 key-value) and form a hybrid system. No special features or software need to be installed on either one. Clients access both data store and index store (via either specific drivers or remote procedure calls) and implements the CRUD operations (Section 2.2) 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.

## 2.1 Data Model

From an application programmer's perspective, a table consists of *application records* (or just *records*). Internally, we represent a record by a *data record* and a set of *index records* (Figure 2). Every data record and every index record is embedded with a *global optimistic lock* (or just *lock*) for concurrency control (Section 2.3). A degenerated form of data record with only an embedded lock, called *dummy data record*, also exists for concurrency control.

- **Application Record (Record)**: It is in the tuple [$pk$, $aks$, $val$], where $pk$ serves as both the primary key and the partition key of a record, $aks$ is the set of alternate keys of the 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 stores).
- **Global Optimistic Lock (Lock)**: It is in the tuple [$pk$, $epoch$, $version$] (or shortly [$pk$, $e$, $v$]), where $pk$ serves as both the primary key and the partition key of a record, $e$ is the generation identifier of the record among all records that have ever associated with $pk$, $v$ is increased upon every update of the record (starting from 0) within a generation.
- **Data Record**: It is augmented from a record [$pk$, $aks$, $val$] with a lock [$pk$, $e$, $v$] embedded to form the tuple [$pk$, $e$, $v$, $aks$, $val$].
- **Index Record**: For each $ak$ in its alternate keys $aks$ of a record [$pk$, $aks$, $val$], an index record is formed with an embedded lock [$pk$, $e$, $v$] into the tuple [$ak$, $pk$, $e$, $v$]. An index record [$ak$, $pk_j$, $e$, $v$] is said to *refer to* a data record [$pk_i$, $e$, $v$, $aks$, $val$] or record [$pk_i$, $aks$, $val$] if $pk_i = pk_j$. Equivalently, a data record [$pk_i$, $e$, $v$, $aks$, $val$] or record [$pk_i$, $aks$, $val$] is said to be *referred by* an index record [$ak$, $pk_j$, $e$, $v$] if $pk_i = pk_j$. A data record [$pk_i$, $e$, $v$, $aks$, $val$] or a record [$pk_i$, $aks$, $val$] is said to *derive* an index record [$ak$, $pk_j$, $e$, $v$] if $ak \in aks$ and $pk_i = pk_j$.
- **Dummy Data Record**: It is a degenerated form of data record that embodies a lock [$pk$, $e$, $v$] in the tuple

Table 1: Operation APIs (operations by primary keys and non-transactional bulk operations are omitted)

| Operation | Behavior | Phases | Section |
|---|---|---|---|
| create(DataRecord *dr*) | return inserted record; fail upon unavailability of data store and/or index store, violation of uniqueness constraint, existence, or victim of concurrency conflicts | initialize dummy data record, indexing (if *dr* has alternate keys), persist data record | 2.4.1 |
| read(String *ak*) | return the latest persisted record (empty if absent) that has alternate key *ak*; fail upon unavailability of index store or data store | read index record, validate | 2.4.2 |
| update(DataRecord *dr*) | return persisted record; fail upon unavailability of data store and/or index store, violation of uniqueness constraint, absence, or victim of concurrency conflicts | read data record, indexing (if new alternate keys are added to *dr* or existing alternate keys of *dr* are changed), persist data record | 2.4.1 |
| delete(String *ak*) | return true if the record that has *ak* is removed, otherwise false; fail upon unavailability of data store or victim of concurrency conflicts | read index record, validate, delete data record | 2.4.3 |

[*pk*, *e*, *v*, ∅, ∅], which is used simply for concurrency control while no real record of *pk* is persisted.

Data records are horizontally partitioned (e.g., range partitioned or hashing partitioned as in [3]) by their primary keys into data store partitions. Likewise, index records are similarly partitioned in index store partitions by their alternate keys. Both data records and index records are persisted into a regular data record table and a regular index record table, respectively, in either underlying relational or key-value partitions with no additional constraint. Conceptually, clients and applications communicate by records, instead of data records. However, in our implementation, clients and applications directly pass back and forth data records with the convention that applications treat the tuple [*e*, *v*] as opaque and never modify it. In this paper, we choose to embed locks into index records and data records for the convenience of presentation. However an implementation can split locks off and store them in a standalone regular lock table.

## 2.2 APIs

By calling APIs (Table 1) implemented by Uniqorn clients, applications can create, read, update, and delete individual records by either primary keys or alternate keys transactionally. Temporary inconsistencies between index records and data records do exist internally, but clients mask them to provide applications with a consistent view of records. Specifically, the modification of a data record and all of its index records must appear atomic, consistent, isolated, and durable to applications. Moreover, if a record with an alternate key is created or updated successfully, no subsequent operations can create another record with the same alternate key until another operation removes this alternate key from the record, updates this alternate key of the record to another value, or deletes the record. A read by primary key or a delete by primary key can directly access the data store partition determined by the primary key. For this reason, we will skip both in this paper.

All APIs 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. 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.

## 2.3 Concurrency Control

Uniqorn supports full spectrum of concurrency among all CRUD operations (Section 2.2) of the same/different primary keys with same/different alternate keys and/or attributes, while warranting global uniqueness among alternate keys in the face of system failures. Due to system failures and operation concurrency, various anomalies may happen. If not handled appropriately, data consistency will be breached and/or system performance will be degraded. In this section, we will (1) first list all possible anomalies in a Uniqorn system and reduce them to a single fundamental anomaly analytically, (2) then introduce global optimistic locks and their properties, and (3) at last show how Uniqorn tames this anomaly to maintain uniqueness using global optimistic locks.

### *2.3.1 Anomalies*

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

- **duplicated**: more than one index records of the same alternate key are persisted in index store.
- **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.
- **valid**: the data record referred by the index record is persisted in data store and has 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.

A record may have multiple alternate keys and their index records may be distributed across multiple index store

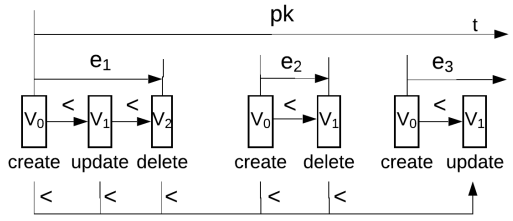



Figure 3: Concept of global optimistic locks and the partial ordering at current time among all previously persisted locks of primary key $pk$ and the currently persisted lock [$pk$, $e_3$, $v_1$].

partitions. Reading or writing alternate keys across multiple partitions without transactions can not be performed consistently. Therefore, index records can not be taken as the source-of-truth of records. Instead, Uniqorn treats *the data record of a record as its* ***source-of-truth*** *for alternate keys, lock, and attributes.*

If we could assume missing index records never occur, we can solve for all other anomalies:

- *at most one persisted index record per alternate key*: Since alternate keys serve as both the partition keys and the primary keys of the logical index record table, 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.
- *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 index record $ir$ of an alternate key $ak$ is not persisted, no data record that has $ak$ is persisted; if the data record $dr$ of primary key $ir.pk$ is persisted and $ak \in dr.aks$, then $ir$ is valid and $dr$ can be returned to applications, otherwise, $ir$ 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 can reduce all anomalies to a single anomaly of missing index records and exclude read and delete operations as anomaly sources.* That is to say, Uniqorn can prevent all of the aforementioned anomalies with a single guarantee:

- **G1**: every create or update operation does not leave any missing index records.

In the following, we will first introduce global optimistic locks and then show how to use them to meet **G1**.

### 2.3.2 Global Optimistic Locks

Uniqorn adopts an optimistic concurrency control method using *non-blocking*, *partially ordered* global optimistic locks (or just locks). Locks are globally and temporally unique (Figure 3). A lock is formed in tuple [$pk$, $epoch$, $version$] (or shortly [$pk$, $e$, $v$]) and embedded in every index record and data record, where $pk$ is both the primary key and the partition key of the data record, $e$ 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. In our implementation, a client generates an epoch in the form ($ts$, $clientId$) for each newly created data record, where $ts$ is a timestamp reading from a logically forward-moving clock at the client and $clientId$ is the unique identifier of the client among all clients. We also utilize the informative elements in each epoch to diagnose production issues at runtime. A data record of primary key $pk$ may not currently be persisted in data store (e.g., it has not been inserted or was deleted), so no lock is currently associated with it. We designate this lock as the *null lock* of $pk$ in the tuple [$pk$, $\emptyset$, $\emptyset$], which means the data record of $pk$ is not persisted currently (therefore all index records referring to $pk$ are garbage).

#### 2.3.2.1 Lock Access.

Locks are accessed with the following rules:

- *lock read*: When a client reads a data record, it reads its embedded lock too.
- *lock create*: A client chooses a new epoch $e = (ts, clientId)$ to form an embedded lock [$pk$, $e$, 0] in a data record, and then inserts the data record into data store.
- *lock initialize*: A client chooses a new epoch $e = (ts, clientId)$ to form a dummy data record of $pk$ as [$pk$, $e$, 0, $\emptyset$, $\emptyset$], and then inserts it into data store.
- *lock update*: When a data record is updated by a client, the version of the lock will be increased by 1 from the version of the lock last perceived by the client, while keeping the same epoch. A client always updates a lock optimistically in a compare-and-swap manner. A client can successfully update a lock only if the persisted lock at the time of lock update matches the lock last perceived by it.
- *lock delete*: When a data record is deleted, its embedded lock is deleted too. After its deletion, a new data record with the same primary key can be recreated.
- *lock uninitialize*: When a dummy data record is deleted, its embedded lock is uninitialized.

We call all lock accesses except lock read collectively as "lock change". The rules ensure that *every data record change will change its embedded lock.*

#### 2.3.2.2 Lock Priority.

We define a partial ordering/priority, perceived by a client, among locks to facilitate concurrency control. Suppose that two locks $l_i = [pk_i, e_i, v_i]$ and $l_j = [pk_j, e_j, v_j]$, their partial ordering, perceived by a client, is defined as follows in precedence (highest precedence first):

1. $l_i = l_j$: $pk_i = pk_j$ and $e_i = e_j$ and $v_i = v_j$.
2. $l_i < l_j$: $pk_i = pk_j$ and $e_i = e_j$ and $v_i < v_j$.
3. $l_i < l_j$: $pk_i = pk_j$ and $e_j$ is current, where $e_j$ is current if (1) it is the epoch of the currently persisted data record of $pk_j$ or (2) no data record of $pk_j$ is currently persisted and $e_j = \emptyset$.

4. $l_i \not\leq l_j$ and $l_j \not\leq l_i$: otherwise.

From the rules, we can see that the lock embedded in a currently persisted data record of primary key $pk$ or the null lock of $pk$ otherwise has the highest priority among all locks that have previously associated with $pk$. Two locks $l_i$ and $l_j$ associated with different primary keys are not comparable (i.e., $l_i \not\leq l_j$ and $l_j \not\leq l_i$). We do not keep multiple versions (with different locks) of a data record in data store or at each client. As time goes, the relative priority of two locks $l_i$ and $l_j$ associated with the same primary key, perceived by a client, may change from comparable (say $l_i < l_j$) to incomparable ($l_i \not\leq l_j$ and $l_j \not\leq l_i$). However, their relative priority will never be reversed (i.e., $l_i < l_j$ changes to $l_i > l_j$). Moreover, no two clients will perceive a non-compatible relative priority, i.e., one perceives $l_i > l_j$ and the other perceives $l_i < l_j$. However, two clients may perceive different but compatible relative priority between two locks (e.g, one perceives $l_i > l_j$ and the other perceives $l_i \not\leq l_j$ and $l_j \not\leq l_i$). Clients may have different but compatible partial orderings among a set of locks. Moreover, the partial orderings among all the locks associated with the same primary key, perceived by all clients, are compatible to one another and can be linearized to their actual total ordering in real time. The partial ordering defined above is weak enough to allow more concurrency and save storage and memory space, but strong enough to just capture possible concurrency conflicts.

Uniqorn could have constructed a lock in the tuple [$pk$, $ts$, $clientId$] and then compare the priorities of two locks of the same primary key by lexicographically comparing their $ts$ and $clientId$. However, Uniqorn avoids this form of lock 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. Locks can be simplified to the tuple [$pk$, $v$] if primary keys are never reused by applications, and their partial ordering can be simplified in a straightforward manner.

### 2.3.3 The Anomaly Prevention

In this section, we will first show a necessary principle **P1** that addresses the issue of *window of missing index records* to meet **G1** (Section 2.3.1). However **P1** alone is insufficient to meet **G1** due to two additional issues: *lost changes of alternate keys* and *unsafe cleanup of garbage index records*. Then we show the other two supplemental principles **P2** and **P3** that use locks to address the two issues respectively. **P1**, **P2**, and **P3** together meet **G1** sufficiently.

#### 2.3.3.1 Window of Missing Index Records.

Since a data record of a record is taken as the **source-of-truth** of the alternate keys of the record (Section 2.3.1), we must persist the derived index records (if they have not) into index store before persisting the data record into data store. As a counter-example, suppose that a create operation first persists a data record and then persists its derived 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. Moreover, these derived index records may be missing indefinitely in the presence of index store failures. We state this necessary principle as follow:

- **P1**: every create or update operation persists all derived index records of a data record, if they have not persisted, before persisting the data record.

However, **P1** alone is insufficient to meet **G1**. It has two issues: *lost changes of alternate keys* and *unsafe cleanup of garbage index records*.

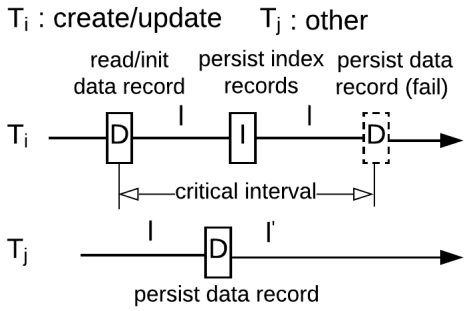



Figure 4: A create/update operation $T_i$ aborts to avoid lost changes of alternate keys since another operation $T_j$ has modified the data record (possibly changed its alternate keys) and hence changed lock $l$ held by $T_i$ to $l'$ during the critical interval. A box with D or I inside means an access (with specific action noted nearby) to data store or index store, respectively. Actions occur from left to right in time with the lock on top of the lines last perceived by $T_i$ and $T_j$ respectively. If no operation had changed lock $l$ during the critical interval, $T_i$ would continue to both persist the data record with an updated lock into data store atomically.

#### 2.3.3.2 Lost Changes of Alternate Keys.

If only following **P1**, Uniqorn may not capture all changes of alternate keys of a persisted data record in the first place, so the index records of those changed alternate keys may not be persisted in index store. Suppose that a data record with an alternate key is persisted into data store and the derived index record is persisted in index store. A client $T_i$ reads the data record, intending to only update its attributes. Immediately after, client $T_j$ removes the alternate key from the data record and then persists only the data record into data store, leaving its index record as garbage for cleanup. Afterwards, the garbage index record is cleaned up asynchronously in best effort. However, $T_i$ is unaware of the alternate key change made by $T_j$ (in other words, the alternate key change is lost). It updates the attributes of the data record only and then directly persists the data record into data store with the illusion that the index record were still persisted in index store, resulting missing index record.

To avoid lost changes of alternate keys (Figure 4), a Uniqorn client follows the following three phases:

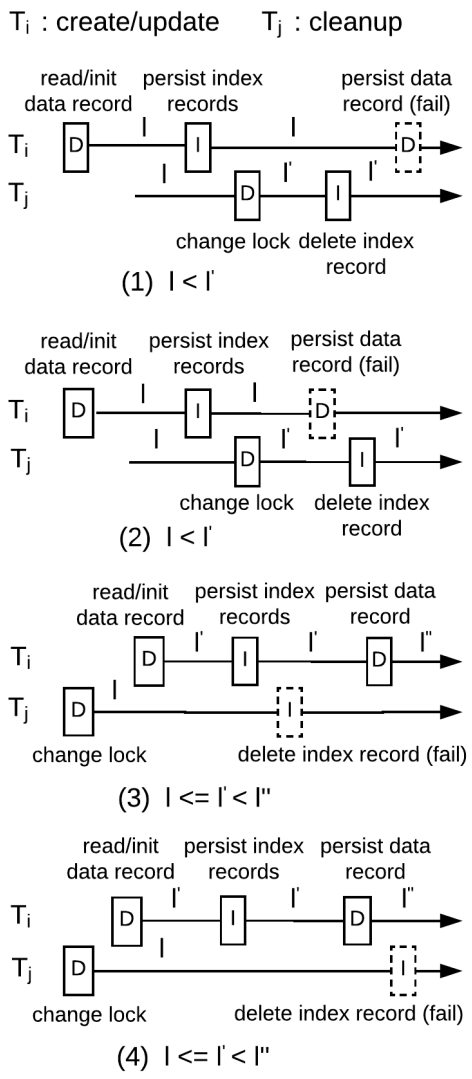



Figure 5: Possible interleavings of a create/update operation $T_i$ that attempts to persist a derived index record $ir$ of data record $dr$ and a garbage cleanup operation $T_j$ that attempts to delete it. A box with D or I inside means an access (with specific action noted nearby) to data store or index store, respectively. Actions occur from left to right in time with the lock on top of the lines last perceived by $T_i$ and $T_j$, respectively. In (1) and (2), $T_i$ fails to persist $dr$ since it does not hold lock $l$. $T_j$ succeeds in deleting $ir$ left by $T_i$ since it changes lock from $l$ to $l'$ ($l' > l$). In (3) and (4), $T_j$ fails to delete $ir$ persisted by $T_i$ (where $ir.lock = l'$) since $l \leq l'$ (in the case of reading $dr$, $l = l'$; in the case of initializing a dummy data record $dr$, $l < l'$). $T_i$ then continues to both persist $dr$ with lock $l'$ changed to $l''$ atomically.

1. *read/initialize data record.* A Uniqorn client first reads out the data record to be updated/created if it is persisted or otherwise initializes (i.e., creates and then persists) a dummy data record with a new lock. The lock embedded in the data record just read or initialized is taken as the lock of the create/update operation.

2. *persist index records.* Then the set difference, called "upsert", is calculated by subtracting the set of alternate keys of the data record that read or initialized in the preceding phase from the set of alternate keys of the data record to be created/updated.

   - If the upsert is empty, this phase can be skipped. For example, if a data record to be created does not have alternate key, the upsert will be empty. If no new alternate keys are added to a data record or no existing alternate keys of a data record are changed, the upsert will be empty.
   - If the upsert is not empty, following **P1**, the index record for each alternate key in upsert, *embedded with the lock of the create/update operation*, must be persisted into index store.

   The opposite of the upsert is the "residual", which is calculated by subtracting the set of alternate keys of the data record to be created/updated from the set of alternate keys of the data record that read or initialized in the preceding phase.

   - *No index record for any alternate key in the residual is deleted in this phase.* If such an index record is deleted and later the data record to be updated fails to be persisted into data store, it turns from a valid index record to a missing index record.

3. *persist data record.* At last, the create/update operation persists the data record with an incremented version of its lock into data store only if it still "holds" the lock (i.e., no other operation has changed the lock since it last accessed the data record and hence its embedded lock). This time interval is called "critical interval". If another operation (e.g., any mix of update/delete/recreate operations) has changed the lock of the data record during the critical interval (any change to the data record, including alternate key changes, will change the lock, see Section 2.3.2.1), this create/update operation will fail to persist the data record into data store. Since this operation never deletes any valid index records before this phase, the failure to persist into data store will not leave missing index records, though may leave garbage index records.

In addition to **P1**, we summarize this principle as:

- **P2**: Every create/update operation can persist a data record only if the lock embedded in the data record has not been changed since its last access of the data record.

#### 2.3.3.3 *Unsafe Cleanup of Garbage Index Records.*

If only following **P1** and **P2**, an index record persisted by a create/update operation may be deleted by a garbage cleanup operation. 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. However, Uniqorn must not delete an index record if it is valid on one

hand. This can be achieved by reading 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. On other hand a garbage cleanup operation must not delete an index record if it may turn into valid from garbage. However, an index record that is treated as garbage now may turn into valid later. If such an index record is unsafely deleted by a garbage cleanup operation, the index record may be missing. A garbage index record may turn into valid in the following case: it is persisted in index store by an ongoing create or update operation that has not persisted the data record referred by the index record into data store. If such an ongoing create or update operation later persists the data record successfully, this persisted index record will transition from garbage to valid. Otherwise it stays as garbage. We say the conflict between the garbage cleanup operation and such an ongoing create/update operation as create/update-cleanup conflict.

A garbage cleanup operation utilizes the following two phases to safely delete garbage index records (Figure 5).

1. *change lock*. To prevent an index record that is garbage now from turning into valid later (and then may disastrously be deleted), a garbage cleanup operation forcefully aborts the ongoing create/update operation by updating or uninitializing the lock of the ongoing create/update operation (by **P2**). The changed lock is taken as the lock of the garbage cleanup operation.

2. *delete index record*. On one side, from now on, the ongoing insert/update operation will definitely fail to persist the data record into data store due to the lost lock. The garbage index records left by it will not turn into valid, therefore the garbage cleanup operation can safely delete them. The garbage cleanup operation does not wait for such an ongoing create/update operation to terminate, concerning of increased operation latency, decreased operation throughput, and complicated recovery. On the other side, the garbage index record may be reused by another ongoing or completed create/update operation after the preceding phase. Such an ongoing or completed create/update operation must have changed the lock held by this garbage cleanup operation (by *P2*). If a garbage cleanup operation attempts to delete an index record, however its lock does not have higher priority than the lock embedded in the index record, it aborts itself. Therefore, the ongoing create/update may continue, if not completed, to persist in data store without the risk of its index records being deleted (by **P2**).

In either case, no valid index records will be possibly deleted. We summarize this principle additionally as:

- **P3**: A garbage cleanup operation can delete a garbage index record only if it holds a lock that has higher priority than the lock embedded in the garbage index record.

## 2.4 The Operations

Based on the concurrency control method (Section 2.3), we present a variant of the implementations of the APIs at client (Section 2.2) in this section, which consists of the application-faced CRUD operations as well as an internal garbage cleanup operation. A pseudocode description of this implementation is presented in Figure 6. For all operations, Uniqorn guarantees *uniqueness* (i.e., no two data records in data store will ever have the same alternate key). We provide an informal proof of this guarantee and a formal TLA+ [15] specification (for model checking) in [24].

### 2.4.1 Create and Update

A create or update operation of a data record $dr$ involves addition or update of alternate keys consists of three phases:

1. *Read Data Record/Initialize Dummy Data Record*. The data record $pdr$ of primary key $pk$ is read out from data store if persisted and its embedded lock is extracted (i.e., $lock = pdr.lock$). If $dr.lock \neq pdr.lock$, the operation is aborted since other concurrent operation has modified the data record and $dr$ is stale. If $pdr$ is not persisted, a lock $lock = [pk, (timestamp, clientId), 0]$ is created in memory, assigned to $dr$ (i.e., $dr.lock = lock$) and then persist in data store as a dummy data record $pdr = [lock, \emptyset, \emptyset]$.

2. *Indexing*. A set of index records are created in memory in the form of $(ak, lock)$ for each alternate key $ak \in dr.aks \setminus pdr.aks$. A client attempts to persist those index records into index store in parallel. If any of those index records fails to persist into index store, the create or update operation will abort. For each index record $ir = (ak, lock)$,

   - if no pre-persisted index record with $ak$ is found in index store, $ir$ will be inserted.
   - if an index record with a lower-priority lock than $lock$ is persisted, it will be updated to $ir$ to mark the lock $lock$ of this operation.
   - if an index record with a higher-priority lock than $lock$ is persisted, this operation is aborted in favor of another higher-priority operation. Since this create/update operation will definitely fail to persist $dr$ to data store due to its lost of the lock, it aborts itself earlier.
   - if an index record that refers to a data record of a different primary key from $pk$, the referred data record is read from data store and checked if it has alternate key $ak$. If it has $ak$, the index record is valid and this operation aborts to conform to uniqueness constraint, otherwise garbage.
   - if an index record that refers to a data record of a different primary key from $pk$ is found and validated as garbage, this operation will run the garbage cleanup operation (Section 2.4.4) to replace it with $ir$ (or equivalently delete it and insert $ir$).

3. *Write Data Record*. The version of the lock is incremented by 1 (i.e., $dr.v = dr.v + 1$). The operation will persist the data record $dr$ optimistically only if it still holds the lock $lock$. Otherwise, this operation aborts itself to avoid lost changes of alternate keys since another higher-priority operation has successfully modified this record since the start of this operation.

As an optimization, in the following two cases, the indexing phases can be skipped since no index records must be added into index store.

Figure 6: Pseudocode for Uniqorn operations

```
1: Class Operation {
2:     Class Lock {String pk; /*primary key*/ String e; /*epoch*/ Long v /*version*/}
3:     Class IndexRecord {String ak; /*alternate key*/ String pk; String e; Long v}
4:     Class DataRecord {String pk; String e; Long v; Set<String> aks; /*set of alternate keys*/ Object val}
5:     /*a lock field is used for IndexRecord or DataRecord to collectively denote their fields [pk, e, v]*/

6:     DataRecord create(DataRecord dr)
7:         dr.lock = [dr.pk, (clientId, timestamp), 0] /*create a new lock in memory*/
8:         DataRecord pdr = DataStore.get(dr.pk) /*read persisted data record if any*/
9:         if(pdr = ∅ and dr.aks = ∅) Return DataStore.insert(dr) /*no data record persisted and dr does not have AKs, directly insert dr*/
10:        elif(pdr = ∅ and dr.aks ≠ ∅) pdr = DataStore.insert([dr.lock, ∅, ∅]) /*no data record persisted but dr has AKs, insert the dummy data record embodied from the new lock*/
11:        elif(pdr.aks = ∅ and pdr.val = ∅ and dr.aks = ∅) Return DataStore.updateOptimistically(dr, pdr.lock) /*a dummy data record persisted and dr does not have AKs, directly replace it with dr*/
12:        elif(pdr.aks = ∅ and pdr.val = ∅ and dr.aks ≠ ∅) pdr = DataStore.updateOptimistically([dr.lock, ∅, ∅], pdr.lock) /*a dummy data record persisted, replace it with the dummy data record embodied from the new lock*/
13:        else Return ∅ /*a real data record already persisted*/
14:        for(ak ∈ dr.aks) /*indexing, execute for each ak in parallel*/
15:            if(!persistIndexRecord([ak, pdr.lock]) Return ∅ /*abort if any ak fails to persist*/
16:        dr.v = dr.v + 1 /*increase the version of the lock of dr*/
17:        Return DataStore.updateOptimistically(dr, pdr.lock) /*replace pdr with dr if pdr has not been modified since last read*/

18:    DataRecord update(DataRecord dr)
19:        DataRecord pdr = DataStore.get(dr.pk); /*read persisted data record pdr if any*/
20:        if(pdr = ∅ or pdr.lock ≠ dr.lock) Return ∅ /*no data record persisted or dr is stale, abort*/
21:        for(ak ∈ dr.aks \ pdr.aks) /*indexing, execute for each new or updated ak in parallel*/
22:            if(!persistIndexRecord([ak, pdr.lock]) Return ∅ /*abort if any ak fails to persist*/
23:        dr.v = dr.v+1 /*increase the version of the lock of dr*/
24:        Return DataStore.updateOptimistically(dr, pdr.lock) /*replace pdr with dr if pdr has not been modified since last read*/

25:    DataRecord read(String ak)
26:        pir = IndexStore.get(ak) /*read persisted index record pir if any*/
27:        if(pir ≠ ∅) pdr = DataStore.get(pir.pk) else Return ∅ /*read referred data record pdr or return empty*/
28:        if(pdr ≠ ∅ and ak ∈ pdr.aks) Return pdr /*pdr has ak, return it*/
29:        Return ∅ /*pir is garbage (pdr does not have ak), return empty*/

30:    Boolean delete(String ak)
31:        pir = IndexStore.get(ak) /*read persisted index record pir if any*/
32:        if(pir ≠ ∅) pdr = DataStore.get(pir.pk) else Return false /*read referred data record pdr or return false*/
33:        if(pdr = ∅ or ak ∉ pdr.aks) Return false /*pir is garbage (pdr does not have ak), return false*/
34:        Return DataStore.deleteOptimistically(pdr, pdr.lock) /*delete pdr if pdr has not been modified since last read*/

35:    Boolean persistIndexRecords(IndexRecord ir)
36:        pir = IndexStore.get(ir.ak) /*read persisted index record pir if any*/
37:        if(pir = ∅) Return IndexStore.insert(ir) /*no index record persisted, directly insert ir*/
38:        if(pir.pk = ir.pk) /*pir and ir refer to the same pk*/
39:            if(pir.e = ir.e) /*pir and ir are in the same epoch*/
40:                if(pir.v < ir.v) Return IndexStore.updateOptimistically(ir, pir.lock) /*pir is stale, replace it with ir directly*/
41:                elif(pir.v > ir.v) Return false /*ir is stale, abort*/
42:                else Return true /*ir already persisted*/
43:            else /*pir and ir are in different epochs*/
44:                pdr = DataStore.get(ir.pk) /*read currently persisted data record pdr (it embeds the highest-priority lock)*/
45:                if(ir.lock ≠ pdr.lock) Return false /*lock marked in ir is lost to another higher-priority operation*/
46:                Return IndexStore.updateOptimistically(ir, pir.lock) /*pir is in a past epoch, replace it with ir if it has not been modified since last read*/
47:        else /*pir and ir refer to different pks*/
48:            pdr = DataStore.get(pir.pk) /*read data record pdr referred by pir*/
49:            if(pdr ≠ ∅) /*validate if pir is garbage or valid by checking pdr*/
50:                if(ak ∈ pdr.aks) Return false /*pdr has ak and hence pir is valid, this operation aborts*/
51:                elif(pdr.aks = ∅ and pdr.val = ∅) DataStore.deleteOptimistically(pdr, pdr.lock) /*pdr is a dummy data record, delete it if it has not been modified since last read*/
52:                else DataStore.updateOptimistically([pdr.pk, pdr.e, pdr.v+1, pdr.aks, pdr.val], pdr.lock) /*pir is garbage, change the lock of pdr.pk for deleting pir if pdr has not been modified since last read*/
53:            IndexStore.deleteOptimistically(pir, pir.lock) /*delete garbage pir if pir has not been modified since last read*/
54:            Return IndexStore.insert(ir)
55:    }
```

- A record to be created does not have any alternate key.
- A record to be updated does not add alternate keys or change any existing alternate keys.

### 2.4.2 Read

For a read operation by an alternate key $ak$, it consists of two phases:

- *Read Index Record.* If the index record $ir$ of $ak$ is not persisted, no data record that has $ak$ is persisted and this operation returns $\emptyset$.
- *Validate.* Read the data record $dr$ of primary key $ir.pk$. If $dr$ is persisted and $ak \in dr.aks$, then this operation returns $dr$, otherwise, $ir$ is garbage and this operation returns $\emptyset$.

### 2.4.3 Delete

For a delete operation by an alternate key $ak$, the data record can be directly deleted, if it has $ak$, without deleting any of its index records. A deletion operation consists of three phases:

- *Read Index Record.* If the index record $ir$ by $ak$ is not persisted, no data record that has $ak$ is persisted and this operation returns false.
- *Validate.* Read the data record $dr$ of primary key $ir.pk$. If $dr$ is not persisted or $ak \notin dr.aks$, then $ir$ is garbage and this operation returns false.
- *Delete Data Record.* The data record $dr$ is deleted optimistically only if $dr.lock$ has not been modified by other operations since the read in the first phase (i.e., this delete operation still holds the lock). Once $dr$ is deleted, all of its index records become orphaned.

### 2.4.4 Garbage Cleanup

Garbage cleanup operation is an internal operation to Uniqorn and invisible to application programmers. It is the only operation that deletes index records. During a CRUD operation as shown above, an index record may be found or left as garbage. If the alternate key of a garbage-suspected index record is to be reused (as in a create/update operation in Section 2.4.1), it has to be deleted in the first place; otherwise, it can be deleted in best-effort. A valid index record never triggers the garbage cleanup operation, so it will not be deleted. A garbage cleanup operation consists of two phases to delete a garbage-suspected index record $ir$:

- *Change Lock.* If the data record $dr$ of primary key $ir.pk$ is not persisted, this phase can be skipped since the data record has been deleted and all of its index records can be safely deleted (no ongoing create/update operations will ever derive the same index record). If the data record $dr$ of primary key $ir.pk$ is persisted, the version of the lock of $dr$ is incremented by 1 (i.e., $dr.v = dr.v + 1$). This cleanup operation will persist the data record $dr$ optimistically only if no other operations have modified the lock since the time $ir$ is validated (by reading the data record of primary key $ir.pk$) as garbage. Otherwise, this cleanup operation aborts itself to avoid risking of deleting a possibly valid index record.
- *Delete Index Record.* The garbage-suspected index record is deleted successfully if the lock embedded in it has not been changed since the time $ir$ is validated; otherwise, the cleanup fails since a higher-priority operation has updated it to reuse its alternate key.

A dummy data record can be deleted directly without any restriction.

## 2.5 Discussions

### 2.5.1 Garbage

Garbage index records may be persisted in index store due to failures and concurrency conflicts (Section 2.3). 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 partitioned 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 a lock 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 3.4). 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 (Section 2.3.3.2). 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.

### 2.5.2 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 partition or data store partition usually comprises a cluster of physical database servers that replicate to one another. Since data store is the source-of-truth of records, the unavailability of relevant data store partitions will fail all CRUD operations that access them. The unavailability of index store partitions affects the availability of different operations differently. We enhance Uniqorn clients (Section 2.4) to maintain high availability for all operations by either sacrificing scalability or consistency temporarily.

Upon the unavailability of relevant index store partitions, 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 partition where the index record of the alternate key is persisted. If the index store partition is unavailable, the read operation will fail (Section 2.4). In this case, we enhance Uniqorn clients to fall back to looking up the alternate key by scattering/gathering across all data store partitions if they provide the local secondary index 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 partitions. However, scattering/gathering across all data store partitions 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 partitions, 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 partitions 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 partitions 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 locks) 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 databases to recover. In Uniqorn, the only additional metadata, locks, are embedded into index and data records, which does not impose additional constraints on the split and merge of partitions for load balancing and system maintenance.

# 3. EVALUATIONS

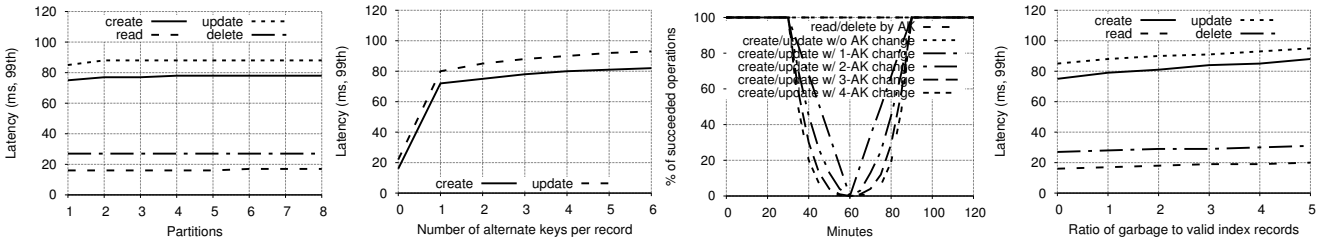
Uniqorn lies in the performance space between a single monolithic database with secondary indexes and a partitioned database system with local secondary indexes on each partition. On one hand, Uniqorn incurs longer latency than a single monolithic database for the benefit of scalability, though retaining the ACID 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 database system performance. On the other hand, Uniqorn has an additional dependency on index store than a partitioned database system with local secondary indexes 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 partitions 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 Walmart private data center. The servers run the Linux operating system on x86 processors. All partitions 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 chose MySQL as the underlying database technologies to run each index store and data store partitions. All MySQL databases 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 partitions to the number of data store partitions as 1:1. Each data store partition has the approximately same number of records during all experiments, so does each index store partition. As the number of data store partitions 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 partitions 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 ca-




(a) Latency of CRUD operations as data store, index store, and clients scale linearly.

(b) Latency of create/update operations as # of alternate keys per record increases.

(c) Operation availability during index store outage and recovery time window.

(d) The effect of garbage index records on latency.

pacity, which falls in the targeting scope of the design of Uniqorn.

## 3.1 Performance Cost

We compare Uniqorn to a single-partitioned (Monolithic) data system, where the sole database has all records and a local secondary index 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 local secondary indexes. 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 2), 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 lock 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 2: Latency (ms, 99th)

| Operation | Monolithic | Uniqorn (1-partition) |
|---|---|---|
| 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 |

## 3.2 Scalability

To emulate a scaling system, we increase both index store and index store partitions 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 partitions 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 7a). The latency of all operations are approximately equal to their latency in Table 2 with a single partition. 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 partitions increases. The increased system load is evenly distributed across all clients and servers.

We also measure the latency in 99 percentile when application increases the number of alternate keys per record (Figure 7b) in the setting of 8 partitions. 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 local secondary index 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.

## 3.3 Availability

We run the workload in the setting of 6 partitions while bring down one index store partition 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 2.5.2) for the improved availability at the cost of temporally reduced consistency. We measure the ratio of succeeded CRUD operations among all issued operations (Figure 7c). As an index store partition goes down, all create operations with at least one of its alternate key on this partition will fail; all update operations with at least one of its changed or newly added alternate key on this partition will fail. However, if an update operation deletes an alternate key on this partition, 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 partitions to look up for a given alternate key when the corresponding index store partition is down. However it incurs prolonged latency and additional resource cost due to the scatter/gather. As more index store partitions 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 partition and more chance to fail. In both cases, the total operation availability dips.

## 3.4 Effect of Garbage Index Records

We run the workload in the setting of 8 partitions 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 lock 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 database technologies, the data system can tolerate high ratio of garbage index records with negligible latency increase (Figure 7d).

# 4. RELATED WORK

Concurrency control methods for generic distributed transactions in a distributed database system [2, 5], such as two-phase locking and timestamp-ordering, can be applied to manage global secondary indexes with uniqueness constraint. Unlike Uniqorn, both two-phase locking and timestamp-ordering methods assume participating data managers (DMs) to support two-phase commit [17]. 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 [17]. Uniqorn, however, addresses a specific problem (i.e., global secondary indexes) 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 [2, 18] 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 [2], Uniqorn only is persisted a single copy of a record (though it embeds a global optimistic lock as a "version" mark into the record) at any time. Industry implementations (such as Java Enterprise Edition [16]) 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 global secondary indexes transactionally, similar to the optimistic concurrency control method in [12, 23]. However, unlike [12], Uniqorn's optimistic concurrency control method is specially designed for a partitioned database system, instead of solely for a single database. In Uniqorn, the global optimistic locks are partially ordered, yet the transaction timestamps are totally ordered in [12]. In [23], a distributed optimistic concurrency control method followed by locking (locking is an integral part of distributed validation and two-phase commit) is proposed to reduce data contention and increase throughput in a distributed database. However, Uniqorn is purely an optimistic concurrency control method from application programmers' perspective.

Traditional monolithic relational database systems like Oracle, MySQL, SQL Server, and PostgreSQL, etc. perform local secondary indexing on internal indexing data structures in the same transaction as the record modification (e.g., [11, 6]). Uniqorn achieves the same semantics by building global secondary indexes on top of over-the-shelf databases without using their support of local secondary indexes.

Key-value database systems, like Cassandra [13], Couchbase [8, 14], and DynamoDB [22], etc. partition data across multiple servers/nodes and usually do not support distributed transactions. They either do not support global secondary index or support with relaxed consistencies. Cassandra [7] and MegaStore [1] store index data along with original data on the same partition (therefore only supports local secondary indexes). DynamoDB [21] and Couchbase [21] support global secondary indexes with eventual consistency, that is, indexes are updated in a best-effort manner. Uniqorn supports global secondary indexes transactionally and peculiarly enforces global uniqueness among all secondary keys.

Percolator [18] implements snapshot isolation by extending multi-version timestamp ordering [2] across a distributed system using two-phase commit. It builds on the single-row transactions of BigTable [4] 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 global secondary indexes with uniqueness constraint. Moreover, Uniqorn does not need multi-version supports from underlying data systems like BigTable.

Similar to SLIK [10], 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.

# 5. CONCLUSION AND FUTURE WORKS

We have built Uniqorn to show that large-scale partitioned database systems need not forgo the benefits of global secondary indexes, 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 ACID supports of all CRUD operations of individual records, at the cost of either slightly reduced write availability or temporally reduced consistency only upon alternate key additions or changes. We are interested in knowing how far we can go with the global optimistic locking mechanism to tame concurrency anomalies to provide transactional multi-record CRUD operations or generic global secondary indexes, 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.

# 6. REFERENCES

[1] 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.

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

[3] 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.

[4] 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.

[5] 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.

[6] C. Diaconu, C. Freedman, E. Ismert, P.-A. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling. Hekaton: Sql server's memory-optimized oltp engine. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 1243–1254, New York, NY, USA, 2013. ACM.

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

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

[9] 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.

[10] 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.

[11] 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.

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

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

[14] 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.

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

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

[17] 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.

[18] 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.

[19] 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.

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

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

[22] 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.

[23] 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.

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