MODULE *uniqorn*

a fully-fledged specification for *uniqorn*. A CRUD operation consists of multiple phases. There are infinite many clients. Each client can issue any CRUD operation to the queue by enqueuing the first phase of the operation into the queue. The state machine (which is modeled by this specification) will randomly pick up any phase of any operation in the queue and execute this phase according to the *uniqorn* protocol described in the draft paper. When an operation finishes a given phase, it queues the next phase. However, it does not remove the completed phase from the queue. As a result, the completed phase can be picked up and run again All of these are intentionally designed to emulate a packet-lost, duplicate, reorder, delayed network/distributed environment.

create/update operations that reuse an alternate key of a garbage index record will not directly delete the garbage index record. Only the garbage cleanup can delete a garbage index record. This behavior has the same effect (since TLA model checking will exhaust all state paths) as each create/update mandatorily delete garbage index record and then insert its own index record

data store or index store partitioning policy does not affect the protocol, so we do not even express them in this specification

an example model values *PERSIST_INDEX_RECORD*, *PERSIST_DATA_RECORD* : model value *CLEANUP_VALIDATE*, *CLEANUP_CHANGE_LOCK*, *CLEANUP_DELETE_GARBAGE*: model value
AKS: 1 . . 16
PKS: 1 . . 8
VAL: 1 *AKS_PER_RECORD*: 2 PLEASE ALSO SET THE FOLLOWING STATE CONSTRAINTS TO LIMIT THE STATE SPACE, *e.g.*
$uuid < 32$

EXTENDS *TLC*, *Integers*, *Sequences*, *FiniteSets*, *Bags*

phases for create and update, the first phase "read data record" or "init a dummy data record" is omitted since it is done at the time when a create or update operation is initiated

CONSTANTS *PERSIST_INDEX_RECORD*, *PERSIST_DATA_RECORD*

phases for cleanup, the first phase "read index record" is omitted since it is done when a cleanup operation is initiated

CONSTANTS *CLEANUP_VALIDATE*, *CLEANUP_CHANGE_LOCK*, *CLEANUP_DELETE_GARBAGE*

a delete operation has only one phase, it can be done when the delete operation is initiated, so we omit it

set of integer keys for primary keys and alternate keys

CONSTANTS *PKS*, *AKS*

a non-zero integer, 0 means the *val* is null/empty, other values means non-empty value

CONSTANTS *VAL*

the number of *aks* per record, a record can have 0 to *AK_PER_RECORD* alternate keys

CONSTANTS *AKS_PER_RECORD*

data or index records in data store partitions or index store partitions

VARIABLES *persistedDataRecords*, *persistedIndexRecords*

seperate queues for all create/update and clenaup operations, delete has only one phase, no need a queue for it.

```
60   an operation can equeue its next phase as it progresses. All create and update operations
61   share the same queue since they follow the same protocol after their initial phases, respectively
62 VARIABLES inprogressCreateUpdates, inprogressCleanups

64   a monotonically increasing integer as a uuid, which will be used as epoch values.
65   For any two epoch values, their relations are only equality and inequality, no other relation exists
66   like greater than, less than, etc. That is, we don't use the monotonicity of the integers for any use
67 VARIABLES uuid

69 RECURSIVE Cat(_)
70 Cat(akset) ≜
71           IF (Cardinality(akset) = 0) THEN ⟨⟩ ELSE
72                LET akchosen ≜ (CHOOSE ak ∈ akset : TRUE)
73                IN   Append(Cat(akset \ {akchosen}), akchosen)

75   the catenation of all aks in a sequence
76 AkSeq ≜ Cat(AKS)

78   randomly choose a set of aks from the provided AKs, can be empty too
79 ChooseAKs(akCount) ≜
80   LET chooseAk[i ∈ 0 .. akCount] ≜
81     IF i = 0 THEN {} ELSE  chooseAk[i − 1] ∪ {AkSeq[RandomElement(1 .. Len(AkSeq))]}
82   IN   chooseAk[akCount]

84   ********** data                store accesses start here**********
85 IsDummy(pk) ≜ IF  ∧ pk ∈ DOMAIN persistedDataRecords
86                   ∧ persistedDataRecords[pk].aks = {}
87                   ∧ persistedDataRecords[pk].val = 0
88                THEN TRUE
89                ELSE  FALSE

91 isLockHeld(pk, epoch, version) ≜ IF  ∧ pk ∈ DOMAIN persistedDataRecords
92                                    ∧ persistedDataRecords[pk].epoch = epoch
93                                    ∧ persistedDataRecords[pk].version = version
94                                 THEN TRUE
95                                 ELSE  FALSE

97 DataStoreDelete(pk) ≜
98   ∧ pk ∈ DOMAIN persistedDataRecords
99   ∧ persistedDataRecords′ = [key ∈ (DOMAIN persistedDataRecords \ {pk}) ↦
100                                                    persistedDataRecords[key]]
101  ∧ UNCHANGED ⟨persistedIndexRecords, inprogressCreateUpdates, inprogressCleanups⟩

103 DataStoreInitLock(pk, aks, epoch, version) ≜
104    ∨ ∧ pk ∉ DOMAIN persistedDataRecords
105      ∧ persistedDataRecords′ = persistedDataRecords @@ (pk :> [pk ↦ pk, epoch ↦ epoch,
106                                                    version ↦ version, aks ↦ {}, val ↦ 0])
```

```
107         ∧ inprogressCreateUpdates′ = inprogressCreateUpdates ∪ {[phase ↦ PERSIST_INDEX_RECORD,
108               pk ↦ pk, newAks ↦ aks, prevAks ↦ {}, upsertAks ↦ {}, epoch ↦ epoch, version ↦ version]}
109           ∨  ∧ IsDummy(pk)
110              ∧ persistedDataRecords′ = [persistedDataRecords EXCEPT ![pk].epoch = epoch,
111                                          ![pk].version = version, ![pk].aks = {}, ![pk].val = 0]
112              ∧ inprogressCreateUpdates′ = inprogressCreateUpdates ∪ {[
113                    phase ↦ PERSIST_INDEX_RECORD, pk ↦ pk, newAks ↦ aks, prevAks ↦ {},
114                                                  upsertAks ↦ {}, epoch ↦ epoch, version ↦ version]}
115           ∨ UNCHANGED ⟨persistedDataRecords, inprogressCreateUpdates⟩

117   DataStoreUpdateOptimistically(pk, aks, epoch, version) ≜
118         ∨ ∧ isLockHeld(pk, epoch, version)
119           ∧ persistedDataRecords′ = [persistedDataRecords EXCEPT ![pk].version = @ + 1,
120                                                                  ![pk].aks = aks, ![pk].val = VAL]
121         ∨ UNCHANGED persistedDataRecords

123   DataStoreValidate(pk, ak, epoch, version) ≜
124     IF pk ∈ DOMAIN persistedDataRecords ∧ (ak ∈ persistedDataRecords[pk].aks) THEN
125           UNCHANGED inprogressCleanups
126      ELSE inprogressCleanups′ = inprogressCleanups ∪ {[phase ↦ CLEANUP_CHANGE_LOCK, pk ↦ pk,
127                                                        ak ↦ ak, epoch ↦ epoch, version ↦ version]}

129   DataStoreChangeLock(pk, ak, epoch, version) ≜
130     IF pk ∈ DOMAIN persistedDataRecords THEN
131       IF ak ∉ persistedDataRecords[pk].aks THEN
132         ∧ IF persistedDataRecords[pk].val = 0 THEN
133               persistedDataRecords′ = [key ∈ (DOMAIN persistedDataRecords \ {pk}) ↦
134                                                        persistedDataRecords[key]]
135             ELSE
136               persistedDataRecords′ = [persistedDataRecords EXCEPT ![pk].version = @ + 1]
137         ∧ inprogressCleanups′ = inprogressCleanups ∪ {[phase ↦ CLEANUP_CHANGE_LOCK, pk ↦ pk,
138                                                        ak ↦ ak, epoch ↦ epoch, version ↦ version]}
139       ELSE UNCHANGED ⟨persistedDataRecords, inprogressCleanups⟩
140      ELSE
141       ∧ inprogressCleanups′ = inprogressCleanups ∪ {[phase ↦ CLEANUP_DELETE_GARBAGE,
142                                                     pk ↦ pk, ak ↦ ak, epoch ↦ epoch, version ↦ version]}
143       ∧ UNCHANGED persistedDataRecords

145   ********** data store accesses end here**********

147   ********** index store accesses start here**********
148   index store has only two accesses methods: insert and delete. Update and replace accesses can be derived from these two.
149   IndexStoreDirectlyInsert(pk, epoch, version, prevAks, newAks, upsertAks) ≜
150     ∧ upsertAks ≠ newAks \ prevAks
151     ∧ LET ak ≜ (CHOOSE anyAk ∈ (newAks \ prevAks) : TRUE)
152       IN  ∨ ∧ ak ∉ DOMAIN persistedIndexRecords
```

```
                ∧ persistedIndexRecords′ = persistedIndexRecords  @@ (ak :> [ak ↦ ak,
                                                    pk ↦ pk, epoch ↦ epoch, version ↦ version])
                ∧ ∨ ∧ (newAks \ prevAks) = (upsertAks ∪ {ak})
                    ∧ inprogressCreateUpdates′ = inprogressCreateUpdates ∪ {[
                                          phase ↦ PERSIST_DATA_RECORD, pk ↦ pk, aks ↦ newAks,
                                                  epoch ↦ epoch, version  ↦ version]}
                  ∨ ∧ (newAks \ prevAks) ≠ (upsertAks ∪ {ak})
                    ∧ inprogressCreateUpdates′ = inprogressCreateUpdates ∪ {[phase ↦
                     PERSIST_INDEX_RECORD, pk ↦ pk, newAks ↦ newAks, prevAks ↦ prevAks,
                        upsertAks ↦ (upsertAks ∪ {ak}), epoch ↦ epoch, version ↦ version]}
                  ∨ UNCHANGED ⟨persistedIndexRecords, inprogressCreateUpdates⟩

IndexStoreDeleteOptimistically(ak, pk, epoch, version) ≜
    IF  ∧ ak ∈ DOMAIN persistedIndexRecords
        ∧ persistedIndexRecords[ak].pk = pk
        ∧ persistedIndexRecords[ak].epoch = epoch
        ∧ persistedIndexRecords[ak].version = version
     THEN
       persistedIndexRecords′ = [key ∈ (DOMAIN persistedIndexRecords \ {ak})
                                                    ↦ persistedIndexRecords[key]]
     ELSE  UNCHANGED persistedIndexRecords

********** index store accesses end here**********

********** various operations start here**********
 issue a create operation
Create(pk, aks) ≜
    ∧ inprogressCreateUpdates′ = inprogressCreateUpdates ∪ {[phase ↦ PERSIST_INDEX_RECORD,
              pk ↦ pk, prevAks ↦ {}, newAks ↦ aks, upsertAks ↦ {}, epoch ↦ uuid, version ↦ 0]}
    ∧ UNCHANGED ⟨persistedDataRecords, persistedIndexRecords, inprogressCleanups⟩

 issue an update operation
Update(pk, epoch, version, prevAks, newAks) ≜
    ∧ inprogressCreateUpdates′ = inprogressCreateUpdates ∪ {[phase ↦ PERSIST_INDEX_RECORD,
                                pk  ↦ pk, prevAks ↦ prevAks, newAks ↦ newAks, upsertAks ↦ {},
                                                    eopch ↦ epoch, version ↦ version]}
    ∧ UNCHANGED ⟨persistedDataRecords, persistedIndexRecords, inprogressCleanups⟩

 issue a cleanup operation
Cleanup(ak, pk, epoch, version) ≜
    ∧   inprogressCleanups′ = inprogressCleanups ∪ {[phase ↦ CLEANUP_VALIDATE, ak ↦ ak,
                                                    pk ↦ pk, epoch ↦ epoch, version ↦ version]}
    ∧   UNCHANGED ⟨persistedDataRecords, persistedIndexRecords, inprogressCreateUpdates⟩

 make a create/update operation go through its phases
RunCreateUpdate(op) ≜
    LET phase ≜ op.phase
```

```
200             pk ≜ op.pk
201             epoch ≜ op.epoch
202             version ≜ op.version
203         IN  ∨ ∧ phase = PERSIST_INDEX_RECORD
204               ∧ LET prevAks ≜ op.prevAks
205                       newAks ≜ op.newAks
206                       upsertAks ≜ op.upsertAks
207                   IN  ∧ IndexStoreDirectlyInsert(pk, epoch, version, prevAks, newAks, upsertAks)
208                       ∧ UNCHANGED ⟨persistedDataRecords, inprogressCleanups⟩
209             ∨ ∧ phase = PERSIST_DATA_RECORD
210               ∧ DataStoreUpdateOptimistically(pk, op.aks, epoch, version)
211               ∧ UNCHANGED ⟨persistedIndexRecords, inprogressCleanups, inprogressCreateUpdates⟩


214  make a garbage cleanup operation go through its phases
215 RunCleanup(cleanupOp) ≜
216     LET phase ≜ cleanupOp.phase
217         pk ≜ cleanupOp.pk
218         ak ≜ cleanupOp.ak
219         epoch ≜ cleanupOp.epoch
220         version ≜ cleanupOp.version
221     IN  ∨ ∧ phase = CLEANUP_VALIDATE
222           ∧ DataStoreValidate(pk, ak, epoch, version)
223           ∧ UNCHANGED ⟨persistedDataRecords, persistedIndexRecords, inprogressCreateUpdates⟩
224         ∨ ∧ phase = CLEANUP_CHANGE_LOCK
225           ∧ DataStoreChangeLock(pk, ak, epoch, version)
226           ∧ UNCHANGED ⟨persistedIndexRecords, inprogressCreateUpdates⟩
227         ∨ ∧ phase = CLEANUP_DELETE_GARBAGE
228           ∧ IndexStoreDeleteOptimistically(ak, pk, epoch, version)
229           ∧ UNCHANGED ⟨persistedDataRecords, inprogressCreateUpdates, inprogressCleanups⟩

231  ********** various operations end here**********

233  ********** state machine starts here**********
234  all aks and pks are initialized in each partition
235 Init ≜ ∧ persistedDataRecords = [pk ∈ {} ↦ {}]
236         ∧ persistedIndexRecords = [ak ∈ {} ↦ {}]
237         ∧ inprogressCreateUpdates = {}
238         ∧ inprogressCleanups = {}
239         ∧ uuid = 0

241  next-state actions
242 Next ≜ ∧ ∨ ∃ pk ∈ PKS : DataStoreDelete(pk)
243           ∨ ∃ pk ∈ PKS : Create(pk, ChooseAKs(AKS_PER_RECORD))
244           ∨ ∃ ak ∈ DOMAIN persistedIndexRecords : Cleanup(ak, persistedIndexRecords[ak].pk,
245                 persistedIndexRecords[ak].epoch, persistedIndexRecords[ak].version)
```

```
246                 ∨ ∃ pk ∈ DOMAIN persistedDataRecords : Update(pk, persistedDataRecords[pk].epoch,
247                                 persistedDataRecords[pk].version, persistedDataRecords[pk].aks,
248                                                                   ChooseAKs(AKS_PER_RECORD))
249                 ∨ ∃ createUpdateOp ∈ inprogressCreateUpdates :  RunCreateUpdate(createUpdateOp)
250                 ∨ ∃ cleanupOp ∈ inprogressCleanups :  RunCleanup(cleanupOp)
251               ∧ uuid′ = uuid + 1

253   specification
254  Spec ≜ Init ∧ □[Next]⟨persistedDataRecords, persistedIndexRecords, inprogressCleanups,
255                                                  inprogressCreateUpdates, uuid⟩
256   ********** state                machine ends here**********

258   no missing index record invariant
259  NoMissing ≜ ∀ pk ∈ DOMAIN persistedDataRecords :
260                   IF persistedDataRecords[pk].ak ≠ 0 THEN
261                       ∃ ak ∈ DOMAIN persistedIndexRecords :
262                         ∧ persistedIndexRecords[ak].pk = pk
263                         ∧ persistedDataRecords[pk].ak = ak
264                    ELSE  TRUE

266  THEOREM Spec ⇒ NoMissing
267 ╚═══════════════════════════════════════════════════════════════════════════
```

* Modification History
* Last modified *Wed Mar* 07 17:10:06 *PST* 2018 by *jyi*
* Created *Mon Feb* 05 09:28:50 *PST* 2018 by *jyi*