## 1 System Goals

Many recent transactional database systems, such as Google Spanner, FaunaDB, and AWS Aurora, guarantee Strict Serializability (or stronger) for application programmers. Our system *guarantees Regular Sequential Serializability*, a consistency model that weakens Strict Serializability but maintains the same set of application invariants. This allows more flexibility in serving reads, potentially decreasing tail-latency.

Both Strict Serializability and Regular Sequential Serializability assume synchronous client interaction. That is, their guarantees only hold for complete operations – those with a matching invocation and response pair. This system aims to provide *correctness for multiple outstanding client operations* in the form of a invocation-order guarantee. As the name suggests, all clients can expect system execution to respect the order of their invocations.

Performance wise, we strive to achieve *high throughput*, minimizing the overhead imposed by providing the invocation-order guarantee. Following the approach of Calvin, we "determinize" the order of transactions through a shared log, eliminating reliance on concurrency control mechanisms in the common case. We also emphasize *data durability*; once returned to the client, a transaction will not be lost unless an improbable number of service components fail.

## 2 Client-Service Interaction

Clients establish a session with the service using a local library, through which all interactions occur. The client-side library transparently maintains this session without the need for user intervention. Clients may have multiple outstanding transaction requests and can expect that the resulting execution preserves invocation order. Read-only transactions are automatically assigned a sequence number respecting real-time invocation order, which we call a CRSN (client read-only sequence number). Similarly, read-write transactions and write-only transactions are stamped with a CWSN (client write sequence number). Both CRSNs and CWSNs are composed of a unique client identifier, $cid$, and a number $num \in \mathbb{N}$ i.e. $(cid, 0), (cid, 1), (cid, 2), ...$ where $(cid, 0) < (cid, 1) < (cid, 2) < ...$ with respect to invocation order.

## 3 System Assumptions and Design Overview

In our system, all transactions are handled by a transaction manager, which is a collection of servers ordered to form a chain. This manager communicates with shard groups to durably store data. Each shard group is a multi-versioned, linearizable, storage service. Only read-write transactions are replicated in the manager chain, and clients can open a session with any manager server *except for* the head and tail.

Assume the transaction manager consists of $n$ servers and data is split among $m$ shard groups. Uniquely identify each node in the chain with an index $i \in [1, n]$ based on their location. That is, the head of chain is 1; the tail is $n$; and the successor of any other node $i$ in the chain is $i + 1$. We also label each shard group with some unique $j \in [1, m]$. Figure 1 outlines the data structures present on service nodes and client sessions.

As in Chain Replication and CRAQ, we assume the transaction manager servers are fail-stop. We also assume the presence of a failure detector for the chain and a coordination service, which maintains the mapping from shards to keyspaces. We will refer to this mapping as $sh(\cdot)$. Our network model is asynchronous, and messages can be dropped, delivered out of order, or both.

Read-write transactions start as *PreCommit* at the head of the chain and are replicated at successive nodes. Once the tail appends a transaction to its log, the transaction is considered committed, and its constituent sub-transactions sent to the relevant shard groups for execution. Completion of sub-transactions are sent to the chain tail. After the tail learns that all of these are complete, a command to transition the transaction to *Executed* is sent backwards through the chain. A transaction whose status changes to *Executed* on the head node subsequently has its completion returned to the client.

The rest of this specification is as follows. We will detail the protocol for write-only and read-only transactions. Then, we describe how read-only and dependent read-only transactions are reduced to the case of write-only transactions.

As a notational convention, we will take superscript "write" to denote the write key-set and superscript "read" to denote the read key-set of a transaction. For maps, superscript "key" represents the domain of the map and "value" to be its range. On maps or sets whose keys admit an ordering, let $\max(\cdot)$ and $\min(\cdot)$ have their usual meaning. If the map or set $S$ is empty, then both $\max(S), \min(S) = -1$.

On **transaction manager node** $i$

1. Write-ahead log of read-write transactions, $\mathcal{L}_i$: All entries on the log have a sequence number, which doubles as an index into the log, and have a type of *PreCommit*, *Executed*, or *Abort*. We defer discussion of these types to the protocol description. For convenience, let $\tilde{\mathcal{L}}_i$ denote the subhistory (not necessarily suffix) consisting of *PreCommit* entries and $\tilde{\mathcal{L}}_i|j$ denote the subhistory of *PreCommit* entries that affect keys belonging to shard group $j$. Observe that $\tilde{\mathcal{L}}_i|j \subseteq \tilde{\mathcal{L}}_i$ and

$$order\left(\tilde{\mathcal{L}}_i|j\right) \subseteq order\left(\tilde{\mathcal{L}}_i\right) \subseteq order(\mathcal{L}_i)$$

   as a consequence of these definitions.[a]

2. FIFO queue for each shard group $j$, $Q_i|j$: Each queue is maintained so that its contents and order are exactly the sequence numbers of $\tilde{\mathcal{L}}_i|j$. Additionally, each queue tracks the log sequence number of the most recently dequeued entry, which we will write as $exec(Q_i|j)$.

3. Map from *PreCommit* transactions to shards, $C_i$: Any given *PreCommit* transaction $T \in \mathcal{L}$ is viewed as a collection of disjoint sub-transactions $\{T|j_p\}_{p=1}^{k}$, where $k \leq m$ and $T|j_p$ is the subset of operations in $T$ that only concern the state on shard $j_p$. This map stores bindings of the form $T.sn \mapsto (shds, cwsn)$, where $shds := \{j_1, ..., j_k\}$ and $cwsn$ is the associated CWSN. For a given *PreCommit* transaction, $shds$ informs us which participating shard groups have not yet executed the necessary operations, and the $cwsn$ enables the update of transaction status in item 5.

4. Map from shards to sequence number, $SSN_i$: In the same way that the client library appends each outgoing transaction with either a CWSN or CRSN, the manager nodes increment $SSN$ on every insert into $Q_i|j$.

5. Map from CWSNs to transactions, $W_i$: This associates a CWSN with either null or its corresponding response $(status, data)$ that has not yet been acknowledged. Define $W_i^{\text{key}}|cid$ as the maximal subset of $W_i^{\text{key}}$ such that all of its elements have prefix $cid$. This additional per-client bookkeeping is essential to guaranteeing exactly-once semantics in the presence of server failures and message loss (§4).

6. Map from $cid$s to $(maxNum, lsn)$ tuple, $R_i$: The component $maxNum$ is the highest CSRN $num$ seen from client $cid$, and $lsn$ is a log sequence number. As we will see in §5, this other form of per-client metadata is used for preserving the invocation-order for reads when messages arrive out of order.

[a] Here, $order(S)$ is the assumed total order over $S$ – a distinguished subset of $S \times S$ that satisfies the usual axioms. The natural order on sequence numbers, inherited from $\mathbb{N}$, induces all orderings we consider herein.

On **shard group** $j$

1. Log sequence number of the most recently completed sub-transaction, $shExec_j$: This is analogous to the queues and $exec(\cdot)$ in the transaction manager nodes.

2. Local sequence number, $ssn_j$: After the completion of any sub-transaction $T|j$ that involves a state mutation, the shard group increments $ssn_j$.

On **client** $cid$

1. Greatest assigned CWSN and CRSN, $cwsn_{\max}$ and $crsn_{\max}$: These are incremented accordingly as the client submits transactions (§4, 5).

2. Ordered set of CWSNs assigned to outstanding transactions, $InProg_{cid}$: This allows computation of the largest CWSN such that all transactions with lower CWSNs have received responses (§4). Set elements are removed on receiving responses.

3. Ordered map from CRSNs to log sequence numbers, $UB_{cid}$: This is a set of upper bounds that ensure the results of a retried read-only transaction do not violate invocation ordering (§5). As above, the client library prunes this map appropriately as it receives responses from the service.

4. Map from CRSNs to response data, $ReadResult_{cid}$: This map is of the form $\{crsn_T \mapsto (num, data, lsn)\}$, where $num$ is the number of participating shards, $data$ contains response data, and $lsn$ is the read's log sequence number constraint (§5).

**Figure 1: Data structures on transaction manager, shard group servers, and clients**

## 4 Simple Read-Write Transactions

We distinguish between two types of read-write transactions: *simple* read-write transactions or *dependent* read-write transactions. For now, we focus on simple read-write transactions; dependent read-write transactions are discussed in section §6. Simple read-write transactions do not rely on the results of any constituent operations to determine the full read/write key-set. For example, there are no dependencies amongst the operations in $T := \{w(x) = 5, r(y), r(z)\}$, so this is of simple type. Another example is a conditional write such as $T := \{\text{if } r(z) \geq 100 \text{ then } w(x) = x - 100\}$. While the write to $x$ might depend on $z$, all keys in the transaction are known.

To submit a simple read-write transaction to the system, the client invokes SessionRuntimeAppend through a wrapper API, which handles timeout and retries.

**Procedure** SessionRuntimeAppend($T$, $isRetry$, $cwsn_T$ | null)

```
   // SessionRuntimeAppend on client cid
 1 if ¬isRetry then
 2 |   cwsn_max := cwsn_max + 1
 3 |   cwsn_T := cwsn_max
 4 ackBound := min(InProg_cid) // could optimize this to clean specific intervals
 5 Send_head(AppendTransact, {T, cwsn_T, ackBound}) // all writes go to head of chain
 6 return cwsn_T
```

Note the variables $cwsn_T$ and $ackBound$. The first of these two guarantees execution preserves invocation order in spite of message reordering, while the second enables the transaction manager to garbage collect no-longer-needed metadata. Additionally, $cwsn_T$ is returned to the client for any possible retries.

**Procedure** AppendTransact($T$, $cwsn_T$, $ackBound$)

```
   // AppendTransact on transaction manager node i ∈ [1, n]
 1 if cwsn_T ≤ max(W_i^key|cid) then
   |   // W_i[cwsn_T] has variant type resp|uint
 2 |   if i = 1 ∧ W_i[cwsn_T] = resp(SUCCESS, data) then
 3 |   |   Send_cid(SessionRespWrite, {cwsn_T, resp})// handle case when i is head
 4 |   return
 5 else
 6 |   wait until max(W_i^key|cid).num + 1 = cwsn_T.num ∧ ind = L_i.len // prevent reordering
 7 end
 8 ind := L_i.len
 9 L_i ← T
10 W_i := W_i ∪ {cwsn_T ↦ null}
11 W_i := W_i \ {(k ↦ W_i[k]) ∈ W_i : k < ackBound} // remove unneeded metadata
12 C_i := C_i ∪ {ind ↦ (cwsn_T, {})}
13 foreach j ∈ [1, m] do
14 |   particip := T^write ∩ sh(j)
15 |   if particip ≠ ∅ then
16 |   |   Q_i|j ← ind
17 |   |   SSN[j] := SSN[j] + 1
18 |   |   C_i[ind].shds := C_i[ind].shds ∪ {j}
19 end
20 if i = n then
21 |   ShDeps := DepAnalysis(T^key) // local and remote dependencies for each participant j
22 |   foreach j ∈ C_i[ind] do
23 |   |   Send_j(ShardExecAppend, {T|j, ind, SSN[j], shDeps[j]})
24 |   end
25 else
26 |   Send_succ(AppendTransact, {T, cwsn_T, ackBound})
27 end
28 return
```

The check in lines 2-5 is necessary, as a dropped response message can lead to multiple client retries. If

the service indiscriminately performs these retries, the resulting execution history might not admit a valid total ordering.

Once the transaction is replicated on the tail node, it is sent to the shard groups for execution. To reduce the size of network messages, only the relevant sub-transaction is transmitted to each participating shard. For transactions whose writes may depend on its read results, the tail also includes the results a read/write set depdendency analysis. This analysis is similar to that presented in Calvin. Retries are piggybacked on subsequent invocations or sent after a preset timeout, whichever happens first.

**Procedure** ShardExecAppend($T|j$, $ind$, $sn$, $deps$)

```
   // ShardExec on shard group j ∈ [1, m]
 1 if sn ≤ ssn_j then
 2 |  return
 3 else
 4 |  wait until ssn_j + 1 = sn // handle manager to shard message reordering
 5 end
 6 recv all deps^local
   // Once dependencies resolved, can execute local operations and serve reads
 7 respData := ∅
 8 foreach op ∈ T|j do
   |  // Calls to storage service will be batched in practice
 9 |  match op with:
10 |  |  put(key, value) ⟹ put(key, value, ind) // versioned put exposed by storage service
11 |  |  get(key) ⟹ respData := respData ∪ {key ↦ get(key)}
12 |  end
13 end
14 foreach op ∈ deps^remote do
15 |  Send_{op.shard}(RemoteRead, {op.key ↦ get(key)})
16 end
17 shExec_j := ind
18 ssn_j := sn
19 Send_tail(ExecNotif, {j, ind, respData})
20 return
```

**Procedure** ExecNotif($j$, $ind$, $respData$)

```
   // ExecNotif on tail node n
 1 if ind ≥ exec(Q_n|j) then
 2 |  deq ← Q_n|j
 3 |  while deq ≠ ind do
 4 |  |  deq ← Q_n|j
 5 |  end
 6 C_n[ind].shds := C_n[ind].shds \ {j}
 7 cwsn := C_n[ind].cwsn
 8 if W_n[cwsn] = null then
 9 |  W_n[cwsn] := resp(INPROG, respData)
10 else
11 |  W_n[cwsn].data := W_n[cwsn].data ∪ respData
12 end
13 if C_n[ind].shds = ∅ then
   |  // Local call to start communication backwards through the chain
14 |  ExecAppendTransact({ind, W_n[cwsn].data})
15 return
```

The tail listens to shard group replies through ExecNotif and updates its $C_n$. After all sub-transactions are finished, transaction completion is propagated backwards to the head through ExecAppendTransact. For each participating shard queue, elements are deqeued up to and including the completed transaction index. This ensures an updated view of all execution statuses, which is critical for correctly serving reads. When the head

is done updating its queues, the response is returned to the client session.

**Procedure** ExecAppendTransact(*ind*, *data*)

```
   // ExecAppendTransact on node i ∈ [1, n]
 1 cwsn := C_i[ind].cwsn
 2 foreach j ∈ C_i[ind].shds : ind ≥ exec(Q_i|j) do
 3 |   deq ← Q_i|j
 4 |   while deq ≠ ind do
 5 |   |   deq ← Q_i|j
 6 |   end
 7 end
 8 C_i := C_i \ {ind ↦ C_i[ind]}
 9 W_i[cwsn] := resp(SUCCESS, data)
10 if i = 1 then
11 |   Send_{cwsn.cid}(SessionRespWrite, {cwsn, W_i[cwsn]})
12 else
13 |   Send_{pred}(TransasctExec, {ind, data})
14 end
15 return
```

**Procedure** SessionRespWrite(*cwsn*, *resp*)

```
   // SessionRespWrite on client cid
 1 InProg_cid := InProg_cid \ {cwsn}
 2 return resp
```

## 5 Read-Only Transactions

SessionRuntimeRead in the client-side library is called on every read-only transaction, including retries. Responses include the log sequence number used in executing the read.

**Procedure** SessionRuntimeRead($T$, $isRetry$, $crsn_T$ | null)

```
   // SessionRuntimeRead on client cid, connected to manager node i
 1 writeDep, lsnConst := −1
 2 isRetry := False
 3 if InProg_cid ≠ ∅ then
 4 |   writeDep := max(InProg_cid)
   // Compute invocation order constraints
 5 if crsn_T = null then
 6 |   crsn_max := crsn_max + 1
 7 |   crsn_T := crsn_max
 8 |   ReadResult_cid := ReadResult_cid ∪ {crsn_T ↦ (−1, {}, lsnConst)}
 9 |   if |ReadResult_cid| > 1 then
10 |   |   UB_cid := UB_cid ∪ {crsn_T ↦ null}
11 else
12 |   isRetry := True
   |   // Check existence of upper bound
13 |   ubKey := min{x ∈ UB_cid^key : crsn_T ≤ x ∧ UB_cid[x] ≠ null}
14 |   if ubKey ≠ null then
15 |   |   lsnConst := UB_cid[ubKey]
16 |   ReadResult_cid[crsn_T] := (−1, {}, lsnConst) // throw away partially completed read
17 end
18 Send_i(ReadOnlyTransact, {T, isRetry, crsn_T, writeDep, lsnConst})
19 return crsn_T
```

The utility of $crsn_T$ is the same as in write-only transactions. For every read-only transaction, *writeDep* captures dependency on the most recent ongoing write. This is not required in all cases. If the write key-set of all ongoing transactions are known, a dependency need only be declared on the latest transaction $X$ such that $X^{\text{write}} \cap T^{\text{read}} \neq \emptyset$. Unfortunately, this optimization is not always possible due to the way we handle read-write and dependent transactions (§6).

**Procedure** SessionRespRead(*data*, $crsn_T$, *fence*, *numShards*)

```
   // SessionRespRead on client cid
 1 (currNum, currData, currLsn) := ReadResult_cid[crsn_T]
 2 if currLsn = −1 ∨ currLsn = fence then
 3 |   if currNum = −1 then
 4 |   |   currNum := numShards
 5 |   currNum := currNum − 1
 6 |   currData := currData ∪ {data}
   |   // Recieved data from all shards, so can return to client
 7 |   if currNum = 0 then
 8 |   |   ReadResult_cid := ReadResult_cid \ {crsn_T ↦ ReadResult_cid[crsn_T]}
 9 |   |   if crsn_T ∈ UB_cid^key then
10 |   |   |   UB_cid[crsn_T] := lsn
11 |   |   |   if crsn_T ≠ crsn_max then
12 |   |   |   |   UB_cid := UB_cid \ {(crsn_T + 1) ↦ UB_cid[crsn_T + 1]}
13 |   |   return data
14 |   else
15 |   |   ReadResult_cid[crsn_T] := (currNum, currData, currLsn)
16 |   end
```

For retries, the client session must also explicitly provide an upper bound on freshness, *lsnConst*, to the transaction manager node. Otherwise, a retry could return information newer than that of later reads, with

respect to invocation order. Consider the set of completed read-only transactions with CRSN greater than $crsn_T$. We take $lsnConst$ as the log sequence number contained in the response to the minimal element of this set. That is, $lsnConst$ can be thought of as the least upper bound on freshness. In practice, we do not track all completed read-only transactions. The set $UB_{cid}$ only contains least upper bounds, which corresponds to transactions whose immediate predecessors have not yet received a response (lines 9-12 of SessionRespRead).

Once a read is retried, responses matching the original invocation must be discarded. Again, this is needed to prevent more recent invocations from matching with a response containing data more stale than older invocations. Observe that this scheme can potentially lead to a large amount of unneccesary, cascading retries. For example, responses for a large set of read transactions may arrive just after all retries are sent out, thereby invalidating their content. However, we expect that manifestation of these behaviors is highly unlikely, assuming a resonable timeout value.

Based on the status of their queues, transaction manager nodes compute a consistent read across shards subject to the client library's $writeDep$ and $lsnConstr$ constraints. This consistent read can be visualized as a "fence" that cuts across the manager's queues.

**Procedure** ReadOnlyTransact($T$, $isRetry$, $crsn_T$, $writeDep$, $lsnConst$)

```
   // ReadOnlyTransact on manager node i ∈ [1, n]
 1 if writeDep ≥ min(W_i^key|cid).num then
 2 |  wait until writeDep ∈ W_i^key or ABORT(writeDep)
 3 fence := −1
 4 ShardSet := ∅
   // Participating shards and preliminary fence computation
 5 foreach j ∈ [1, m] do
 6 |  particip := T^read ∩ sh(j)
 7 |  if particip ≠ ∅ then
 8 |  |  ShardSet := ShardSet ∪ {j}
 9 |  |  fence := max{fence, exec(Q_i|j)}
10 end
   // Modify constraints based on retry flag and R_i
11 if isRetry then
12 |  fence := lsnConst
13 if crsn_T.cid ∈ R_i^key then
14 |  if crsn_T.num ≤ R_i[crsn_T.cid].num then
15 |  |  if ¬isRetry then
16 |  |  |  fence := R_i[crsn_T.cid].lsn
17 |  else
18 |  |  if ¬isRetry then
19 |  |  |  fence := max{fence, R_i[crsn_T.cid].lsn}
20 |  |  R_i[crsn_T.cid] := fence
21 |  end
22 else
23 |  R_i := R_i ∪ {crsn_T.cid ↦ fence}
24 end
   // Sends reads and constraints to shards
25 numShards := |ShardSet|
26 foreach j ∈ ShardSet do
27 |  Send_j(ShardExecRead, {T|j, crsn_T, fence, numShards})
28 end
29 return
```

A shard group simply waits until the manager-determined constraint is satisfied before executing its portion of the read. Shards return directly to the client, $crsn_T.cid$, which originated the request. Because correctness depends only on execution up to a specified log sequence number, writes on the shard can continue uninterrupted in parallel.

**Procedure** ShardExecRead($T|j, crsn_T, fence, numShards$)

```
   // ShardExecRead on shard group j
1  wait until shExec_j ≥ fence
2  data := ∅
3  foreach op ∈ T|j do
       // multi-versioned get exposed by storage service, returns value with greatest
          version less than fence
       // as before, calls batched in practice
4      data := data ∪ {op.key ↦ get(op.key, fence)}
5  end
6  Send_{crsn_T.cid}(SessionRespRead, {data, crsn_T, fence, numShards})
7  return
```

## 6 Dependent Read-Write Transactions

In constrast to the simple case, a transactions is *dependent* if the full read/write key-set depends on some of its own reads. A scenario where this can arise is secondary index lookups. We buffer these types of transactions at the head node, which fully reduces the transaction into a simple read-write transaction. More specifically, the head node makes the necessary read requests to satisfy all intra-transaction dependencies. For deduplication, the head maintains a set $BF$ of all buffered transactions, in addition to the state described in figure 1. A more client-driven approach would enable the *writeDep* optimization described in the previous section, but the tradeoff would be an increase in the number of client to manager node round trips, which runs contrary to our goal of acheiving higher throughput.

Unfortunately, contention and aborts are possible when clients elect to use these types of transaction. Since the reduction process involves one or more reads, any write that affects the read keys must trigger a transaction abort. Otherwise, we would introduce safety violation. We use a priority-based mechnism so that dependent transactions are not constantly aborted in times of high contention. When a transaction aborts, the head sends a notification down the chain, enabling constraint removal on affected reads (line 2 of ReadOnlyTransact). To submit a dependent read-write transaction, the client uses a slightly modified SessionRuntimeAppend, with ReduceTransact replacing AppendTransact in the call on line 5.

**Procedure** ReduceTransact($T$, $cwsn_T$, $ackBound$)

```
1  if cwsn_T ∈ BF then
2      return
3  BF := BF ∪ cwsn_T
4  toRead := DepAnalysis(T^key) // get read dependencies
5  depResults := ∅
6  W_i := W_i \ {(k ↦ W_i[k]) ∈ W_i : k < ackBound} // remove unneeded metadata
7  foreach j, ops ∈ toRead do
8      Send_j(ShardExecHeadRead, {ops, tail(Q_1|j)})
9  end
10 if ABORT(cwsn_T) then
11     BF := BF \ cwsn_T
12     Send_{cwsn_T.cid}(SessionRespWrite, {cwsn_T, (ABORT, ∅)})
13     Send_succ(NotifAbort, {cwsn_T, ackBound})
14     return
15 else
16     recv all depResults
17 end
18 T_reduce := T[depResults] // resolve read/write set
19 AppendTransact(T_reduce, cwsn_T, ackBound)
```

## 7 Proof of Safety

We now turn to proving that the protocol described above guarantees MD-RSS. First, we recall some definitions. *TODO*

**Theorem (Safety):** Any well-formed execution, $\boldsymbol{\sigma}$, of the protocol satisfies MD-RSS.

To prove this theorem, we will first define a total ordering on $\boldsymbol{\sigma}$, using log indices as logical timestamps. We will then show that the resulting sequential execution $\boldsymbol{\sigma}_{\text{ord}}$ is equivalent to $\boldsymbol{\sigma}$, and $\boldsymbol{\sigma}_{\text{ord}}$ is in our desired sequential specification. Finally, we will verify that $\boldsymbol{\sigma}_{\text{ord}}$ satisfies both the RSS and Multi-Dispatch properties.

*Proof of Theorem:* Let $W$ be the set of simple read-write invocation and response pairs in $\sigma$, and let $R$ be the set of read-only invocation and response pairs. Observe that pair in $W$ corresponds to a unique index in the transaction manager log $\mathcal{L}$. Similarly each pair in $R$ has an associated $lsnConst$, which is also an index into the transaction manager log $\mathcal{L}$. Note, however, that for read-only pairs this index may not be unique. In either case, for pair $a \in \sigma$, denote the associated index $\text{ind}(a)$. We can treat these as logical timestamps to define an strict total order over $\boldsymbol{\sigma}$.

Concretely, let $\rightarrow$ be the binary relation over $\boldsymbol{\sigma} \times \boldsymbol{\sigma}$, such that for $a, b \in \boldsymbol{\sigma}$ one has $a \sqsubset b$ if and only if one of the following hold

i.) $\text{ind}(a) < \text{ind}(b)$

ii.) $\text{ind}(a) = \text{ind}(b)$ where $a \in W$, $b \in R$

iii.) $\text{ind}(a) = \text{ind}(b)$ where $a, b \in R$, both are from the same client, and $\text{CRSN}(a) < \text{CRSN}(b)$.

iv.) $\text{ind}(a) = \text{ind}(b)$ where $a, b \in R$, the pairs are from the different clients, and the *cid* of the client who issued $a$ is strictly less than that of $b$ i.e. we arbitrarily break ties.

Claim that this is a well-defined strict total order. *TODO*

## 8 Extensions: Checkpointing and Metadata Garbage Collection