Integrating Snapshot Isolation Into Transactional Federations

Ralf Schenkel and Gerhard Weikum
University of the Saarland
P.O.-Box 15 11 50
D-66041 Saarbrücken
Fax +49 681 302 4014
email {schenkel,weikum}@cs.uni-sb.de

Abstract. This paper reconsiders the problem of transactional federations, more specifically the concurrency control issue, with particular consideration of component systems that provide only snapshot isolation, which is the default setting in Oracle and widely used in practice. The paper derives criteria and practical protocols for guaranteeing global serializability at the federation level. The paper generalizes the well-known ticket method and develops novel federation-level graph testing methods to incorporate sub-serializability component systems like Oracle. These contributions are embedded in a practical project that built a CORBA-based federated database architecture suitable for modern Internet- or Intranet-based applications such as electronic commerce. This prototype system, which includes a federated transaction manager coined Trafic (Transactional Federation of Information Systems Based on CORBA), has been fully implemented with support for Oracle and O_2 as component systems and using Orbix as federation middleware. The paper presents performance measurements that demonstrate the viability of the developed concurrency control methods.

1 Introduction

1.1 Objectives of a Transactional Federation

With the ever-increasing demand for information integration both within and across enterprises, on one hand, and the proliferation of gateway and other middleware technologies such as ODBC, DCOM, JEB, or CORBA, on the other hand, there is renewed interest in providing seamless access to multiple, independently developed and largely autonomously operated databases [Ston98,HSC99,MK+99]. 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* [SL90,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]; 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.

Problems of ensuring the global serializability and atomicity of federated transactions have been intensively studied in the past. The proposed solutions range from imposing additional constraints on the transaction protocols of the component systems to building an additional transaction manager in the federated software layer [Weihl89, BGRS91,BS92,Raz92,GRS94]. However, none of the proposed approaches can by any means be considered as the universally "best" strategy for federated concurrency control. Rather the choice of the most appropriate strategy depends on the transaction protocols of the component systems, the operational characteristics of the applications, and other factors. Therefore a federated transaction manager should support a suite of different strategies and allow the application builder (or administrator staff) to select the most suitable protocols for the federation. In addition, it should be extensible to incorporate new transaction management strategies, specifically tailored to the needs of the application at hand. Furthermore, the federation must be able to cope with different isolation levels [BBGM+95] in the underlying component systems, for example, the "snapshot isolation" provided by Oracle [Orac99]. Such options for relaxed (local) serializability are widely popular in practice. They have only recently attracted also the research community with emphasis on formal properties [ALO00,ABJ97,BLL00, FLO+99,SW+99], but have still been more or less disregarded in the literature on federated transaction management.

1.2 Problems With Component Systems Providing Snapshot Isolation

This paper specifically addresses the problem of federated concurrency control for transactional federations based on component systems some of which provide only snapshot isolation. Note that Oracle falls into that category; although it can provide also full serializability using table locking, snapshot isolation is the best setting that can be achieved without extra coding in the application programs and widely used in practice. Further note that already running global transactions against multiple, autonomous Oracle instances all of which use the same snapshot-isolation protocol poses severe correctness problems as the resulting global execution cannot even be guaranteed to be globally snapshot-isolated. When different Oracle instances employ different isolation levels or the federation includes also database systems that provide full serializability, it becomes even more unclear how to guarantee global data consistency. As for global transaction atomicity and durability, on the other hand, a transactional federation can rely on the logging and recovery capabilities of the underlying component systems in conjunction with a standardized distributed commit protocol like XA or its CORBA counterpart OTS, as supported by virtually all commercial database systems. Therefore, this paper focuses on the concurrency control dimension of transactional federations.

For (local) transactions run under the *snapshot isolation* level, or *SI* for short, 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 acquiring a write lock: if the data object to be locked 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. This check aims to provide an additional level of sanity. Nonetheless, the protocol cannot ensure full (multiversion) serializability, with the following schedule as a counterexample (operation subscripts are transaction identifiers, x_i denotes the version of x generated by transaction t_i, and t_0 is a fictitious initializing transaction):

$$r_1(x_0)\ r_1(y_0) \qquad r_2(x_0)\ r_2(y_0) \qquad w_1(x_1)\ c_1 \qquad w_2(y_2)\ c_2$$

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. Given that such anomalies appear to be infrequent in practice, the protocol is widely used in Oracle applications. However, although application architects accept the risk of inconsistencies, they are not happy with his state of affairs, especially as applications become more complex, span organizational boundaries, and become even more mission-critical for business success.

1.3 Contribution and Outline of the Paper

The paper's contributions are threefold:

- We develop a formal model that allows us to reason about local versus global snapshot isolation and serializability in the context of federated transactions.
- Based on this model, we develop novel algorithms for federated concurrency control to ensure global serializability. Our algorithms leverage prior proposals, specifically the ticket method [GRS94] and federation-level graph testing with edges derived from SQL-

statement predicates [BGS92,SSW95], and generalize them for the newly considered setting with component systems that provide only local snapshot isolation (e.g., Oracle). In particular, we introduce a novel type of *online Snapshot-Isolation Multiversion-Serializability Graph (OSI-MVSG)* that allows us to enforce global serializability on top of component systems that merely provide snapshot isolation.

- We have implemented a full-fledged prototype of a transactional federation supporting Oracle and O_2 as component systems and using Orbix as middleware technology. We present performance measurements in a real system environment (as opposed to mere simulations) that demonstrate the viability of the developed methods and compare their performance properties.

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. Section 3 reconsiders established protocols for global serializability in the case that some systems provide only snapshot isolation, and develops a practically viable protocol to guarantee global serializability in this case. Section 4 presents the architecture of the fully operational prototype implementation. In Section 5, we discuss performance measurements of the presented algorithms. We conclude the paper with an outlook on future work.

2 Formal Underpinnings and Basic Techniques

2.1 Notation

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. <. 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 <.

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 $t_i^{(k)}$. For the scope of this paper, we assume that all LTs are known to the federation layer. This can be accomplished by simple request/reply re-routing without any modification of the code of LT applications.

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]. A means of testing this is the *multiversion serialization graph* (MVSG) for a schedule, which is acyclic if and only if the schedule is multiversion serializable.

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

(SI-V) *SI 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,

(SI-W) *disjoint writesets*: The writesets of two concurrent transactions are disjoint.

MVSR and SI are incompatible in that neither of the two classes contains the other. In [SW+99], we developed a graph characterization for schedules in SI, the so-called SI-MVSG, that extends the MVSG from multiversion serializability theory.

2.2 Characterization of Serializability on Top of Snapshot Isolation

In our context of transactional federations we are faced with the problem of having to guarantee serializability on top of an existing mechanism that we know to guarantee snapshot isolation. This problem is of practical interest already for a single component system like Oracle if the application demands full serializability. Oracle's recommendation to this end is to have the application programs explicitly lock, e.g., through the SQL "Select … For Update …" command, all relevant data in a transaction including read-only data. Another approach, inspired by earlier work on graph-based techniques for multidatabase transactions [BGS92], that we are going to exploit in our protocols is to maintain an online version of the SI-MVSG on top of the component system(s) and test it for cycles. Note that the earlier work was restricted to conventional conflict graphs aiming at conflict-serializability, whereas we consider a multiversion serialization graph. We introduce our novel concept here for the centralized (i.e., single-component) case, and will later leverage the result for the distributed (i.e., multi-component) case.

Definition: (OSI-MVSG)
The Online Snapshot Isolation Multiversion Serialization Graph (*OSI-MVSG)* for a given execution s satisfying (SI-V) is a directed graph with transactions as nodes and edges built by the following rules, as operations are submitted:

(i) When a transaction begins, it is added to the graph.

(ii) When a transaction t_i issues operation $r_i(x)$, then for all transactions t_j in the graph that issued an operation $w_j(x)$ before,
   a) an edge $t_i \rightarrow t_j$ is added to the graph if t_j and t_i are concurrent,
   b) an edge $t_j \rightarrow t_i$ is added to the graph if t_j was committed before t_i began,
   and each of the edges is labeled with x.

(iii) When a transaction t_i issues operation $w_i(x)$, then for all transactions t_j in the graph that issued an operation $r_j(x)$ before, an edge $t_j \rightarrow t_i$ labeled with x is added to the graph.
   Additionally, if there is a transaction t_j in the graph concurrent to t_i that issued $w_j(x)$ before, an edge $t_j \rightarrow t_i$ labeled with x is added to the graph.

(iv) A transaction is removed from the graph as soon as it is committed, all the transactions running concurrently with it are committed, and it is a source in the graph. ■

Note that although the OSI-MVSG is very similar to a standard conflict graph, it differs from a conflict graph in a subtle but important way by the edges created according to rule (ii) a) which capture the (SI-V) property of snapshot isolation. The practical relevance of the OSI-MVSG construction stems from the following theorem.

Theorem 1 (correctness of OSI-MVSG cycle testing):
A concurrency control algorithm based on OSI-MVSG allows only executions that are
(i) serializable if it rejects an operation when the added edges lead to a cycle in the graph,
(ii) snapshot isolated if it rejects an operation on object x when the added edges lead to a cycle that consists only of edges labeled with x. ■

The OSI-MVSG can be implemented on top of one or more component systems. The resulting global OSI-MVSG is a nicely versatile data structure as it can be used for 1) testing if an execution is serializable (i.e., no cycle exists) and 2) testing if an execution is snapshot isolated (i.e., no cycle exists where all predicates on the edges are conflicting with each other), whichever is considered the appropriate correctness criterion for the transactional federation.

As we cannot directly observe the underlying read and write accesses to the stored data, we need to build an approximation of the reads-from relationship or the conflict relations of the various component systems based on the operations that global transactions submit. In our implementation, we have adopted the approach of [SSW95] for analyzing potential conflicts between the predicates of SQL commands. We monitor the actions of global subtransactions and derive predicates that characterize the objects accessed by those subtransactions. Read operations are usually caused by SQL select statements, so we can use the search predicate of that statement to characterize all objects returned by that statement. For example, if a subtransaction submits the query "select p from parts where p.color=green or (p.price>100 and p.weight>7)", the predicates "color=green" and "price=100 and weight<7" characterize exactly those subsets of the complete relation "parts" that the subtransaction reads. When the query involves a join, the join predicates are decomposed into separate predicates on each of the joined tables thus disregarding the join predicate itself but keeping all filter predicates on each of the tables; these filter predicates are themselves converted into disjunctive normal form for efficient bookkeeping and conflict testing. Using the same technique, we can analogously derive predicates for SQL updates, inserts and deletes. Based on those predicates, we then say that a subtransaction t_j *potentially reads from* another committed subtransaction t_i, if the subsets that t_i updated and those that t_j read are overlapping, i.e., the conjunction of the predicates is satisfiable. This may actually add nonexisting reads-from or conflict relations, because we can only approximate the set of objects being accessed. However, this approximation is conservative in capturing all real reads-from relationships, so that we can guarantee the correctness of an algorithm that is based on the approximated reads-from or conflict relation.

3 Guaranteeing Global Serializability

Although SI is a popular option in practice, many mission-critical applications cannot afford the (whatever small) risk of data becoming inconsistent and therefore require full serializability. Then global SI as a correctness criterion, albeit "almost correct" with regard to the applications' needs, will not be good enough.

As an example consider the following federated version of the introduction's example for a non-serializable but SI schedule:

DB_1 $r_1^{(1)}(x)$ $r_2^{(1)}(x)$ $w_1^{(1)}(x)$ c_1 c_2
DB_2: $r_1^{(2)}(y)$ $r_2^{(2)}(y)$ $w_2^{(2)}(y)$ c_1 c_2

In this scenario, a global consistency constraint that relates the values of objects x and y could not be guaranteed. For example, a condition such as $x + y < 100$ could be violated although each individual transaction would preserve the constraint if it were executed alone. The problem could be rectified, of course, by requesting the application programmers to take additional measures such as manually acquiring stronger locks in the underlying component systems or extending the programs by an application-level notification or other synchronization mechanism. However, this would seriously impede application development productivity, and make applications more complex and thus more software-failure-prone. Federated systems should become no more difficult to use than today's centralized systems, neither at the application nor the system-administration level.

3.1 Extending the Ticket Technique

The ticket technique was introduced by Georgakopoulos et al [GRS94] as an elegant and flexible way to derive local serialization orders and, based on that, guarantee global serializability, for local SR schedulers. It requires that each global subtransaction reads the current value of a dedicated counter-type object, the so-called ticket, and writes back an increased value at some point during its execution. The ordering of the ticket values read by two global subtransaction reflects the local serialization order of the two subtransactions. Incompatible serialization orders are detected by the means of a *ticket graph* that is maintained at the federation level and whose edges reflect the local serialization orders. The global schedule is serializable if and only if the ticket graph does not contain a cycle.

This technique is easy and efficient to implement. For component systems with special properties (e.g., allowing only rigorous local schedules) further optimizations are possible, so that the ticket technique has the particularly nice property of incurring only as much overhead as necessary for each component system. 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. Second and much more severely, having to write the ticket turns read-only transactions into read-write transactions.

When the ticket technique is applied on top of local SI schedulers, it is evident that every global subtransaction in a component database writes at least one common object, namely the ticket. Because SI enforces disjoint writesets of concurrent transactions, all but one of several concurrent subtransactions will be aborted by the local scheduler. 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. Nevertheless, sequentializing all global subtransactions in SI component systems is a dramatic loss of performance and would usually be considered as an overly high price for global consistency.

The results we have presented up to now may lead to the impression that the ticket technique is unusable for SI component systems. In [SW+99] we developed an extension of the ticket technique that can overcome its disadvantages for many typical applications; we only give a short overview here. In fact, many real-life application environments are dominated by read-only transactions, but exhibit infrequent read-write transactions as well. 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 global transaction is read-only if all its subtransactions are read-only; otherwise it is a global read-write transaction.

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. Each read-write subtransaction of a global transaction takes a ticket as in the standard ticket, its read-only subtransactions are treated as those of a global read-only transaction. 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 SI component systems are part of the federation.

In our extension of the ticket versions, read-only subtransactions, on the other hand, need only read the ticket. This avoids making them read-write and being forced sequential. 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. The fact that this may result in multiple read-only subtransactions with the same ticket value is acceptable in our protocol.

3.2 Multilevel Transactions

Multilevel transactions have been pursued for transactional federations in the earlier work of [DSW94,SSW95] as a means for reducing the duration for which locks are held in the underlying component systems (or, more generally, the scope of the local concurrency control measures if a non-locking protocol is used). To this end, each high-level operation that is passed to a component system is handled as if it were a separate, purely local and fairly short, transaction. This ensures the atomicity and isolation of individual high-level operations. On top of this, the federation layer needs to keep additional, long-duration higher-level locks or take equivalent steps such as cycle testing on a graph in order to guarantee that the entire, multi-operation transactions appear atomic and semantically serializable to the clients. The high-level operations at the federation layer could be semantically rich methods on appropriately defined object types such as deposit or withdraw operations on bank account objects, or they could correspond to the SQL commands that typically constitute the interface of a component system. In the latter case, which has been studied in detail in [SSW95], the federation layer needs to extract appropriate predicates from the SQL commands as the basis for its own additional predicate locking. An inevitable consequence of committing each high-level operation as a separate transaction as early as possible is that undoing an entire client transaction, e.g., on behalf of a client-requested rollback, entails issueing compensating actions for the already committed transactions of the underlying component systems.

In the context of component systems that merely support SI, multilevel transactions do not necessarily guarantee semantic serializability. The reason is that the individual high-level operations from different client transactions are not fully isolated. So the pathological (i.e., consistency-violating) scenario shown at the beginning of Section 3 could arise between two individual high-level operations (on behalf of two different client transactions), and this effect would not be known to the federation layer. However, if such critical cases can be identified by the application architect, the federation layer could acquire appropriately strong semantic locks so as to prevent such concurrent scenarios in the underlying component systems. So whenever local SI is potentially insufficient, the federation layer would simply block the second high-level operations from being issued to the component system until the ongoing operation is completed. Obviously this approach requires a deep understanding of the application's semantics and consistency requirements. Thus we do not consider the multilevel-transaction protocol as the method of choice for transactional federations with local SI schedulers, but include it in the spectrum of supported methods for specific application cases. For the experiments reported in Section 5, we have designed our benchmark application and the form of high-level locks kept at the federation layer such that semantic serializability for entire client transactions is guaranteed even with local SI schedulers.

3.3 Global Cycle Testing Based on the Online SI-MVSG

The online SI-MVSG, or OSI-MVSG for short, introduced in Section 2 is a means for ensuring serializability on top of an SI scheduler. As explained in Section 2, we simply need to test the OSI-MVSG for cycles and reject operations that would lead to a cycle by aborting the corresponding transaction. In a transactional federation, the additional problems arise that the OSI-MVSG needs to be built across all underlying component systems and needs to include also component systems that do provide local serializability, not just the ones that provide local SI. It turns out, however, that both problems are relatively easy to solve:

- The global OSI-MVSG to be maintained at the federation layer is simply the union of the OSI-MVSGs that we would maintain on top of each component system separately. We need to ensure, however, that nodes are not removed too early from this union graph: for the generalization to federations, the rule is that a transaction can be removed from the graph only if all transactions that have run concurrently with it in *anyone* of the underlying databases are committed, and the transaction to be deleted is a source in the global graph.
- As for the incorporation of component systems that guarantee full local conflict-serializability, the construction of edges in the OSI-MVSG requires an additional mechanism for observing conflict orders rather than the version-based reads-from relationship. In our implementation, we require such component systems to use tickets (possibly the implicit tickets given by the commit order if the component system has the rigorousness property). The OSI-MVSG adds edges for subtransactions on these component systems such that the edges correspond to the observed ticket orders between subtransactions.

Putting these considerations together yields the following result: a global concurrency control algorithm based on cycle testing on a global OSI-MVSG for transactional federations with component systems that guarantee either local snapshot isolation or local conflict-serializability allows only executions that are globally serializable if it rejects an operation when the added edges lead to a cycle in the graph. As discussed already in Section 2, the construction of the graph's edges requires observing reads and writes at the component system's interfaces. In our setting, this entails extracting simple predicates from the SQL operations that are issued to the various component systems. Section 2 already discussed how this can be accomplished.

3.4 Comparison and Combination of Protocols

None of the three presented families of protocols dominates the others; rather each of them has specific advantages in certain situations but also drawbacks with regard to certain aspects. In principle, the most powerful approach in terms of possible concurrency is the *multilevel transaction protocol*, as the locks in the underlying component systems are held only for the duration of an operation and the federation layer can exploit the semantics of high-level operations to a large extent. However, its drawback is that it requires significant additional care for setting up the appropriate style of operations and corresponding locks, the compensating actions to be logged, etc. Even if the approach is restricted to standard SQL operations, for which all these aspects can be set up generically (as opposed to application-specific methods such as withdrawals and deposits), one still needs to specify the high-level conflicts between these operations such that local SI can be tolerated for the isolation of operations, which may be application-specific. In addition, the management of high-level locks at the federation layer and the logging of compensating actions may incur significant overhead. The *family of ticket methods*, on the other hand, is clearly the best in terms of low overhead at the federation layer. In particular, its overhead "scales" with the correctness degrees of the underlying component systems: for systems with locally rigorous schedules, no explicit tickets are needed, for systems with the formal property of avoiding cascading aborts no edges need to be kept in the global ticket-order graph. However, tickets may well incur substantial performance degradation. Finally, the *cycle testing protocol* on the global online SI-MVSG is positioned in between the other two approaches with regard to both potential concurrency, which we would expect to be higher than with tickets, and overhead as well as application-specific setup complexity, which should be lower than with multilevel transactions.

The three protocol families are complementary not only in that differently configured federations may choose different protocols, but also in that different protocols may be combined within the same federation, either on a per transaction or per component system basis. So a single transaction that accesses multiple component systems may use different protocols on different component systems depending on the properties of these systems.

4 Prototype Implementation

The practical viability of our theoretical results is demonstrated by a federated transaction manager, coined TRAFIC, that we have built as part of the VHDBS system [HWW98], a comprehensive prototype system for federated databases that has been developed by the Fraunhofer Institute for Software and Systems Engineering and the Research Lab of the German Telekom AG. The VHDBS architecture is based on the wrapper-mediator paradigm [Wied92]. System-specific adapters wrap the existing component systems and translate data definitions and queries between the federated level and the native languages of the underlying database systems. Currently, there are adapters to integrate Oracle 8i and O_2 Release 5. The VHDBS federation server acts as a mediator on the federated level, integrating metadata and decomposing federated queries and transactions into the corresponding subqueries and subtransactions. The common data model at the federation level is a subset of the ODMG object model, the query language is essentially ODMG's OQL, extended by operations to modify object attributes. The databases in a federation remain locally accessible, in addition to the integrated access through the federation layer.

Transactions are supported in VHDBS by the federated transaction manager *Trafic* [SW99]. Whenever a client embeds a data-access operation in the context of a transaction, VHDBS passes them through Trafic. There, appropriate steps are taken to guarantee global atomicity and global serializability. Trafic consists of several modules, which are specified in CORBA's IDL interface definition language. All modules can be integrated into a single CORBA server, or each module can be instantiated as a separate CORBA server with all servers distributed over a number of hosts.

Trafic generally provides a clear separation between transaction management *mechanisms* and *strategies*. Mechanisms such as lock management, graph cycle testing, or logging provide generic functionality; strategies make use of those mechanisms to guarantee a globally correct execution. Trafic supports a suite of different strategies; currently, there are implementations of all the strategies presented in Section 3.

To guarantee the atomicity of distributed transactions, Trafic makes use of Iona's OrbixOTS, an implementation of OMG's Object Transaction Service OTS. OTS essentially provides a two-phase commit protocol with CORBA-style interfaces and allows integrating existing resource managers via the standardized XA interface. Trafic uses OTS in the following way. When a client requests to start a new transaction using Trafic's transaction factory, Trafic begins a corresponding OTS transaction. All subsequent data-access operations are then passed through Trafic. Whenever a database is accessed for the first time in the context of the transaction, the local OTS agent automatically registers it with the transaction. Finally, upon the transaction's commit, OTS coordinates the subtransactions in the involved component systems, applying the standard two-phase commit protocol. Thus, our prototype architecture requires that all component systems provide an XA interface; this is the case for the currently supported systems Oracle and O_2.

All the logging that is necessary for the two-phase commit protocol is done by the underlying component system and OTS (for the coordinator log). Additionally, Trafic provides its own LogManager to support compensating actions for the multilevel transaction management strategy presented in Section 3.2. With this strategy, a client transaction is broken down into a sequence of independently committed OTS transactions. The standard case for all other strategies presented in this paper, however, is that each client transaction constitutes a single OTS transaction, and this guarantees global atomicity by OTS and the underlying component systems alone.

5 Experimental Evaluation

5.1 Benchmark Setup

The experimental evaluation of the presented suite of protocols has been carried out on the full-fledged implementation within the VHDBS federated database system on top of Oracle 8i databases. To this end, we designed an oversimplified stock brokerage scenario with three databases and very few data so that data contention would be the main performance bottleneck in multi-user access. So the experiments were designed as extreme stress tests rather than trying to capture all aspects of a real application.

We consider two stock brokers and a bank. Each stock broker manages 100 customers, and each one of them operates an Oracle 8i database with the following schema (identical for both of them):

- A first relation `Stocks(StockID, Price)` contains the available stocks, identified by the primary key `StockID`, and the current price for each stock. This relation contains 100 tuples.
- The second relation `StockList(CustomerID,StockID,Amount)` holds information about the stocks owned by the customers. Each customer owns ten randomly selected, different stocks.

In addition to these two stock brokers, the bank operates another Oracle 8i database with the following schema:

- A relation `Portfolio(CustomerID,StockID,Amount)` holds information about the stocks owned by the customers. It is essentially the union of the stock brokers' `StockList` relations.

On top of these three databases, we ran two classes of transactions: The *Investment* transaction models one customer buying several shares of one stock, by reading the current stock price in the one of the two stock broker's database and updating the customer's entry in the broker's and the bank's databases. The *Value* transaction computes the total value of all shares that a customer owns by querying both stock brokers' databases for the stocks that the customer owns and their current prices. These two transaction types are sketched in Figure 1.

```
Investment(cid,sid,amount):
determine stock broker database SB from sid;
select price from SB.Stocks where StockID=sid;  number:=amount/price;
if (customer already owns shares of this stock)
{ update SB.StockList set Amount=Amount+number; }
else { insert into SB.StockList values(cid,sid,number); }
if (customer already owns items of this stock)
{ update Bank.Portfolios set Amount=Amount+number; }
else { insert into Bank.Portfolios valules(cid,sid,number); }

Value(cid):
totalvalue:=0;
for StockID,Amount in (select StockID,Amount from SB1.StockList where CustomerID=cid)
{ select Price from SB1.Stocks where stockid=StockID;  totalvalue+=Amount*Price; }
for StockID,Amount in (select StockID,Amount from SB2.StockList where CustomerID=cid)
{ select Price from SB2.Stocks where stockid=StockID;  totalvalue+=Amount*Price; }
```

Figure 1 – Pseudocode of the Two Benchmark Transactions

All the benchmarks were run with Trafic's modules implemented as separate CORBA servers, distributed over five Sun Sparcstations in such a way that the CPU load was balanced across the machines, to the best possible extent. As for the underlying database servers, the Oracle instances were run on a Sun Ultra Enterprise 4000 with eight processors and two PCs running Windows NT. As it turned out during the measurements, all these machines were far from being performance bottlenecks.

5.2 Performance Measurements

This section presents results from the performance measurements for the following four experiments:

- a multi-user combination of Value and Investment transactions with input parameters chosen according to a 90-10 skewed distribution, with the read-write Investment transactions as the dominating load,
- the same multi-user combination of Value and Investment transactions, but with the read-only Value transactions as the dominating load,
- a multi-user load of a conflict-free variation of Investment transactions such that no two concurrent transactions accessed any common stocks or customers, as a means for assessing the pure bookkeeping overhead of the various techniques,
- a multi-user load of read-only Value transactions, to assess the bookkeeping overhead for the important special case of read-only transactions.

All experiments measured transaction throughput for a particular number of clients each of which spawns a new transaction upon the completion of the last transaction issued by the same client. So we used a fixed multiprogramming level (MPL) in a single run, and then varied this MPL to produce a complete performance chart. For the read-write dominated mixed workload experiments, the MPL of the Value transactions was constantly five, and only the MPL of the Investment transactions was varied. For the read-only dominated mixed workload experiments, the MPL of the Investment transactions was constantly five. Note that in all experiments the absolute throughput figures are fairly low for several reasons: VHDBS is merely a prototype system that incurs substantial overhead (e.g., copying fine-grained Orbix objects across machines), the experiments were run on rather low-end hardware, and our experimental setup really was an extreme stress test in terms of data contention.

In the following, we use *ETT* for the extended ticket technique presented in Section 3.1, *OTT* for the traditional (optimistic) ticket technique, *MLTM* for the multilevel transaction strategy presented in Section 3.2, and *MVSG* for the Online SI-MVSG strategy presented in Section 3.3. To assess the overhead of the various strategies, we also measured a strawman strategy *NONE* that completely ignores global concurrency control (and thus cannot guarantee any global correctness criterion at all), but simply drives the two-phase commit protocol at commit time.

Figure 2 shows the throughput results for the read-write dominated (left chart) and the read-only dominated (right chart) settings. Throughput of Value transactions turned out to be roughly the same for all strategies, so it was omitted from the charts. As for the Investment transactions in the read-write dominated setting, we see that the OSI-MVSG technique outperforms all other techniques until a certain MPL, after that, MLTM allows the highest throughput of Investment transactions. The ticket method performs poorly for read-write transactions because of the additional contention for ticket objects. In the read-only dominated setting, the OSI-MVSG strategy again shows the best throughput for Investment transactions. MLTM's throughput rapidly drops with increasing MPL of Value transactions, because an increasing amount of lock conflicts with the long-running Value transactions causes long blockings of Invest transactions.

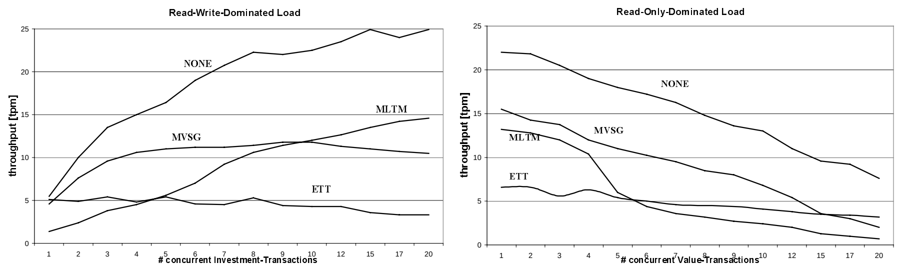

Figure 2 – Benchmark results for read-write dominated load

To study the overhead of the strategies for read-only transactions, we also measured the throughput of Value transactions alone for varying MPL. Here, the extended ticket technique developed in this paper performs equally well as the MVSG technique, whereas the original ticket technique again suffers from the bottleneck of the ticket objects. MLTM shows a relatively low throughput because of the additional cost of committing subtransactions (note that a single transaction comprised 60 subtransactions on average). Similar measurements with the conflict-free variant of Investment Transactions convey the overhead of the various strategies for read-write transactions. Again, MVSG shows the clearly best performance. The additional logging of MLTM accounts for its lower performance.

We conclude this section with a summary of our experimental findings:

- The Online SI-MVSG has proven to be the method of choice for mixed environments with both read-write and read-only transactions. It even outperforms the strategies for global SI, even though providing globally serializable executions.
- The extended ticket technique has the best performance for read-only transactions, but has major performance disadvantages with read-write transactions.
- Multilevel transactions exhibit decent overall performance, without any extraordinary results, however, which is partly caused by the fact that our benchmark did not include any operations with special semantics such as withdrawals or deposits that could have been exploited for higher concurrency.

6 Conclusion

In this paper we have re-opened the subject of transactional federations with particular focus on the previously neglected issue of component systems that support "only" snapshot isolation (SI) rather than full serializability (SR). We have presented a suite of techniques for ensuring global serializability on top of heterogeneous federations with a mix of both SI- and SR-oriented components. Our techniques leverage prior work, most notably, the ticket method and the technique of deriving reads-from and conflict relations from observing SQL-statement predicates at the federation layer. In addition, we have presented a generalization of the ticket technique and a novel technique based on cycle testing in a federation-level online multiversion-serialization graph. All techniques have been implemented in a full-fledged federated database system, and we have presented stress-test performance measurements on this real-system platform to demonstrate the viability and performance advantages of the novel techniques.

Federated database systems were considered impractical for much of the last decade, but the proliferation of gateway and other middleware technology and especially the pressing application demand for integrated access to multiple data sources have recently led to a revival of the federation concept [Ston98,MK+99,HSC99]. This paper has addressed a significant part of the transactional aspects of federations, and we believe that the incorporation of snapshot isolation is of particular practical interest given its wide use for operational (Oracle) databases. Our future work will mostly aim to make our framework, protocols, and also prototype implementation more comprehensive in that we also want to support other forms of relaxed isolation levels such as the ANSI SQL "read committed" level and especially combinations of different isolation levels on a per component-systems or transaction-class basis.

References

[ALO00] A. Adya, B. Liskov, P. O'Neil: *Generalized Isolation Level Definitions.* ICDE, San Diego, 2000.

[ABJ97] V. Atluri, E. Bertino, S. Jajodia: *A Theoretical Formulation for Degrees of Isolation in Databases*, Information and Software Technology 39(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*. SIGMOD, San Jose, 1995.

[BLL00] A.J. Bernstein, P.M. Lewis, S. Lu: *Semantic Conditions for Correctness at Different Isolation Levels.* ICDE, San Diego, 2000.

[BGRS91] Y. Breitbart, D. Georgakopoulos, M. Rusinkiewciz, A. Silberschatz: *On Rigorous Transaction Scheduling*. IEEE Transactions on Software Engineering 17(9), 1991.

[BGS92] Y. Breitbart, H. Garcia-Molina, A. Silberschatz: *Overview of Multidatabase Transaction Management*. VLDB Journal 1(2), 1992.

[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,* RIDE, Tempe, 1992.

[DSW94] A. Deacon, H.-J. Schek, G. Weikum: *Semantics-based Multilevel Transaction Management in Federated Systems*. ICDE, Houston, 1994.

[FLO+99] A. Fekete, D. Liarokapis, E. O'Neil, P. O'Neil, D. Shasha: *Making Snapshot Isolation Data Item Serializable*, Manuscript, 1999.

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

[HSC99] J.M. Hellerstein, M. Stonebraker, R. Caccia: *Independent, Open Enterprise Data Integration*, IEEE Data Engineering Bulletin 22(1), 1999.

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

[MK+99] N. M. Mattos, J. Kleewein, M. T. Roth, K. Zeidenstein*: From Object-Relational to Federated Databases*. Invited Paper, in: A. P. Buchmann (Ed.): *German Database Conference (BTW)*, 1999.

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

[Orac99] Oracle Corporation: *Oracle8i Concepts: Chapter 27, Data Concurrency and Consistency*, 1999.

[Raz92] Y. Raz: *The Principle of Commit Ordering or Guaranteeing Serializability in a Heterogeneous Environment of Multiple Autonomous Resource Managers Using Atomic Commitment*. VLDB, 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*. RIDE, Taipeh, 1995.

[Ston98] M. Stonebraker: *Are We Working On The Right Problems? (Panel)*. SIGMOD, Seattle, 1998.

[SW99] R. Schenkel, G. Weikum*: Experiences With Building a Federated Transaction Manager Based on CORBA OTS*, in: Proceedings of the 2nd International Workshop on Engineering Federated Information Systems, Kühlungsborn, 1999.

[SW+99] R. Schenkel, G. Weikum, N. Weißenberg, X. Wu*: Federated Transaction Management With Snapshot Isolation*, in: Proceedings of the 8th International Workshop on Foundations of Models and Language for Data and Objects – Transactions and Database Dynamics, Schloß Dagstuhl, Germany, 1999.

[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.