Specification and Implementation of Uniqorn CRUD Operations

Paper # 222

1 Specification

We call the embedded version ([g, v] where where g is the generation identifier of the data record among all data records that have ever associated with primary key pk, v is increased upon every update of the data record (starting from 0) within a generation) in a data record or index record as a lock in the proof (hope it is more understandable).

By calling CRUD APIs 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 Description

Based on the approach of global optimistic indexing, we present a variant of the implementations of the CRUD APIs at client 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 1. 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+ specification (for model checking) in another document.

2.1 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. *Persist Index Records.* 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.

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	2.1
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	2.2
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	2.1
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	2.3

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) 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.

- 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.

2.2 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 $\emptyset$.

2.3 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 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 2.1), 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 4 phases to delete a garbage-suspected index record ir:

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

- **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 garbage cleanup operation fails.

- *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.

3 Pseudocode

Figure 1: Pseudocode for Uniqorn operations

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

    DataRecord create(DataRecord dr)
        dr.lock = [dr.pk, (clientId, timestamp), 0] /*create a new lock in memory*/
        DataRecord pdr = DataStore.get(dr.pk) /*read persisted data record if any*/
        if(pdr = $\emptyset$ and dr.aks = $\emptyset$) Return DataStore.insert(dr) /*no data record persisted and dr does not have AKs, directly insert dr*/
        elif(pdr = $\emptyset$ and dr.aks $\neq \emptyset$) pdr = DataStore.insert([dr.lock, $\emptyset$, $\emptyset$]) /*no data record persisted but dr has AKs, insert the dummy data record embodied from the new lock*/
        elif(pdr.aks = $\emptyset$ and pdr.val = $\emptyset$ and dr.aks = $\emptyset$) Return DataStore.updateOptimistically(dr, pdr.lock) /*a dummy data record persisted and dr does not have AKs, directly replace it with dr*/
        elif(pdr.aks = $\emptyset$ and pdr.val = $\emptyset$ and dr.aks $\neq \emptyset$) pdr = DataStore.updateOptimistically([dr.lock, $\emptyset$, $\emptyset$], pdr.lock) /*a dummy data record persisted, replace it with the dummy data record embodied from the new lock*/
        else Return $\emptyset$ /*a real data record already persisted*/
        for(ak $\in$ dr.aks) /*indexing, execute for each ak in parallel*/
            if(!persistIndexRecord([ak, pdr.lock]) Return $\emptyset$ /*abort if any ak fails to persist*/
        dr.v = dr.v + 1 /*increase the version of the lock of dr*/
        Return DataStore.updateOptimistically(dr, pdr.lock) /*replace pdr with dr if pdr has not been modified since last read*/

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

    DataRecord read(String ak)
        pir = IndexStore.get(ak) /*read persisted index record pir if any*/
        if(pir $\neq \emptyset$) pdr = DataStore.get(pir.pk) else Return $\emptyset$ /*read referred data record pdr or return empty*/
        if(pdr $\neq \emptyset$ and ak $\in$ pdr.aks) Return pdr /*pdr has ak, return it*/
        Return $\emptyset$ /*pir is garbage (pdr does not have ak), return empty*/

    Boolean delete(String ak)
        pir = IndexStore.get(ak) /*read persisted index record pir if any*/
        if(pir $\neq \emptyset$) pdr = DataStore.get(pir.pk) else Return false /*read referred data record pdr or return false*/
        if(pdr = $\emptyset$ or ak $\notin$ pdr.aks) Return false /*pir is garbage (pdr does not have ak), return false*/
        Return DataStore.deleteOptimistically(pdr, pdr.lock) /*delete pdr if pdr has not been modified since last read*/

    Boolean persistIndexRecords(IndexRecord ir)
        pir = IndexStore.get(ir.ak) /*read persisted index record pir if any*/
        if(pir = $\emptyset$) Return IndexStore.insert(ir) /*no index record persisted, directly insert ir*/
        if(pir.pk = ir.pk) /*pir and ir refer to the same pk*/
            if(pir.e = ir.e) /*pir and ir are in the same epoch*/
                if(pir.v < ir.v) Return IndexStore.updateOptimistically(ir, pir.lock) /*pir is stale, replace it with ir directly*/
                elif(pir.v > ir.v) Return false /*ir is stale, abort*/
                else Return true /*ir already persisted*/
            else /*pir and ir are in different epochs*/
                pdr = DataStore.get(ir.pk) /*read currently persisted data record pdr (it embeds the highest-priority lock)*/
                if(ir.lock $\neq$ pdr.lock) Return false /*lock marked in ir is lost to another higher-priority operation*/
                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*/
        else /*pir and ir refer to different pks*/
            pdr = DataStore.get(pir.pk) /*read data record pdr referred by pir*/
            if(pdr $\neq \emptyset$) /*validate if pir is garbage or valid by checking pdr*/
                if(ak $\in$ pdr.aks) Return false /*pdr has ak and hence pir is valid, this operation aborts*/
                elif(pdr.aks = $\emptyset$ and pdr.val = $\emptyset$) DataStore.deleteOptimistically(pdr, pdr.lock) /*pdr is a dummy data record, delete it if it has not been modified since last read*/
                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*/
            IndexStore.deleteOptimistically(pir, pir.lock) /*delete garbage pir if pir has not been modified since last read*/
            Return IndexStore.insert(ir)
    }
```