1 System Goals

Many recent transactional database systems, such as Google's Spanner, FaunaDB, and AWS DynamoDB, 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.

Furthermore, 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 *stronger semantics 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 meaining. If the map or set S is empty, then both $\max(S), \min(S)$ equal -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 writes in T that only mutate the state of 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 writes, 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 CWSNs with either their corresponding ongoing write or write response that has not been acknowledged. Define $W_i^{\text{key}}|cid$ as the maximal subset of W_i^{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 cids 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 $T|j$, 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 Write-Only Transactions

First, the client invokes SessionRuntimeAppend through a wrapper API, which handles timeout and retries.

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

`// SessionRuntimeAppend on client` cid
if $\neg isRetry$ **then**
    $cwsn_{\max} := cwsn_{\max} + 1$
    $cwsn_T := cwsn_{\max}$
$ackBound := \min(InProg_{cid})$
$\text{Send}_{\text{head}}(\text{AppendTransact}, \{T, cwsn_T, ackBound\})$ `// all writes go to head of chain`
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` $i \in [1, n]$
1 **if** $cwsn_T \le \max(W_i^{key}|cid)$ **then**
    `//` $W_i[cwsn_T]$ `has variant type resp|uint`
2   **if** $i = 1 \wedge W_i[cwsn_T] = \text{resp}\{\text{STATUS}\}$ **then**
3         $\text{Send}_{\text{cid}}(\text{SessionResp}, \{\text{STATUS}\})$`// handle case when` i `is head`
4   **return**
5 **else**
6   **wait** until $\max(W_i^{\text{key}}|cid).num + 1 = cwsn_T.num$ `// prevent reordering`
7 **end**
8 $\text{ind} := \mathcal{L}_i.len$
9 $\mathcal{L}_i \leftarrow T$
10 $W_i := W_i \cup \{cwsn_T \mapsto \text{ind}\}$
11 $C_i := C_i \cup \{ind \mapsto (cwsn_T, \{\})\}$
12 **foreach** $j \in [1, m]$ **do**
13     $\text{particip} := T^{\text{write}} \cap sh(j)$
14   **if** $\text{particip} \neq \emptyset$ **then**
15         $Q_i|j \leftarrow \text{ind}$
16         $SSN[j] := SSN[j] + 1$
17         $C_i[\text{ind}].shds := C_i[\text{ind}].shds \cup \{j\}$
18 **end**
19 **if** $i = n$ **then**
20   **foreach** $j \in C_i[\text{ind}]$ **do**
21         $\text{Send}_j(\text{ShardExecAppend}, \{T|j, \text{ind}, SSN[j]\})$
22   **end**
23 **else**
24     $\text{Send}_{\text{succ}}(\text{AppendTransact}, \{T, cwsn_T\})$
25 **end**
26 $W_i := W_i \setminus \{(k \mapsto W_i[k]) \in W_i : k < ackBound\}$ `// remove unneeded metadata`
27 **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 using ShardExecAppend. To reduce the size of network messages, only the relevant sub-transaction is transmitted to each participating shard. Similar to before, a shard-specific sequence number sn is also included. Retries are piggybacked on subsequent invocations or sent after a preset timeout, whichever happens first.

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

```
// ShardExec on shard group j ∈ [1, m]
if sn ≤ ssn_j then
    return
else
    wait until ssn_j + 1 = sn
end
foreach op ∈ T|j do
    // abstract put exposed by storage service, these calls will be batched in practice
    put(op.key, op.value)
end
ssn_j := sn
Send_tail(ExecNotif, {j, ind})
return
```

Procedure ExecNotif(j, *ind*)

```
// ExecNotif on tail node n
deq ← Q_n|j
while deq ≠ ind do
    deq ← Q_n|j
end
C_n[ind].shds = C_n[ind].shds \ {j}
if C_n[ind].shds = ∅ then
    Send_tail(TransactExec, {ind})
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*)

```
// TransactExec on node i ∈ [1, n]
cwsn := C_i[ind].cwsn
foreach j ∈ C_i[ind].shds do
    deq ← Q_i|j
    while deq ≠ ind do
        deq ← Q_i|j
    end
end
C_i := C_i \ {ind ↦ C_i[ind]}
W_i[cwsn] := resp{SUCCESS}
if i = 1 then
    Send_cwsn.cid(SessionRespWrite, {cwsn, W_i[cwsn]})
else
    Send_pred(TransasctExec, {ind})
end
return
```

Procedure SessionRespWrite(*cwsn*, *resp*)

```
// SessionRespWrite on client cid
InProg_cid := InProg_cid \ {cwsn}
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
$writeDep, lsnConst := -1$
$isRetry := False$
if $InProg_{cid} \neq \emptyset$ **then**
  $writeDep := \max(InProg_{cid})$
`// Compute invocation order constraints`
if $crsn_T = null$ **then**
  $crsn_{\max} := crsn_{\max} + 1$
  $crsn_T := crsn_{\max}$
  $ReadResult_{cid} := ReadResult_{cid} \cup \{crsn_T \mapsto (-1, \{\}, lsnConst)\}$
  if $|ReadResult_{cid}| > 1$ **then**
    $UB_{cid} := UB_{cid} \cup \{crsn_T \mapsto \text{null}\}$
else
  $isRetry := True$
  `// Check existence of upper bound`
  $\text{ubKey} := \min\{x \in UB^{\text{key}}_{cid} : crsn_T \leq x \wedge UB_{cid}[x] \neq \text{null}\}$
  if $\text{ubKey} \neq \text{null}$ **then**
    $lsnConst := UB_{cid}[\text{ubKey}]$
  $ReadResult_{cid}[crsn_T] := (-1, \{\}, lsnConst)$ `// throw away partially completed read`
end
$ackBound := \min(InProg_{cid})$ `// also piggyback write acknowledgements on reads`
$\text{Send}_i(\text{ReadOnlyTransact}, \{T, isRetry, crsn_T, ackBound, writeDep, lsnConst\})$
return $crsn_T$

The utility of $crsn_T$ and $ackBound$ are 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($crsn_T$, $fence$, $data$, $numShards$)

`// SessionRespRead on client` cid
$(\text{currNum}, \text{currData}, \text{currLsn}) := ReadResult_{cid}[crsn_T]$
if $\text{currLsn} = -1 \vee \text{currLsn} = fence$ **then**
  if $\text{currNum} = -1$ **then**
    $\text{currNum} := numShards$
  $\text{currNum} := \text{currNum} - 1$
  $\text{currData} := \text{currData} \cup \{data\}$
  `// Recieved data from all shards, so can return to client`
  if $\text{currNum} = 0$ **then**
    $ReadResult_{cid} := ReadResult_{cid} \setminus \{crsn_T \mapsto ReadResult_{cid}[crsn_T]\}$
    if $crsn_T \in UB^{\text{key}}_{cid}$ **then**
      $UB_{cid}[crsn_T] := lsn$
      if $crsn_T \neq crsn_{\max}$ **then**
        $UB_{cid} := UB_{cid} \setminus \{(crsn_T + 1) \mapsto UB_{cid}[crsn_T + 1]\}$
    return $data$
  else
    $ReadResult_{cid}[crsn_T] := (\text{currNum}, \text{currData}, \text{currLsn})$
  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 4-5 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, $isRetry$, $crsn_T$, $ackBound$, $writeDep$, $lsnConst$)

`// ReadOnlyTransact on manager node` $i \in [1, n]$

if $writeDep \geq \min(W_i^{key}|cid).num$ **then**

  wait until $writeDep \in W_i^{\text{key}}$

$fence := -1$

ShardSet $:= \emptyset$

`// Participating shards and preliminary fence computation`

foreach $j \in [1, m]$ **do**

  particip $:= T^{\text{read}} \cap sh(j)$

  if particip $\neq \emptyset$ **then**

    ShardSet $:=$ ShardSet $\cup \{j\}$

    $fence := \max\{fence, exec(Q_i|j)\}$

end

`// Modify constraints based on retry flag and` R_i

if $isRetry$ **then**

  $fence := lsnConst$

if $crsn_T.cid \in R_i^{\text{key}}$ **then**

  if $crsn_T.num \leq R_i[crsn_T.cid].num$ **then**

    if $\neg isRetry$ **then**

      $fence := R_i[crsn_T.cid].lsn$

  else

    if $\neg isRetry$ **then**

      $fence := \max\{fence, R_i[crsn_T.cid].lsn\}$

    $R_i[crsn_T.cid] := fence$

  end

else

  $R_i := R_i \cup \{crsn_T.cid \mapsto fence\}$

end

`// Sends reads and constraints to shards`

$numShards := |\text{ShardSet}|$

foreach $j \in ShardSet$ **do**

  Send$_j$(ShardExecRead, $\{T|j, crsn_T, fence, numShards\}$)

end

return

6 Read-Write and Dependent Transactions

A 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-manager node round trips. In view of acheiving higher throughput, we opt not to take this approach.