# Federated Transaction Management with Snapshot Isolation

**Ralf Schenkel, Gerhard Weikum**
University of the Saarland
email {schenkel,weikum}@cs.uni-sb.de

**Norbert Weißenberg**
Fraunhofer ISST
email norbert.weissenberg@do.isst.fhg.de

**Xuequn Wu**
Deutsche Telekom AG
email wu@tzd.telekom.de

## Abstract

Federated transaction management (also known as multidatabase transaction management in the literature) is needed to ensure the consistency of data that is distributed across multiple, largely autonomous, and possibly heterogeneous component databases and accessed by both global and local transactions. While the global atomicity of such transactions can be enforced by using a standardized commit protocol like XA or its CORBA counterpart OTS, global serializability is not self-guaranteed as the underlying component systems may use a variety of potentially incompatible local concurrency control protocols. The problem of how to achieve global serializability, by either constraining the component systems or implementing additional global protocols at the federation level, has been intensively studied in the literature, but did not have much impact on the practical side. A major deficiency of the prior work has been that it focused on the idealized correctness criterion of serializability and disregarded the subtle but important variations of SQL *isolation levels* supported by most commercial database systems.

This paper reconsiders the problem of federated transaction management, more specifically its concurrency control issues, with particular focus on isolation levels used in practice, especially the popular *snapshot isolation* provided by Oracle. As pointed out in a SIGMOD 1995 paper by Berenson et al., a rigorous foundation for reasoning about such concurrency control features of commercial systems is sorely missing. The current paper aims to close this gap by developing a formal framework that allows us to reason about local and global transaction executions where some (or all) transactions are run under snapshot isolation. The paper derives criteria and practical protocols for guaranteeing global snapshot isolation at the federation level. It further generalizes the well-known ticket method to cope with combinations of isolation levels in a federated system.

## 1 Introduction

### *1.1 Reviving the Problem of Federated Transactions*

With the ever-increasing demand for information integration both within and across enterprises, there is renewed interest in providing seamless access to multiple, independently developed and largely autonomously operated databases. Such a setting is known as a database federation or heterogeneous multidatabase system. More specifically, the approach of building an additional integration software layer on top of the underlying component systems is referred to as a *federated database system* [LMR90, SL90, HBP93, BE96, Conr97, ÖV98]. Among the challenges posed by such a system architecture is the problem of enforcing the consistency of the data across the boundaries of the individual component systems. Transactions that access and modify data in more than one component system are referred to as *federated* or *global transactions* [BGS92, BGS95]; for example, an electronic commerce application could require a transaction to update data in a merchant's database as well as the databases of a credit card company and a service broker that pointed the customer to the merchant and requests a provisioning fee for each sale. Providing the usual ACID properties for such federated transactions is inherently harder than in a homogeneous, centrally administered distributed database system, one reason being that the underlying component systems of a federation may employ different protocols for their local transaction management. A canonical example is the following schedule of three transactions, $t_1$, $t_2$, and $t_3$, that read and write data

objects x, y, and z in two databases, $DB_1$ and $DB_2$, managed by two different database systems (or differently configured instances of the same database system) with different concurrency control protocols:

$DB_1$ $r_1(x)$ $w_1(x)$ $C_1$ $r_2(x)$ $w_2(x)$ $C_2$
$DB_2$: $r_3(y)$ $r_1(y)$ $w_1(y)$ $C_1$ $r_2(z)$ $w_2(z)$ $C_2$ $r_3(z)$ $C_3$

Both local schedules, as seen by each of the two component systems alone, are serializable, but the problem is that the resulting serialization orders are incompatible from a global viewpoint. In $DB_1$, the serialization order requires that $t_1$ precede $t_2$, whereas in $DB_2$, the only possible serialization order is the one in which $t_2$ precedes $t_3$ and $t_3$ precedes $t_1$. Thus, the overall execution of the three transactions is not globally serializable and may potentially result in data inconsistencies.

These kinds of problems with regard to federated transactions have been intensively studied in the late eighties and early nineties. The proposed solutions range from imposing additional constraints on the transaction protocols of the underlying component systems to building an additional transaction manager in the federated software layer on top of the component systems to reconcile or control the underlying local executions. The most notable result in the first category probably is that global serializability is self-guaranteed if all component systems allow only conflict-serializable schedules where the commit order coincides with the serialization order [BGRS91, BS92, Raz92, Weihl89], with rigorous schedules being the most important special case [BGRS91]. In the second category, the family of ticket methods [GRS94] has been among the most promising approaches, complementing knowledge about local serialization orders with graph-cycle testing at the federation level.

All this ample work has led to interesting theoretical insights, but appears to have made little impact on the practical side. Consequently, the subject of federated transactions has not been pursued further by the research community in the last few years. The current paper aims to revive the subject, pushing it in a new and practically relevant direction. We believe that the prior work has not succeeded in the intended technology transfer because it made some fundamental assumptions that did not match the typical setting of commercial database systems and their real-life applications. Essentially, the prior work made too liberal assumptions on the recovery protocols of practically used systems, and it made too restrictive assumptions on their local concurrency control protocols:

- On the recovery side, the point that the underlying component systems are largely autonomous and may therefore not necessarily be willing to cooperate for ensuring the atomicity of global transactions has been way overrated. Today, standardized distributed commit protocols like XA or its CORBA counterpart OTS [OMG97] are supported by virtually all commercially relevant database systems, request brokers (TP monitors, ORBs, etc.), and even packaged business-object software systems. Rather than building complex recovery protocols at the federation level, it is much easier and more manageable to rely on those protocols for global atomicity. For long-running workflow-style applications, a distributed commit protocol would admittedly incur severe performance problems, but for reasonably short federated transactions (e.g., the kind that arises in electronic commerce applications) it is a perfectly viable solution.
- On the concurrency control side, virtually all prior work assumed that the component systems would guarantee at least conflict-serializability for their local schedules. In real life, however, various kinds of isolation levels are provided by commercial database systems and widely used by performance-conscious application developers. These include the isolation levels of the SQL standard, such as “read committed” (known as “cursor stability”

in some commercial systems), as well as vendor-specific options like Oracle's snapshot isolation feature. None of the prior work on federated transaction management has taken these isolation level options into account, despite their undebatable practical relevance. Even worse, there is generally a wide gap in the theoretical foundation of transaction management as far as such concurrency tuning options are concerned. To our knowledge, the only exceptions are the remarkable paper by Berenson et al. [BBGM+95], discussing SQL-standard as well as vendor-specific isolation levels from a conceptual viewpoint without truly foundational ambitions, however, and the work by Atluri et al. [ABJ97] extending the classical serializability theory to incorporate weaker isolation levels from a formal and rather abstract viewpoint. Both papers restrict themselves to a centralized database setting.

### *1.2 Contribution and Outline of the Paper*

The current paper aims to narrow the aforementioned gap in the foundation of transaction management, while also pursueing practically viable solutions for federated transaction management in the presence of isolation levels different from standard serializability. We essentially concentrate on concurrency control issues, disregarding recovery for the fact that standardized distributed commit protocols like XA and OTS can ensure global atomicity across system boundaries, as discussed above. We focus on conventional types of transactions, the key problem being their federated, heterogeneous nature, and do not address workflow-style activities that may run for hours or days. Further note that the problem of federated concurrency control arises even if the same database system is used for all component databases of the federation, since the system may be configured with different isolation level options for different databases (and transaction classes).

Our approach is largely driven by considering the capabilities of commercial systems. In particular, Oracle provides interesting options that are fairly representative for the subtle but important deviations from the pure school of serializability theory. Like several other commercial systems, Oracle supports transient versioning for enhanced concurrency, and exploits versions, by means of a timestamp-based protocol, to offer the following two isolation level options [Orac95, Orac97]:

1. A transaction run under the *"read committed"* option reads the most recent versions of the requested data objects that were committed at the time when the read operation is issued. Note that these read accesses to committed versions can proceed without any locking. All updates that the transaction may invoke are subject to exclusive locking of the affected data objects, and such locks are held until the transaction's commit.

2. For transactions run under the *"snapshot isolation"* level, all operations read the most recent versions as of the time when the transaction began, thus ensuring a consistent view of the data through the transaction. A particularly beneficial special case is that all read-only transactions are perfectly isolated in the sense of the multiversion serializability theory [BHG87]. For read-write transactions, on the other hand, the sketched protocol cannot ensure (multiversion) serializability. In addition, Oracle performs the following check upon commit of a transaction: if any data object written by the transaction has been written by another, already committed transaction that ran concurrently to the considered one (i.e., committed after the considered transaction began), then the current transaction is aborted and rolled back. Additionally, exclusive locks are used for updates and held until the transaction's commit. If a transaction has to wait for a lock and the transaction holding that lock commits, the waiting transaction is aborted. This is a special case of the commit-time test, allowing prevention of concurrent writes before they happen.

Disallowing concurrent writes aims to provide an additional level of sanity, and Oracle even advocates this option under the name "serializability". Nonetheless, the protocol cannot ensure full (multiversion) serializability, with the following schedule as a counter-example ($x_i$ denotes the version of x generated by transaction $t_i$, and $t_0$ is a fictitious initializing transaction):

$r_1(x_0)\ r_1(y_0)\quad r_2(x_0)\ r_2(y_0)\quad w_1(x_1)\ C_1\quad w_2(y_2)\ C_2$

The example may lead to inconsistent data, for example, violating a constraint such as $x + y < 100$ although both transactions alone would enforce the constraint. However, given that such anomalies are very infrequent in practice, the protocol is widely popular in Oracle applications.

As pointed out by Berenson et al. [BBGM+95], such non-serializable isolation levels, and especially the snapshot isolation option, are extremely useful in practice, despite their lack of theoretical underpinnings. Since Oracle would obviously be a major player also in a federated database setting, we concentrate in this paper on understanding the impact of snapshot isolation for federated transaction management. In fact, the work presented here is part of a larger effort to build a federated database architecture, coined VHDBS, that currently supports Oracle and O2 databases [Wu96, HWW98, SW99].

The paper´s "plan of attack" proceeds in three steps, leading to the following contributions:

- We develop a formal model that allows us to reason about isolation levels in the context of federated transactions, and we derive results that relate local and global isolation levels. Since snapshot isolation exploits transient versioning, all our formal considerations are cast in a multiversion schedule framework.
- Based on these theoretical results, we develop new algorithms for federated concurrency control, to ensure global snapshot isolation or global (conflict-) serializability, whatever the application demands. The latter algorithm is based on a generalization of the ticket method to cope with component systems that provide snapshot isolation.

The rest of the paper is organized as follows. In Section 2, we introduce the basic model and notations, and we develop the theoretical underpinnings for coping with snapshot isolation in concurrent transaction executions. In Section 3, we derive results on how to relate the isolation levels in local schedules with the desired correctness criteria at the federation level, and develop an algorithm for ensuring global snapshot isolation under the assumption that all component systems guarantee local snapshot isolation. Section 4 develops a practically viable protocol to guarantee global serializability in a federated system with some component systems providing snapshot isolation, extending and generalizing the ticket method of [GRS94]. We conclude the paper with an outlook on future work. All proofs of theorems are given in the paper's Appendix.

## 2 Basic Model and Notation

This section introduces the formal apparatus that is necessary for our study of snapshot isolation. We briefly introduce the notation and some results of the standard theory of (multiversion) serializability in Subsection 2.1, and then develop a formal characterization of snapshot isolation in Subsection 2.2.

## 2.1 Preliminaries

**Definition: (transaction)**

A *transaction* $t_i$ is a sequence, i.e., total order <, of read and write actions on data objects from one or more databases, along with a set of begin and commit actions, one for each database accessed by $t_i$, such that all begin actions precede all read and write actions in the order < and all commit actions follow all other actions w.r.t. <. The *readset* and *writeset* of $t_i$ are the sets of objects for which $t_i$ includes read and write actions, respectively.

We denote the j-th read or write step of $t_i$ by $a_{ij}(x)$ where x is the accessed object, or $r_{ij}(x)$ (for reads) and $w_{ij}(x)$ (for writes) when the kind of actions is relevant. When we want to indicate the database to which a step refers, we extend the notation for an action as follows: $a_{ij}^{(k)}(x)$ denotes the j-th step of transaction $t_i$ accessing object x that resides in database $DB_k$. The begin and commit actions for database $DB_k$, explicitly denoted whenever they are relevant, are written as $B_i^{(k)}$ and $C_i^{(k)}$, respectively.

■

For all transactions, we further restrict the sequence of read and write accesses to allow a write on object x only if there the transaction includes also a read on x that precedes the write in the action order <. Furthermore, we allow at most one read and at most one write on the same object within a single transaction. None of these two properties present serious restrictions of the executions that we can model; in practice, writes are usually preceded by reads, and multiple reads or writes to the same object can be easily eliminated by using temporary program variables.

Note that we do not consider partial orders within a transaction for the sake of simpler notation, although this relaxation could be incorporated in the model quite easily. Similarly, disallowing multiple reads or writes on the same object is also only a matter of notation. The only "semantically" relevant restriction of our model is that we do not consider transaction aborts and assume all transactions to be committed. This paper addresses federated concurrency control; extensions to incorporate recovery issues in the formal model would be the subject of future work.

**Definition: (global and local transactions)**

A *global transaction (GT)* is a transaction that accesses objects from at least two different databases. In contrast, a *local transaction (LT)* accesses only a single database. The projection of a global transaction $t_i$ onto a database $DB_k$ is the set of actions of $t_i$ that refer to objects from $DB_k$, along with their corresponding order <. This projection will be referred to as a *global subtransaction (GST)* and denoted by $t_i^{(k)}$.

■

Note that the dichotomy between GTs and LTs is a purely syntactic one in our definition. In practice, an additional key difference is that LTs are not routed through an additional federation software layer but rather access a database directly through the native database system and nothing else.

**Definition: (schedule, multiversion schedule, monoversion schedule)**

A *schedule* of transactions $T=\{t_1, ...\}$ is a sequence, i.e., total order <, of the union of the actions of all transactions in T such that the action ordering within transactions is preserved.

A *multiversion schedule* of transactions $T=\{t_1, ...\}$ is a schedule with an additional *version function* that maps each read action $r_i(x)$ in the schedule to a write action $w_j(x)$ that precedes

the read in the order <. The read action is then also written as $r_i(x_j)$ with $x_j$ denoting the version created by the write of $t_j$.

A *monoversion schedule* of transactions T={$t_1$, ...} is a multiversion schedule whose version function maps each read action $r_i(x)$ to the most recent write action $w_j(x)$ that precedes it (i.e., $w_j(x) < r_i(x)$ and there is no other write action on x in between).

■

The usual correctness criterion for multiversion concurrency control is that a given multiversion schedule should be view-equivalent to a serial monoversion schedule, with view-equivalence being defined by the reads-from relation among the actions of a schedule [BHG87, Pa86].

**Definition: (MVSR)**

A multiversion schedule s is multiversion serializable (MVSR), if it is view-equivalent to a monoversion serial schedule.

■

**Definition: (version order, MVSG)**

Given a multiversion schedule s and an object x, a *version order* $\ll$ is a total order of the versions of x in s. A *version order* for s is the union of the version orders for all objects.

Given a multiversion schedule s and a version order $\ll$ for s, the *multiversion serialization graph for s and* $\ll$*, MVSG(s,* $\ll$*),* is the directed graph with the transactions as nodes and the following edges:

For each operation $r_j(x_i)$ in the schedule there is an edge $t_i \rightarrow t_j$ (WR pair).

For each pair of operations $r_k(x_j)$ and $w_i(x_i)$ where i, j and k are distinct, there is an edge

(i) $t_i \rightarrow t_j$, if $x_i \ll x_j$ (WW pair),

(ii) $t_k \rightarrow t_i$, if $x_j \ll x_i$ (RW pair).

■

The following theorem is the most important characterization of MVSR by means of the MVSG, serving as the basis for correctness proofs of practical multiversion concurrency control protocols (see, e.g., [BHG87]).

**Theorem: (characterization of MVSR [BHG87])**

A multiversion schedule s is in MVSR if and only if there exists a version order $\ll$ such that MVSG(s,$\ll$) is acyclic.

■

When we consider only (non-serial) monoversion schedules (for which each write is preceded by a read of the same transaction) and wish to reason about their correctness, the criterion of MVSR automatically degenerates into the well-known notion of conflict-serializability (SR) [Pa86]. This results from the restriction of the version function. So with this restriction, MVSR becomes equivalent to the standard definition of conflict-serializability (SR) based on read-write, write-write, and write-read conflicts (and the corresponding conflict-graph construction.

## *2.2 Formalizing the Snapshot Isolation Level (SSI)*

So far we have introduced the traditional apparatus of multiversion serializability. We are now ready to discuss relaxations, in the sense of isolation levels, within this model. In this paper we concentrate on the situation where an entire schedule satisfies a certain relaxed isolation level. The general case where only a specific subset of transactions tolerates a relaxed isola-

tion level and the “rest of the schedule” should still be serializable or MVSR is the subject of future work.

**Definition: (snapshot isolation)**

A multiversion schedule of transactions $T=\{t_1, ...\}$ satisfies the criterion of *snapshot isolation (SSI)* if the following two conditions hold:

(SSI-V) *SSI version function*: The version function maps each read action $r_i(x)$ to the most recent committed write action $w_j(x)$ as of the time of the begin of $t_i$,

or more formally: $r_i(x)$ is mapped to $w_j(x)$ such that $w_j(x)<C_j<B_i<r_i(x)$ and there are no other actions $w_h(x)$ and $C_h$ ($h \neq j$) with $w_h(x)<B_i$ and $C_j<C_h<B_i$.

(SSI-W) *disjoint writesets*: The writesets of two concurrent transactions are disjoint,
or more formally: if for two transactions $t_i$ and $t_j$, either $B_i<B_j<C_i$ or $B_j<B_i<C_j$, then $t_i$ and $t_j$ must not write a common object x.

■

SSI is weaker than MVSR in the sense that it allows non-serializable schedules, for example the following:

$$r_1(x_0)\ r_1(y_0)\ r_2(x_0)\ r_2(y_0)\ w_1(x_1)\ w_2(y_2)\ C_1\ C_2$$

This schedule satisfies both (SSI-V) and (SSI-W), but it is not equivalent to a serial monoversion schedule. On the other hand, SSI is not a superset of MVSR, because there are serializable multiversion schedules that are not in SSI, for example

$$r_1(x_0)\ r_2(y_0)\ w_2(y_2)\ C_2\ r_1(y_2)\ w_1(y_1)\ C_1$$

This schedule is not in SSI because $r_1(y)$ is mapped to $w_2(y_2)$ rather than $w_0(x_0)$, and because $t_1$ and $t_2$ are concurrent and write the same object y. It is, however, equivalent to the serial monoversion schedule

$$r_2(y)\ w_2(y)\ C_2\ r_1(x)\ r_1(y)\ w_1(y)\ C_1$$

The key point here is that this schedule uses a version function different from the SSI version function.

We now characterize the SSI membership of a given multiversion schedule by the absence of cycles in a graph.

**Definition: (SSI version order)**

The SSI version order $\ll_s$ is the order that is induced by the order of the commit operations of the transactions that wrote the versions, or more formally:
$x_i \ll_s x_j :\Leftrightarrow C_i < C_j$

■

**Definition: (SSI-MVSG)**

The SSI multiversion serialization graph *SSI-MVSG* for a given multiversion schedule s satisfying the property (SSI-V) is a directed graph with the transactions as nodes and the following edges:

For each operation $r_j(x_i)$ in the schedule there is an edge $t_i \rightarrow t_j$, labeled with “x” (WR pair).

For each pair of operations $r_k(x_j)$ and $w_i(x_i)$, there is an edge

(i) $t_i \rightarrow t_j$, if $x_i \ll_s x_j$ (WW pair),

(ii) $t_k \rightarrow t_i$, if $x_j \ll_s x_i$ (RW pair),

labeled with "x" in both cases.

Edges in the graph that are labeled with "x" are called *x-edges*. A cycle in the graph that consists completely of x-edges is called an *x-cycle*.

■

SSI-MVSG is exactly the same as the usual multiversion serialization graph MVSG defined above, extended by the edge labels. Similarly to the MVSG theorem cited in Subsection 2.1, the following theorem gives a characterization for schedules that are snapshot isolated:

**Theorem 1: (Equivalence of SSI and cycle-free SSI-MVSG)**

A multiversion schedule satisfying (SSI-V) is

a) in MVSR, if and only if its corresponding SSI-MVSG is acyclic,

b) in SSI, if and only if there is no object x such that the SSI-MVSG has an x-cycle

■

As an example, consider the two schedules

$$s_1 := r_1(x_0)\ r_1(y_0)\ r_2(x_0)\ r_2(y_0)\ w_1(x_1)\ w_2(y_2)\ C_1\ C_2$$
$$s_2 := r_1(x_0)\ r_1(y_0)\ r_2(x_0)\ r_2(y_0)\ w_1(x_1)\ w_2(x_2)\ C_1\ C_2$$

$s_1$ is in SSI while $s_2$ is not (the transactions are concurrent and both write x). For both schedules, the corresponding SSI-MVSG, shown in Figure 1, contains a cycle; thus neither of the two schedules is MVSR. But only the MVSG for $s_2$ (on the right of Figure 1) contains a cycle that consists solely of edges labeled by the same object x. So $s_2$ is not SSI whereas $s_1$ is SSI.

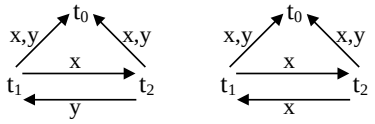


**Figure 1 - SSI-MVSG for two schedules**

# 3 Guaranteeing Global Snapshot Isolation

## *3.1 The Problem of Global Snapshot Isolation*

The goal of this section is to derive necessary and sufficient conditions for schedules of federated transactions to be globally SSI, provided that all underlying component systems generate only schedules that are locally SSI. In the following, we assume that all federated transactions employ an atomic commit protocol (ACP) across the affected component systems with a conceptually centralized coordinator such that

- the relative order of the local commit operations of different global transactions is the same in all component systems (i.e., if $t_i$ commits before $t_j$ in $DB_1$, then $t_i$ must commit before $t_j$ in $DB_2$ etc. as well) and
- the local commit operations are totally ordered among all transactions, i.e., no two commit operations happen concurrently.

With these assumptions, the local commit operations of a transaction can be safely replaced by the global commit operation in all local schedules.

The following example shows that it is not at all trivial to guarantee global SSI even if all component systems enforce local SSI.

$DB_1$: $r_1^{(1)}(a_0)\ r_2^{(1)}(x_0)\ w_2^{(1)}(x_2)$ $C_2\ r_1^{(1)}(x_0)$ $C_1$

$DB_2$: $r_2^{(2)}(y_0)\ w_2^{(2)}(y_2)\ C_2$ $r_1^{(2)}(y_2)\ w_1^{(2)}(y_1)\ C_1$

In database $DB_1$, the subtransactions $t_1^{(1)}$ and $t_2^{(1)}$ of the two global transactions $t_1$ and $t_2$ run concurrently, so they both read from a prior transaction (here this is the fictitious, initializing transaction $t_0$). Only $t_2^{(1)}$ writes an object. This subschedule therefore is SSI. The other subschedule, in database $DB_2$, is SSI, too: the two subtransactions $t_1^{(2)}$ and $t_2^{(2)}$ run serially and they both read the correct values according to the SSI version function. However, combining the two subschedules into a global schedule (with database superscripts omitted) yields:

$$r_1(a_0)\ r_2(x_0)\ w_2(x_2)\ r_2(y_0)\ w_2(y_2)\ C_2\ r_1(x_0)\ r_1(y_2)\ w_1(y_1)\ C_1$$

This schedule is not in SSI: $t_1$ reads y from $t_2$ rather than $t_0$ (so (SSI-V) does not hold), and both transactions run concurrently and both write y (so (SSI-W) does not hold either). The problem lies in the fact that the local scheduler in $DB_2$ does not know when $t_2$ started at the global federation level, so it fails to assign the globally correct versions, restricting itself to local correctness. Analogous arguments hold for the consideration of global writesets versus local writesets and global concurrency versus local concurrency. The following two subsections present two algorithms that overcome these problems and guarantee that the global schedule is SSI. The first, pessimistic approach, discussed in Subsection 3.2, achieves its objective by synchronizing the local begin operations of transactions (in addition to the implicit synchronization of the local commits that results from the ACP). The second, optimistic approach, discussed in Subsection 3.3., is based on testing the potential violation of global SSI.

### *3.2 Pessimistic Approach: Synchronizing Subtransaction Begins*

An idea towards a globally correct execution is to start all local subtransactions $t_i^{(k)}$ upon the first operation of the global transaction $t_i$. The intuition behind this is that we want all subtransactions of a global transaction to have the same start time as the global transaction in the global schedule. If we can achieve this, any violation of the SSI property of the global schedule could be mapped to a corresponding violation in one of the local schedules, and the latter would be prevented by the fact that all local schedulers generate SSI schedules.

The obvious way to "simultaneously" start the local subtransactions is by issueing an explicit local begin in each component system $DB_k$ on which the transaction will possibly issue read or write operations. Unfortunately, this alone does not entirely solve the problem. It may occur that a global transaction $t_1$ commits while another transaction $t_2$ is in the process of submitting its local begin operations. If in some database the begin operation is executed before the commit of $t_1$ and this order is reversed in another database, global SSI property can still not be guaranteed. The following part of a schedule illustrates this problem:

$DB_1$: $r_1^{(1)}(x_0)\ w_1^{(1)}(x_1)\ B_2^{(1)}\ C_1$ $r_2^{(1)}(x_0)$ $C_2$

$DB_2$: $r_1^{(2)}(y_0)\ w_1^{(2)}(y_1)$ $C_1\ B_2^{(2)}$ $r_2^{(2)}(y_1)\ w_2^{(2)}(y_2)\ C_2$

This execution is not globally SSI. However, both local schedules are SSI, so merely adding the local begin operations did not fix the problem.

Obviously, the above problem arises because a global commit is executed in between several local begin operations of the same federated transaction. So we need to delay the commit request of a global transaction when another transaction is executing its begin operations, until

all local begin operations have returned. Likewise, we have to delay the begin operations if another transaction is already in the process of committing, until the commit is finished.

With this kind of begin synchronization and the implicit synchronization by the ACP, we can now assume that all subtransactions of a global transaction $t_i$ start at the same time $B_i$ and end at the same time $C_i$ in all underlying component systems. Therefore, the version functions of the various component systems are "synchronized" as well. This consideration leads us to a criterion for globally correct execution, as expressed by the following theorem.

**Theorem 2:**
Let s be a schedule in a federated database system whose component systems guarantee local SSI for the local schedules (and with an ACP based on a conceptually centralized coordinator). If each federated transaction issues local begin operations on each potentially accessed component system and the federation level prevents global commit operations from being concurrently executed with a transaction's set of local begin operations, then s is SSI.

■

This approach is relatively easy to implement, but it has potential performance problems: At the start time of a federated transaction, it is not necessarily known which component systems the transaction may access during its execution (nor can this information be easily inferred from the transaction program). To be on the safe side, local begin operations would have to be executed in a conservatively chosen set of component systems if not even all component systems of the federation. Although each of the local begin operations is fairly inexpensive, the total overhead may incur a significant penalty in terms of the transaction throughput. Even worse, since global commit operations may have to be delayed by begin operations, there is also a potentially severe performance drawback as far as transaction response time is concerned.

### *3.3 Optimistic Approach: Testing for Global SSI Violations*

Looking again at the example in the previous Subsection 3.2, we observe a specific situation: the two transactions are concurrent in one database ($DB_1$), while having a reads-from relationship in the serial execution in the other database ($DB_2$). This type of situation is in fact the only case when global SSI can be violated despite all component systems enforcing local SSI. Note that the concurrent execution in one database implies that the writesets of the two transactions must be disjoint because of the local SSI property. However, this concurrent execution may result in a version function that is incompatible with the version function of the second component system. Further note that in the component system where the subtransactions are concurrent, no reads-from relationship is possible between the two transactions. So it seems that all we need to do is to avoid that two transactions are a) concurrent in one database and b) have a (serial) reads-from relationship in another database. Unfortunately, the following example shows that this considerations does still not capture all relevant cases, as it disregards schedules where one transaction accesses only a proper subset of the databases accessed by the other transaction (e.g., only one out of DB1 and DB2):

$DB_1$: $r_1^{(1)}(a_0)$ $C_1$

$DB_2$: $r_2^{(2)}(x_0)$ $w_2^{(2)}(x_2)$ $C_2$ $r_1^{(2)}(x_2)$ $w_1^{(2)}(x_1)$ $C_1$

This global schedule is not SSI, because both transactions are concurrent and write the same object x. However, our above consideration does not capture this unacceptable case, because all operations of $t_1$ are strictly after those of $t_2$ in $DB_2$, and $t_2$ does not have any operations in $DB_1$. In the approach of the previous Subsection 3.2 with synchronized begin operations, such an execution would be impossible because the local begin operation of $t_1$ in $DB_2$ would make

$t_1$ locally start before $t_2$ so that $t_1$ would read $x_0$ rather than $x_2$. However, the synchronized begin operations would have performance drawbacks that we would like to avoid in the current approach. Here, to fix the above problem, we first extend our model of a schedule by introducing additional pseudo-operations $B_i^{(k)}$ and $C_i^{(k)}$ for each transaction $t_i$ and each database $DB_k$ that is *not* accessed by $t_i$. The ordering of the artificial $C_i^{(k)}$ operations is such that it is compatible with all real commit operations in the other databases and arbitrary with regard to read and write operations, and the artificial $B_i^{(k)}$ operations are placed immediately before the corresponding local commit. Unlike the local begin operations of the previous Subsection 3.2, the newly introduced operations here are indeed only placeholders for the purpose of correctness reasoning about federated schedules; they do no have to be issued in the real system. The net effect of this syntactical extension is that we can now easily tell, in our model, for each pair of transactions $t_i$ and $t_j$ and each $DB_k$ if $t_i^{(k)}$ fully precedes $t_j^{(k)}$, denoted $t_i^{(k)} < t_j^{(k)}$ (or more precisely: $C_i^{(k)} < B_j^{(k)}$), or if $t_j^{(k)}$ fully precedes $t_i^{(k)}$, or if $t_i^{(k)}$ and $t_j^{(k)}$ are concurrent, the latter being denoted as $t_i^{(k)} \parallel t_j^{(k)}$.

**Theorem 3: (characterization of the global SSI version function)**

Let s be a schedule, extended in the above way, that is executed in a federated database system (using an ACP) whose component systems guarantee local SSI. The version function of s, that is, the union of the version functions of the underlying local schedules, satisfies (SSI-V) if and only if there are no two transactions $t_i$ and $t_j$ and no two databases $DB_k$ and $DB_l$ such that $t_i^{(k)} < t_j^{(k)}$, $t_j^{(k)}$ reads from $t_i^{(k)}$, and $t_i^{(l)} \parallel t_j^{(l)}$.

■

From this necessary and sufficient condition for a global schedule to satisfy (SSI-V), we can further derive the following theorem that tells us when a global schedule is SSI.

**Theorem 4: (sufficient and necessary condition for global SSI)**

Let s be an extended schedule in a federated database system (using an ACP) whose component systems guarantee local SSI. Then s is SSI if and only if there are no two transactions $t_i$ and $t_j$ and no two databases $DB_k$ and $DB_l$ such that $t_i^{(k)} < t_j^{(k)}$, $t_j^{(k)}$ reads from $t_i^{(k)}$, and $t_i^{(l)} \parallel t_j^{(l)}$.

■

This theorem forms the basis of an algorithm to enforce global SSI, under the assumption that all component systems guarantee local SSI. The idea is to monitor the three conditions of the theorem and take appropriate actions at the federated level when all three are conjunctively violated for a pair of transactions. Monitoring the property whether two transactions execute serially or concurrently in one component system is straightforward. Monitoring the reads-from relationship, however, is very difficult if not infeasible in a practical system setting. The problem is that we cannot tell by merely observing the operation requests and replies at a component system's interface which version, identified by the creating transaction's number, was read by a read operation. So we need to build an actual algorithm for global SSI on a coarser version of Theorem 4 where the reads-from condition is omitted and thus the set of allowed schedules is restricted even further.

**Corollary 1: (sufficient condition for global SSI)**

Let s be an extended schedule in a federated database system (using an ACP) whose component systems guarantee local SSI. Then s is SSI if there are no two transactions $t_i$ and $t_j$ and no two databases $DB_k$ and $DB_l$ such that $t_i^{(k)} < t_j^{(k)}$ and $t_i^{(l)} \parallel t_j^{(l)}$.

■

So for global SSI to hold, the situation to be disallowed is that the subtransactions of two global transactions are running concurrently in one database and serial in another one, regard-

less of whether one transaction reads from the other transaction or not. This leads to the following algorithm for ensuring the SSI property of global schedules.

In the federation layer, we assign a unique timestamp $B_i^{(k)}$ to the subtransaction of transaction $t_i$ in database $DB_k$ before it issues its first operation there. This does not require any additional operation on the component system, only some bookkeeping by at the federation level. When transaction $t_i$ requests its global commit, we assign a timestamp $C_i$ to it. In addition, we keep information about the relative execution order of subtransactions in an array R[i,j], where

$$R[i,j] = \begin{cases} \textit{serial}, & \text{if there is a database } DB_k \text{ where } C_i < B_j^{(k)} \\ \textit{concurrent}, & \text{if there is a database } DB_k \text{ where } B_i^{(k)} < B_j^{(k)} < C_i \text{ or } B_j^{(k)} < B_i^{(k)} < C_j \\ \textit{undefined}, & \text{if at least one of the transactions has not yet submitted any operation} \end{cases}$$

Initially, there are no transactions in the system, so R is empty. Once a transaction $t_i$ enters the system, all the $B_i^{(k)}$ and $C_i$ values are set to $\infty$, and R[i,j]=R[j,i]=undefined for all active transactions $t_j$. Upon the first operation of transaction $t_i$ in database $DB_k$, the timestamp $B_i^{(k)}$ is assigned and R is updated as follows:

```
for all transactions t_j in the system do
    if (C_j < B_i^(k)) then // serial execution
       if (R[i,j]=concurrent) then abort t_i else R[i,j] := R[j,i] := serial
    else if (B_j^(k) < B_i^(k)) then // concurrent execution
       if (R[i,j]=serial) then abort t_i else R[i,j] := R[j,i] := concurrent
```

When transaction $t_i$ attempts to commit, our bookkeeping needs to add "pseudo operations" for each database $DB_k$ that $t_i$ has not accessed at all. To do so, we set $B_i^{(k)}=C_i$ and update R as if $t_i$ just submitted an operation to this database. If adding these pseudo operations does not force $t_i$ to abort, we allow it to commit.

Obviously this algorithm can cause unnecessary aborts, since it does not take into account the actual reads-from relationship. However, as noted before, this is an inescapable consequence of the fact that the federation layer cannot easily observe the details of the local executions in a real-life system setting. Despite this drawback of possibly unnecessary aborts, the algorithm appears to be significantly more efficient than the begin-synchronization approach of the previous Subsection 3.2. In particular, the presented algorithm incurs overhead only for those component systems that are actually accessed, and does not delay any begin or commit requests.

### *3.4 Remarks On Global Snapshot Isolation with Non-SSI Component Systems*

In the previous two subsections we have shown how to guarantee global SSI when all component systems guarantee local SSI. It would, of course, also be a desirable option to enforce global SSI even if some component systems may support a correctness criterion other than local SSI. A particularly interesting combination would be one database supporting local SSI and another database supporting standard conflict-serializability. Assume that a component system guarantees conflict-serializability in combination with the avoidance of cascading aborts (ACA). Then the first read operation of a transaction establishes a reads-from relationship that constrains the feasibility of subsequent reads if we want to guarantee that the resulting schedule is also locally SSI (i.e., actually a member of the schedule class SR $\cap$ ACA $\cap$ SSI). The transaction must not read any data that are committed later than at the time of that first read. Brute-force methods to enforce this constraint are conceivable (e.g., by aborting all transactions that initiate a commit between our transaction's first read and its commit), but it is very likely that they restrict the possible concurrency in an undue manner. This problem

alone has discouraged us from proceeding further along these lines. In addition, one would have to ensure that the version functions of the different local schedules are compatible and that writesets of concurrent transactions are disjoint, both at the level of global transactions. Obviously, this setting calls for future research.

# 4 Guaranteeing Global Serializability

Although SSI is a popular option in practice, there are certainly many mission-critical applications that still demand the more rigid correctness criterion of global (conflict-) serializability (SR). In this section, we study the problem of ensuring global SR in the presence of component systems that merely provide SSI. We develop an algorithm that extends the well-known ticket method [BGRS91, GRS94] so that it supports local SSI schedulers, while guaranteeing global SR. In Subsection 4.1, we briefly review the standard ticket method and discuss its benefits and potential shortcomings. In Subsection 4.2, we show that, with a minor extension, the ticket method yields correct executions even when applied to local SSI schedulers, but point out that this approach has certain severe drawbacks. In Subsection 4.3, we finally present a generalization of the ticket method that increases performance for applications with a large fraction of read-only (sub-)transactions.

## *4.1 Benefits and Potential Shortcomings of the Ticket Method*

A ticket is a dedicated data object of type "counter" (or some other numerical type) in a component, used solely for the purpose of global concurrency control. Each global subtransaction must read the current value of the ticket and write back an increased value at some point during its execution (the so-called *take-a-ticket* operation). The ordering of the ticket values read by two global subtransaction reflects the local serialization order of the two subtransactions. A *ticket graph* is maintained at the federation level to detect incompatible local serialization orders. This graph has the global transactions as nodes, and there is an edge from transaction $t_i$ to transaction $t_j$ if the ticket of $t_i$ is smaller than that of $t_j$ in some component database. It has been shown in [GRS94] that the global schedule is serializable if and only if the ticket graph does not contain a cycle, provided that all component systems generate only locally SR schedules.

This method is easy and efficient to implement; one merely needs to add the take-a-ticket operation to each subtransaction and to manage the global ticket graph, i.e., searching for cycles when a global transaction attempts to commit (or at some earlier point). Upon detecting a cycle, one (or more) of the involved transactions must be aborted. For component systems with the property of allowing only rigorous local schedules, it is not even necessary to take a ticket at all; rather the time of the commit operation can be used as an implicit ticket. So the ticket method has the particularly nice property of incurring overhead only for non-rigorous component systems, even with transactions that access both rigorous and non-rigorous component systems. This makes the ticket method a very elegant and versatile algorithm for federated concurrency control.

The ticket method has two potential problems, however. First, the ticket object may be a potential bottleneck in a component database, since all global subtransactions have to read and write the ticket. Second and much more severely, having to write the ticket turns read-only transactions into read-write transactions, thus preventing the possibility of specific optimizations for read-only (sub-) transactions (e.g., Oracle's "Set Transaction Read Only" option).

### 4.2 Applying the Ticket Method to SSI Component Systems

The standard ticket method requires that local schedulers generate (conflict-) serializable schedules. Thus, it is not clear if the method can incorporate component systems that provide non-serializable SSI schedules. Consider the effects of the additional take-a-ticket operation to the execution of global subtransactions on an SSI component system. For each pair of concurrent global subtransactions on such a database, both write at least one common object, the ticket, so that one of them must inevitably be aborted to ensure local SSI. The resulting local schedule is therefore trivially serializable, because it is in fact already serial, and the ordering of the ticket values of two global subtransactions reflects their local serial(ization) order. Note that this does not make the entire global transactions serial; concurrency is still feasible in other (non-SSI, but SR) databases. Nevertheless, sequentializing all global subtransactions in an SSI component system is a dramatic loss of performance and would usually be considered as an overly high price for global consistency.

What about local transactions on an SSI component system, i.e., transactions that solely access this database and are not routed through the federation level? Those transactions do not have to take a ticket in the original ticket method (as any additional overhead for them could possibly be considered as a breach of the local database autonomy). Nevertheless, global serializability is still ensured by means of the ticket-graph testing, as cases where local transactions cause the serialization order of global transactions to be reversed would lead to local cycles and are thus detectable. Unfortunately, with SSI component system, tickets for global subtransactions alone are insufficient to detect those critical situations caused by local. An example schedule for an SSI database with ticket object T, global subtransactions $t_1$ and $t_2$, and a local transaction $t_3$ is the following:

$$r_3(y_0)\ r_1(y_0)\ w_1(y_1)\ r_1(T_0)\ w_1(T_1)\ C_1\ r_2(x_0)\ r_2(T_1)\ w_2(T_2)\ C_2\ r_3(x_0)\ w_3(x_3)\ C_3$$

The schedule including the ticket operations is SSI and even SR with the ticket operations removed, but the ticket ordering of $t_1$ and $t_2$ contradicts the serialization order of the ticket-less schedule.

A possible and in fact the only solution for this problem is to add take-a-ticket operations also to all local transactions. This can be done without modifying the application programs themselves, for example, by changing the stub code of the commit. However, there is still a major problem unsolved, as the forced serial execution on an SSI component system, discussed above, would now apply to both global subtransactions and local transactions. So the resulting performance loss would involve the local transactions as well, and this is surely unacceptable in almost all cases. In the next subsection we will present a generalization of the ticket method to cope with SSI databases while avoiding such performance problems for the most important case of read-only (sub-) transactions. There is, however, no panacea that can cope with the most general case without any drawbacks.

### 4.3 Generalizing Tickets for Read-Only Subtransactions on SSI Component Systems

In this subsection, we present a generalization of the ticket method that allows read-only transactions to run as concurrently as the component systems allow them (i.e., without additional restrictions imposed by the federated concurrency control). So for application environments that are dominated by read-only transactions but exhibit infrequent read-write transactions as well, our approach reconciles the consistency quality of global serializability with a sustained high performance.

Each global subtransaction and local transaction has to be marked “read-only” or “read-write” at its beginning; an unmarked transaction is supposed to be read-write by default. A transaction that is declared as read-only before can be re-labeled as read-write at any point during its execution (with certain consequences in terms of its performance, however). A global transaction is read-only if all its subtransactions are read-only; otherwise it is a global read-write transaction. We will first discuss how to deal with read-write transactions in a rather crude, unoptimized way, and then show how read-only transactions can be handled in a much more efficient manner.

**Read-Write Transactions:**

Our extended ticket method requires that all global read-write transactions are executed serially. That is, the federated transaction manager has to ensure that at most one global read-write transaction is active at a time. There is no such restriction, however, for global read-only transactions or for local (read-write) transactions. Each read-write subtransaction of a global read-write transaction takes a ticket as in the standard ticket method. Read-only subtransactions of read-write transactions only need to read the corresponding ticket object, as further discussed below. Note that tickets are still necessary for global read-write transactions to correctly handle the potential interference with local transactions.

Although the sequentialization of global read-write transactions appears very restrictive, it is no more restrictive than in the original ticket method if SSI component systems are part of the federation. On the other hand, our generalized method is much less restrictive with regard to read-only transactions and local transactions.

**Read-Only Transactions:**

The problem with read-only transactions is that the usual updating of ticket objects would turn them into read-write transactions, with the obvious adverse implications. A careful analysis of the possible cases, however, shows that it is sufficient if read-only (sub-) transactions merely read the ticket object.

A subtransaction’s ticket value shows its position in the locally equivalent serial schedule. When a component system generates SSI schedules, the ticket value that a transaction $t_i$ reads from the corresponding database $DB_k$ depends only on the position of its first local operation $B_i^{(k)}$. The transaction reads from other transactions that were committed before $B_i^{(k)}$, so its position in the equivalent serial schedule must be behind them. Its ticket value must therefore be greater than that of all transactions that committed earlier. On the other hand, the transaction $t_i$ does not see updates made by transactions that commit after $B_i^{(k)}$; so it must precede them in the equivalent serial schedule, hence its ticket value must be smaller than the tickets of those transactions. Thus, a feasible solution is to assign to a read-only subtransaction a ticket value that is strictly in between the value that was actually read from the ticket object and the next higher possible value that a read-write subtransaction may write into the ticket object. This approach can be implemented very easily. For example, if ticket objects are of type integer, we can restrict the actual ticket-object values to even integers with read-write subtransactions always incrementing the ticket object by two, and for read-only subtransactions we can use the odd integer that immediately follows the actually read ticket value for the purpose of building the global ticket graph. The fact that this may result in multiple read-only subtransactions with the same ticket value is acceptable in our protocol.

Figure 2 shows an example, with the ticket values that the read-write transactions write in the databases denoted by the numbers below the transaction boxes. The smaller white and black boxes denote the first operations of two read-only transactions $t_W$ and $t_B$ that span all three databases. Looking at the white transaction in the first database, we see that its ticket value

must be greater than 6 (which is the value that $t_1$ wrote), but smaller than 8 (which is what $t_2$ wrote), so a possible ticket value for $t_W$ is 7. the tickets for the other subtransactions are assigned analogously.

Using these ticket values, we can now test if the execution of the two read-only transactions in Figure 2 leads to a globally serializable execution, i.e., a cycle-free global ticket graph. First consider the black read-only transaction $t_B$. In the databases $DB_1$ and $DB_3$, it starts its execution after the commit of $t_2$ (and $t_1$), so its ticket is greater than that of $t_2$ and $t_1$ (for example, the ticket of $t_2$ in $DB_1$ is 8, while that of $t_B$ would be 9). In $DB_2$, $t_2$ had no operations, and $t_B$ started after $T_1$ committed. As a result, $t_B$ is always executed after $t_1$ and $t_2$ committed; the corresponding ticket value graph has no cycle. In an equivalent global serial schedule, $t_B$ must be executed after $t_1$ and $t_2$.

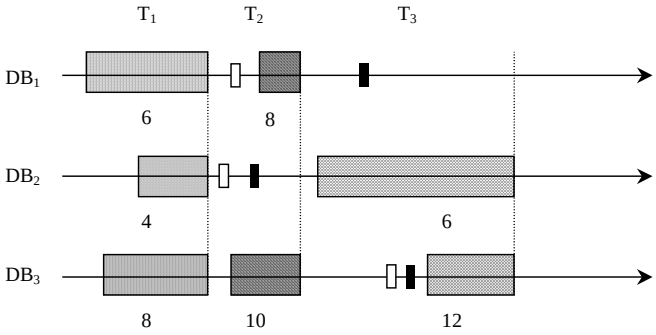


**Figure 2 - Assigning a ticket value to read-only transactions**

As for the white transaction $t_W$, it executes its first operations in $DB_2$ and $DB_3$ before $t_3$ committed, so its ticket value is smaller than that of $t_3$ in both databases. On the other hand, it makes its first operation in $DB_1$ before $t_2$ committed, but in $DB_3$ after $t_2$ committed. It is therefore possible that $t_W$ reads values written by $t_2$ in $DB_3$ but not in $DB_1$, so the state of the global database seen by $t_W$ could be inconsistent. This is captured by the ordering of the tickets of these transactions. The ticket of $t_W$ in $DB_1$ is smaller than that of $t_2$ (7 vs. 8), while it is greater in $DB_3$ (11 vs. 10). The corresponding ticket graph therefore has a cycle between these two transactions which is detected when $t_W$ attempts to commit.

Local transactions are incorporated in this protocol as before: read-write transactions need to take tickets as usual, whereas read-only transactions merely need to read the ticket object. These considerations hold for SSI component systems. Our method can, however, easily be combined with standard tickets for other types of component systems, if such databases participate in the federation. For example, if some database generates schedules where the serialization order of the transactions reflects the order of their commit operations, the “implicit ticket” method [GRS94] can be applied for this database (i.e., no explicit tickets need to be taken). In databases for which conventional (conflict-) serializability is guaranteed, all global subtransactions submit the usual take-a-ticket, whereas local transactions do not need to take a ticket as in the original ticket method.

## 5 Concluding Remarks

In the last few years the subject of federated transactions has been largely disregarded by the research community, despite the fact that the results from the late eighties and early nineties do not provide practically viable solutions. A major reason for this situation probably is that

many application classes are perfectly satisfied with a loose coupling of database, no or very little care about mutual consistency, and possibly application-level solutions to avoid special types of severe inconsistencies. The expected proliferation of advanced applications like virtual-enterprise workflows, electronic-commerce agents and brokers, etc. should rekindle the community's interest in federations that span widely distributed, highly heterogeneous component systems while also requiring a highly dependable IT infrastructure and thus highly consistent data. The problems of federated information systems are inherently hard, but we should not give up too early and rather pursue long-term efforts towards well-founded but also practically viable solutions. This paper should be understood as a first step along these lines. In particular, we have aimed to incorporate important aspects of commercial database systems into a systematic and rigorously founded approach to federated transaction management.

## References


[ABJ97] V. Atluri, E. Bertino, S. Jajodia: *A Theoretical Formulation for Degrees of Isolation in Databases*, Information and Software Technology Vol.39 No.1, Elsevier Science, 1997.

[BBGM+95] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, P. O'Neil: *A Critique of ANSI SQL Isolation Levels*. ACM SIGMOD International Conference on Management of Data, San Jose, 1995.

[BE96] O.A. Bukhres, A.K. Elmagarmid (eds): *Object Oriented Multidatabase Systems: A Solution for Advanced Applications*. Prentice Hall, 1996.

[BGRS91] Y. Breitbart, D. Georgakopoulos, M. Rusinkiewciz, A. Silberschatz: *On Rigorous Transaction Scheduling*. IEEE Transactions on Software Engineering Vol. 17 No. 9, 1991.

[BGS92] Y. Breitbart, H. Garcia-Molina, A. Silberschatz: *Overview of Multidatabase Transaction Management*. VLDB Journal, Volume 1, No. 2, 1992.

[BGS95] Y. Breitbart, H. Garcia-Molina, A. Silberschatz: *Transaction Management in Multidatabase Systems*. In: W. Kim (Ed.): *Modern Database Systems*, ACM Press, 1995.

[BHG87] P.A. Bernstein, V. Hadzilacos, N. Goodman: *Concurrency Control and Recovery in Database Systems*. Addison Wesley Press, 1987.

[BS92] Y. Breitbart, A. Silberschatz: *Strong Recoverability in Multidatabase Systems*. Second International Workshop on Research Issues on Data Engineering: Transaction and Query Processing, Tempe, 1992.

[Conr97] S. Conrad: *Federated Database Systems* (in German), Springer, 1997.

[DSW94] A. Deacon, H.-J. Schek, G. Weikum: *Semantics-based Multilevel Transaction Management in Federated Systems*. IEEE International Conference on Data Engineering, Houston, 1994.

[Elm92] A.K. Elmagarmid (Ed.): *Database Transaction Models For Advanced Applications*, Morgan Kaufmann Publishers, 1992.

[GRS94] D. Georgakopoulos, M. Rusinkiewicz, A.P. Sheth: *Using Tickets to Enforce the Serializability of Multidatabase Transactions*. IEEE Transactions on Knowledge and Data Engineering, Volume 6, Number 1, February 1994.

[HBP93] A.R. Hurson, M.W. Bright, S.H. Pakzad: *Multidatabase Systems – Advanced Solution for Global Information Sharing*. IEEE Computer Society Press, 1993.

[HWW98] B. Holtkamp, N. Weißenberg, X. Wu: *VHDBS: A Federated Database System for Electronic Commerce*, EURO-MED NET, 1998.

[LMR90] W. Litwin, L. Mark, N. Roussopoulos: *Interoperability of Multiple Autonomous Databases*. ACM Computing Surveys, Vol. 22, 1990.

[ÖV98] M.T. Özsu, P. Valduriez: *Principles of Distributed Database Systems*. 2nd Edition, Prentice Hall, 1998.

[OMG97] Object Management Group, Inc: *Object Transaction Service 1.1*, 1997.

[Orac95] Oracle Corporation: *Concurrency Control, Transaction Isolation and Serializability in SQL92 and Oracle7*. White Paper. 1995.

[Orac97] Oracle Corporation: *Oracle8 Concepts, Release 8.0: Chapter 23, Data Concurrency and Consistency*, 1997.

[Pa86] C. Papadimitriou: *The Theory of Database Concurrency Control*. Computer Science Press, 1986.

[Raz92] Y. Raz: *The Principle of Commit Ordering or Guaranteeing Serializability in a Heterogeneous Environment of Multiple Autonomous Resource Managers Using Atomic Commitment*. International Conference on Very Large Databases, Vancouver, 1992.

[SL90] A.P. Sheth, J.A. Larson: *Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases*. ACM Computing Surveys, 22(2), 1990.

[SSW95] W. Schaad, H.-J. Schek, G. Weikum: *Implementation and Performance of Multi-level Transaction Management in a Multidatabase Environment.* IEEE International Workshop on Research Issues in Data Engineering, Taipeh, 1995.

[SW99] R. Schenkel, G. Weikum: *Experiences with Building a Federated Transaction Manager based on CORBA OTS*, in: Proc. of the 2nd International Workshop of Engineering Federated Information Systems (EFIS '99), Kühlungsborn, 1999.

[VEH92] J. Veijalainen, F. Eliassen, B. Holtkamp: *The S-transaction Model*. In: [Elm92]

[Weihl89] W.E. Weihl: *Local Atomicity Properties: Modular Concurrency Control for Abstract Data Types*. ACM Transactions on Programming Languages and Systems, 11(2), 1989.

[Wied92] G. Wiederhold: *Mediators in the Architecture of Future Information Systems*. IEEE Computer, 25(3), 1992.

[WW97] X. Wu, N. Weißenberg: *A Graphical Interface for Cooperative Access to Distributed and Heterogeneous Database Systems.* International Database Engineering and Applications Symposium, Montreal, 1997.

[Wu96] X. Wu: *An Architecture for Interoperation of Distributed Heterogenous Database Systems*. International Conference on Database and Expert Systems Applications, Zürich, 1996.

# Appendix: Proofs of Theorems

### Proof of Theorem 1:

*a)* With the (SSI-V) property as our premise and our overall assumption that each write in a transaction must be preceded by a read of the same object, the SSI version order $\ll_s$ is the only version order that can render the MVSG of a schedule acyclic. The theorem then follows immediately from the standard MVSR theorem (see Subsection 2.1), because SSI-MVSG contains the same edges as the ordinary MVSG.

*b) "if":*

Assume that the schedule is not SSI. Then there must be at least two concurrent transactions $t_i$ and $t_j$ that write the same object x. Because of the read-before-write restriction, both transactions read x before writing it, say $t_i$ reads $x_k$ and $t_j$ reads $x_l$. By the definition of the version function, $t_k$ must have been committed when $t_i$ started, so $x_k \ll_s x_i$ by the definition of the SSI version order $\ll_s$. The same holds for $t_l$ and $t_j$, so we obtain $x_l \ll_s x_j$. Additionally, $t_k$ must commit before $t_j$, because $t_i$ reads from $t_k$ (so $t_k$ was committed when $t_i$ started) and $t_j$ runs in parallel with $t_i$, so it commits after the start of $t_i$. This yields $x_k \ll_s x_j$ and, analogously, $x_l \ll_s x_i$. The SSI-MVSG now contains the edges

- $t_i \rightarrow t_j$ labeled with "x", because $r_i(x_k)$, $w_j(x_j)$ and $x_k \ll_s x_j$, and
- $t_j \rightarrow t_i$ labeled with "x", because $r_j(x_l)$, $w_i(x_i)$ and $x_l \ll_s x_i$.

But this is an x-cycle and therefore a contradiction to the precondition.

*"only if":*

Assume there is an x-cycle $t_{i1} \rightarrow t_{i2} \rightarrow ... \rightarrow t_{in}=t_{i1}$ in SSI-MVSG of the schedule s, $s \in$ SSI. If there is more than one, select one with minimal length. Without loss of generality, assume that the transactions in the cycle are (re-)numbered such that $t_1 \rightarrow t_2 \rightarrow ... \rightarrow t_n=t_i$.

We show first that there must be one edge in this cycle that was added due to the (RW)-rule in the definition of SSI-MVSG. Both the (WR)-rule and the (WW)-rule add edges from a transaction that writes an older version of x to a transaction that writes a younger version of x. Because the SSI version order $\ll_s$ is defined by the commit order of the transaction, one of those edges from $t_i$ to $t_j$ means that $t_i$ commits before $t_j$, formally $C_i<C_j$. But

we have a cycle. This would mean that $C_1 < C_1$, which is a contradiction, so there must be at least one edge $t_i \rightarrow t_{i+1}$ where $C_i > C_{i+1}$, we choose the first such edge.

This edge was added because we have $r_i(x_l)$ and $w_{i+1}(x_{i+1})$ in the schedule, and $C_l < C_{i+1}$. We also have $C_l < B_i$ and there is no other commit of a transaction in between them that writes x, because $t_i$ reads x from $t_l$. Putting this together, we obtain $C_l < B_i < C_{i+1} < C_i$. We have thus shown that $C_i$ and $C_{i+1}$ run concurrently. However, this does not mean that the schedule is not in SSI, because $t_i$ does not necessarily write x.

The read-before-write rule means that $t_{i+1}$ must read a version $x_p$ of x before writing it. To show a contradiction, we now discuss the different possibilities for the transaction $t_p$ from which transaction $t_{i+1}$ can read.

(i) If $t_{i+1}$ reads from a transaction that committed before $t_l$, we get $C_p < B_{i+1} < C_l$. But this means that $t_l$ and $t_{i+1}$ are concurrent and write the same object x, so the schedule s cannot be in SSI, which is a contradiction.

(ii) If $t_{i+1}$ reads x from $C_l$, we have l=p and $C_l < B_{i+1} < C_{i+1}$. This means that there is an x-edge in the SSI-MVSG from $t_l$ to $t_{i+1}$ because of the (WR)-rule. But then $t_l$ cannot be part of the cycle, or we could replace the edges $t_l \rightarrow t_i \rightarrow t_{i+1}$ by the edge $t_l \rightarrow t_{i+1}$ and would get a shorter cycle, contrary to the minimality of the selected cycle.

So we know that $t_i$ has an incoming x-edge, and that this edge does not come from $t_l$, which is $t_i$'s only incoming x-edge of (WR) type ($t_i$ reads x only from $t_l$). Therefore the incoming x-edge must be an edge of type (RW) or (WW). But the definition of the SSI-MVSG says that $t_i$ must write x in order to have such an incoming edge, so we have shown that the concurrent transactions $t_i$ and $t_{i+1}$ both write x, which is a contradiction.

(iii) If $t_{i+1}$ reads x from a transaction $t_q$ that committed after $t_i$ started, we have the ordering $C_l < B_i < C_q < B_{i+1} < C_{i+1} < C_i$. If $t_l$ was not part of the cycle, we could show as before that $t_i$ must write x, so that s would not be in SSI, and $t_l$ is on the cycle.

If there is no other transaction in between $t_l$ and $t_q$ that writes x, the graph contains the edges $t_l \rightarrow t_q$ ($t_q$ reads x from $t_l$) and $t_q \rightarrow t_{i+1}$ ($t_{i+1}$ reads x from $t_q$). If we replace the edges $t_l \rightarrow t_i \rightarrow t_{i+1}$ by the edges $t_l \rightarrow t_q \rightarrow t_{i+1}$, we get a cycle with the same length. Additionally, we can replace the edge from a larger to a smaller version ($t_i \rightarrow t_{i+1}$) by edges that respect the version order ($C_l < C_q < C_{i+1}$). As shown before there must be another version-order-reversing edge in the cycle, so we can restart the proof at the beginning. Since the cycle has finite length, we can do so only a finite number of times until one of the other cases applies.

If there is a sequence of transactions $t_{r1}$ ... $t_{rm}$ between $t_l$ and $t_q$ that write x, $t_{r1}$ must read x from $t_l$, $t_{r2}$ must read x from $t_{r1}$, and so on, until $t_q$ must read x from $t_{rm}$. Therefore there are edges $t_l \rightarrow t_{r1}$ and $t_{r1} \rightarrow t_{ij+1}$ (because $r_{r1}(x_l)$, $w_{ij+1}(x_{ij+1})$ and $x_{ij+1} \gg_s x_l$). Now we can replace $t_l \rightarrow t_{ij} \rightarrow t_{ij+1}$ by $t_l \rightarrow t_{r1} \rightarrow t_{ij+1}$ which are edges that respect the version order, and the same argument as before applies.

■

**Proof of Theorem 2:**

We show that the global schedule is SSI by showing that both SSI properties (SSI-V) and (SSI-W) hold for the global schedule.

(SSI-V): Assume that there is a transaction $t_i$ in the global schedule that reads a version of object x in database $DB_l$ that another transaction $t_j$ wrote, but that $t_j$ is not the "right"

transaction in the sense of (SSI-V). Then $t_j$ is either uncommitted, committed after the begin of $t_i$, or another transaction $t_k$ wrote x and committed between the commit of $t_j$ and the begin of $t_i$. But the begin and commit operations in all databases are synchronized, so if any of these three cases holds globally, it does also hold in $DB_l$. Therefore the local subschedule in $DB_l$ does not satisfy (SSI-V), contrary to the assumption that all local schedules are SSI.

(SSI-W): Assume that there are two transactions $t_i$ and $t_j$ in the global schedule that execute concurrently and that write an object x in database $DB_l$. Because the begin and commit operations in all databases are synchronized, $t_i^{(l)}$ and $t_j^{(l)}$ are executing concurrently, too. But then the local schedule is not SSI, which is a contradiction.

■

**Proof of Theorem 3:**

"There are no ..." ⇒ "(SSI-V) holds":

Assume (SSI-V) does not hold. Then there is a transaction $t_i$ that reads a version of an object x that is either "too old" or "too new", with x being part of database $DB_l$. All reads and writes of x are therefore part of subtransactions in this database.

If the version is too old, $t_i$ does not read x from the last transaction $t_j$ that wrote x and committed before $t_i$ started. So there must be another transaction $t_k$ that committed in between: $C_j < C_k < B_i$, and both $t_j$ and $t_k$ write x. The subtransaction $t_i^{(l)}$ of transaction $t_i$ in $DB_l$ cannot begin before the begin of $t_i$ itself, so $B_i < B_i^{(l)}$. With the ACP assumption, together this yields $C_j^{(l)} < C_k^{(l)} < B_i^{(l)}$. This means that $t_i^{(l)}$ does not read x from the last transaction that committed before $t_i^{(l)}$ started in the local database system, but this is a contradiction to $DB_l$ guaranteeing SSI for all local subtransactions.

If the version is too new, $t_i$ reads x from a transaction $t_j$ that was not yet committed when $t_i$ globally started. Because the local schedulers guarantee SSI, $t_j^{(l)}$ must have been committed when $t_i^{(l)}$ executed its first operation, so $C_j^{(l)} < B_i^{(l)}$, and therefore $t_j^{(l)} < t_i^{(l)}$. On the other hand, there must be at least one database $DB_k$ where $t_i^{(k)}$ started before $t_j^{(k)}$ committed, because globally $t_i$ started before $t_j$ committed, so $B_i^{(k)} < C_j^{(k)}$. Together with $B_j^{(k)} < C_j^{(k)}$ and $C_j < C_i$ this yields $t_i^{(k)} \parallel t_j^{(k)}$, which is a contradiction.

"(SSI-V) holds" ⇒ "There are no ...":

Assume that there are databases $DB_l$ where $t_i^{(l)} < t_j^{(l)}$ and $t_j^{(l)}$ reads x from $t_i^{(l)}$, and $DB_k$ where $t_i^{(k)} \parallel t_j^{(k)}$. Because $t_j$ reads from $t_i$ in $DB_l$, we have $B_i^{(l)} < C_i^{(l)} < B_j^{(l)}$. In database $DB_k$, $t_i^{(k)}$ and $t_j^{(k)}$ run concurrently, so that either $B_i^{(k)} < B_j^{(k)} < C_i^{(k)}$ or $B_j^{(k)} < B_i^{(k)} < C_i^{(k)}$. Whatever case applies, we see that $t_j^{(k)}$ begins before $t_i^{(k)}$ commits, so that the global transactions $t_i$ and $t_j$ run concurrently. This means that $t_j$ reads x from a globally concurrent transaction, which is a contradiction to the version function satisfying (SSI-V).

■

**Proof of Theorem 4:**

By Theorem 3, we already know that s satisfies (SSI-V). To show that s also satisfies (SSI-W), we use the characterization by the SSI-MVSG that we introduced in Section 2.2. The SSI-MVSG of the global schedule is the union of the SSI-MVSGs of the local subschedules. The SSI property of the global schedule then follows immediately from the SSI property of the local subschedules: If all local subschedules are SSI, there is no x so that there is an x-edge in one of the local SSI-MVSGs. But since an object exists in exactly one database, there is no x-edge in the global SSI-MVSG, too. So by Theorem 1 the global schedule is SSI.

■