VLDB Journal manuscript No.
(will be inserted by the editor)

Distributed Snapshot Isolation

Global Transactions Pay Globally, Local Transactions Pay Locally

Carsten Binnig · Stefan Hildenbrand · Franz Färber · Donald Kossmann · Juchang Lee · Norman May

Received: date / Accepted: date

Abstract Modern database systems employ Snapshot Isolation to implement concurrency control and isolation because it promises superior query performance compared to lock-based alternatives. Furthermore, Snapshot Isolation never blocks readers, which is an important property for modern information systems, which have mixed workloads of heavy OLAP queries and short update transactions. This paper revisits the problem of implementing Snapshot Isolation in a distributed database system and makes three important contributions. First, a complete definition of Distributed Snapshot Isolation is given, thereby extending existing definitions from the literature. Based on this definition a set of criteria is proposed to efficiently implement Snapshot Isolation in a distributed system. Second, the design space of alternative methods to implement Distributed Snapshot Isolation is presented based on this set of criteria. Third, a new approach to implement Distributed Snapshot Isolation is devised; we refer to this approach as *Incremental*. The results of comprehensive performance experiments with the TPC-C benchmark show that the Incremental approach significantly outperforms any other known method from the literature. Furthermore, the Incremental approach requires no a priori knowledge of which nodes of a distributed system are involved in executing a transaction. Also, the Incremental approach can execute transactions that involve data from a single node only with the same efficiency as a centralized database system. This way, the Incremental approach takes advantage of sharding or other ways to improve data locality. The cost for synchronizing transactions in a distributed system is only paid by transactions that actually involve data from several nodes. All these properties make the Incremental approach more practical than related methods proposed in the literature.

Carsten Binnig
DHBW Mannheim
E-mail: carsten.binnig@dhbw-mannheim.de

Stefan Hildenbrand, Donald Kossmann
Systems Group, ETH Zurich

Franz Färber, Juchang Lee, Norman May
SAP AG

Keywords Distributed Databases · Concurrency Control · Snapshot Isolation

1 Introduction

Motivation: Modern database systems must operate in a distributed setting for several reasons. The first reason is scalability with the number of users. In large-scale applications that involve a high workload from many users, it is critical to distribute requests across a potentially large number of machines. Facebook is a prominent example of a system that scales out in this way. Secondly, distributed database technology is important for main-memory database systems. While a database may not fit into the main memory of a single machine, a database may fit into the aggregate main memory of several machines. Even if a database fits into the main memory of a single machine, it is nevertheless often important to deploy such a database as a distributed system on a single machine [26]. With current hardware trends towards many cores and non-uniform main memory access (i.e., NUMA), it is often better to have cores independently serve requests to the same database.

The classic way to organize data in distributed databases is sharding [4]. Sharding is a special form of partitioning. The goal of sharding is to co-locate data that is frequently used together in the same node. For instance, the data of an *Order processing* database could be partitioned by *Customer*, thereby partitioning both the *Order* and *Customer* tables by *customer-id*. This way, transactions that involve only a single customer can be processed by a single node. A special form of sharding are distributed database systems that support multi-tenancy [4]. In this case, all the data of a tenant is stored by one node. Thus, all requests of that tenant are processed by that node, and the data is possibly replicated at other nodes for fault-tolerance.

While sharding is effective in many scenarios, there are always situations in which a transaction involves data from multiple nodes. In the *Order processing* example mentioned in the previous paragraph, for instance, a transaction may aggregate order data from all customers of a certain region. In a multi-tenant system, an administrator may want to change the configuration of a set of tenants stored on different nodes. In such situations, support for distributed transactions is needed that synchronize concurrent reads and updates across nodes. In such scenarios, it is important that transactions that involve only a single node are executed with the highest possible efficiency because these transactions make up for the bulk of the workload. The price for distributed transaction processing should only be paid for (rare) global transactions that indeed involve data from multiple nodes.

Obviously, there has been a vast amount of work on distributed transactions in the past. Bernstein et al. formalized the correctness criterion of one-copy serializability [6]. In the more recent past, two other models have become popular: Snapshot Isolation [5] and Eventual Consistency [32]. Eventual Consistency is appropriate for application domains where users accept to see potentially inconsistent data. However, many applications require stronger consistency. Thus, this paper focuses on Snapshot Isolation and how to implement it best in a distributed database system.

Snapshot Isolation was first devised and implemented as part of the Oracle database system in the late Eighties. As stated in [5], it is not equivalent to serializability because it may result in a phenomenon called *write skew*. Nevertheless, it supports a high level of consistency which is sufficient for most business applications. Its biggest advantage is that it allows for implementations that do not block any read-only transactions. As a result, Snapshot Isolation, if implemented correctly, supports a high level of concurrency. This feature is particularly critical in a distributed database system.

Contributions: The first major contribution of this paper is a precise formalization of Snapshot Isolation for distributed systems. Snapshot Isolation in distributed systems has been studied in the past [27]. As shown in this paper, the definitions in [27] are incomplete because they do not consider an important class of phenomena that can occur in a distributed system.

The second contribution of this paper is to define and explore the design space of possible ways to implement Distributed Snapshot Isolation. The foundation for this discussion is laid by giving a set of criteria that, if met by an implementation, guarantee correct Snapshot Isolation. Even though these criteria are not equivalent to Snapshot Isolation (i.e., the criteria are more restrictive), they can be efficiently implemented in a distributed system as shown in this paper.

The third and arguably most important contribution of this paper is a new approach to implement Distributed Snapshot Isolation, referred to as *Incremental*. The key idea of this approach is to build up the snapshot seen by a transaction *incrementally* as the transaction requests data from different nodes. In contrast, many related approaches to implement Distributed Snapshot Isolation create complete Snapshots *upfront*, thereby leading to high overheads at the beginning of a transaction and/or requiring a high level of a priori knowledge of which transactions access data from which nodes. All known approaches that avoid this upfront investment can result in high abort rates. The Incremental approach can, thus, be seen as a way to combine the advantages of both camps of approaches discussed in the literature.

One particular advantage of the Incremental approach is that it works particularly well for sharded database systems: transactions that involve only data from a single shard can be executed as efficiently with the Incremental approach as in a traditional, centralized single-node database system. Only transactions that involve multiple shards need to pay the price for distributed coordination; in this case, the cost for coordination grows proportionally with the number of shards accessed. Only in the extreme case of transactions that access data from all shards, the Incremental approach performs in the same way as traditional (upfront) approaches to Distributed Snapshot Isolation, but never worse. This paper presents the results of comprehensive performance experiments carried out with the TPC-C benchmark using a commercial distributed database system. These experiments assessed the performance of the Incremental approach as compared to the best known alternative approaches from the literature to implement Distributed Snapshot Isolation.

Outline: The remainder of this paper is organized as follows: Section 2 describes the architecture and system properties assumed throughout this work. Section 3 recaps the definition of Snapshot Isolation using a special form of conflict graphs originally defined in [1]. Section 4 gives the formal foundations of this work based on [1]: a definition of Distributed Snapshot Isolation and a set of correctness criteria for implementing Distributed Snapshot Isolation. Moreover, the properties of an ideal implementation are discussed and the design space of possible approaches is presented. Sections 5 to 7 present enhanced and adopted versions of the approaches in [27] to implement Distributed Snapshot Isolation in a distributed database. Section 8 presents our new proposed approach, the Incremental approach. Section

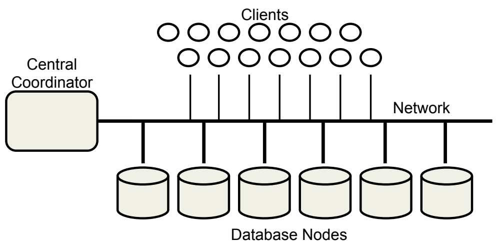


Fig. 1 Distributed Database System

9 discusses the results of performance experiments. Section 10 gives an overview of related work. Section 11 contains conclusions and suggests possible avenues for future work. Appendix A gives a proof for the correctness criteria of Section 4. Appendix B presents algorithms for the approaches to implement Distributed Snapshot Isolation presented in this paper.

2 Background

2.1 System Architecture

The architecture of the distributed database system used in this paper is shown in Figure 1. It is a collection of database nodes each running a (centralized) database system. The data of the distributed database system is *partitioned* (e.g., sharded) over multiple database nodes, whereas single database nodes are virtually (or physically) separated. Furthermore, the architecture involves a central coordinator which is used for handling (distributed) transactions as well as for other tasks such as distributed deadlock detection or for detecting failing nodes. All nodes are connected via a communication network (e.g., Ethernet).

Each database node of the distributed database system implements local snapshot isolation (SI) as described in [19]: The database nodes store modifications of the same tuple using multiple tuple versions (i.e., a MVCC scheme is used for concurrency control). In order to identify a tuple version, a tuple is tagged with a unique transaction identifier (TID) and a commit identifier (CID). While CIDs define the commit order of all transactions, TIDs identify a single transaction without any order. To that end, TIDs are used to tag non-committed tuple versions and CIDs are used to tag committed tuple versions.

When a transaction x begins, a CID is attached to the transaction in order to identify the snapshot the transaction reads from. Write/write-conflicts of concurrent transactions are detected as follows: whenever transaction x wants to update a tuple t it checks whether a new version of the tuple has been committed since x has started based on its snapshot (i.e., the CID it reads from). Thus, if a more recent committed version of that tuple exists, transaction x is aborted early. Moreover, write-locks on a tuple-level are used to avoid potential conflicts with concurrent non-committed transactions: if a transaction y updates a tuple t it first must acquire a write-lock, then it updates t and holds the lock until committing. If a concurrent transaction x wants to update the same tuple t after y has acquired the write-lock, then x waits until y commits or aborts. If y commits, x is aborted and restarted with a new snapshot. Otherwise, if y is aborted x is notified to continue with its old snapshot.

The goal of this work is to find the best way to implement Distributed Snapshot Isolation (DSI) in a distributed database system architecture as described before. In this work, we explicitly study scenarios in which the individual database nodes can collaborate with the central coordinator. Thus, we want to implement new protocols in each database node to support distributed transaction processing. This assumption is in contrast to federated database system designs in which the individual database nodes are considered to be black boxes that cannot be modified. Such a federated design was used in [27] and limits the possibilities to implement Distributed Snapshot Isolation.

In distributed database systems, replication is a common technique to improve performance and availability. There are many ways to effect replication; e.g., read/write quora or primary copies. However, we will ignore replication in this work. Studying the impact of replication on the alternative techniques to implement Distributed Snapshot Isolation is left for future work.

2.2 Expected Workload

As mentioned in the introduction, we would like to optimize for sharded or multi-tenant databases in which most transactions access data from a single node only. In such workloads, only few transactions require consistent access to more than one node or even the whole database, e.g., for updates of the mentioned system data. Those *distributed* or *global* transactions should pay the price for such distributed synchronization and local transactions which make up the bulk of the workload should be processed with the highest possible efficiency.

A typical example for such a shardable workload is the workload simulated by the TPC-C benchmark. In that benchmark, the data can be sharded by warehouse because most transactions access only data from one warehouse (i.e., one node). Only a few transactions require data from different warehouses (i.e., from more than one node). We will use the TPC-C benchmark in our performance experiments (Section 9). However, in our experiments, we modify the TPC-C benchmark to test the robustness of our approaches with regard to a growing percentage of global transactions.

2.3 Terminology and Notation

In this paper, we consider a database as a collection of objects (i.e., attributes of tuples) that can be accessed by transactions. Following the database literature, a *transaction* is a set of totally ordered operations on the objects of the database. Operations are *reads* and *writes* of database objects. Transactions always start with a *begin* operation and end with a *commit* or an *abort* operation. According to Snapshot Isolation, each read accesses a certain version v (i.e., a snapshot) of a database object o whereas writes install a new object version. Moreover, a transaction can modify the same object o multiple times (i.e., a transaction can create multiple versions for the same object).

In terms of notation, this paper uses the letters s,t,x,y,z for transactions. b_x represents the begin of transaction x while c_x represents the commit of transaction x and a_x the abort of transaction x. For reads (r) or writes (w) the subscript on a database object o reflects the version read or installed, respectively, by the operation. That is, $r_x(o_v)$ means that transaction x reads version v of object o. $w_x(o_v)$ indicates that transaction x installed version v of object o.

A superscript i (or j) on an operation of a transaction means that the given operation is executed on a particular node i, e.g., b_x^i is the begin of transaction x on node i. $N(x)$ contains all nodes accessed by transaction x, i.e., all nodes on which x has accessed an object. $SN(x,y)$ contains all nodes accessed (shared) by both transactions x and y, i.e., $SN(x,y) = N(x) \cap N(y)$. If only one node is involved in an example, we might omit the superscript i on the operations. Moreover, if it is clear from the example on which node an operation is executed, we might also skip the superscript i.

A *local transaction* is a transaction that accesses (reads or writes) only database objects from a single node whereas a *distributed transaction* accesses (reads or writes) database objects from more than one node.

A *schedule* (i.e., a history) for a set of transactions (local or global) is defined as a partial order of all operations (i.e., read, write, begin, abort, commit) of all involved transactions. A schedule has to preserve the order of all operations within a transaction including the commit and abort operations. A schedule also defines a *version order* as a total order on the object versions created by committed transactions in a schedule. The version order for one object o contains only those versions for that object. For example, a schedule contains two committed transactions s and t. If transaction s first commits version a_1 and b_1 and then transaction t commits version a_2, then the version order for object a is defined as $a_1 \ll a_2$. We define a *global schedule*, G, as a schedule which may contain operations of transactions from different nodes. Correspondingly, a *local schedule*, S^i, only contains operations from a single node i. If two local schedules S^i and S^j with $i \neq j$ contain operations of the same transaction x, then x is a global transaction.

Finally, we define a partial order of transactions on each node: A transaction x is *before* a transaction y on node i if $c_x^i < b_y^i$. That is, transaction y sees all updates carried out by transaction x on objects stored by node i. We use $x <^i y$ as a short notation. Two transactions x,y are concurrent on node i if $b_x^i < c_y^i$ and $b_y^i < c_x^i$. We use $x||^i y$ as a short notation for this kind of concurrency between x and y. For global schedules, we omit the superscript i and use $x < y$ and $x||y$ respectively to indicate if two transactions are globally serial (all local commits of x come before all local begins of y on all nodes) or concurrent.

All symbols used in this paper are summarized in the table shown in Figure 2.

Symbol	Details
o_v	Database object o in version v
$o_1 \ll o_2$	A version order for database object o
s,t,x,y,z	Local and global transactions (total order of operations in a transaction)
$r_t(o_v)$	Read operation of transaction t reading version v of object o
$w_t(o_v)$	Write operation of transaction t installing version v of object o
op_t^i	Local operation of transaction t on node i (op is one of b, c, a, r, w)
op_t	Global operation of transaction t (op is one of b, c, a, r, w)
S^i	Local schedule on node i (partial order of operations)
G	Global schedule on multiple nodes (partial order of operations)
$N(x)$	All nodes accessed by transaction x
$SN(x,y)$	All nodes accessed (shared) by both transactions x and y
$op\_1_s^i < op\_2_t^i$	Order relation of two operations op_1 and op_2 in a local schedule S^i
$op\_1_s < op\_2_t$	Order relation of two operations op_1 and op_2 in a global schedule G
$s\|\|^i t$	Two concurrent transactions in a local schedule S_i on node i
$s <^i t$	Two serial transactions in a local schedule S_i on node i
$s\|\|t$	Two concurrent transactions in a global schedule G
$s < t$	Two serial transactions in a global schedule G

Fig. 2 Definition of Symbols

3 Local Snapshot Isolation

Snapshot Isolation (SI) is a method of multiversion concurrency control in a single-node DBMS. If a transaction x executes under SI two rules must hold: (1) Read operations of x must read the version v of a database object o (i.e. o_v) which was last committed before b_x. (2) Moreover SI also enforces a second rule to avoid write/write conflicts: The write sets of each pair of committed concurrent transactions x and y must be disjoint.

The implementation of Snapshot Isolation in a single-node system (also called *Local Snapshot Isolation*) has been extensively studied in the past. Many major database products support Snapshot Isolation. Therefore, we can build on the fact that within each node of a distributed system, *Local Snapshot Isolation* is implemented correctly. For the theory and formal definitions of Local Snapshot Isolation, we rely on the framework defined in [1]. We briefly revisit this framework in this section and build ontop of it to define Distributed Snapshot Isolation in the next section.

According to [1], the correctness of an implementation of Local Snapshot Isolation is defined using a so-called *start-ordered serialization graph (SSG)* which is based on dependencies between transactions. Section 3.1 defines the SSG for a single-node system, and Section 3.2 shows how it is used to define Snapshot Isolation in a single-node system.

In this section we omit all superscripts that indicate on which node an operation is executed because all actions are executed on the same node in a single-node system. Definitions for the distributed case are given in Section 4.

3.1 Start-Ordered Serialization Graph

The following gives the definition of the Start-Ordered Serialization Graph (SSG). The SSG was first presented in [1].

Definition 1 (Start-Ordered Serialization Graph) We define the Start-Ordered Serialization Graph arising from a schedule S, denoted SSG(S), as follows. Each node in SSG(S) corresponds to a committed transaction in S and directed edges correspond to different types of dependencies. These dependencies are defined for two committed transactions x and y as follows:

- *Direct Read-Dependency:* A transaction y directly read-depends on transaction x (denoted by $x \xrightarrow{wr} y$) if x installs some object version o_x and y reads o_x.
- *Direct Anti-Dependency:* A transaction y directly anti-depends on transaction x (denoted by $x \overset{rw}{\dashrightarrow} y$) if x reads some object version o_z and y installs o's next version (after o_z) in the version order.
- *Direct Write-Dependency:* A transaction y directly write-depends on x (denoted by $x \xrightarrow{ww} y$) if x installs a version o_x and y installs o's next version (after o_x) in the version order.
- *Start-Dependency:* A transaction y start-depends on x (denoted by $x \xrightarrow{s} y$) if $c_x < b_y$ in schedule S, i.e., if y starts after x commits.

3.2 Definition of Snapshot Isolation

In [1], a schedule S and its $SSG(S)$ are used to define Snapshot Islation as follows:

Definition 2 (Snapshot Isolation) Snapshot Isolation is defined as the absence of the following anomalies in a given schedule S:

- *G1a (Aborted Reads):* A schedule S exhibits anomaly G1a if it contains an aborted transaction x and a committed transaction y such that y has read a version of an object created by x.
- *G1b (Intermediate Reads):* A schedule S exhibits anomaly G1b if it contains a committed transaction y that has read a version of object o written by transaction x that was not x's final modification of o.
- *G1c (Circular Information Flow):* A schedule S exhibits anomaly G1c if SSG(S) contains a directed cycle consisting entirely of read- and write-dependency edges.
- *G-SIa (Interference):* A schedule S exhibits anomaly G-SIa if SSG(S) contains a direct read- or write-dependency edge from x to y without there also being a start-dependency edge from x to y.
- *G-SIb (Missed Effects):* A schedule S exhibits anomaly G-SIb if SSG(S) contains a directed cycle with exactly one anti-dependency edge.

A schedule, S is valid under Snapshot Isolation iff G1(a-c) and G-SI(a-b) are not present in S and its $SSG(S)$. The absence of these anomalies guarantees that the two rules (1) and (2) mentioned at the begin of this section hold for a given schedule S.

4 Distributed Snapshot Isolation

This section defines Distributed Snapshot Isolation (DSI), thereby expanding the definition of Local Snapshot Isolation given in the previous section. In order to avoid potential phenomena that can occur in a distributed database system if only Local Snapshot Isolation is enforced by the individual nodes without any additional coordination, we define a set of correctness criteria that are sufficient to enforce global schedules that are correct according to our definition of Distributed Snapshot Isolation. Furthermore, we present two phenomena that can occur in a distributed database system if it does not implement the correctness criteria presented before. Finally, this section lists a set of properties that a good implementation of Distributed Snapshot Isolation should have, and it discusses the design space of possible implementations. Sections 5 to 8 give a more detailed description of the concrete schemes to implement Distributed Snapshot Isolation that we studied as part of this work.

4.1 Definition of Distributed Snapshot Isolation

The following definition extends Local Snapshot Isolation for distributed systems.

Definition 3 (Distributed Snapshot Isolation) A set of local schedules $S = \{S^1, S^2, ..., S^n\}$ that are each correct under Local Snapshot Isolation is also correct under Distributed Snapshot Isolation if and only if there exists a view-equivalent global schedule G that combines all operations of the given local schedules and the $SSG(G)$ of the global schedule G does not show any of the anomalies G1(a-c) and G-SI(a-b) presented in Section 3.

We use the same definition of view-equivalence as it can be found in textbooks on transaction theory; e.g., [7]. That is, a set of local schedules $\{S^1, \ldots S^n\}$ and a global schedule G are view-equivalent if they have the same reads-from relationships and the same final write sets. Since the reads-from relationships in SI is defined by the version of a database object that is read by a transaction, we use the following definition for view-equivalence:

1. All read operations $r_t(o_v)$ of a transaction t in a local schedule S^i must return the same version v of a database object o as in the global schedule G.
2. All final write operations $w_t(o_v)$ to a database object o of a transaction t in a local schedule have to install the same version v as in the global schedule G.

To prove that a set of local schedules that each conform to Local Snapshot Isolation is correct under Distributed Snapshot Isolation, we need to prove that a view-equivalent global schedule exists that has no anomalies. Typically, several such correct view-equivalent global schedules exist. See appendix A for the proof.

The definition of view-equivalence does not imply that all operations in all S^is are executed in the same order in G. For example, if transaction x does not read or write any object written by transaction y in a local schedule S^i, the begin of transaction x could be moved to either before or after the commit of y in the global schedule without changing the view or the effects of the transactions.

Many other existing definitions for Distributed Snapshot Isolation focus on Snapshot Isolation in replicated databases only. One example is what Lin et al. [21] call *one-copy-snapshot-isolation* as well as other definitions (e.g., in [13, 14]). We discuss that work in Section 10.

4.2 Correctness Criteria

As a first step towards implementing a protocol for Distributed Snapshot Isolation, we develop a set of correctness criteria. If an implementation meets these criteria, then that implementation is correct according to Definition 3. In the literature, there have been proposals [27] that do not meet these criteria and it turns out that these proposals are not correct according to Definition 3.

$$\exists i : c_x^i < c_y^i \rightarrow c_x < c_y \quad (1)$$
$$\exists i : b_x^i < c_y^i \rightarrow b_x < c_y \quad (2)$$
$$\exists i : c_x^i < b_y^i \rightarrow c_x < b_y \quad (3)$$
$$\exists i : b_x^i \leq b_y^i \rightarrow b_x \leq b_y \quad (4)$$
$$c_x < c_y \rightarrow \forall j \in SN(x,y) : c_x^j < c_y^j \quad (5)$$
$$b_x < c_y \rightarrow \forall j \in SN(x,y) : b_x^j < c_y^j \quad (6)$$
$$c_x < b_y \rightarrow \forall j \in SN(x,y) : c_x^j < b_y^j \quad (7)$$
$$b_x < b_y \rightarrow \forall j \in SN(x,y) : b_x^j < b_y^j \quad (8)$$

Fig. 3 Correctness Criteria

Assume that a set of local schedules that are correct according to Local Snapshot Isolation is given. If the rules in Figure 3 are met by all these local schedules, we can construct a global schedule that is view-equivalent to all local schedules and is correct according to Snapshot Isolation (as defined before):

$SN(x,y)$ is the set of nodes that both transactions x and y access.

Rules (1)-(4) construct a partial global order for all begin and commit operations of all distributed transactions. Rules (5)-(8) enforce the same partial order on all local nodes. Appendix A contains a proof that shows that the correctness criteria are sufficient.

Note that it is not sufficient to define only a set of combined rules of the form $\exists i : c_x^i < c_y^i \rightarrow \forall j \in SN(x,y) : c_x^j < c_y^j$ in all four variants of begins and commits. The intermediate step of constructing the global partial order is necessary. To see why, consider three nodes with three transactions and the following order: on node 1: $x < y$, on node 2: $y < z$, and on node 3: $z < x$. Since there is always only one node that two transactions access together, the combined rules are trivially true. But from a global perspective there is a cycle since $x < y < z$ (transitive from nodes 1 and 2) and at the same time $z < x$ (on node 3). Thus only the complete set of eight rules above construct a partial global order in which all distributed transactions are registered and basically constructs the transitive closure over all distributed transactions on all nodes.

One small exception from these formal rules is allowed: if there is no commit operation (either locally or globally) between two begin operations, the order of these begin operations may be changed (i.e., this allows simultaneous begins). Consider this example: transaction x begins on node 1 and transaction y begins on node 2. Then transaction x also begins on node 2. If now transaction y also accessed node 1, the order of the begin operations is not the same on the two nodes. This is allowed if on one of the two nodes no commit happened between the two begin operations (because the two transactions read the same snapshot if no commit happened in between and therefore this does not influence the correctness).

If the above rules are satisfied for all transactions x, y and all nodes i in the system that are accessed by the the two distributed transaction x and y together, a set of local schedules that are correct according to Local Snapshot Isolation is correct according to Distributed Snapshot Isolation. Note that the rules are sufficient but not necessary due to ignoring the freedom from the reads-from-relationship, i.e., there are correct schedules that do not fulfill these rules. Especially rule 8 will lead to rejected schedules (i.e., aborts of single transactions) that are correct according to Definition 3. Alternative solutions take the reads-from relationship into account, which is expensive to monitor. Furthermore, this is difficult to formalize since the reads-from relationship needs to be evaluated in a transitive way on all nodes. However, as the experiments show, the restrictive variant of the rules has little negative impact on throughput for our expected workloads.

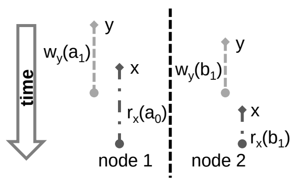
Finally, to see why the global order is only partial let us look at the following scenario: Assume that there exists a partitioning of the nodes of a distributed database such that one set of transactions accesses only nodes from one partition and another set of transactions accesses a completely different set of nodes, then no order needs to be defined between those transactions since they are completely independent from each other.

4.3 Phenomena in a Distributed Setting

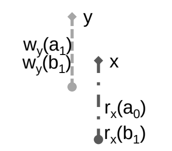
This section presents two phenomena in the distributed setting if the individual nodes only enforce Local Snapshot Isolation and no centralized coordination is applied. For these phenomena we show that all possible view-equivalent global schedules contain anomalies that are disallowed for Distributed Snapshot Isolation as described in Definition 3.

Serial-Concurrent-Phenomenon: One phenomenon that can occur in a federated database was already identified in [27]. This phenomenon is called *serial-concurrent* phenomenon. As shown by the local schedules in Figure 4[1], the phenomenon occurs if a transaction x runs concurrent to another transaction y on node 1 (i.e., $x||^1y$) and serial to y on node 2 (i.e., $x <^2 y$). Therefore, transaction x does not read what transaction y writes on node 1 but reads what y wrote on node 2.

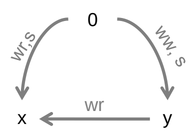
[1] For all figures in this section, we skip the node information on the actions since it is clear on which node an action is executed. Moreover, all transactions shown in any schedule are committed transactions (i.e., the begin is shown as rhombus while the commit is shown as a circle).



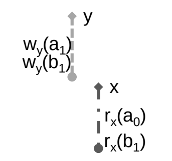
(a) Local schedules



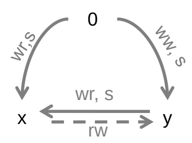
(b) Global schedule G1



(c) SSG(G1)



(d) Global schedule G2



(e) SSG(G2)

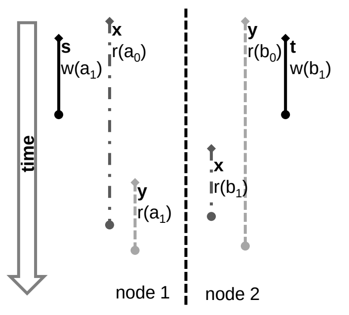
Fig. 4 Serial-Concurrent-Phenomenon

This happens if transaction y commits before transaction x starts on node 2.

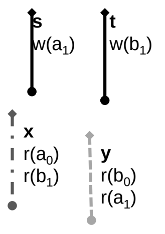
As the two transactions x, y in Figure 4 are concurrent on node 1 and serial on node 2 there are only two classes of global schedules that are view-equivalent with the two local schedules. The global schedules $G1$ and $G2$ in Figure 4 (b) and (d) represent one instance for each class of view-equivalent global schedules that can be constructed representing the version order $a_0 \ll a_1$ for object a and $b_0 \ll b_1$ for object b. Transaction 0 in this figure is the initial transaction that installs version 0 as the initial database state.

In the two classes of global schedules, the two transactions x, y are either concurrent (as shown by the global schedule $G1$ in Figure 4 (b)) or serial (as shown by the global schedule $G2$ in Figure 4(d)). However, neither of the two classes of view-equivalent global schedules is correct according to Distributed Snapshot Isolation as described in Definition 3. In order to prove this, we show for both classes of global schedules that the Start-ordered Serialization Graphs $SSG(G1)$ and $SSG(G2)$ contains at least one of the disallowed anomalies.

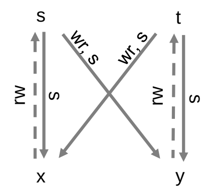
Figure 4(c) shows the Start-ordered Serialization Graph $SSG(G1)$ for the concurrent case: The definition of Distributed Snapshot Isolation requires that there is no interference (G-SIa in Section 3). This means that if there is a read- or write-dependency (wr/ww) edge from transaction y to transaction x, there has to be a start-dependency (s) edge from transaction y to transaction x. However, this is not the case



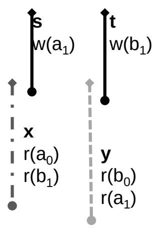
(a) Local schedules



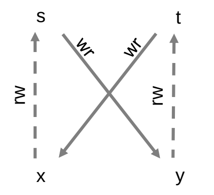
(b) Global schedule G1



(c) SSG(G1)



(d) Global schedule G2



(e) SSG(G2)

Fig. 5 Cross-Phenomenon

in Figure 4(c) since there is a read-dependency edge from transaction y to transaction x but there is no start-dependency from transaction y to transaction x. Figure 4(e) shows the Start-ordered Serialization Graph $SSG(G2)$ for the serial case. Now the schedule contains missed effects (anomaly G-SIb in Section 3) because there is a cycle consisting of exactly one anti-dependency edge)

Cross-Phenomenon: Another phenomenon that can arise in a distributed setting was not considered in the literature before. Figure 5 presents this phenomenon which we call the *cross-phenomenon*. In the example in Figure 5, we see again two local schedules which are each correct according to Local Snapshot Isolation but none of the possible global schedules is correct with regard to Distributed Snapshot Isolation.

The problem arises since the distributed transaction x does not read from the local transaction s on node 1 and reads from the local transaction t on node 2. At the same time, the distributed transaction y reads from transaction s on node 1 and does not read from transaction t on node 2. More formally, we assume $x||^1s$, $x||^1y$ and $s <^1 y$ on node 1 and $y||^2t$, $y||^2x$ and $t <^2 x$ on node 2. As shown in Figure 5, this phenomenon can result in y reading a fresher version of a than x on node 1 whereas x reads a fresher version of b than y on node 2. In other words, x and y see different global snapshots even though they run concurrently on all nodes. Furthermore, there is no way to define a total order on all snapshots seen by transactions in this example which is an important requirement for temporal queries such as those suported in Oracle Flashback and many other commercial database systems that are based on Snapshot Isolation.

In the following, we show that our theory captures this inconsistency and that all possible view-equivalent global schedules for these two local schedules contain at least one anomaly which is disallowed with regard to our definition of Snapshot isolation. The global schedules $G1$ and $G2$ in Figure 5 (b) and (d) represent one instance for different classes of view-equivalent global schedules that can be constructed.

Figure 5(b) shows one way to combine the two local schedules in a global schedule $G1$ which is view-equivalent as defined in the Section before. Figure 5(c) shows the corresponding $SSG(G1)$ (edges from the initial transaction 0 are omitted for readability). This variant is not a correct schedule according to Distributed Snapshot Isolation because it has missed effects (G-SIb in in Section 3): The $SSG(G1)$ contains two directed cycles with exactly one anti-dependency (rw) edge and one start-dependency edge. In order to generalize this observation, any global schedule with $s < x$ or $t < y$ will contain at least on such cycle.

In order to get rid of this cycle, one could rearrange the global schedule as shown in Figure 5(d) by global schedule $G2$ which is also view-equivalent to the local schedules. Again, Figure 5(e) shows the $SSG(G2)$. In the new schedule, the start-dependency edges that caused the cycle disappeared. There are no more start-dependency edges from transaction x to transaction t and from transaction y to transaction s. But, at the same time, the start-dependency edges from transaction x to transaction s and transaction y to transaction t disappeared as well. Now the schedule contains interference (G-SIa in [1]) because there are read-dependency edges from transaction s to transaction x and transaction t to transaction y without start-dependency edges. This problem occurs in any global schedule with $s||y$ or $t||x$.

Note that a mixture of the two discussed global schedules $G1$ and $G2$ will not be correct either since in any other variant, both issues occur at the same time: we will not get rid of the missed effects completely while introducing interference. Furthermore, any global schedule where $y < s$ or $x < t$ does not make sense because then transaction x (respectively transaction y) reads a version that is not yet written. These three classes cover all possible schedules, thus none of the possible variants are correct according to Distributed Snapshot Isolation.

The cross-phenomenon is only an issue if the transactions x, y actually read something that transactions s, t write. However, monitoring the reads-from relationship is expensive in a distributed database. Thus, in order to avoid the *cross-phenomenon*, the system has to make sure that the order of the begin operations of distributed transactions is the same on all involved nodes. This phenomenon was not considered in [27] since the local transactions are (usually) not visible to the central coordinator in a federated system.

Approaches / Properties		Local vs. Global	Low Abort Rate	Low Initial Overhead	High Fault Tolerance
Centralized (w/o a priori)		no	☺	😐	☹
Pessimistic	w/o a priori	no	☺	☹☹	☹
	w local/ global	yes	☺	☹	☺
	w nodes	yes	☺	😐	☺
Optimistic	w/o a priori	no	☹	☺	☹
	w local/ global	yes	☹	☺	😐
Incremental (w/o a priori)		yes	☺	☺	😐

Fig. 6 Approaches and Properties

4.4 Goals and Design Space

In addition to the correctness criteria discussed above, an approach which implements Distributed Snapshot Isolation should have the following properties:

1. **Local vs. Global:** Local transactions should only have to pay the costs for local coordination while global (distributed) transactions should pay the overhead for global coordination. In practice, we expect the majority of transactions to be local and we wish to leverage the locality of multi-tenant databases and sharding.
2. **No A priori Knowledge:** Ideally, transactions do not need to provide any a priori knowledge of whether the transaction is a local or a global transaction and in the global case which nodes a global transaction will access. Consequently, transactions should be able to start as a local transaction and on the fly be able to upgrade to a global transaction and incrementally access data from other nodes.
3. **Low Abort Rate:** Ideally, approaches for DSI should have a low abort rate as well as low overhead to start new transactions (see next point). Typically, optimistic concurrency protocols tend to trade a high abort-rate for low initial overhead. Thus, if the workload does not contain many conflicts optimistic protocols are better suited than pessimistic protocols.
4. **Low Initial Overhead:** Approaches for DSI, which should favor low rates of global transactions, should also have low initial synchronization overhead for starting global transactions. The rationale is that we then can expect to have only a few conflicts for global transactions. Thus, an optimistic scheme with low initial overhead will result overall in a higher throughput than a pessimistic scheme which tries to avoid all potential conflicts by paying a higher cost for initial synchronization. For example, while the pessimistic scheme described in Section 6 accesses all nodes at transaction begin to construct a consistent global snapshot, the optimistic scheme in Section 7 just reads the most recent snapshot and incrementally checks for actual conflicts.
5. **High Fault Tolerance:** Ideally, an approach which implements DSI has a high fault tolerance (i.e., no single point of failure exists). However, approaches for distributed transaction handling often need to rely on a central coordinator which is a single point of failure. In an ideal case, active global transactions should be able to continue running even if the central coordinator fails while local transactions are not affected at all.

Figure 6 summarizes the four different approaches for Distributed Snapshot Isolation and their properties that are presented in the following sections of this paper. All these approaches rely on a central coordinator which is e.g. responsible to execute an atomic distributed commit protocol (e.g., 2PC). The approaches differ in the way how they guarantee the correctness criteria presented in the section before.

It can be observed that there is no perfect way to implement Distributed Snapshot Isolation. However, our novel Incremental scheme does well in all regards. The Optimistic approach gets close, but it requires a priori knowledge of which transactions are global. The experiments presented in Section 9 confirm this result.

5 Centralized Coordination

5.1 Overview

As shown in [19] the simplest approach to implement Distributed Snapshot Isolation (DSI) is with a centralized coordinator scheme. This scheme implements Distributed Snapshot Isolation in similar way to a single-node database system using globally unique commit identifiers (CIDs). The idea is that *all* transactions are synchronized by a central coordinator when they begin and commit, no matter whether they are local or global: The main task of the central coordinator is to assign a global snapshot to every transaction x when it begins (i.e., the coordinator attaches the most recent committed global CID to x) and to coordinate the commit of x (i.e., the coordinator assigns a new globally unique CID to x for tagging its tuple versions). Figure 7 shows a typical call sequence when a transaction begins and commits successfully in a distributed database system with two database nodes.

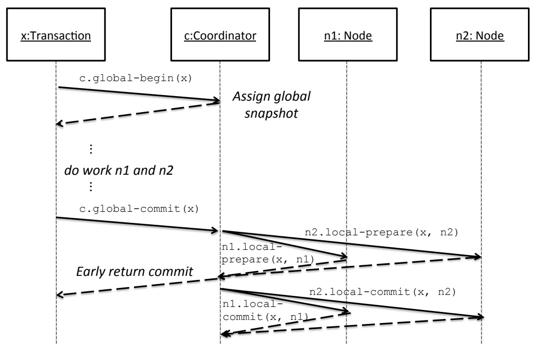


Fig. 7 Sequence Diagram: Centralized Coordination

In order to begin a new transaction *x*, the transaction calls the operation *global-begin(x)* of the central coordinator. This operations assigns a snapshot to *x* that it reads from based on the most recent committed snapshot (i.e., the last issued global CID) to transaction *x*. Moreover, a global TID is assigned to *x* as well, which is used to tag the non-committed tuple versions on node *i*. After calling the *global-begin* operation, transaction *x* can execute its read- and write-operations on all nodes of the distributed database system without contacting the central coordinator anymore.

In order to commit a transaction *x*, the transaction calls the operation *global-commit(x)* of the central coordinator, which first assigns a new globally unique CID to *x* that is used to tag all the tuple versions created by *x*. Afterwards, the central coordinator executes an optimized two phase commit protocol (2PC) using a synchronous prepare phase and an asynchronous a commit phase. All optimizations of the 2PC protocol such as early commit notification after the prepare phase are discussed in [19] in detail. The same optimized 2PC protocol is used for all other approaches presented in the following sections as well.

One important aspect of the 2PC is that a transaction *x* tags its created tuple versions using the global CID already in the prepare phase on every database node it visited (i.e., when the operation *local-prepare* is called). However, the newly tagged tuple versions still have the status *in doubt* after tagging, which makes them invisible for other transactions since some database nodes still might fail during the prepare phase (e.g., due to a node failure or network partitioning). Once the prepare phase finished, the *local-commit* operation is called on every database node involved in a transaction. This operation finally makes all tuple versions with the status *in doubt* visible to other transactions. Write-locks to detect write-write conflicts are also released in the commit phase by every database node.

Another important aspect of the Centralized Coordination scheme is that write/write-conflicts are detected locally (and not globally) as described in Section 2.1). However, additionally a global deadlock detection mechanism is implemented as described in [19].

The listing in appendix B.1 shows the algorithms that are relevant to implement the Centralized Coordination scheme in detail (including latches for synchronization). While this approach implements Snapshot Isolation in a similar way as a centralized system, compared to a centralized system it is able to farm out computation and query processing, as well as conflict detection to all the nodes of the distributed database system, thereby taking advantage of all the resources as long as the centralized coordinator does not become a bottleneck.

5.2 Correctness

With regard to concurrency control, Centralized Coordination behaves exactly like a centralized system. As a result, this approach trivially enforces all criteria for Distributed Snapshot Isolation (Definition 3 and the Rules 1-8 of Section 4.2).

5.3 Discussion

The Centralized Coordination scheme has the following properties:

1. **Local vs. Global:** Local and global transactions are not differentiated. All transactions (including local transactions) have to access the central coordinator to begin, commit and abort. Thus, the central coordinator can become a bottleneck.
2. **A priori Knowledge:** The protocol does not require any a priori knowledge since local and global transactions are handled in the same way in the system.
3. **Abort Rate:** Central coordination has a low abort rate. Transactions are only aborted as a result of write/write-conflicts that happen on a single node. Again, this approach behaves like Local Snapshot Isolation in this regard.
4. **Initial Overhead:** This protocol has a high initial overhead since all transactions need to access the central coordinator at transaction begin and to commit.
5. **Fault Tolerance:** Since all transactions rely on the central coordinator, no transaction can begin or commit while the central coordinator is down. During a failure of the central coordinator, active transactions can continue until they attempt to commit.

6 Pessimistic Coordination

6.1 Overview

The Centralized Coordination approach assigns a global snapshot to every transaction centrally using a globally unique CID. In contrast, with the *Pessimistic* Coordination scheme no globally unique CID exists to identify global snapshots. Instead every database node independently provides its own local snapshots using locally unique CIDs. The main idea of the *Pessimistic* Coordination scheme is to assign local snapshots for all nodes involved in a global transaction atomically whenever a global transaction begins. That way, local transactions can run independently without any central coordination by using the local snapshots of one node. A global transaction, however, needs to contact the central coordinator for coordinating the global begin and commit operations. Figure 8 shows a typical call sequence when a global transaction begins and commits successfully using the Pessimistic Coordination scheme in a distributed database system with two database nodes where no a priori knowledge is provided.

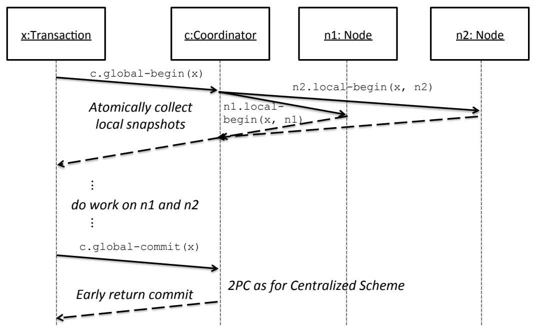


Fig. 8 Sequence Diagram: Pessimistic Coordination

Whenever a new global transaction *x* begins, it first calls the *global-begin* operation in the central coordinator. This operation atomically calls a *local-begin* operation on each node that transaction *x* intends to visit. The *local-begin* operation then assigns a local snapshot to *x* based on its local CIDs. Moreover, a local TID is assigned to *x* as well per database node, which is used to tag non-committed tuple versions on each node. Afterwards, a transaction *x* can execute on all database nodes that it acquired a local snapshot from without contacting the central coordinator. One important aspect is that write/write-conflicts are detected locally per database node during transaction execution the same way as it is done for the Centralized Coordination scheme.

At commit time, the central coordinator carries out a 2PC as described for the Centralized approach before. The only difference is that no global CID is assigned to a global transaction by the central coordinator. Instead each database node locally assigns a local CID per database node by calling the *local-prepare* operation (see Listing 5 in appendix B.2). All tuple versions are tagged with the new local CID in the *local-prepare* operation but again they do have a status *in doubt* until they are set visible by the *local-commit* operation.

In contrast to the Centralized Coordination scheme, in the Pessimistic Coordination scheme a local transaction does not need to contact the central coordinator when beginning and committing. However, a transaction needs to know upfront if it is global or local and if it is global which nodes it will visit. Otherwise, if this a priori knowledge is not available, a transaction must be conservative and must always contact the central coordinator to get a snapshot from all database nodes in the distributed database system. Moreover, once a transaction has already started it can not expand this set of nodes while running.

The listings in appendix B.2 show the algorithms that are relevant to implement the Pessimistic Coordination scheme in detail (including latches for synchronization). The basic idea of this approach was already presented in [27] using sub-transactions to represent global transactions in a federated database system. In this paper, we adopted the idea such that global transactions use a set of local CIDs (i.e., one for each node) to represent a global snapshot. This variant has less overhead then starting a new sub-transaction on each single database node that is involved in a global transaction.

6.2 Correctness

The Pessimistic approach enforces rules (1-8) in Section 4.2 by explicitly ordering the begin and commit operations using the central coordinator. Since no other global transaction can begin or commit while the transaction context is prepared for another global transaction *x* or during the commit phase of a global transaction, all begin- and commit-operations are executed in the same order on all nodes. Thus, rules (1-8) in Section 4.2 are again trivially true for all global transactions.

6.3 Discussion

The *Pessimistic* Coordination scheme has the following properties:

1. **Local vs. Global:** The advantage of this approach is that it can handle local and global (distributed) transactions in a different way, and thus allows the local transactions to run without any additional overhead.
2. **A priori Knowledge:** The disadvantage is that the system requires more knowledge about the transactions in advance (i.e., if a transaction is local or global and which

nodes it will access). Even if this knowledge is available, which in many scenarios is not the case, the client protocol (and therefore the application code) needs to be changed to have the application communicate this knowledge to the database. Such a change is difficult to introduce in existing systems.

3. **Abort Rate:** *Pessimistic* Coordination has a low abort rate. Transactions are only aborted due to write/write-conflicts that happen on a single node like with local Snapshot Isolation.
4. **Initial Overhead:** This protocol has a very high initial overhead if no a priori knowledge is available. In this case the system has to behave conservatively and issue a local-begin operation for all transactions (local and global) on all nodes. If a priori knowledge is available the begin has to be coordinated only for global transaction on the involved nodes that are accessed by the transaction. Compared to the *Centralized* approach, a single begin operation of a distributed transaction is much more expensive since local nodes have to be accessed to assign a local snapshot on each involved node.
5. **Fault Tolerance:** Without any a priori knowledge all transactions are global and all transactions require the coordinator to begin as in the *Centralized* approach. Thus, without a priori knowledge neither global nor local transactions can begin or commit if the central coordinator fails. However, if a priori knowledge is provided (i.e., whether a transaction is local or global), local transactions do not need to contact the central coordinator at all, and thus can be executed normally in case that the central coordinator fails. Moreover, active global transactions can continue running as well as long as they do not need to expand to other nodes. However, active global transactions cannot commit until the central coordinator is recovered.

7 Optimistic Coordination

7.1 Overview

In the *Optimistic* Coordination scheme, every database node independently provides its own local snapshots using local CIDs in the same way as for the *Pessimistic* Coordination scheme. However, different from the *Pessimistic* Coordination scheme, a global transaction does not atomically collect a set of local snapshots when it begins. Instead, the most recent committed local snapshot of a database node is used by a global transaction, whenever it first accesses that database node. In order to provide correct distributed snapshot isolation, a transaction therefore needs to contact the coordinator in the following situations: when a global transaction *x* begins, incrementally whenever *x* first accesses a new database node and finally when *x* commits. That way, the coordinator monitors the order of all global begin and commit operations on every database node and checks potential violations of one of the rules (1-8) of Section 4.2. In order to be efficient, a transaction needs to know upfront if it is global or local. However, a major advantage over the *Pessimistic* Coordination scheme is that a transaction does not need to know which nodes is going to access. Instead it can incrementally expand to other database nodes. Figure 8 shows a call sequence when a global transaction begins, accesses data on two different database nodes and then commits successfully.

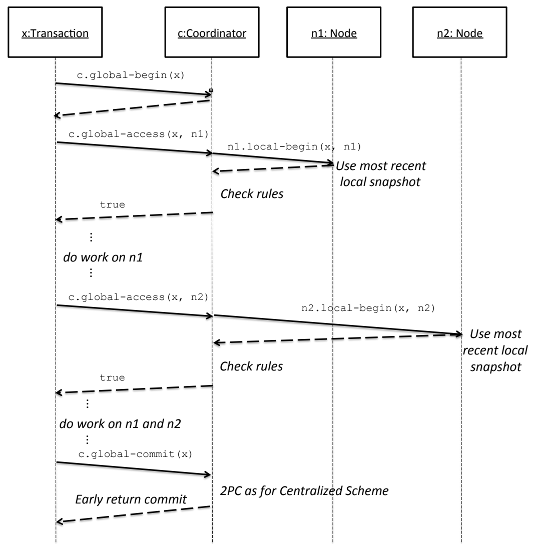


Fig. 9 Sequence Diagram: Optimistic Coordination

When a global transaction *x* begins, it calls the operation *global-begin* of the central coordinator before it accesses data on any database node. This operation assigns a globally unique TID to *x*, which represents a begin timestamp for all global transactions. Moreover, the operation *global-begin* also constructs a list of global transactions that are concurrent to *x* at the moment when it begins. In contrast to the Centralized and the Pessimistic Coordination scheme, the transaction needs to call the operation *global-access* in the coordinator, before a global transaction *x* is allowed to execute on a database node *i*.

A transaction can extend its scope to a new database node *j* while running by calling *global-access* for that node, which is not possible in the approaches discussed before. The *global-access* operation assigns the latest local committed snapshot on node *i* to transaction *x* by calling the *local-begin* operation on that node. This *local-begin* operation is the same as for the *Pessimistic* approach (see Listing 5 in appendix B.2). Afterwards, the operation *global-access* performs the following two checks to find potential violations of DSI. If one of these checks fails, transaction *x* is aborted.

1. First, it ensures that the order of global begin operations is the same on every database node accessed by transaction x by using the last begin timestamp for each database node.
2. Second, it uses the list of global transactions concurrent to x to check if the order of all global begin and commit operations is the same on all shared nodes.

For committing a global transaction, the Optimistic Coordination scheme uses the same *global-commit* operation as the Pessimistic Coordination scheme before.

The listing in appendix B.3 shows the algorithms that are relevant to implement the Optimistic Coordination scheme in detail (including latches for synchronization). Our version of the *Optimistic* approach for DSI was again inspired by the Optimistic approach presented in [27]. Both versions are similar in the aspect that they both involve a central coordinator to keep information about the relationship of concurrent distributed transactions. However, our approach extends the approach in [27] such that it is not vulnerable to the *cross-phenomenon* (Section 4.3). To this end, the central coordinator additionally records for each node i, which global transaction started last on that node. This information is used to ensure that the global begin order is the same on all nodes.

7.2 Correctness

We now show that the *Optimistic* Coordination scheme enforces an order among all global operations: as stated above, in the Optimistic approach, a global transaction has to tell the system, that it is distributed beforehand or otherwise all transactions must be treated as a global transaction.

Thus, if a distributed transaction x begins the operation *global-begin(x)* is called (see Listing 6 in appendix B.3). This operation assigns a unique begin timestamp for the begin of x.

If transaction x wants to access a node i the first time, the operation *global-access(x, i)* is called (see Listing 6 in appendix B.3). Using this operation, the coordinator enforces that the same order of global begins holds on all nodes. This is implemented by using the global begin timestamp $global - tid$ and the timestamp of the last begin on node i $last\_begin[i]$. This guarantees that rules (4) and (8) in Section 4.2 hold.

Moreover, the operation *global-access(x, i)* executes the following check for any global transaction y that has not committed before the global begin-timestamp of transaction x: If transaction y committed on node i already but was marked as *concurrent* by the *global-begin* operation for transaction x, then x is aborted. Consequently, once the *concurrent* relationship between two transactions x and y is established, it must be obeyed on all accessed nodes until x commits otherwise the transaction x is aborted. This guarantees that rules (2-3) and (6-7) in Section 4.2 hold.

Finally, the use of an atomic commit protocol also enforces an order among all commit operations of distributed transactions. This guarantees that rules (1) and (5) in Section 4.2 hold.

Compared to the *Optimistic* approach described in [27], we additionally monitor the order of the global begin operations (and not only the relationship between global transactions), and thus we are not vulnerable to the cross-phenomenon.

7.3 Discussion

The *Optimistic* Coordination scheme has the following properties:

1. **Local vs. Global:** Similar to the Pessimistic approach, local transactions can be handled separately and run independently from distributed transactions.
2. **A priori Knowledge:** The major advantage over the Pessimistic approach is that distributed transactions can access any nodes without having to tell the system in advance. However, at the beginning of a transaction a priori knowledge is needed in order to determine whether a transaction is local or distributed. If no a priori knowledge is available, the Optimistic approach must treat all transactions as distributed transactions, thereby paying the performance penalty for distributed transactions for all transactions.
3. **Abort Rate:** The main disadvantage of the Optimistic approach is the central coordinator aborts many distributed transactions when they access a new node for the first time. Consequently, starvation could be a problem in this approach with long-running distributed transactions.
4. **Initial Overhead:** This protocol has a much lower initial overhead compared to the Central Coordination and Pessimistic approaches because distributed transactions incrementally pay the costs for coordination when they access data on a new node.
5. **Fault Tolerance:** This approach has similar properties with regard to fault tolerance as the *Pessimistic* approach (e.g., no begins and commits of global transactions are possible if the central coordinator fails). Moreover, an additional restriction is that global transactions cannot expand to other nodes while the central coordinator is not available.

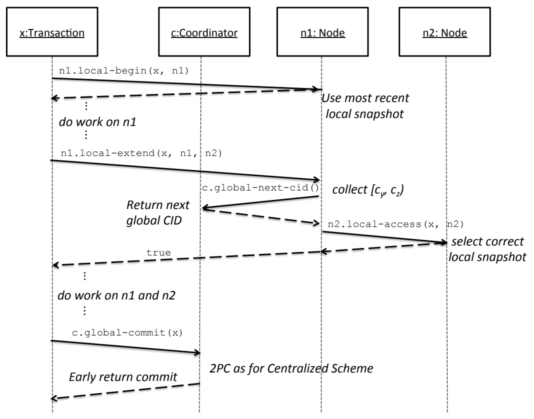


Fig. 10 Sequence Diagram: Incremental Snapshots

8 Incremental Snapshots

8.1 Overview

The *Incremental* Snapshots scheme is a novel approach to enforce DSI in distributed databases. The main idea is similar to the Optimistic Coordination scheme: a transaction can start locally on a database node and incrementally extend its snapshot to other database nodes. A major difference to the Optimistic Coordination scheme is that a global transaction does not choose the most recent local snapshot, whenever it first accesses a database node, but it selects a local snapshot that does not violate any of the rules presented in Section 4.2. If there exists no such snapshot, the transaction is aborted. Consequently, the *Incremental* Snapshots scheme has a significantly lower abort rate than the Optimistic Coordination scheme. Moreover, this approach does not need any a priori knowledge of whether a transaction is local or global and which nodes a transaction will access. Figure 10 shows a diagram of a call sequence when a global transaction x begins, accesses data on two different nodes and then commits successfully.

Compared to all other approaches discussed before, a transaction x can start locally without any central coordination. In order to begin a transaction x, the transaction simply calls the *local-begin* operation of the database node which it first accesses, which can be any node in the distributed database system. In Figure 10 this is node $n1$. The *local-begin* operation assigns the most recent local snapshot that exists on node $n1$ to transaction x (as well as a local TID). Afterwards, transaction x can execute read and write operations on node $n1$.

When transaction x decides to access another node (called the target node), the transaction calls the operation *local-extend* on the source node (i.e., $n1$ in our case) to collect the information about its snapshot on that node. This information consists of an interval $[c_y, c_z)$ of global CIDs. This interval represents a range of global CIDs that a transaction x is allowed to read from without violating any of the rules in Section 4.2. When global transaction x accesses the target node (i.e., $n2$ in our case), transaction x needs to read from a snapshot within the same global CID interval. The operation *local-extend* is called once in the lifetime of a transaction x, whenever x decides to get global to access data on a second node. For accessing subsequent nodes this operation does not need to be called anymore.

After collecting the global CID range for transaction x on node $n1$, the *local-access* operation is called on the target node $n2$ to actually select the local snapshot for the given global CID range. Each database node therefore stores a mapping from global CIDs to local CIDs of global transactions that committed on that node. For some cases, that we discuss later in this section, it might not be possible to select a snapshot that does not violate any of the rules presented in Section 4.2. In this case transaction x must be aborted.

Finally, to commit the global transaction x, the same 2PC protocol is used as mentioned for the Centralized Coordination scheme before.

8.2 Implementation

The following table summarizes the attributes of a transaction x used to implement the Incremental Snapshot scheme. Again, as for the Optimistic Coordination scheme a transaction keeps global and local information. As global information, a transaction stores the attribute *global-cid-range*, which represents those global CIDs a transaction x is allowed to read from on every database node in the cluster. Moreover, the attribute *global-cid* represents the actual global CID attached to x when it commits, which is used for creating the mapping of global CIDs to local CIDs in each database node accessed by x. The local information stored by transaction x is the same as for the Pessimistic and the Optimistic approach before.

Attribute	**Details**
global-cid-range[]	The global CID range $[LOW, UP)$ to select a local snapshot
global-cid	A globally unique CID used for mapping global to local snapshots assigned by the operation *global-commit*
local-snapshot[i]	A local snapshot read by x on node i assigned by the operation *local-begin* for the first node and by the operation *local-access* for all subsequent nodes
local-tid[i]	A local TID for tagging non-committed tuple versions on node i assigned by the operation *local-begin* for the first node and by the operation *local-access* for all subsequent nodes
local-cid[i]	Local CID for tagging committed tuple versions on node i assigned by the operation *local-prepare*

Listing 1 and Listing 2 show the operations executed by the database nodes in the *Incremental* Snapshots scheme. The variables defined in Listing 1 are shared with the operations shown in Listing 2. The methods of the central coordinator are not shown since it only implements a *global-commit* operation in this scheme, which is the same as for the Centralized Coordination approach. However, the *local-prepare* operation that is called by the *global-commit* is different from the other approaches as we will explain in the following. As discussed before, the operation *global-begin* is not needed in the central coordinator at all since any transaction (local and global) can start without any central coordination.

The operation *local-begin* shown in Listing 1 is called only once on that database node *i* where a transaction *x* starts. This operation first acquires the *local-latch* and then initializes the attributes *local-tid* and *local-snapshot* for node *i*. Moreover, the operation updates the variable *readsFromGcid* in the database node *i* to store the information from which global snapshot *x* read from (represented by the variable *LAST-GCID*) . Finally, operation *local-begin* initializes the attribute *global-cid-range* using the last committed global CID *LAST-GCID* as lower bound and *MAX_INT* as upper bound and then releases the latch.

The operation *local-prepare* is called by the *global-commit* operation (which sets the attribute *global-cid* of transaction *x*). The operation *local-prepare* is similar to the operation *local-prepare* implemented by the Pessimistic and the Optimistic approach: All tuple versions created by a transaction *x* are tagged by with the new local CID calling the operation *writeCIDinDoubt*, which sets the status of all tagged tuples to *in doubt* (i.e., they are still invisible for other transactions). After a successful prepare phase, the *local-commit* operation then simply sets all *in doubt* tuple versions of *x* to visible.

However, one difference is that the operation adds a mapping from attribute *global-cid* to *local-cid[i]* of transaction *x* to the variable *gcid2lcid* on database node *i*. This mapping is used on node *i* for selecting a local snapshot for a given CID range. Another difference is that the *local-prepare* operation refines the upper bound *global-cid-range[UP]* for all global transactions *y* that read from the *LAST-GCID*.

Listing 2 shows two additional operations of a database node: the *local-extend* and the *local-access* operation. The operation *local-extend* is called only once in the lifetime of a transaction *x* on the first node that *x* accesses in order to refine the global CID range before it accesses a second node. The operation *local-access* is called every time a transaction accesses a new database node in order to select the local snapshot on that node.

In the following, we describe how the upper bound of the CID range for a transaction *x* is refined by the operation *local-extend*. For determining the upper bound *global-cid-range[UP]* of *x*, two cases are possible: initially, when calling the *local-begin* operation, the upper bound for *x* is set to *MAX_INT*. However, if another global transaction *y* on node *i* has committed next after *x* started (and before *local-extend* is called), the upper bound *global-cid-range[UP]* is already set to be the global CID of that transaction *y* by the *local-commit* operation. If the upper bound is still set to *MAX_INT*) when the *local-extend* operation is called for *x*, the idea is to ask the central coordinator what global CID

Listing 1 Incremental Snapshots: database node operations (part 1)

```
//variables per database node
int LOCAL-CID=0;
int LOCAL-TID=0;
int LAST-GCID=0;

//map: upper bound -> designated local snapshot
Map<int,int> gcidup2lsnap = new Map<int,int>();
//map: global CID -> local CID (commit)
Map<int,int> gcid2lcid = new Map<int,int>();
//map: global CID -> txs that started afterwards
Map<int,List<Transaction>> readsFromGcid =
  new Map<int,List<Transaction>>();

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;
  readsFromGcid[LAST-GCID].add(x);
  x.global-cid-range[LOW] = LAST-GCID;
  x.global-cid-range[UP] = MAX_INT;
  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);

    //get x.global-cid set by global-commit
    int newGcid = x.global-cid
    gcid2lcid[newGcid] = LOCAL-CID;
    for(each y in readsFromGcid[LAST-GCID]){
      y.global-cid-range[UP]=newGcid;
    }
    LAST-GCID = newGcid;
  }
  catch(Exception e){
    success = false;
  }
  finally{
    local-latch.release();
  }
  return success;
}
```

Listing 2 Incremental Snapshots: database node operations (part 2)

```
//collect interval on node i to accesses node j
bool local-extend(Transaction x, Node i, Node j){
  bool success = true;
  local-latch.acquire();
  int gcid-up = x.global-cid-range[UP];
  int gcid-low = x.global-cid-range[LOW];
  if(gcid-up==MAX_INT){
    local-latch.release();
    gcid-up = global-nextCid();
    local-latch.acquire();
  }
  //if some other global tx committed
  //while latch was released
  if(LAST-GCID<gcid-up &&
     LAST-GCID>gcid-low){
     gcid-up = LAST-GCID;
  }
  x.global-cid-range[UP] = gcid-up;

  //abort tx x if x used another
  //local snapshot in the given interval
  if(gcidup2lsnap.containsKey(gcid-up) &&
    gcidup2lsnap[gcid-up] !=
    x.local-snapshot[i]){
    success=false;
  }
  //use local snapshot as global one
  //for the upper limit  gcid-up
  else{
    gcidup2lsnap[gcid-up] = x.local-snapshot[i];
  }

  //call local-access on node j
  //to select a local snapshot there
  if(success){
    j.local-access(x, j);
  }

  local-latch.release();
  return success;
}

//local access node j for the first time
void local-access(Transaction x, Node j){
  local-latch.acquire();
  int gcid-up = x.global-cid-range[UP];
  //use designated local snapshot
  //for global CID range
  if(gcidup2lsnap.containsKey(gcid-up)){
    x.local-snapshot[j] = gcidup2lsnap[gcid-up];
  }
  x.local-tid[j] = ++LOCAL-TID;
  //define designated local snapshot
  //using only the upper bound of the
  //global CID range
  else{
   x.local-snapshot[j] =
       local-find-snapshot(gcid-up, gcid2lcid);
    gcidup2lsnap[gcid-up] = x.local-snapshot[j];
  }
  local-latch.release();
}
```

it will assign *next* (by calling the operation *global-nextCid*). The communication with the coordinator is asynchronous, i.e., the local latch is released which allows other global transactions to commit on node *i*. Therefore, it is necessary to check again locally once the reply from the coordinator is received, if another global transaction z committed in the meantime. If yes, the global CID of transaction z is used as upper bound. If a global transaction z commits that did not access node i, then this transaction z does not need to be considered.

Once the global CID range is collected, the global transaction transforms the information about the global CID interval into a concrete local CID by calling the operation *local-access* on node *j*. To select a local snapshot, every node keeps a mapping between committed global and local CIDs in variable *gcid2lcid*. When selecting a snapshot, the system needs to ensure that all begins of global transactions are properly ordered. The Incremental approach does this in a simple but restrictive way: within one interval $[c_y, c_z)$ (or more precisely, for each distinctive *end* of an interval c_z) only one local snapshot per node is allowed to be used: For each database node, the first global transaction with a given end of the global CID interval defines the local snapshot to be read and all other global transactions with the same end of the interval have to use the same snapshot on that node. In order to implement that the same local snapshot is selected we satisfy the following two criteria: (1) if a local snapshot has already been selected for the upper bound c_z, the map *gcidup2lsnap* is used to map upper bound global CID c_z to the same local snapshot (i.e., a local CID). (2) Moreover, when a new local snapshot must be found for a given global CID range $[c_y, c_z)$, the operation *local-find-snapshot* actually selects a local snapshot in the more restrictive interval $[c_z - 1, c_z)$. This has the following consequence: It may happen that a local transaction cannot become global because in the same interval $[c_y, c_z)$ another local snapshot is already chosen by another transaction with the same upper bound c_z. Therefore, the local transaction is not allowed to become distributed and gets aborted.

This restriction is necessary to avoid the cross-phenomenon. Consider the following example (see Figure 11): Assume that we have two transactions x and y, which incrementally want to become global in a database system with two database nodes ($n1$ and $n2$). First, transaction y becomes global from node $n2$ using the CID interval $[0, 1)$ and selects a local snapshot on node $n1$ that includes the changes committed by a local transaction s (i.e., $c_s^{n1} < b_y^{n1}$). Afterwards, transaction x becomes global from node $n1$ using the same CID interval $[0, 1)$ and selects a local snapshot on node $n2$ that includes the changes committed by a local transaction t (i.e., $c_t^{n2} < b_x^{n2}$). Moreover, $c_s^{n1} > b_x^{n1}$ and $c_t^{n2} > b_y^{n2}$ holds since x started before s on node $n1$ and y started before t on node $n2$. Since different local snapshots are chosen for the

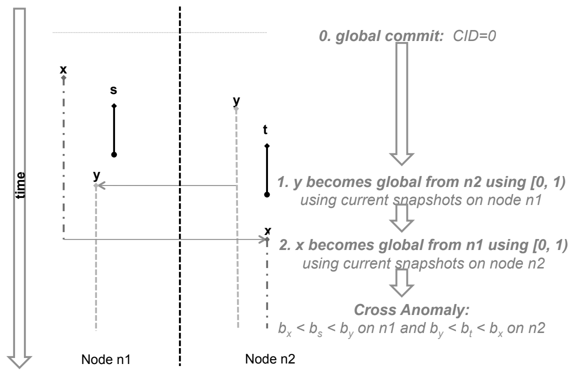

0. global commit: CID=0
x
s
y
t
time
y
1. y becomes global from n2 using [0, 1)
using current snapshots on node n1
x
2. x becomes global from n1 using [0, 1)
using current snapshots on node n2
Cross Anomaly:
$b_x < b_s < b_y$ on n1 and $b_y < b_t < b_x$ on n2
Node n1
Node n2

Fig. 11 Cross Anomaly: Wrong Snapshot Selection

global transactions x and y in the same global CID interval $[0,1)$, the *Cross Anomaly* occurs. Based on the rules defined in Section 4.2, the global begin operations have a different order on node $n1$ and $n2$ (i.e., $b_x^{n1} < b_y^{n1}$ and $b_y^{n2} < b_x^{n2}$)

In order to avoid this anomaly, we currently support only one *designated* local snapshot within the same global CID interval on each node, which is selected by the first transaction that becomes global in that interval. Thus, in our example, we would need to abort transaction x since transaction y already selected the local snapshot on node $n1$ before transaction x wants to become global. Of course this is very restrictive but our experiments show that this does not lead to an increased abort rate for many scenarios. One possible optimization is that a transaction knows a priori if it will become global. The transaction can then inform the local database node during its begin of transaction that it wants to access data from another node. The database node can then ensure that the same local snapshot is assigned to all transactions in the same CID interval.

8.3 Correctness

The *Incremental* approach enforces an order among all distributed operations for the following reasons: the order among global commits is enforced using the atomic commit protocol (rules (1) and (5) in Section 4.2). The order between global begins and commits is enforced because all local begins of a global transaction are scheduled before the same global CID. Thus, Rules (2-3) and (6-7) in Section 4.2) hold. The order between global begins is not directly enforced but the number of possible begins between two global CIDs on one database node is restricted to one snapshot and therefore the order among all begin operations holds on all nodes. Thus, for any two global begin operations b_x^i and b_y^i of transactions that started in the same CID range, the relationship $b_x^i = b_y^i$ holds. Thus, Rules (4) and (8) in Section 4.2 hold as well.

8.4 Discussion

The Incremental coordination scheme has the following properties:

1. **Local vs. Global:** Local transactions can run independently from distributed transactions.
2. **A priori Knowledge:** The major advantage over the Optimistic and Pessimistic approach is that transactions do not need to provide any a priori knowledge whether transactions are local or distributed.
3. **Abort Rate:** The disadvantage is that the expansion to other nodes for a distributed transaction does not work in all cases, which leads to a higher abort rate than the *Centralized* or *Pessimistic* approach but a lower abort rate than for *Optimistic* approach since Incremental uses a correct local snapshot according to DSI if it exists.
4. **Initial Overhead:** This approach has the lowest initial overhead compared to all other approaches since any transaction can start locally without any central coordination involved.
5. **Fault Tolerance:** A local transaction can still be executed while the central coordinator is not available. Moreover, as in the Optimistic approach, distributed transactions can continue running as well as expand incrementally to other nodes while the central coordinator is not available in case that the central coordinator is not involved to determine the interval of valid commit timestamps or the global transaction already knows the interval when the coordinator fails. However, in order to commit the central coordinator must be available again. Since the central coordinator does not store any transaction information (other than the last global commit timestamp) it can be recovered easily.

8.5 Optimizations

The *Incremental* approach solves the issue of the order of the begins in a very restrictive way. This may lead to an increased abort rate because not all local transactions can become distributed. The experiments show that this is not an issue in many scenarios. For scenarios where this increased abort rate is an issue, we briefly sketch a possible extension of the Incremental approach. The idea is to enumerate the possible snapshots between two distributed commits on a node in an order-preserving way. To achieve this with only local information, a combined enumeration is used: the commits of the distributed transactions serve as the major clock tick and are available on all nodes. The minor clock tick uses a simple order-preserving enumeration of the local commits. Not all commits are enumerated but only those that are used by distributed transactions. If the enumeration scheme supports updates, the system can still allow local transactions to

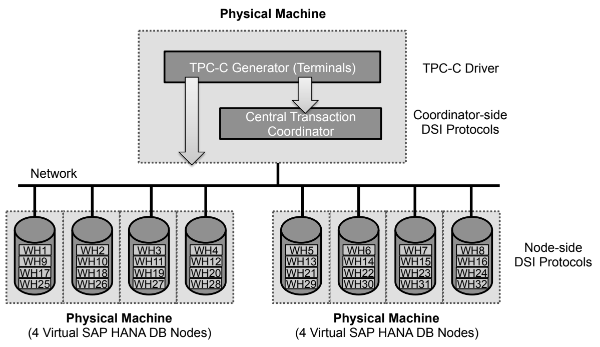


Fig. 12 Benchmark setup

become distributed. The combined identifier of global commit ID and local enumeration can then be used on another node. The only requirement is that on all accessed nodes, the order of snapshots is the same. Depending on how well the used enumeration scheme supports updates, almost all local transactions can now become distributed. One possibility for such an order-preserving enumeration is to use integers and leave gaps to enable future insertions. This is a very limited variant but might be sufficient for a number of scenarios. For more demanding scenarios, other more flexible order-preserving enumeration schemes can be used, e.g., Dewey Codes [24]. We do not discuss this extension further in this paper as the experiments show that the Incremental approach with a single snapshot per interval and node performs quite well.

9 Evaluation

In this section, we summarize the results of experiments that compare the performance of all the approaches presented in Sections 5 to 8. These experiments show that the *Incremental* approach significantly outperforms all other approaches for our targeted usage scenario with few distributed transactions and no a priori knowledge. Furthermore, these experiments show that the *Incremental* approach has the same or slightly better performance in all other cases; e.g., with an increasing fraction of distributed transactions. For these experiments, we used a variant of the TPC-C benchmark [31]. We used a variant of the TPC-C benchmark in order to simulate scenarios with a varying fraction of distributed transactions. (In the original TPC-C benchmark, sharding works extremely well and almost all transactions can be executed on a single node.)

9.1 Benchmark Environment

For the performance evaluation, a variant of the TPC-C benchmark [31] is used. In the experiments we used $8-32$ virtualized SAP HANA database nodes (DBs), whereas we deployed four virtualized database nodes on the same physical machine with 32 cores (4 Intel Xeon X7560 CPUs with hyper-threading) and 256 GB main memory. SAP HANA is a main-memory database system and the main memory was large enough to hold the entire database so that disk I/O was not relevant for all experiments reported in this paper.

For the experiments, we extended the code base of SAP HANA which implements local snapshot isolation (SI) (as described Section 2.1). Moreover, in SAP HANA the coordinator stores the metadata about data distribution and individual database nodes cache the metadata of their database objects. Therefore a transaction uses the local metadata caches to detect if it needs to become distributed. Only the first access of a database node needs a round trip to the master node (i.e., the transaction coordinator), which stores the metadata persistently. For implementing the different DSI approaches, we added our protocols used for beginning and committing transactions to the database nodes and the central coordinator as depicted Figure 12. For the communication of all nodes (coordinator and virtualized database nodes) a real physical network link is used. Moreover, we restricted each virtualized database node to not use more than 4 threads in parallel. Thus, the fact that the virtualized database nodes are not in all cases physically separated does not impact the performance results.

Another server with 8 cores (2 Intel Xeon X5450 CPUs without hyper-threading) and 16 GB of main memory is used as the driver machine (i.e., executing the terminals): it runs a Java implementation of the TPC-C framework based on prepared statements (i.e., no stored procedures are used). On the same machine, we also deployed the central SAP HANA transaction coordinator. A setup for 8 virtualized database nodes is depicted in Figure 12. Having the TPC-C terminals and the coordinator on the same machine favors DSI approaches, where terminals frequently contact the coordinator. In our experiments these are the Pessimistic as well as the Optimistic approach with no a priori knowledge and the centralized approach, which need to contact the coordinator for all transactions (local and global).

For all experiments, we populated each virtual database node with four warehouses (i.e., the database was sharded by warehouses). For distributing the different warehouses (WHs) to the virtual database node a round-robin scheme was used (as defined in the TPC-C specification). Finally, for each virtual database node, we created four terminals (i.e., simulated clients). We ran the TPC-C for 1 minute to warm-up and then for 10 minutes; we only report results for the 10 minutes. During both the warm-up and real experimental phases we ran all five different types of transactions defined in the benchmark.

We varied the fraction of transactions that are distributed. To do so, we had to slightly change the benchmark in the following way: According to the TPC-C specification, 10% of the new orders contain at least one item that is shipped from a remote warehouse. We used this parameter to vary

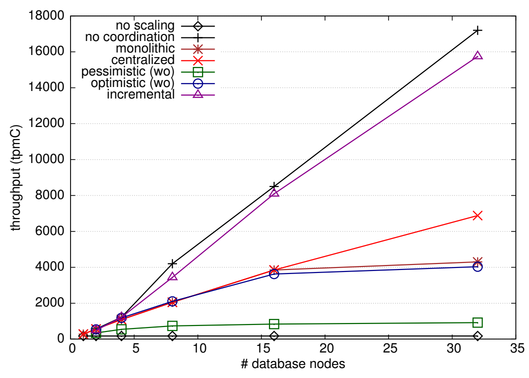


Fig. 13 Throughput (tpmC): vary # database nodes (no a priori, 1-32 database nodes, 4-128 WHs, 5% distributed)

the fraction of distributed transactions. In our variant of the TPC-C benchmark, x% of the new orders contain at least one item that is shipped from a remote warehouse on another virtual database node (i.e., the new-order transaction becomes distributed). The payment transaction can also be distributed. The fraction of distributed payment transactions is not changed except that in the case of 0% distributed new-order transactions, all payment transactions are local as well. In all other cases, the default setting of the specification is used, i.e., 15% of all payment transactions are distributed and always involves two nodes. A similar variant of TPC-C was used in [17].

9.2 Experiment 1: Horizontal Scalability

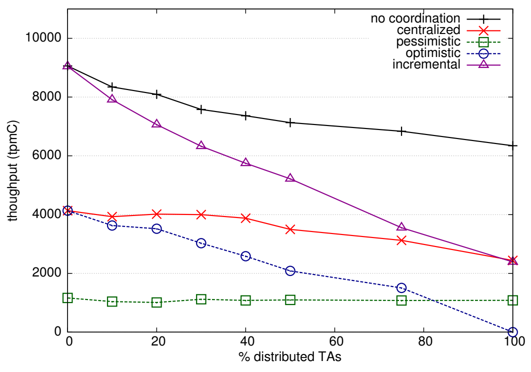
In the first experiment, we studied a typical scale-out scenario where the data size (represented by the number of warehouses) is growing proportionally with the number of nodes. In this experiment, SAP HANA had no a priori knowledge of which nodes were affected by a transaction. We expect this case to be the most common scenario. We varied the number of virtual database nodes from 1 to 32 running on 1 to 8 physical machines as described in the previous section. Each database node stored 4 warehouses resulting in setups ranging from 4 warehouses (1 database node) to 128 warehouses (32 database nodes). Moreover, a database node in this experiment was restricted to use only 4 threads corresponding to the power of a weak commodity machine with 4 cores. We ran the TPC-C benchmark with 5% distributed new-order transactions. We chose this setting because 5% distributed transactions are common for many SAP HANA deployments for sharded, multi-tenant applications.

Figure 13 shows the throughput results (i.e., tpmC) for this experiment. This figure shows that the *Incremental* approach clearly outperforms all other approaches. In fact, it comes extremely close to the ideal (denoted as "no coordination" in Figure 13) which carries out transactions without any synchronization and, thus, results in inconsistencies. In the ideal case (i.e., *no coordination*), each database node is also limited to use only 4 threads.

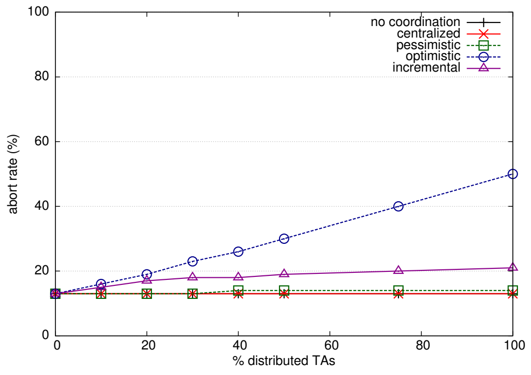
The second best approach in this experiment was the *Centralized* Coordination scheme described in Section 5. This approach scaled better and outperformed the *Optimistic* and *Pessimistic* approaches for more than 16 database nodes. The reason is that the costs for transaction coordination in the *Optimistic* and *Pessimistic* scheme (without a priori knowledge) is much higher than for the *Centralized* approach: In the *Pessimistic* approach the coordinator needs to contact all database nodes in the system when a transaction begins. The *Optimistic* approach has less overhead than the *Pessimistic* approach since it only contacts those database nodes actually involved in a transaction and not all nodes. However, compared to the *Centralized* approach, the *Optimistic* approach has higher network costs since the coordinator needs to contact the database nodes each time a transaction first accesses a database node whereas the *Centralized* approach does not require any network communication with the database nodes to begin a transaction. As shown later (Experiment 3), the *Pessimistic* and *Optimistic* approaches rely heavily on the availability of a priori knowledge at the beginning of a transaction of which database nodes are involved in the transaction (i.e., which data the transaction is going to access).

In addition to "ideal" (no coordination), we added two additional baselines: *no scaling* represents a configuration in which the number of database nodes is fixed to 1. As shown, this configuration is clearly outperformed by all other variants that make use of the additional cores and hardware resources of the system. The *monolithic* setup tested a scale-up scenario with a single multi-core machine. It represents the performance that can be achieved by a traditional database system deployed on a single multi-core machine. In the monolithic setup, we also limited the computing resources available for the database instance to the number of threads that the distributed approaches where using (i.e., 4 threads per virtual database node). As shown in Figure 13, the throughput of the monolithic setup up to 16 nodes is limited by the number of available threads and shows the same throughput as the distributed Central approach. Starting from 16 database nodes (which corresponds to 64 threads used by the monolithic approach), the performance of the *monolithic* approach flattens. The reason is that the resources of the one physical machine used for *monolithic* is limited to 32 cores with hyper-threading (i.e. the one physical machine optimally supports up to 64 parallel threads).

Starting with eight database nodes, the *Incremental* approach significantly outperforms *monolithic* which indicates that the *Incremental* approach and distributed snapshot isolation are advantageous even for deployments on a single (multi-core) machine.



(a) Throughput (tpmC)



(b) Abort Rate

Fig. 14 Varying the fraction of distributed transactions (no a priori, 16 database nodes, 64 WHs, $0-100\%$ distributed)

9.3 Experiment 2: Vary Number of Distr. Transactions

In the second experiment, we studied the effect of a varying fraction of distributed transactions on the throughput (tpmC) and the abort rate, again if no a priori knowledge was available to the database system (i.e., SAP HANA). This experiment used 64 warehouses that were deployed on 16 database nodes running on 4 physical machines.

Throughput (Figure 14a): The throughput of the *Incremental* approach was again close to the ideal (i.e., no coordination) for low fractions of distributed transactions (i.e., $\leq 5\%$ or less). For an increasing number of distributed transactions, the throughput of the *Incremental* approach degraded gracefully until it reached a throughput that was similar to that of the *Centralized* approach.

The *Pessimistic* and *Optimistic* approaches were again outperformed by the *Centralized* in this experiment because of their reliance on a priori knowledge of which data a transaction accesses at the beginning of a transaction. The *Pessimistic* approach had a low throughput of only approximately 1100 tpmC, independent of the fraction of distributed transactions because of its high coordination costs that have to be paid to set-up the transaction at all database nodes at the beginning. The *Optimistic* approach performed better if only a few transactions are distributed and most transactions are local. However, the throughput degraded for an increasing fraction of distributed transactions because it results in very high abort rates.

Abort Rate (Figure 14b): Significant abort rates are inherent to Snapshot Isolation and cannot be avoided, regardless of which approach is taken. However, due to their nature, the *Incremental* and *Optimistic* approaches may result in even higher abort rates for trying to reconcile Snapshots late and hoping that this will always be possible. The *Pessimistic* and *Centralized* approaches on the other hand have the lowest possible abort rate (same as the "no coordination" case), 13% in this experiment, that any Snapshot Isolation scheme can have as a result of write/write-conflicts. However, the abort rate *Incremental* increases much slower as for *Optimistic*. The rationale is that with higher rates of (short running) distributed transactions more transactions can become global (from the same database node) without being aborted in the *Incremental* approach.

In Figure 14b, the increase in abort rate was moderate for the *Incremental* approach, even if 100% new-order transactions are distributed. In contrast, the *Optimistic* approach had a strong increasing abort rate with an increasing number of distributed transactions. This advantage of the *Incremental* approach stems from the fact that the *Incremental* approach tries to find a snapshot that does not violate the DSI criterion as opposed to the *Optimistic* approach that always selects the most recent local snapshot which indeed often violates the DSI criterion and thus results in an abort. The slight increase in the abort rate for the *Incremental* approach can be explained as follows: Transactions that started locally on one node are aborted when they want to access data from another node and another transaction already selected the local snapshot for the same global CID interval. At 100% distributed new-order transactions, about 50% of the aborts of the *Incremental* approach are caused by this phenomenon. Section 8 describes a variant of the *Incremental* approach that remedies this issue and results in even lower abort rates; for brevity, we do not describe results with this variant here as the basic *Incremental* approach already showed very good and robust performance.

9.4 Experiment 3: Impact of A Priori Knowledge

This section shows the results of executing the TPC-C benchmark with a varying number of database nodes and a varying fraction of distributed transactions if SAP HANA was given full a priori knowledge of which data on which database nodes a transaction accesses at the beginning of a transaction. This a priori knowledge favors the *Pessimistic* and *Optimistic* approaches because they can exploit this knowledge to avoid unnecessary communication overhead with

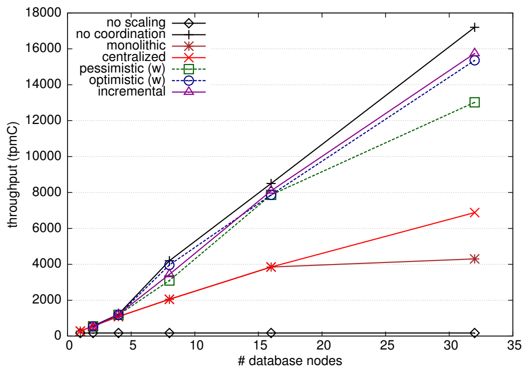


Fig. 15 Throughput (tpmC): vary # database nodes
(full a priori, 1-32 database nodes, 4-128 WHs, 5% distributed)

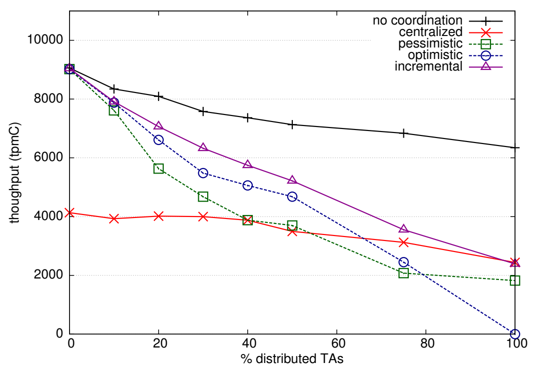
database nodes that are guaranteed to be not involved in the transaction.

Figure 15 shows the throughput results (i.e., tpmC) for the scalability experiment with full a priori knowledge. In this experiment, we scaled the number of virtual database nodes from 1 to 32. As expected, compared to the scalability experiment where no a priori knowledge was provided, the *Optimistic* and *Pessimistic* approach benefit the most from using a priori knowledge. For the *Pessimistic* approach, the improvement results from reducing the initial coordination costs when beginning transactions. For the *Optimistic* approach, the improvement results from a much lower abort rate due to less violation of the rules introduced in Section 4.2. Thus, both approaches achieve a throughput which is close to the *Incremental* approach and the ideal (i.e., no coordination).

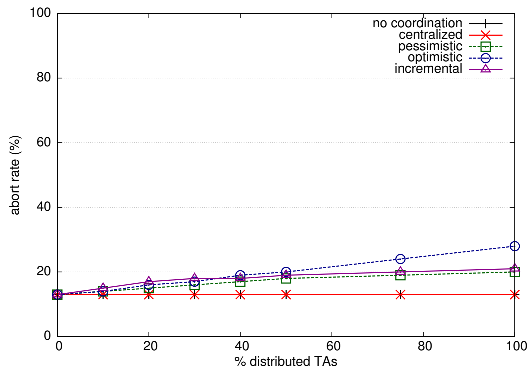
Figure 16 shows the results with a varying fraction of distributed transactions. As in experiment 2, we used 64 warehouses that were deployed on 16 virtual database nodes. Again, we can see that the *Pessimistic* and the *Optimistic* approach benefited from the provided knowledge while the *Centralized* and *Incremental* approach show the same performance as with no a priori knowledge (see Figure 14).

For the *Pessimistic* approach, providing full a priori knowledge leads to a huge increase in throughput whereas workloads with lower fractions of distributed new-order transactions benefit much more since then expensive global begin operations need to be carried out only for few transactions. With full a priori knowledge, the throughput of the *Optimistic* approach is similar to the throughput of the *Incremental* approach.

However, the abort rate of the *Optimistic* approach is still slightly higher than the abort rate of the *Incremental* approach as shown in Figure 16b, which again is caused by the selection of the most recent local snapshots (even when providing full a priori knowledge). The *Pessimistic* and *Centralized* approaches again have the lowest possible abort rate (same as the "no coordination" case) resulting from write/write-conflicts only.



(a) Throughput (tpmC)



(b) Abort Rate

Fig. 16 Varying the fraction of distributed transactions
(full a priori, 16 database nodes, 64 WHs, $0-100\%$ distributed)

9.5 Experiment 4: Varying the Number of Nodes in a Distributed Transaction

The last set of experiments studied the effect of varying the number of nodes participating in a distributed transaction. Figure 17 shows the results of the corresponding experiment with 64 warehouses distributed over 16 database nodes running on 4 physical machines when full a priori knowledge was provided.

For this experiment, we modified the TPC-C benchmark such that the maximal number of nodes that participate in a new-order transaction can be varied. In the original benchmark specification an order can have $5-15$ items in total, whereas each item can come from a different warehouse. In this experiment, we change the upper bound of items. Thus, the x-axis represents an upper limit of participating nodes in any distributed new-order transaction. As upper limit Figure 17 shows 16 nodes on the x-axis, which results from one database node for the coordinating warehouse plus maximally 15 other database nodes for other individual warehouses to fill the items.

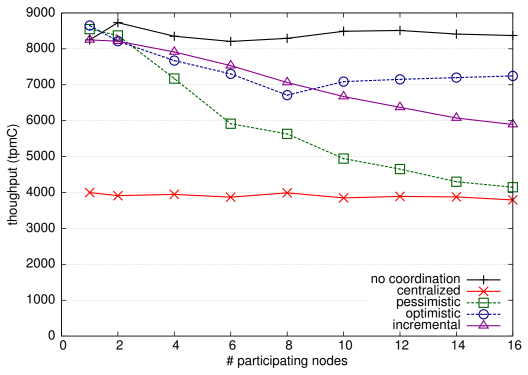


Fig. 17 Throughput (tpmC): vary # participating nodes 1-15 (full a priori, 16 database nodes, 64 WHs, 20% distributed)

During the entire experiment, the actual computing power was constant, but the effort for one transaction changed as more nodes were accessed during one distributed transaction. The fraction of distributed new-order transactions was fixed to 20% (80% were local). We used a high distributed transaction rate of 20% to amplify the results, even though our experience with real SAP HANA deployments have a much lower fraction of distributed transactions. Moreover, we chose to provide full a priori knowledge since this represents the best case for all approaches; in particular, the *Pessimistic* and *Optimistic* approaches.

As expected, the throughput of the *Centralized* approach was independent of the number of nodes that participated in a distributed transaction because its coordination overhead is independent of this parameter. The *Pessimistic* approach does not scale well with the number of participating nodes each database node involved in a distributed transaction involves two extra round-trip messages to set up the transaction context at the node and to commit the transaction at that node. The *Incremental* approach performed better than the *Pessimistic* approach, but the number of participating nodes is an important factor for this approach, too, because the complexity of reconstructing a snapshot grows with the number of nodes. Like the *Centralized* approach, the *Optimistic* approach is not affected by the number of participating nodes because the size of the matrix does not depend on the number of participating nodes and the validation at commit-time is fast even with many participating nodes.

10 Related Work

The topic of Distributed Snapshot Isolation received attention recently. The renewed interest started with early work on Snapshot Isolation in a federated setup [27] and also in a restricted setup [12, 17] or in the context of replication [2]. Snapshot Isolation in column stores received only limited attention so far [33, 34]. Moreover, many other approaches for partitioned distributed databases exist, which do not use Snapshot Isolation. Most of these approaches [17, 30] focus on ordering the transactions before applying them on a node in a distributed setting, which is similar to the ideas of our correctness criteria for DSI.

10.1 Concepts of Snapshot Isolation

First, in [19] the centralized approach discussed in this paper is presented in detail including many possible optimizations.

Similar to some ideas of the Incremental approach presented in this paper is the idea of *Clock-SI* presented in [14] where the begin of the transaction (i.e., the selection of the snapshot) is decoupled from the first actual operation. The Incremental approach discussed in this paper generalizes this idea by placing the begin of transaction on a certain node when it is suitable instead of when the first operation is performed. Different from [14] where the begin may only be earlier, the Incremental approach also allows the begin of a transaction to be later (in terms of wall clock time) to ensure the most recent possible snapshot that is still correct according to the definition of Distributed Snapshot Isolation (see Section 4). Moreover, the Incremental approach does not add any delays when beginning or committing transactions as it is done in Clock-SI. Especially local transactions in Clock-SI could potentially be delayed due to a pending commit of a distributed transaction, which is also not the case in the Incremental approach

Another approach which uses timestamp ranges based on a multi-version concurrency scheme to select a version of a tuple is described in [22]. In [22], the timestamp range is used to adjust the version accessed by a transaction to avoid conflicts that are disallowed in different SQL isolation levels. Compared to the Incremental approach, [22] does not support distributed transaction processing where each local database node has its independent versions. In this case one must ensure that the order of global begin- and commit-operations is the same on all nodes and this does not allow to adjust the version accessed by a distributed transaction to avoid aborts in many cases.

Furthermore the definition of *Session Snapshot Isolation* [13] applies to the system presented in this paper. A system ensures session Snapshot Isolation if consecutive transactions of the same client (i.e., the same session) see what previous transactions wrote and Snapshot Isolation holds. This definition is useful since normal Snapshot Isolation does not require that a client gets the most recent snapshot. [13] call it *strong Snapshot Isolation* if every transaction gets the most recent snapshot. The goal of the Incremental approach is similar since it tries to provide every transaction with the most recent snapshot that is still correct according to the definition of Distributed Snapshot Isolation. In some cases this can even be a more recent snapshot than the definition of strong Snapshot Isolation in [13] would allow. This is because local transactions can be considered (i.e., a more recent snapshot can be used) as long as that view does not break global consistency.

Since the database on which the approaches are implemented and tested is organized as a column store, the work of Zhang and de Sterck [33, 34] on Snapshot Isolation on HBase, an open-source column store, is related. Their approach is close to what is identified as central coordination in this paper since all information about transactions is kept in a bunch of (conceptually) centralized tables. The column store used in this paper implements Snapshot Isolation with centralized coordination successfully for some time now.

10.2 Distributed SI in Restricted Setups

Some of the approaches in this paper are inspired by ideas from Schenkel et al. [27]. In their paper, Schenkel et al. present algorithms to achieve Snapshot Isolation for federated databases. The databases are treated as black boxes and local transactions are not considered. For such a federation of Snapshot Isolation databases, they derive algorithms to achieve global serializability. Due to the black box approach, they have limited possibilities but also do not need to care about local transactions (or more precisely, the federation layer is not aware of any local transactions). This leads to potential violations of the correctness criteria given in Subsection 4.2: in the Optimistic approach in [27], the begins of distributed transactions are not ordered the same on all nodes and therefore the cross-phenomenon (see Figure 5) can occur. This paper extends their approaches to work in the distributed setting where local transactions have to be considered.

Optimizing for local transactions is considered in [23] where a system for a cloud database is presented where transactions that share data from different transaction managers can only operate at a lower consistency level. Snapshot Isolation is mentioned as future work for their system.

In [17], Jones et al. discuss concurrency control in partitioned main memory databases. They restrict the possible transactions to executions of predeclared stored procedures without user interaction which represents a higher dimension of a priori knowledge than the one considered in this paper. Throughput is improved by making use of wait cycles while a remote node executes its part of the current transaction. This is demonstrated using a similar experimental setup than used in this paper.

Restricting the possible operations in order to simplify concurrency control is taken to an extreme in [12] where a fixed group of operations from small set of operators with limited connection among each other, called mini-transactions, are submitted all at once. Another approach is taken in [11]: the database is automatically partitioned with the goal to minimize the amount of distributed transactions. In that sense, this paper can be seen as an extension since the approaches presented in this paper work best if the fraction of distributed transactions is low, but still cannot be neglected. One interesting aspect in [12] is the concept of hierarchical transaction managers. As future work, the ideas in this paper could be extended in a similar way to have a hierarchy of transaction coordinators in order to reduce the pressure on a single central coordinator. This could for example be useful in the so called “multi-tenancy scenario” where the database is provided as a service and shared with multiple customers (i.e., tenants). If a tenant is distributed over multiple nodes, one could introduce a “tenant-global” transaction manager in order to avoid the need to go to the central coordinator for the transactions accessing the nodes of the tenant.

10.3 Snapshot Isolation in Replicated DBMS

The *Pessimistic* approach is used in [3] to build a partial database replication protocol, i.e., on begin of a transaction, all potentially involved nodes are informed.

A similar idea to the Incremental technique is used in [28] but instead of actually reconstructing the snapshot on other nodes, their system just opens dummy transactions after each commit, which can then later be used to process requests from other nodes that need an older snapshot. They are more concerned about the aspect of replication[2] and less about the distinction between local and distributed transactions. In order to keep the nodes synchronized, they use group communication to commit all transactions on all nodes while the Incremental technique uses a centralized coordinator to keep track of the global counters.

A new approach called Parallel Snapshot Isolation (PSI) to guarantee Snapshot Isolation over multiple geo-replicated sites is presented in [29]. PSI extends Snapshot Isolation by allowing different sites to have different commit orderings. This is benefitial for the asynchronous replication of transactions that are executed independently on different database objects at different sites (i.e., they do not share a common write-set). This relaxation could also be interesting if asynchronous replication should be integrated into the approaches presented in this paper.

There are many solutions for concurrency control in replicated databases using group communication, for example [8, 13, 14, 18, 20, 25], or see [2] for an overview of Snapshot Isolation-oriented approaches. But the issue of scheduling is different from the issue considered in this paper since in these systems all data is fully replicated.

Our work differs in one important aspect from other work that uses Snapshot Isolation-based replicas but achieves a higher isolation level in the overall system (e.g. [8, 18]): our approaches minimize the information about the transactions shipped between nodes (e.g., read- or write-sets). This is an

[2] Replication is not considered in this paper, but support of eager replication is straight-forward.

essential design decision in our work and one of the reasons for the improved performance. At the same time, this also implies that our approaches cannot directly achieve a higher isolation level than what the underlying systems provide since that would require additional information.

Some approaches support partial replication (e.g. [15, 16]) but aim for serializability instead of Snapshot Isolation as concurrency requirement. Sprint [10] uses similar techniques, i.e., group communication and predefined transactions, and also aims for serializability. Relevant to the scenario in this paper is that Sprint makes a distinction between distributed and local transactions. A local transaction can access data from other nodes by informing all other nodes using total order multicast. If the order of the operations is wrong on any node, the transaction is aborted. This is similar to the design decision made in this paper not to monitor actual conflicts between transactions since this is expensive. Just as in Sprint, we abort a transaction as soon as the order of the transaction begin or commit operations is wrong.

11 Conclusion

This paper introduced three approaches to improve Distributed Snapshot Isolation in partitioned databases: *Pessimistic* and *Optimistic* are based on ideas developed for federated databases; *Incremental* is a new approach. The main difference between the approaches is the amount of a priori knowledge about transactions they need. Pessimistic only works well if the client provides the list of accessed nodes at the begin of a transaction. Optimistic works well if the client tells the system whether a transaction is distributed or local. Incremental finally does not need any a priori knowledge. Incremental is as efficient as the other approaches in scenarios where a lot of information about the intentions of a transaction are available at the beginning of the transaction. In the general scenario, i.e., if no information is available beforehand, the Incremental approach outperforms the other approaches. Furthermore, the Incremental approach is completely transparent to the application. All approaches provide clear consistency guarantees, i.e., snapshot isolation is provided as if the transactions were executed only locally.

The new techniques can be used to scale-out column stores if transactional guarantees offered by Snapshot Isolation are desirable. Especially the fact that the *Incremental* approach can be implemented transparently allows adoption in current databases. It does not change any of the properties provided by a normal database while improving performance in certain scenarios, e.g., when scaling the database to many nodes.

For now the Incremental technique supports Snapshot Isolation. A possible avenue of future work is to extend the Incremental technique to also support Serializable Snapshot Isolation [9] or other isolation levels (e.g., Parallel Snapshot Isolation [29]).

Bibliography

[1] Atul Adya. “Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions”. PhD thesis. Massachusetts Institute of Technology, 1999.

[2] J. E. Armendáriz-Iñigo, J. R. Juárez-Rodríguez, J. R. González de Mendívil, et al. “A formal characterization of SI-based ROWA replication protocols”. In: *Data and Knowledge Engineering* 70 (1 2011), pp. 21–34.

[3] J. E. Armendáriz-Iñigo, A. Mauch-Goya, J. R. González de Mendívil, et al. “SIPRe: a partial database replication protocol with SI replicas”. In: *Proceedings of the 2008 ACM symposium on Applied computing*. Fortaleza, Ceara, Brazil: ACM, 2008, pp. 2181–2185.

[4] Stefan Aulbach, Michael Seibold, Dean Jacobs, et al. “Extensibility and Data Sharing in evolving multi-tenant databases”. In: *ICDE*. 2011, pp. 99–110.

[5] Hal Berenson, Phil Bernstein, Jim Gray, et al. “A critique of ANSI SQL isolation levels”. In: *Proceedings of the 1995 ACM SIGMOD international conference on Management of data*. SIGMOD ’95. San Jose, California, United States: ACM, 1995, pp. 1–10.

[6] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[7] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[8] Mihaela A. Bornea, Orion Hodson, Sameh Elnikety, et al. “One-copy serializability with snapshot isolation under the hood”. In: *IEEE 27th International Conference on Data Engineering*. ICDE ’11. IEEE Computer Society, 2011, pp. 625–636.

[9] Michael J. Cahill, Uwe Röhm, and Alan D. Fekete. “Serializable isolation for snapshot databases”. In: *ACM Transactions on Database Systems* 34 (4 2009), 20:1–20:42.

[10] Lásaro Camargos, Fernando Pedone, and Marcin Wieloch. “Sprint: a middleware for high-performance transaction processing”. In: *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*. EuroSys ’07. Lisbon, Portugal: ACM, 2007, pp. 385–398.

[11] Carlo Curino, Evan Jones, Yang Zhang, et al. “Schism: a workload-driven approach to database

replication and partitioning". In: *Proceedings of the VLDB Endowment* 3 (1-2 2010), pp. 48–57.

[12] Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. "ElasTraS: an elastic transactional data store in the cloud". In: *HotCloud*. San Diego, California: USENIX Association, 2009.

[13] Khuzaima Daudjee and Kenneth Salem. "Lazy database replication with snapshot isolation". In: *Proceedings of the 32nd international conference on Very large data bases*. VLDB '06. Seoul, Korea: VLDB Endowment, 2006, pp. 715–726.

[14] Sameh Elnikety, Willy Zwaenepoel, and Fernando Pedone. "Database Replication Using Generalized Snapshot Isolation". In: *Proceedings of the 24th IEEE Symposium on Reliable Distributed Systems*. Washington, DC, USA: IEEE Computer Society, Oct. 2005, pp. 73–84.

[15] Udo Jr. Fritzke and Philippe Ingels. "Transactions on Partially Replicated Data based on Reliable and Atomic Multicasts". In: *International Conference on Distributed Computing Systems* (2001), p. 0284.

[16] JoAnne Holliday, Divyakant Agrawal, and Amr El Abbadi. "Partial Database Replication using Epidemic Communication". In: *International Conference on Distributed Computing Systems*. Washington, DC, USA: IEEE Computer Society, 2002, pp. 485–.

[17] Evan P.C. Jones, Daniel J. Abadi, and Samuel Madden. "Low overhead concurrency control for partitioned main memory databases". In: *Proceedings of the 2010 international conference on Management of data*. SIGMOD '10. Indianapolis, Indiana, USA: ACM, 2010, pp. 603–614.

[18] Hyungsoo Jung, Hyuck Han, Alan Fekete, et al. "Serializable Snapshot Isolation for Replicated Databases in High-Update Scenarios". In: *PVLDB* 4.11 (2011), pp. 783–794.

[19] Juchang Lee, Yong Sik Kown, Franz Farber, et al. "SAP HANA Distributed In-Memory Database System: Transaction, Session, and Metadata Management". In: *ICDE*. 2013.

[20] Yi Lin, Bettina Kemme, Ricardo Jiménez-Peris, et al. "Snapshot isolation and integrity constraints in replicated databases". In: *ACM Transactions Database Systems* 34 (2 2009), 11:1–11:49.

[21] Yi Lin, Bettina Kemme, Marta Patiño Martínez, et al. "Middleware based data replication providing snapshot isolation". In: *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*. SIGMOD '05. Baltimore, Maryland: ACM, 2005, pp. 419–430.

[22] David B. Lomet, Alan Fekete, Rui Wang, et al. "Multi-version Concurrency via Timestamp Range Conflict Management". In: *ICDE*. 2012, pp. 714–725.

[23] David B. Lomet, Alan Fekete, Gerhard Weikum, et al. "Unbundling Transaction Services in the Cloud". In: *Fourth Biennial Conference on Innovative Data Systems Research*. CIDR '09. 2009.

[24] Patrick O'Neil, Elizabeth O'Neil, Shankar Pal, et al. "ORDPATHs: insert-friendly XML node labels". In: *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*. SIGMOD '04. Paris, France: ACM, 2004, pp. 903–908.

[25] Christian Plattner and Gustavo Alonso. "Ganymed: scalable replication for transactional web applications". In: *Proceedings of the 5th ACM/IFIP/USENIX international conference on Middleware*. Middleware '04. Toronto, Canada: Springer-Verlag New York, Inc., 2004, pp. 155–174.

[26] Tudor-Ioan Salomie, Ionut Emanuel Subasu, Jana Giceva, et al. "Database engines on multicores, why parallelize when you can distribute?" In: *EuroSys*. 2011, pp. 17–30.

[27] Ralf Schenkel, Gerhard Weikum, Norbert Weißenberg, et al. "Federated Transaction Management with Snapshot Isolation". In: *Transactions and Database Dynamics*. Vol. 1773. Springer Berlin / Heidelberg, 2000, pp. 1–25.

[28] D. Serrano, M. Patino-Martinez, R. Jimenez-Peris, et al. "Boosting Database Replication Scalability through Partial Replication and 1-Copy-Snapshot-Isolation". In: *Proceedings of the 13th Pacific Rim International Symposium on Dependable Computing*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 290–297.

[29] Yair Sovran, Russell Power, Marcos K. Aguilera, et al. "Transactional storage for geo-replicated systems". In: *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. SOSP '11. Cascais, Portugal: ACM, 2011, pp. 385–400.

[30] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, et al. "Calvin: fast distributed transactions for partitioned database systems". In: *SIGMOD Conference*. 2012, pp. 1–12.

[31] Transaction Processing Performance Council. *TPC Benchmark C, revision 5.11*. 2010.

[32] Werner Vogels. "Eventually consistent". In: *Commun. ACM* 52.1 (2009), pp. 40–44.

[33] Chen Zhang and H. de Sterck. "Supporting multi-row distributed transactions with global snapshot isolation using bare-bones HBase". In: *11th IEEE/ACM International Conference on Grid Computing*. GRID '10. Oct. 2010, pp. 177 –184.

[34] Chen Zhang and Hans de Sterck. "HBaseSI: Multi-Row Distributed Transactions with Global String Snapshot Isolation on Clouds". In: *Scalable Computing: Practice and Experience* 12 (2 2011).

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 $\{S^1, S^2, ..., S^n\}$ 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 $S^i \in \{S^1, S^2, ..., S^n\}$ 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 $S^i \in \{S^1, S^2, ..., S^n\}$ 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 $\{S^1, S^2, ..., S^n\}$, 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 $\{S^1, S^2, ..., S^n\}$ (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 $op\_1 < op\_2 < ... < op\_n$ 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 $op\_1 < op\_2 < ... < op\_n$, 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 $S^i \in \{S^1, S^2, ..., S^n\}$ that contain the first global operation (i.e., op_1) in the given total order until (excluding that global operation op_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 $S^i \in \{S^1, S^2, ..., S^n\}$ must not be changed in G. Then, we consume the given global operation op_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 op_2 in the global order until all global operations in the given total order are consumed (i.e., until op_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 $\{S^1, S^2, ..., S^n\}$. 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 $\{S^1, S^2, ..., S^n\}$, we have to show that a) all read operations $r_t(o_v) \in G$ return the same versions v of object o as the corresponding read operation in the local schedules $\{S^1, S^2, ..., S^n\}$ and that b) the final state of the database is the same after executing the schedules, i.e., the last write $w_t(o_v)$ of any transactions $t \in G$ to an object o is the same as in the local schedules $\{S^1, S^2, ..., S^n\}$. 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 $S^i \in \{S^1, S^2, ..., S^n\}$ 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 $SSG(G)$. 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) can not occur in G and its $SSG(G)$. In the following, we discuss each anomaly for Snapshot Isolation separately.

G1a: Aborted Reads. Since local nodes provide proper Snapshot Isolation, any read $r_x(o_v)$ of a transaction x that appears in a local schedule $S^i \in \{S^1, S^2, ..., S^n\}$ is mapped to a write $w_y(o_v)$ of a transaction y that committed before transaction x started (on node i). Since we assume that data is partitioned (and not replicated) and the order of operations in a local $S^i \in \{S^1, S^2, ..., S^n\}$ 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 $S^i \in \{S^1, S^2, ..., S^n\}$. Since we assume that data is partitioned (and not replicated) and the order of operations in a local $S^i \in \{S^1, S^2, ..., S^n\}$ is not changed in G, transaction x reads the same version in G as in S^i (i.e., only committed object versions are read).

G1c: Circular Information Flow. All read- and write-dependency edges from transaction x to transaction y in any $SSG(S^i)$ imply that $c_x < b_y$ in any correct local schedule $S^i \in \{S^1, S^2, ..., S^n\}$ that contains operations from transaction x and transaction y (i.e., node $i \in SN(x,y)$). Since the construction above checks that the same order relation $c_x < b_y$ holds in any of these local schedule S^i and the order relation $c_x < b_y$ is also used to construct the global schedule G, there can not exist a directed cycle of read- and write-dependency edges between x and y in $SSG(G)$ as well. This holds for any transaction x and transaction y where $c_x < b_y$ even if there is no direct read- or write-dependency edge from x tp 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 $SSG(S^i)$ of any $S^i \in \{S^1, S^2, ..., S^n\}$ implies that there must also be a start-dependency edge in S^i (i.e., since G-SIa must hold in a correct local $SSG(S^i)$). Such a start-dependency edge from transaction x to transaction y implies that $c_x < b_y$ in S^i. Since the same order relation also holds in the global schedule G (by its construction), there is also a start-dependency edge in $SSG(G)$ from transaction x to transaction y. Thus, G-SIa can not occur in $SSG(G)$ as well.

G-SIb: Missed Effects. If there is an anti-dependency edge from transaction y to transaction x in a local $SSG(S^i)$ and SI holds for $S^i \in \{S^1, S^2, ..., S^n\}$, this implies that $b_y < c_x$. If $SSG(G)$ would contain the anomaly G-SIb, a cycle with exactly one anti-dependency edge from transaction y to transaction x 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 $\{S^1, S^2, ..., S^n\}$ 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 $SSG(G)$ consisting only of read-/ write-/ start-dependency edges. Any read-/ write-/ start-dependency edge from a transaction s to transaction t on that path implies that $c_s < b_t$. Thus, if there exists a path of read-/ write-/ start-dependency edges from transaction x to transaction y in $SSG(G)$, we can derive that $c_x < b_y$ 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:

Attribute	Details
global-snapshot	Global snapshot read by transaction *x* (i.e., a global CID) assigned by operation *global-begin*
global-tid	A globally unique transaction identifier (i.e., a global TID) used for tagging non-committed tuple versions of *x* assigned by operation *global-begin*
global-cid	A globally unique commit identifier (i.e., a global CID) used for tagging committed tuple versions of *x* assigned by operation *global-commit*

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){
  global-latch.acquire();
  bool success = true;
  x.global-cid=++GLOBAL-CID;

  //synchronous prepare phase:
  //tag tuple version using global-cid
  for( each node i in N(x)){
    success = i.local-prepare(x, i);
    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.

Attribute	Details
local-snapshot[i]	Local snapshot read by transaction *x* on node *i* assigned by operation *local-begin*
local-tid[i]	Local TID for tagging non-committed tuple versions on node *i* assigned by operation *local-begin*
local-cid[i]	Local CID for tagging committed tuple versions on node *i* assigned by operation *local-commit*

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

```
//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 $N(x)$ ){
    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

```
//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. While global information is used by the coordinator to detect violations of the rules (1-8) in Section 4.2, local information is used on each database node to select and snapshot and tag tuple versions created by a transaction *x*.

Attribute	**Details**
global-tid	A globally unique TID used as global begin timestamp assigned by operation *global-begin*
global-cid	A globally unique CID used as global commit timestamp assigned by operation *global-commit*
global-crt	A list of concurrent global transactions of *x* modified by operation *global-begin*
local-snapshot[i]	A local snapshot read by transaction *x* on node *i* assigned by operation *local-begin*
local-tid[i]	A local TID for tagging non-committed tuple versions on node *i* assigned by operation *local-begin*
local-cid[i]	A local CID for tagging committed tuple versions on node *i* assigned by operation *local-prepare*

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
  ...
}
```