A Proof of the Correctness Criteria

**Proof** To proof that the criteria for Distributed Snapshot Isolation (DSI) given in this paper in Section 4.2 are sufficient, we first construct all possible global schedules from a set of local schedules \( \{S^1, S^2, \ldots, S^n\} \) where each global schedule \( G \) that is constructed adheres to the criteria in Section 4.2. Then we show that any of these global schedules \( G \) that can be constructed is view-equivalent to each of the local schedules \( S' \in \{S^1, S^2, \ldots, S^n\} \) and that \( G \) is correct according to Snapshot Isolation with regard to the definition of Snapshot Isolation in Section 3.2. We assume that the data is partitioned and not replicated (i.e., the database objects accessed by different local schedules are disjoint). Moreover, we assume that the local nodes provide proper Snapshot Isolation (i.e., each \( S' \in \{S^1, S^2, \ldots, S^n\} \) is correct according to Local Snapshot Isolation).

**Construction of all possible global schedules \( G \).** In order to construct all global schedules \( G \) from a set of local schedules \( \{S^1, S^2, \ldots, S^n\} \), we do a (sorted) merge of the local schedules where the global begin and commit operations of the distributed transactions are fix-points when merging the local schedules. Therefore, we first need to find all global transactions in the set of local schedules \( \{S^1, S^2, \ldots, S^n\} \) (i.e., transactions that appear in multiple local schedules). For those global transactions, we check that all begin and commit operations adhere to the same partial order on every node as defined in Section 4.2. If not, no global schedule \( G \) can be constructed that satisfies the criteria in Section 4.2. Otherwise we construct all possible total orders \( o_{P,1} < o_{P,2} < \ldots < o_{P,n} \) of all global begin and commit operations of all distributed transactions such that each total order adheres to the criteria in Section 4.2.

For each constructed total order \( o_{P,1} < o_{P,2} < \ldots < o_{P,n} \), we then merge the local schedules as follows in order to produce a global schedule \( G \): first, we merge all operations from all local schedules \( S' \in \{S^1, S^2, \ldots, S^n\} \) that contain the first global operation (i.e., \( o_{P,1} \)) in the given total order until (excluding that global operation \( o_{P,1} \)). When merging these operations from all local schedules into the global schedule \( G \), it is important that the order of operations given by a local schedule \( S' \in \{S^1, S^2, \ldots, S^n\} \) must not be changed in \( G \). Then, we continue the given global operation \( o_{P,1} \) from all local schedules and add it once to the global schedule \( G \). Afterwards, we continue this procedure with the next global begin or commit operation \( o_{P,2} \) in the global order until all global operations in the given total order are consumed (i.e., until \( o_{P,n} \) is consumed). At the end, we add the remaining operations of all local schedules to the global schedule \( G \).

We now show that the construction leads to a set of global schedules whereas each global schedule \( G \) in that set is view-equivalent to the local schedules \( \{S^1, S^2, \ldots, S^n\} \). Then we show that any of these global schedules \( G \) is correct according to Snapshot Isolation as defined in Section 3.

*Any constructed global schedule \( G \) is view-equivalent to all local schedules \( \{S^1, S^2, \ldots, S^n\} \):* To show that any of the possible global schedules \( G \) that we constructed before is view-equivalent to the local schedules \( \{S^1, S^2, \ldots, S^n\} \), we have to show that a) all read operations \( r_i(o) \in G \) return the same versions \( v \) of object \( o \) as the corresponding read operation in the local schedules \( \{S^1, S^2, \ldots, S^n\} \) and that b) the final state of the database is the same after executing the schedules, i.e., the last write \( w_i(o) \) of any transactions \( t \in G \) to an object \( o \) is the same as in the local schedules \( \{S^1, S^2, \ldots, S^n\} \). Both rules are trivially satisfied in any of the global schedules \( G \) since data is partitioned and not replicated and the order of all operations in a local schedule \( S' \in \{S^1, S^2, \ldots, S^n\} \) is not changed in any global schedule \( G \) (including the global begin and commit operations).

**Any global schedule \( G \) is correct according to Snapshot Isolation:** We now show that none of the anomalies in Section 3.2 can occur in any of the constructed global schedules \( G \) and its \( SS(G) \). As stated in [1] and quoted in Section 3, Snapshot Isolation holds for a schedule \( G \) if the anomalies G1(a-c) and G-SI(a-b) do not occur in \( G \) and its \( SS(G) \). In the following, we discuss each anomaly for Snapshot Isolation separately.

**G1a:** Aborted Reads. Since local nodes provide proper Snapshot Isolation, any read \( r_i(o) \) of a transaction \( t \) that appears in a local schedule \( S' \in \{S^1, S^2, \ldots, S^n\} \) is mapped to a write \( w_i(o) \) of a transaction \( t \) that committed before transaction \( t \) started (on node \( i \)). Since we assume that data is partitioned (and not replicated) and the order of operations in a local \( S' \in \{S^1, S^2, \ldots, S^n\} \) is not changed in \( G \), all reads in a global schedule \( G \) return only object versions written by committed transactions.

**G1b:** Intermediate Reads. Using a similar argument as above: since local nodes provide proper Snapshot Isolation, a transaction \( x \) always reads the latest version written by a transaction \( y \) that committed last before transaction \( x \) started in a local schedule \( S' \in \{S^1, S^2, \ldots, S^n\} \). Since we assume that data is partitioned (and not replicated) and the order of operations in a local \( S' \in \{S^1, S^2, \ldots, S^n\} \) is not changed in \( G \), transaction \( x \) reads the same version in \( G \) as in \( S' \) (i.e., only committed object versions are read).

**G1c:** Circular Information Flow. Any read- and write-dependency edges from transaction \( x \) to transaction \( y \) in any \( SS(G) \) imply that \( c_x < b_x \) in any local schedule \( S' \in \{S^1, S^2, \ldots, S^n\} \) that contains operations from transaction \( x \) and transaction \( y \) (i.e., \( i \in SN(x,y) \)) since the construction above checks that the same order relation \( c_x < b_x \) holds in any of these local schedule \( S' \) and the order relation \( c_x < b_x \) is also used to construct the global schedule \( G \). There can exist no directed cycle of read- and write-dependency edges between \( x \) and \( y \) in \( SS(G) \) as well. This holds for any transaction \( x \) and transaction \( y \) where \( c_x < b_x \) even if there is no direct read- or write-dependency edge from \( x \) to \( y \) but there is a path of read- and write-dependency edges from \( x \) to \( y \).

**G-SIa:** Interference. A read- or a write-dependency edge in a \( SS(G) \) of any \( \{S^1, S^2, \ldots, S^n\} \) implies that there must also be a start-dependency edge in \( S' \) (i.e., since G-SIa must hold in a correct local \( SS(G) \)). Such a start-dependency edge from transaction \( x \) to transaction \( y \) implies that \( c_x < b_x \) in \( S' \). Since the same order relation also holds in the global schedule \( G \) (by its construction), there is also a start-dependency edge in \( SS(G) \) from transaction \( x \) to transaction \( y \). Thus, G-SIa can not occur in \( SS(G) \) as well.

**G-SIb:** Missed Effects. If there is an anti-dependency edge from transaction \( x \) to transaction \( y \) in a local \( SS(G) \) and SI holds for \( S' \in \{S^1, S^2, \ldots, S^n\} \), this implies that \( b_x \neq c_x \). If \( SS(G) \) would contain the anomaly G-SIb, a cycle with exactly one anti-dependency edge from transaction \( x \) to transaction \( y \) and a path with only read-/write-start-dependency edges must exist from transaction \( x \) to transaction \( y \). We now show that no such cycle can exist in \( G \), if all local schedules \( \{S^1, S^2, \ldots, S^n\} \) are correct under SI.

The proof is by contradiction: Assume there is an anti-dependency edge from transaction \( y \) to transaction \( x \) and a path from transaction \( x \) to transaction \( y \) in \( SS(G) \) consisting only of read-/write-start-dependency edges. Any read-/write-start-dependency edge from a transaction \( x \) to transaction \( t \) on that path implies that \( c_x < b_x \). Thus, if there exists a path of read-/write-start-dependency edges from transaction \( x \) to transaction \( y \) in \( SS(G) \), we can derive that \( c_x < b_x \) must hold by transitivity. However, by the construction of \( G \) which uses a given total order of global begin and commit operations no anti-dependency edge can exist since this would require \( b_y < c_x \). Thus, no such a cycle can exist in \( G \).

This concludes the proof that any global schedule \( G \) that satisfies the correctness criteria in Section 4.2 is correct according to Distributed Snapshot Isolation.
B Algorithms for DSI

B.1 Centralized Coordination

B.1.1 Attributes of a transaction

The following table summarizes the attributes of a transaction x used to implement the Centralized Coordination scheme. All attributes of a transaction are initialized by the global coordinator:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>global-snapshot</td>
<td>Global snapshot read by transaction x (i.e., a global CID) assigned by operation global-begin</td>
</tr>
<tr>
<td>global-tid</td>
<td>A globally unique transaction identifier (i.e., a global TID) used for tagging non-committed tuple versions of x assigned by operation global-begin</td>
</tr>
<tr>
<td>global-cid</td>
<td>A globally unique commit identifier (i.e., a global CID) used for tagging committed tuple versions of x assigned by operation global-commit</td>
</tr>
</tbody>
</table>

B.1.2 Algorithms

The algorithms in Listing 3 show the implementation of two operations executed by the centralized coordinator: global-begin and global-commit. Both operations share the same latch global-latch for synchronization. The operation global-begin acquires the latch, then sets the attribute global-tid as well as global-snapshot of transaction x and then releases the latch. During that time no other transaction can begin or commit. The operation global-commit works as follows: it first acquires the same latch, then issues a new global CID, which is assigned to the attribute global-cid of transaction x to tag its tuple versions. Then the operation executes the synchronous prepare phase (which actually tags the tuple versions) on each database node involved in x. After the prepare phase the latch is released and the asynchronous commit phase is executed.

```
Listing 3 Centralized Coordination: coordinator operations

// global variables in the coordinator
int GLOBAL-TID=0;
int GLOBAL-CID=0;
Lock global-latch = new Lock();

// global begin in coordinator
void global-begin(Transaction x){
    global-latch.acquire();
    x.global-tid = ++GLOBAL-TID;
    x.global-snapshot = GLOBAL-CID;
    global-latch.release();
}

// global commit in coordinator
void global-commit(Transaction x){
    bool success = true;
    x.global-cid = ++GLOBAL-CID;
    // synchronous prepare phase:
    // tag tuple version using global-cid
    for( each node i in N(x)){
        if(!success)
            break;
    }
    global-latch.release();
    // return control to client
    signal_client();
    // asynchronous commit or abort phase
    for( each node i in N(x)){
        if(success)
            local-commit(x, i);
        else
            local-abort(x, i);
    }
}
```

B.2 Pessimistic Coordination

B.2.1 Attributes of a transaction

The following table summarizes the attributes of a transaction x used to implement the Pessimistic Coordination scheme. In contrast to the Centralized Coordination scheme, a transaction does not store global information (i.e., a global snapshot, a global TID and a global CID) but it stores local information for each database node it visits:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>local-snapshot[i]</td>
<td>Local snapshot read by transaction x on node i assigned by operation local-begin</td>
</tr>
<tr>
<td>local-tid[i]</td>
<td>Local TID for tagging non-committed tuple versions on node i assigned by operation local-begin</td>
</tr>
<tr>
<td>local-cid[i]</td>
<td>Local CID for tagging committed tuple versions on node i assigned by operation local-commit</td>
</tr>
</tbody>
</table>

B.2.2 Algorithms

The algorithms in Listing 4 show the implementation of two operations executed by the centralized coordinator: global-begin and global-commit. Both operations share the same latch global-latch for synchronization.

The operation global-begin acquires the latch, then calls the operation local-begin (to assign local information) for every every node x intends to visit and then releases the latch. During that time no other transaction can begin or commit. The operation global-commit works as described before for the Centralized Coordination scheme.

The algorithms in Listing 5 show the implementation of the relevant operations executed by the database nodes: local-begin and local-prepare. The operation local-begin is called by global-begin and local-prepare is called by global-commit. Both operations share the same latch local-latch for synchronization one one database node.

The operation local-begin first acquires the latch, then assigns a local snapshot and TID to transaction x, and finally releases the latch. During that time no other local or global transaction can begin or commit on that node. The operation local-prepare first acquires the same latch, then assigns a new local CID and then tags all tuple version on node i using that CID by calling the method writeCIDinDoubt. As mentioned before, all tuple version still have a status in doubt, which makes them invisible to other transactions since the prepare phase could fail. Finally, the latch is released. The operation local-
Listing 4 Pessimistic Coordination: coordinator operations

```java
// global variables in the coordinator
Lock global-latch = new Lock();

// global begin in coordinator
void global-begin(Transaction x){
    global-latch.acquire();
    for( each node i in SN(x)){
        i.local-begin(x,i);
    }
    global-latch.release();
}

// global commit same as for
// Centralized Coordination
// w/o updating the global-cid

Listing 5 Pessimistic Coordination: database node operations

```java
// variables per database node
int LOCAL-CID=0;
int LOCAL-TID=0;
Lock local-latch = new Lock();

// local begin in a database node i
void local-begin(Transaction x, Node i){
    local-latch.acquire();
    x.local-tid[i] = ++LOCAL-TID;
    x.local-snapshot[i] = LOCAL-CID;
    local-latch.release();
}

// local prepare in a database node i
bool local-prepare(Transaction x, Node i){
    bool success = true;
    try{
        local-latch.acquire();
        x.local-cid[i] = ++LOCAL-CID;
        x.writeCIDinDoubt(i);
    } catch(Exception e){
        success = false;
    } finally{
        local-latch.release();
    }
    return success;
}
```

commit (which is not shown) simply sets all in doubt tuple versions of 
```
x to visible.
```

B.3 Optimistic Coordination

B.3.1 Attributes of a transaction

The following table summarizes the attributes of a transaction \( x \) used to implement the Optimistic Coordination scheme. In contrast to both schemes before, a transaction holds global and local information.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>global-tid</td>
<td>A globally unique TID used as global begin timestamp assigned by operation global-begin</td>
</tr>
<tr>
<td>global-cid</td>
<td>A globally unique CID used as global commit timestamp assigned by operation global-commit</td>
</tr>
<tr>
<td>global-crt</td>
<td>A list of concurrent global transactions of ( x ) modified by operation global-begin</td>
</tr>
<tr>
<td>local-snapshot[]</td>
<td>A local snapshot read by transaction ( x ) on node ( i ) assigned by operation local-begin</td>
</tr>
<tr>
<td>local-tid[]</td>
<td>A local TID for tagging non-committed tuple versions on node ( i ) assigned by operation local-begin</td>
</tr>
<tr>
<td>local-cid[]</td>
<td>A local CID for tagging committed tuple versions on node ( i ) assigned by operation local-prepare</td>
</tr>
</tbody>
</table>

B.3.2 Algorithms

The algorithms in Listing 6 show the implementation of two operations executed by the centralized coordinator: global-begin and global-commit. The operation global-begin is called once to begin a transaction, while global-access is called whenever transaction \( x \) accesses a new database node for the first time. The code for the global-commit operation is not shown since the control flow is the same as for the 2PC of the Centralized Coordination scheme. The only difference is, that the local-prepare operation that is called by the global-commit is assigning local CIDs (which is not the case for the Centralized Coordination scheme). All three operations share the same latch global-latch for synchronization.

The operation global-begin acquires the latch, then it initializes the attribute global-tid with a unique global TID and sets the attribute global-cid to MAX_INT that indicates that a transaction did no yet commit. Finally, the operation initializes the list global-crt with global transactions that are concurrent to \( x \) and releases the latch. This list is updated whenever a new global transaction \( y \) starts while \( x \) has not yet committed.

The operation global-access is called each time, transaction \( x \) accesses a new database node for the first time. This operation first acquires the same latch as the global-begin operation. Next it calls the local-begin operation on node \( i \) that \( x \) wants to access in order to assign the most recent local CID on node \( i \) to the attribute local-snapshot of \( x \) and to get a local-tid attribute for node \( i \). Afterwards, the operation global-access checks if another transaction \( y \) with a global-tid greater than the one of transaction \( x \) has accessed node \( i \) since \( x \) started in order to ensure a proper begin ordering for global transactions. If the begin order is not correct, transaction \( x \) must be aborted (i.e., attribute success is set to false). This is implemented by storing the largest global-tid that accessed node \( i \) in variable last-begin[i]. Afterwards, the operation global-access checks if any of the concurrent global transactions has already committed on the new node \( i \). In that case the transaction \( x \) must be aborted as well. Finally, if all checks are successful last-begin[i] is updated and the latch is released.

Once the operation global-access is called and returns true, transaction \( x \) can read and write data on that node until \( x \) ends without calling global-access again.
Listing 6 Optimistic Coordination: coordinator operations

// global variables in the coordinator
Transaction activeGlobalTAs[];
int GLOBAL–TID = 0;
int GLOBAL–CID = 0;
int last_begin[]; // per node
Lock global–latch = new Lock();

// global begin in coordinator
void global–begin(Transaction x) {  
global–latch.acquire();
x.global–tid = ++GLOBAL–TID;
x.global–cid = MAX_INT;
for (each transaction y in activeGlobalTAs){  
x.global–crt.add(y);
y.global–crt.add(x);
}  
activeGlobalTAs.add(x);
global–latch.release();
}

// global checks in coordinator
// before x accesses node i the first time
bool global–access(Transaction x, Node i){
global–latch.acquire();
bool success = true;
local–begin(x, i);

// check begin order
if (last_begin[i] > x.global–tid) {
global–abort(x);
success = false;
}

// check begin and commit order
if (success){  
for (each Transaction y in x.global–crt) {  
if (i in N(y)) \{ // y also accessed node i
// abort, if x is now serial to y
// but was concurrent to y at its begin
if (y.global–cid < MAX_INT) {  
success = false;
break;
}
}
}

// update begin order
if (success){  
last_begin[i] = x.global–tid;
}
global–latch.release();
return success;
}

// global commit in coordinator
void global–commit(Transaction x){
global–latch.acquire();
// same as for Centralized Coordination
...

// remove x from activeGlobalTAs
activeGlobalTAs.remove(x);
global–latch.release();

// same as for Centralized Coordination
...