Taming the Tiger: How to Cope with Real Database Products in Transactional Federations for Internet Applications

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

Data consistency in transactional federations is a key requirement of advanced E-service applications on the Internet, such as electronic auctions or real-estate purchase. Federated concurrency control needs to be aware of the fact that virtually all commercial database products support sub-serializability isolation levels, such as Snapshot Isolation, and that applications make indeed use of such local options.
This paper discusses the problems that arise with regard to global serializability in such a setting, and proposes solutions. Protocols are developed that can guarantee global serializability over component systems that provide only weaker isolation levels. A full-fledged implementation is presented that makes use of OrbixOTS and runs on top of Oracle8i and O_2 databases. Performance measurements with this prototype indicate the practical viability of the developed methods.

1 Motivation

1.1 Why Transactional Federations Are Important

Ubiquitous consistent data access across organizational boundaries is of paramount importance for internet-base E-Commerce, advanced business-to-business E-Services, and virtual enterprises as well as mergers of companies. These modern applications require access to multiple, independently developed and largely autonomously operated databases [Conr97, Ston98, CSS99, HSC99, MK+99, DE00, SW00b], which is known as a database federation or multidatabase system. 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.

Existing products for the federation of independent database systems, such as IBM's DataJoiner, Oracle's Heterogeneous Services [Orac99a,Orac99b], or implementations of the CORBA Object Transaction Service [OMG97], offer only a limited support for federated transactions. They are restricted to driving the two-phase commit protocol between the databases, and essentially ignore global serializability. Usually, those systems rely solely on the concurrency control components of the local database systems. For example, IBM's DataJoiner has a locking component, but only for its own local database [IBM98]. Oracle even warns that in federations with Microsoft SQL Server, consistency is no longer guaranteed [Orac96], while they can guarantee consistency in Oracle-only federations [Orac99a].

Problems of ensuring the global serializability and atomicity of federated transactions have been intensively studied in the past. For global atomicity, standardized a two-phase commit (i.e., X/Open XA or CORBA OTS) has proven to be a viable solution. For global concurrency control, numerous protocols have been proposed (see, e.g., [Weihl89, BGRS91, BS92, Raz92, GRS94]). However, these solutions completely ignored the frequent use of different isolation levels [BBGM+95] in the underlying component systems, like the "read committed" mode supported by virtually all products or the "snapshot isolation" mode provided by the market leader Oracle [Orac99c]. Such options for relaxed (local) serializability are widely popular in practice, because they typically allow a much better performance at the price of a "slightly reduced" level of data consistency. 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 ignored in federated transaction management. This paper shows what problems can arise, and presents techniques to solve these problems.

1.2 Problems Arising With Local Snapshot Isolation

Oracle advertises its multiversion-based Snapshot Isolation mode as "serializable" [Orac97], but they use the ANSI SQL definition of serializable executions, that is, the avoidance of dirty reads, unrepeatable reads, and phantoms [ANSI92]. It has been shown in [BBGM+95] that this is a weaker criterion than multiversion serializability (MVSR) in the usual definition (as, e.g. in [BHG87], allowing executions that are not in MVSR, and may indeed lead to inconsistent data.

A transaction run under Snapshot Isolation (SI) sees the most recent committed version of the objects in the database as of the time when the transaction began, thus ensuring a consistent view of the database. Read-only transactions are thus perfectly isolated. For read-write transactions, Oracle's transaction manager enforces that no two concurrently running transactions write a common object; if it happens, one of them is forced to abort. However, while this catches most possible cases where inconsistencies can appear, it still allows executions that are not serializable and that may violate constraints. To illustrate this problem, we use the following example of the interleaved execution of two banking transactions that withdraw money from two accounts x and y, with regard to the constraint that the sum of the balances of x and y must be greater than 1000€ (time advances from top to bottom):

Transaction 1:	**Transaction 2:**
Determine current balance of accounts x and y	
	Determine current balance of accounts x and y
	Check constraint: bal(x)+bal(y) > 2000 €
Check Constraint: bal(x)+bal(y) > 1500 €	
	If true, withdraw 1000 € from account x
If true, withdraw 500 € from account y	

This execution can occur if the database guarantees Snapshot Isolation, because both transactions write different objects. However, it may violate the balance-sum constraint, although each transaction alone would enforce the constraint. For example, if the balance of each of x and y was 1000 € before, the above execution would produce a balance of 0 € for x and 500 € for y.

To prevent such situations, Oracle recommends programmers to explicitly acquire an exclusive lock (with a SELECT FOR UPDATE SQL call) for data that are critical with regard to such anomalies. With this precaution, this would lock out the read operations of transaction t_2 until transaction t_1 is completely finished, thereby guaranteeing a correct (even serial) execution, at the price of lower concurrency. However, determining which objects to lock is typically a hard problem in real-life settings with complex application programs and many tables. If some data is left out by accident, inconsistencies can still happen. So application programmers and users have learnt to live with this kind of problem, but are not at all happy with it.

1.3 Snapshot Isolation in Federations

For federated applications, Oracle's recommendation to explicitly lock data is even harder to apply. The federated DBMS would have to acquire these locks in a (semi-)automatic way, as federated applications usually don't know what kind of database systems they really access, and federated data may be spread across several, possibly different, databases. The last resort for a provably correct execution would be to conservatively lock everything before access, which would result in a dramatic loss of concurrency, and therefore is not really option.

If some or all component systems use Snapshot Isolation, potential problems are not even limited to corrupting local data or violating local constraints. In this section, we present two typical problems that violate global consistency. We illustrate this using the example of a world wide online bookstore with branches in Germany, France, and Spain. All these branches have their own local databases which are integrated into a federated database. In the following, assume that the German and the French database use Snapshot Isolation, and the Spanish database uses strong two-phase locking (SS2PL), i.e. 2PL with both read and write locks held until transaction commit.

The first case where problems can arise is that a federated transaction spans databases generating serializable schedules and databases guaranteeing only snapshot isolation. As an example, assume that transaction t_1 checks the total number of copies of a certain book the global store has in stock, maybe to check if new copies have to be ordered. Another transaction t2 updates the database after some copies of the same book have been moved from the German store to the Spanish store, decrementing the amount in one database and incrementing it in the other database. The resulting execution may look like this (t_1's operations are printed in italics, and time advances from top to bottom):

German Store (using SI)	**Spanish Store (using SS2PL)**
	t_2: Increment number of copies of the book
t_1: Determine number of copies of the book	
t_2: Decrement number of copies of the book	
	t_2: Commit t_2
	t_1: Determine number of copies of the book
	t_1: Commit t_1

Transaction t_1 doesn't see t_2's update of the database in the German store, but it sees the update in the Spanish store, because t_1 has to wait until t_2 releases the update lock at its commit time. So t_1 sees an inconsistent state of the database and, based on the information read, may further damage the database consistency by writing back "wrong" values. If the German store used SS2PL for concurrency control, too, this could not happen; then the transactions would run into a global deadlock, and one of them would be aborted by the federated database system.

Even if all local databases use Snapshot Isolation, problems can occur. In our example, assume that there is a global constraint enforcing that there are at least a certain number of copies of a certain book in stock. If the total amount drops below that limit, new copies are ordered. The application programmer decided to put this test into the procedure for processing the sale of a book, which could include the following code:

```
procedure sell(bookid, amount_sold)
begin
  count1:= select amount from StoreGermany.stock where id=bookid;
  count2:= select amount from StoreFrance.stock where id=bookid;
  //select one of the stores and decrement its amount by amount_sold
  if (count1+count2≥limit) and (count1+count2-amount_sold<limit)
  then trigger order-new-copies
end
```

Now assume that two such transactions run concurrently for the same book, where the first decides to sell the book from the German store, and the other uses the French store. Then both executions in the local databases are correct in the sense of Snapshot Isolation, as no two concurrent transactions write a

common object. However, it is possible that the total stock of this book drops below its limit, but none of the transactions detects this, because it does not see the changes made by the concurrent transaction. So it is possible that an execution violates a constraint that spans more than one database, even though the transactions do their best to detect the violations. In the specific example, once the event of the quantity-in-stock dropping below the threshold has been missed, all subsequent transactions would fail to initiate the reordering, too.

It is of course possible for a skillful application designer to detect these problems and find solutions to circumvent them. However, typical real-life applications are too complex to find all such problematic cases, so a general system-provided solution is highly desirable. Especially for mission-critical applications, one is very interested in protocols that can be proven to be correct for every possible execution. We present such protocols in this paper.

1.4 Contribution and Outline of the Paper

The paper's contributions are threefold:

- We develop new algorithms to cope with local Snapshot Isolation in federated database systems. With Oracle being the market leader, ensuring global serializability over local Snapshot Isolation is obviously of extreme practical relevance.
- We present the architecture and implementation of a prototype of a flexible transaction manager that can handle a variety of component systems with different local concurrency control protocols.
- Based on this prototype, we measure the performance of our techniques to show their viability in real-life settings.

The rest of the paper is organized as follows. In Section 2, we present algorithms for global concurrency control that can cope with the fact that some of the local systems provide only Snapshot Isolation. In Section 3, we present the architecture of a flexible transaction manager and give some details of its implementation. In Section 4, we present the results of benchmark measurements that show the viability of our algorithms.

2 Algorithms for Global Serializability

In this section we discuss how global serializability can be achieved when some or all component databases guarantee only Snapshot Isolation. First, we recall the previously published ticket technique for global concurrency control and show that it is not appropriate for this setting. Then we extend the ticket technique such that it efficiently supports read-only subtransactions. Next, we present a novel graph approach based on observing the reads-from relation. We conclude this section with a comparison of the protocols and show how the protocols can be combined into a unified framework.

2.1 The Ticket Technique and Snapshot Isolation

The common definition of global serializability requires that for a "correct" global schedule, all the subschedules in the local databases be serializable and the serialization order of the transactions in those subschedules be compatible. This means that when a transaction is serialized before another transaction in some local database, they must be serialized in the same way in all other databases where they both issued operations. While this is an elegant definition, it is not at all easy to extract information about serialization orders from commercial database systems.

The ticket technique was introduced by Georgakopoulos et al [GRS94] as an elegant and flexible way to derive local serialization orders and, based on this information, guarantee global serializability over local schedulers generating serializable schedules. 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 subtransactions 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 further optimizations are possible. 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. 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.

If 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. However, sequentially executing 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.

2.2 Extending the Ticket Technique for Read-Only Transactions

The results we have presented up to now may lead to the impression that the ticket technique is unusable for SI component systems. We now present an extension of the ticket technique than can overcome its disadvantages for many typical applications. In fact, many real-life application environments are dominated by read-only transactions, but exhibit infrequent read-write transactions as well. In our extension, the performance of read-only transactions is not degraded, so for those class of transactions, 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 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 technique, read-only subtransactions, on the other hand, need only read the ticket. This avoids making them read-write and being forced sequential. Why is this sufficient for the tickets to reflect the correct local serialization order? A subtransaction’s ticket value shows its position in the locally equivalent serial schedule. When a component system DB_k generates SI schedules, the values a read-only transaction t_i reads there depends only on the position of its first local operation $B_i^{(k)}$. The transaction reads from read-write 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 read-write 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. The fact that this may result in multiple read-only subtransactions with the same ticket value is acceptable in our protocol.

2.3 Using a Graph-Based Technique

In this section, we present a graph-based technique to guarantee global serializability if all local component systems guarantee snapshot isolation. We show how to include component databases with SR schedulers in the following Section.

In [SW+99], we developed a variant of the standard multiversion serializability graph [BHG87], coined SI-MVSG, that allows us to decide if a given multiversion schedule is serializable, snapshot isolated or none of them, provided that its version function satisfies the SI property. We showed there that such a schedule is serializable if and only if the corresponding SI-MVSG is acyclic. Inspired by earlier work on graph-based techniques for multidatabase transactions [BGS92], the protocol we present now maintains an online version of this graph and tests 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.

Our protocol maintains an Online Snapshot Isolation Multiversion Serialization Graph *(OSI-MVSG)* for a schedule s. This is a directed graph that has the 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 a read operation, then for all transactions t_j in the graph whose write set overlaps the set of objects read by the operation,
   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,

(iii) When a transaction t_i writes an object, then for all transactions t_j in the graph that read that object before, an edge $t_j \rightarrow t_i$ is added to the graph.
   Additionally, if there is a transaction t_j in the graph concurrent to t_i that wrote the same object before, an edge $t_j \rightarrow t_i$ 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 snapshot semantics of read operations.

The global execution is serializable iff this graph has no cycles. As soon as a cycle appears, one of the participating transactions must be rolled back.

The implementation of this algorithm requires that the read and write sets of global transactions are known at the federated layer. As there is usually no easy way to extract this information from the local database systems, we derive an approximation of the read and write sets from the predicates of SQL statements the transactions submit. We present details of this technique in Section 3.3.3.

2.4 Comparison and Combination of the Techniques

None of the presented families of protocols dominates the other; rather each of them has specific advantages in certain situations but also drawbacks with regard to certain aspects.

The *family of ticket methods* 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. However, tickets may well incur substantial performance degradation. The *cycle testing protocol* on the global online SI-MVSG, on the other hand, can achieve a higher degree of concurrency, but incurs substantially more overhead. Especially deriving read and write sets from SQL predicates requires significantly more work at the federation layer than managing the ticket values.

To gain the best results, the protocols can be combined so that the "best" protocol for a given application and database is used. For example, in a federation with both local Snapshot Isolation and local Serializability, the OSI-MVSG strategy can be used for the SI subsystems, and one of the ticket techniques for the other systems. Both techniques generate a graph, and the global execution is serializable iff the union of the graphs is acyclic. If it is known in advance that most transactions will only read in a subsystem running Snapshot Isolation, the extended ticket technique can be used there. This combined algorithm requires minimal overhead while still guaranteeing a globally correct execution, so it is the method of choice for general-purpose settings.

3 Prototype Implementation

We have built a prototype system to demonstrate the practical viability of our theoretical results. This federated transaction manager, coined TraFIC (*Transactions in Federated Information Systems using CORBA*), has been built as part of the VHDBS System, a comprehensive prototype system for federated database systems. We give a brief introduction into the architecture of VHDBS in Section 3.1. Next, we present the TraFIC prototype architecture in Section 3.2. We conclude the section with some details of the implementation in Section 3.3.

3.1 The VHDBS Prototype

VHDBS [HWW98] is a federated system that can incorporate pre-existing, largely autonomous databases managed by different database systems. Its architecture is based on the wrapper-mediator paradigm [Wied92]. Existing component systems are wrapped by means of system-specific DBMS adapters that translate data definitions and queries between the federated level and the native languages of the underlying database systems. At the federation level a mediator, called the federation server, and decomposes federated queries and transactions into the corresponding subqueries and subtransactions. The common language at the federation level is a subset of the ODMG data model with its query language OQL, extended by SQL-like means to update object attributes. Currently supported component systems are Oracle 8i and O_2 Release 5. The databases of a federation remain locally accessible, in addition to the integrated access through the federation level. The architecture of VHDBS is depicted in Figure 1 and consists of the following major modules:

- The **federation server** is the central component. It comprises all services provided by the federation layer, most importantly the schema subsystem, the query subsystem and the transaction manager TraFIC. The federation server decomposes a query into subqueries for the affected component databases, and executes them in parallel through the corresponding DBMS adapters. The results of the subqueries are imported into a designated **federation database**, where they are combined into the final query answer. The transaction manager guarantees the atomicity and serializability of global transactions.

- The **meta server** manages metadata, which is used to provide the information about definitions of schemas/types, databases, and users.

- The component databases are plugged into the system via **DBMS adapters** that implement a uniform interface, providing abstract operations to access the data within the component database. Each DBMS adapter for a specific database system (or system class) maps the federated data model to the data model of the component system.

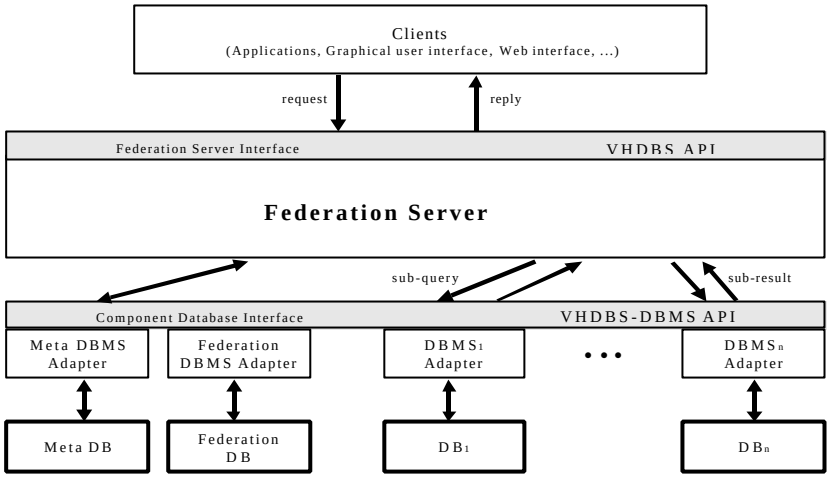

Figure 1 - Architecture of the VHDBS System

All modules of the architecture and all data objects are specified using the IDL interface definition language of CORBA. This permits a fine-grained combination of these components into CORBA servers, specifically tailored for the needs of the applications. All communication between such server processes is based on CORBA, so that servers may be distributed across local or wide-area networks. The same holds for the communication with clients of the federation, which allows easy access from both application programs (e.g. using IDL to C++ mapping) and Web clients (using IDL to Java mapping).

3.2 The Transaction Manager TraFIC

VHDBS was originally built as a query-only system without the need for federated transactions. This was revised when update operations were integrated into VHDBS, which require additional protocols to guarantee global serializability and atomicity for update transactions. In the following section we briefly show why the transaction support built into CORBA is not sufficient for our case. Next, we present the architecture of the VHDBS transaction manager TraFIC.

3.2.1 Why OTS is Not the Solution

Following the CORBA approach, a straightforward solution to integrate support for transactions into VHDBS would be to apply the Object Transaction Service OTS [OMG97]. This can be done smoothly, because the components of OTS themselves are specified as IDL interfaces. To integrate transactional support into VHDBS, the following modifications to the system are needed:

- Mark all interfaces that may be used in the context of a transaction as transactional, by inheriting from the pseudo-interface `TransactionalObject`.
- Modify the database wrappers so that they use the database's XA interface for transaction control, not explicit SQL statements.

- Add transaction control statements for the begin and commit of transactions to client programs. OTS then implicitly adds the current transaction context to every call of a transactional object, so there is no need to modify the actual calls of VHDBS methods.

While this is an easy way to integrate transactions, it is only half of the solution. The Object Transaction Service drives the standardized two-phase commit protocol between the participating databases at commit time, thereby guaranteeing atomicity for global transactions. However, all it provides for concurrency control are means to lock CORBA objects in shared and exclusive mode, which doesn't help at all for federated transactions accessing data in local databases. OTS does neither offer protocols to guarantee global serializability, nor does it allow an easy integration of such protocols by hand. As we showed in Section 1, such protocols are absolutely necessary for a correct execution; so OTS alone does not solve this problem.

3.2.2 Transaction Manager Architecture

Instead of using the transaction control mechanisms of OTS, we decided to include a dedicated federated transaction manager, coined TraFIC, in the VHDBS federation server. This manager offers a suite of strategies for concurrency control and atomicity, among which the client program can choose the one which best fits its needs.

```
module Trafic
{ interface Transaction
{ void CommitTransaction();
void AbortTransaction();
// transactional methods of VHDBS database adapters
dbCollection select(in string query
in dbAdapter db);
// ...
};
interface TransFactory
{ Transaction BeginTransaction();
// ...
}
```

Figure 2 - IDL specification of the transaction and the factory objects (excerpt)

Transactions are represented by means of a transaction object in TraFIC (see Figure 2). When a client wants to begin a new transaction, it requests TraFIC's transaction factory to create a new transaction object. This object provides the same set of methods as the database adapters in VHDBS, extended by a reference to the database adapter which should execute the operation. The interface of the federation server is extended by a second version of its original methods that take the current transaction object as an additional parameter. When an operation is embedded in the context of a transaction, the query subsystem passes the operation or the resulting suboperations through the transaction manager by calling the appropriate method of the transaction object, instead of submitting them immediately to the database adapters. TraFIC then performs the necessary steps for federated concurrency control and atomicity, depending on the currently selected strategy.

The transaction manager consists of several modules all of which provide IDL interfaces. These modules can be combined to CORBA servers as the application demands. If one of the modules turns out to use too much CPU time, for example, it can easily be separated out into a distinct CORBA server running on another machine. The internal architecture of TraFIC and its interactions with the other major components of VHDBS are depicted in Figure 3. TraFIC generally provides a clear separation between transaction management *mechanisms* and *strategies*. Mechanisms such as predicate management or graph cycle testing provide generic bookkeeping functionality that can be (re-)used for a variety of strategies. Strategies such as the family of ticket techniques and the graph-based technique pre-

sented in the previous section are responsible for the correct execution of transaction, i.e. to guarantee a globally serializable and atomic execution. TraFIC supports a suite of different strategies, each of them being an instance of the generic IDL interface `CCStrategy`. Each strategy provides its own implementation of the transaction interface. We have implemented the strategies covered in the Section 2 and several other strategies that are beyond the scope of this paper. New strategies can be added easily by providing implementation classes. The transaction manager offers a Configuration Manager to select and fine-tune the best-fitting strategy, while taking care that the selected options are compatible. The rationale for this flexibility and customizability of TraFIC is that federated database systems must be able to cope with a rich spectrum of options and tuning preferences in the underlying component databases.

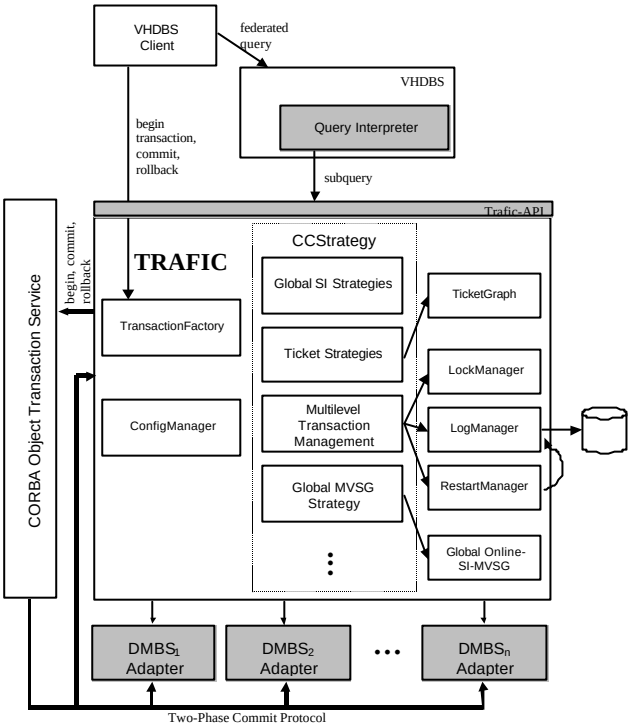

Figure 3 - Architecture of the Trafic Prototype

3.3 Implementation Issues

In this section, we present some implementation issues of the transaction manager and its strategies. First, we give a short overview how we make use of the Object Transaction Service. Then, we describe the transparent integration of the transaction functionality into the existing VHDBS system. Finally, we show how we derive read- and writesets of transactions from SQL predicates and decide if two such sets overlap.

3.3.1 OTS as a Building Block for Atomicity

Even though OTS alone is not sufficient for federated transaction management, it is a robust and easy-to-use means to guarantee the atomicity of federated transactions. TraFIC therefore makes extensive use of it.

As with the pure OTS solution, the interfaces of the database adapters are made transactional by inheriting from the pseudo-interface TransactionalObject. Subsequent method calls in the context of a

transaction are then implicitly integrated into this transaction. When a VHDBS client initiates a new federated transaction, the currently selected strategy creates a corresponding OTS transaction. Upon the first call of a transactional method of a database agent, OTS automatically begins a (sub-)transaction in the associated database system and registers it with the OTS server. Finally, upon the transaction's commit (or rollback), OTS coordinates the subtransactions in the involved component systems, applying the standard two-phase commit protocol. Thus, our prototype architecture requires that component systems can serve as XA participants; this is the case for the currently supported systems Oracle and O_2.

It should be noted that this technique has problems when multiple threads are active within a single transaction. This would be the case, for example, when a query is split into several subqueries at different databases all of which should execute in parallel. The OTS specification does not state how to cope with such multi-threaded transactions; rather it appears to be limited to a single thread suspending and resuming a transaction. One possible solution to this problem is an Orbix-specific extension of the Object Transaction Standard that provides a mechanism for attaching multiple threads to the same transaction. Another way to solve it could be to use a separate OTS subtransaction for each thread.

3.3.2 Transparent Integration into VHDBS

While the concept of a transaction object that takes all transactional method calls is elegant, it is somewhat inelegant that nontransactional methods are still sent directly to the federation server. It would be much better if a concept similar to the OTS approach could be applied: the client merely has to begin and commit transactions, but can use the same style of method calls for both transactional and non-transactional methods. The system itself should detect transactional methods and reroute those calls through the transaction manager. This should happen completely transparently for the client program.

In order to achieve this, we use the concept of Smart Proxies provided by Orbix, Iona's implementation of CORBA. This is a means to intercept outgoing method calls and replace them. Whenever the client starts a new transaction (using TraFIC's Transaction Factory), our glue code attaches the corresponding transaction object to the current thread. Every "transactional" method call is redirected to TraFIC by a SmartProxy, using the associated transaction object (see Figure 4). TraFIC then determines the appropriate OTS transaction, executes the original method in the context of this transaction, and returns the result to the caller.

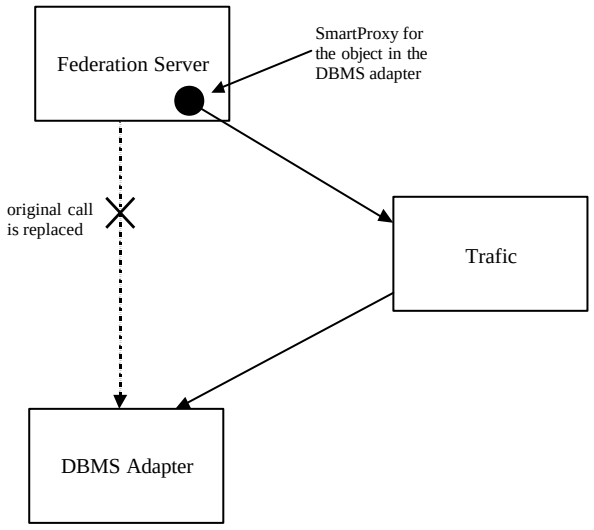

Figure 4 - Using Smart Proxies to integrate Trafic into VHDBS

By using those smart proxies, the necessary operations on the client side are essentially the same as for the OTS implicit propagation of the transaction context. The client merely needs to explicitly call methods of the transaction manager to begin and commit or rollback a transaction.

3.3.3 Deriving Read- and Write-Sets From Predicates

In this section, we present how to determine read and write sets of transactions and how to detect overlapping read and write sets. This is necessary for the implementation of the graph-based technique shown in Section 2.3, and is reused in several other strategies that are not covered in this paper.

Due to the autonomy of the databases, it is impossible to directly observe read and write accesses to data objects in the local databases. Therefore, we need to build an approximation of the read and write sets of federated transactions in the various component systems based on the operations they submit. In our implementation, we have adopted the approach of [SSW95] for analyzing potential conflicts between the predicates of SQL commands. Because every operation of a federated transaction is routed through the transaction manager, we can monitor the actions and derive predicates that characterize the objects accessed by those transactions. 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, when 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 table "parts" that the transaction 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 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 these predicates, we can determine if the write set of t_i, i.e. the subset that t_i updated, and the read set of t_j are *potentially* overlapping, i.e., the conjunction of the predicates is satisfiable. This may actually signal false overlappings, because we can only approximate the set of objects being accessed. However, this approximation is conservative in capturing all real overlappings, so that we can guarantee the correctness of an algorithm that is based on the approximated results.

The viability of this mechanism, i.e., the ability to handle a sufficiently broad class of practically relevant SQL commands with reasonable accuracy and low overhead, has been demonstrated already in the experimental work of [SSW95]. Our experiments fully confirmed that the predicate-based approach is not a bottleneck at all.

4 Benchmark Results

In this section, we present benchmark results that show the viability of our methods and implementation. In Subsection 4.1, we briefly introduce the setting in which the benchmark was taken. In Section 4.2, we present the measured results.

4.1 Benchmark Setting

We evaluated our protocols using the full-fledged implementation of the VHDBS federated database system on top of Oracle 8i databases. The scenario for the benchmarks was 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. 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 (see Figure 5). 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.

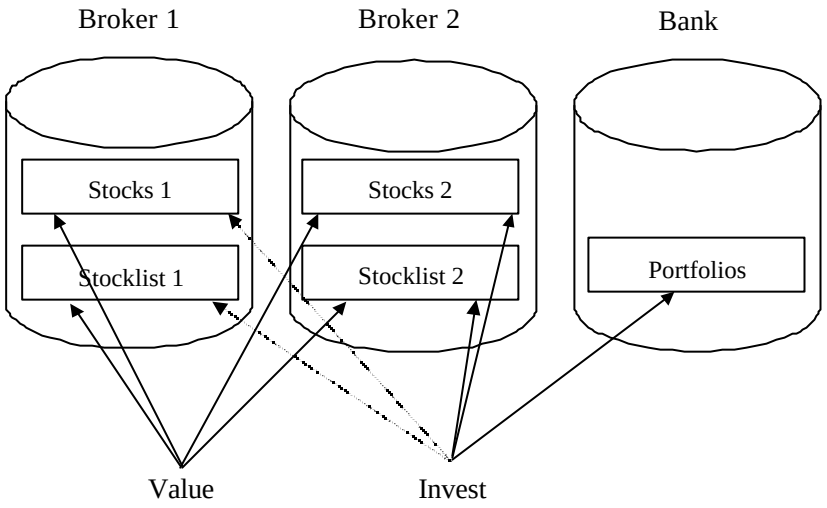

Figure 5 – Stock Brokerage Scenario for the Benchmarks

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. The Oracle instances were run on a Sun Ultra Enterprise 4000 with eight processors and two PCs running Windows NT.

4.2 Performance Measurements

This section presents selected results from our performance measurement, a more comprehensive discussion can be found in [SW00a]. We present the results for the following two 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.

All experiments measured transaction throughput for a particular number of clients. Each client spawns a new transaction upon the completion of the last one. 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 con-

stantly 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 0, *OTT* for the traditional (optimistic) ticket technique, and *MVSG* for the Online SI-MVSG strategy presented in Section 0. 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.

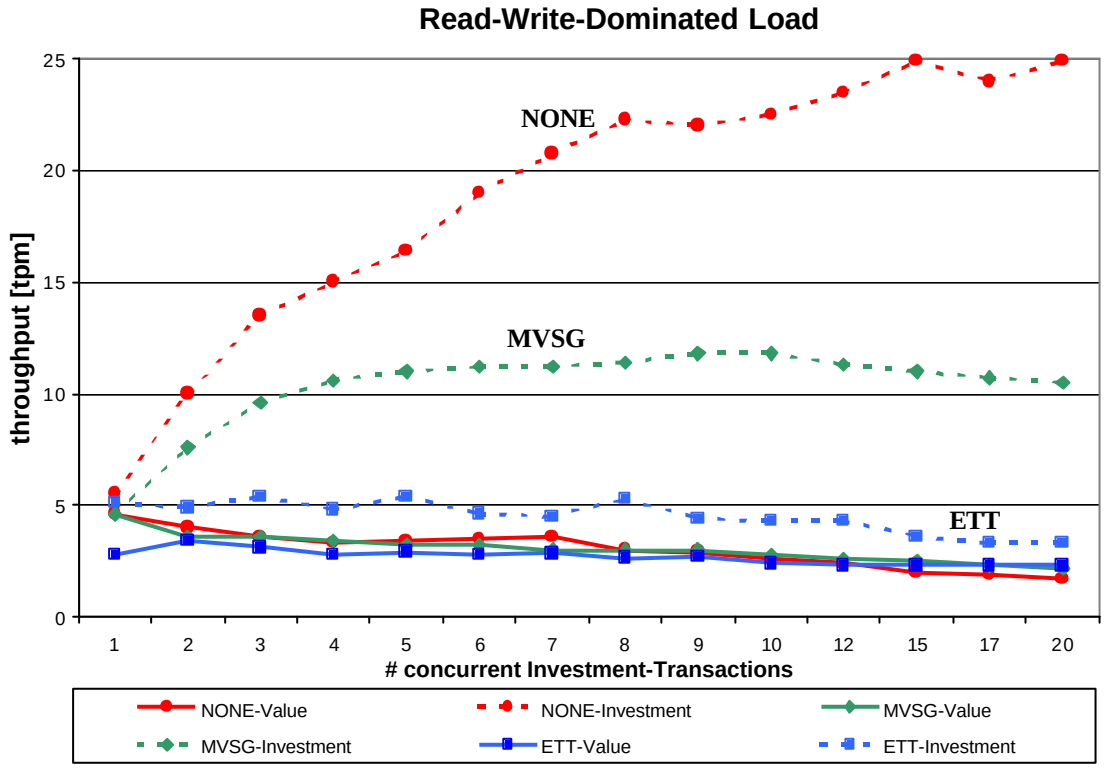

Figure 6 - Performance Results for the Read-Write Dominated Setting

Figure 6 shows the throughput results for the read-write setting. As for the Investment transactions (dashed lines), we see that the OSI-MVSG technique outperforms all other techniques until a certain MPL. Throughput of Value transactions (solid lines), on the other hand, is roughly the same for all other strategies. The ticket method performs poorly for read-write transactions because of the additional contention for ticket objects.

Figure 7 shows both Investment (dashed lines) and Value (solid lines) throughput for the read-only dominated setting. The OSI-MVSG strategy again shows the best throughput for Investment transactions. As in the read-write dominated case, throughput for Value transactions is roughly the same for all strategies.

To summarize our experimental findings, we found that the Online SI-MVSG has proven to be the method of choice for mixed environments with both read-write and read-only transactions. The ex-

tended ticket technique has the best performance for read-only transactions, but has major performance disadvantages with read-write transactions. Both strategies show reasonably lower throughput than the strawman strategy NONE; this is the price to pay for the additional correctness guarantee they provide.

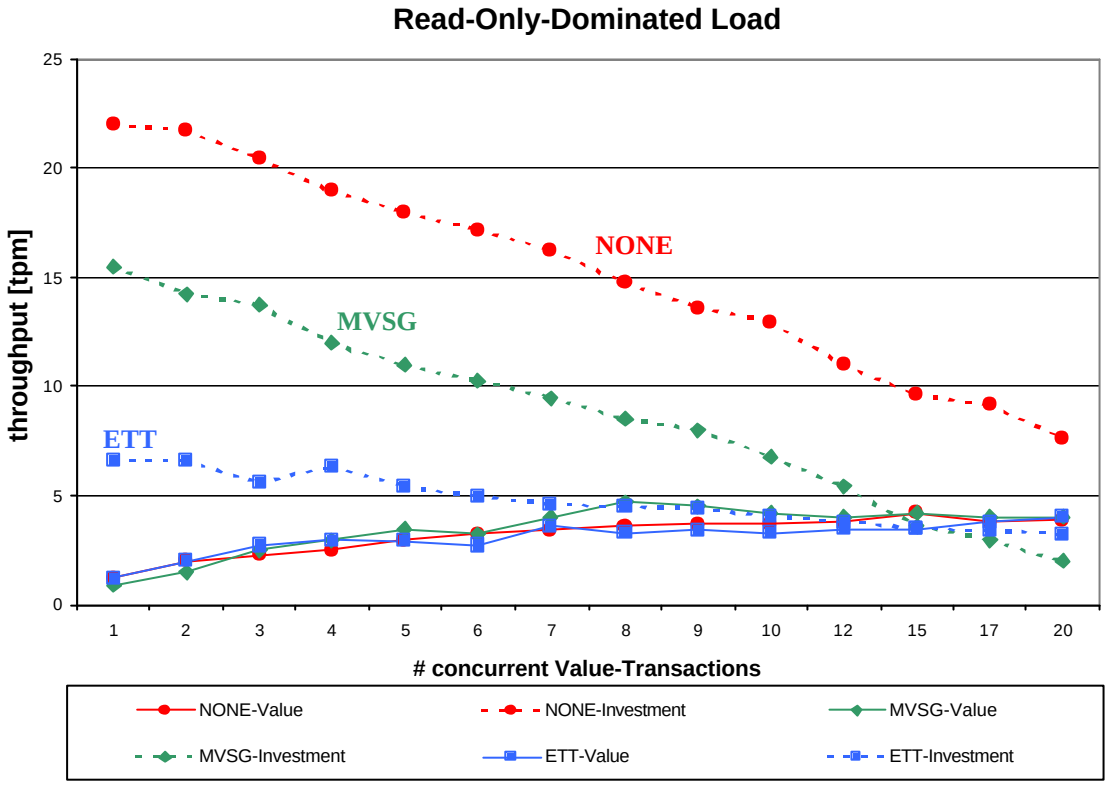

Figure 7 - Performance Results for the Read-Only Dominated Setting

5 Concluding Remarks

There is a practice that application requirements on data consistency can sometimes be relaxed and traded off for performance or simplicity of the overall system. For example, when airlines use to overbook their flights anyway, they might as well tolerate a very small risk of ending up with inconsistent passenger counters due to overly loose concurrency control. This attitude also explains the popularity of sub-serializability isolation levels in real applications. However, this kind of practice is treacherous. Once in a while mission-critical databases exhibit minor but nevertheless troublesome inconsistencies. This is often attributed to software bugs, but it is not unlikely that insufficient concurrency control causes a good fraction of these problems [BBGM+95]. So relaxing transactional isolation requires extreme care, deep insight into the application's nature, detailed understanding of the entire application system, and often time-consuming intellectual analysis of application program logic. This practice is best characterized as black art, as opposed to scientifically founded engineering.

With the increasing shortage of qualified people, the high cost of human labor, the inherent probability of human oversight, and the need for easy adaptation of application logic to meet rapidly changing business requirements, intellectually relaxing data consistency and transactional isolation is clearly an anachronistic approach. The only hope for keeping the most important asset of modern Internet applications, the data, valid is to rely on generic, system-provided concurrency control with guaranteed serializability in the strict sense. The research on transactional federations had received much attention

in the late eighties and early nineties, under the label of "multidatabase transactions" then. It contributed some nice results (e.g., the theorems about commit-order-preserving and rigorous schedules or the ticket method), but did not have much impact later on as it overlooked some practically crucial aspects, most notably, the fact that real database products use fancy isolation levels rather than local serializability. Consequently, this line of research has been criticized by practitioners, and eventually gave up too early. With the ongoing megatrends towards Internet-based transactional federations, we believe that it is vital to revive this research area (see also [Ston98, MK+99]) and that this paper makes a useful contribution in the right direction.

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.

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

[ANSI92] ANSI X3.1335-1992: American National Standard for Information Systems – Database Language – SQL, November 1992.

[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.

[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.

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

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

[BS92] Y. Breitbart, A. Silberschatz: *Strong Recoverability in Multidatabase Systems,* RIDE, Tempe, 1992.

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

[CSS99] S. Conrad, G. Saake, K.-U. Sattler*: Informationsfusion – Herausforderung an die Datenbanktechnologie*, in: A. P. Buchmann (Ed.): *German Database Conference (BTW)*, 1999.

[DE00] Data Engineering Bulletin, *Special Issue on Database Technology in Electronic Commerce*, IEEE Computer Society, March 2000.

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

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

[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.

[IBM98] IBM Corp.: *DB2 DataJoiner® Administration Supplement Version 2.1.1*, July 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.

[OMG97] Object Management Group, Inc: *CORBA Services: Common Object Services Specification, Chapter 10: Transaction Service*, Revision 1.1, November 1997.

[Orac96] Oracle Corp.: *Oracle Transparent Gateway for Microsoft SQL Server Installation and User's Guide*, Release 4.0, 1996.

[Orac99a] Oracle Corp.: *Oracle8i Transparent Gateways*, White Paper, August 1999.

[Orac99b] Oracle Corp.: *Oracle8i Distributed Database Systems*, 1999.

[Orac99c] 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.

[SW00a] R. Schenkel, G. Weikum*: Integrating Snapshot Isolation Into Transactional Federations*, in: Proceedings of the 5th IFCIS International Conference on Cooperative Information Systems (CoopIS 2000), Eilat, Israel, September 6-8, 2000.

[SW00b] H. Schöning, J. Wäsch: *Tamino – An Internet Database System.* Proc. Of the 7th International Conference on Extending Database Technology, Konstanz, March 2000.

[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.