CONCURRENT CHECKING OF GLOBAL CROSS-DATABASE INTEGRITY CONSTRAINTS

Stefan Böttcher
University of Paderborn
Fachbereich 17 (Mathematik-Informatik)
Fürstenallee 11 , D-33102 Paderborn , Germany
email : stb@uni-paderborn.de

Abstract Global integrity of data across the boundaries of single database systems is an important requirement in multi-database systems, but cannot be achieved without transaction synchronization across the boundaries of database systems. The problem is to guarantee that global transactions leave these multiple databases in a globally consistent state and to avoid that global integrity checks unnecessarily block other application transactions. We present a solution that offers both, unlimited concurrency between global integrity constraint checks and local transactions, and increased concurrency of global integrity checks and global application transactions, thereby contributing to a higher performance of global integrity checks. We show that the key idea of our approach, i.e., to lock the integrity constraint itself, leads to a correct and efficiently implementable lock protocol for concurrent integrity constraint checks crossing database system boundaries. Since our approach blocks significantly less resources for global integrity checking than the conventional approach, we consider it to be an important contribution to guarantee global cross-database integrity.

Key words global integrity of data, transaction synchronization, concurrent integrity checking.

1. INTRODUCTION

1.1 Problem origin and motivation

Due to acquisitions and mergers, an increasing number of companies has enterprise data separated on different databases which have not yet been integrated or cannot be integrated into a distributed database running on a single database system. Whenever data in one or more of the involved

---

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: 10.1007/978-0-387-35693-8_16

M. Gertz (ed.), *Integrity and Internal Control in Information Systems V*

databases is modified, *global data integrity* involving multiple databases on possibly different DBMS is a key requirement. Whenever the check of global integrity constraints is time consuming, but essential for data consistency of an enterprise, it is desired that global data integrity checks can be done concurrently to other applications, e.g. to local applications accessing only a single database. The concurrent execution of global integrity checks and transactions of other applications requires an appropriate concurrency control strategy.

Our work is motivated by a company with multiple divisions, each containing its own local product database, and a database in the headquarters, which summarizes some of the product data from several divisions for other purposes like external product information, marketing, etc.. Whenever a transaction runs on a database of a division and deletes, inserts or modifies product information, it has to be checked whether or not the corresponding information in the headquarters' database is still valid or has to be changed too. And vice versa, whenever a transaction running on the production headquarters' database changes information that is relevant to a specific division, the validity of corresponding information in the database of this division has to be checked too. While our contribution guarantees that such global transactions leave these multiple databases in a globally consistent state, it additionally allows for increased concurrency of global integrity checks and application transactions.

1.2 Relation to other work and our focus

There has been a lot of research on guaranteeing data integrity in multi-database and distributed transaction environments. Some of these contributions cover semantic integrity control, the majority however contributes protocols for transaction synchronization.

Since semantic integrity control usually involves queries on large amounts of data, two kinds of optimization techniques have been proposed: first, to prove at compile-time that update transactions cannot violate certain given integrity constraints [4,15,30], and second to reduce the complexity of the remaining queries at run-time [17,31,32,33]. While the strategies for run-time integrity control optimization focus on query simplification for integrity checks, our protocol focuses on the concurrent execution of integrity checks. Nevertheless, our protocol is compatible with this approach, because it is orthogonal to approaches used for the run-time optimization of integrity constraint checks, i.e., these optimization strategies can be combined with our protocol.

Transaction synchronization protocols are classified according to at least five criteria [14]: their synchronization strategy (e.g. 2-phase locking or validation), synchronization granularity (e.g. tuple, page, object or XML fragment), whether or not they are multi-level synchronization strategies, whether or not they use a global scheduler on top of local schedulers, and whether or not a single unique synchronization technique is used for all pairs of conflicting operations. Our contribution to concurrent global integrity checks is an improvement along the last mentioned criterion, i.e., we use different synchronization techniques for different pairs of conflicting operations. Note that since our contribution is orthogonal to the first four criteria, it can be combined with any choice for the first four criteria (e.g. to locking *or* validation, e.g. to objects in OODBMS *or* XML fragments in XML databases, etc.).

Additionally to other contributions (e.g. [6]) that suggest to synchronize read-write conflicts different from write-write conflicts, we distinguish two kinds of read operations - ordinary queries and integrity checks - and propose an improved strategy for the concurrent execution of (global) integrity checks and other operations. In this aspect, our approach is completely different from other work on semantic based concurrency control in (multi) database systems [11,16,18,19,20,21,22,24]. Our contribution is an add-on-protocol which can be used in combination with any existing synchronization protocol in a participating database that guarantees serializable schedules for write operations and other read operations (except global integrity constraints).

Furthermore, different serializability levels are distinguished [14] (-1: unrestricted concurrency, 0: avoid lost updates, 1: guarantee committed read, 2: repeatable read, 3: serializable). While a variety of approaches to global synchronization relax serializability in order to increase concurrency (e.g. [12, 24, 2, 1, 5, 29]), most contributions argue that it is desirable to accept only serializable schedules [3, 23, 28, 35, 10, 9, 27, 13]. Our add-on-protocol offers both, it allows for unrestricted concurrency of global integrity checks with local transactions (i.e. global integrity checks can read database data without setting any locks), *and* it guarantees serializable schedules. Therefore, our contribution allows for a higher degree of application parallelism *and* guarantees global data consistency.

2. FUNDAMENTAL PRINCIPLES AND PROBLEM DESCRIPTION

2.1 The underlying transaction model

We consider transactions as sequences of read operations, integrity checks, and committed write operations. We assume, that only those transactions commit which have previously checked all integrity constraints successfully, and other transactions are aborted. We furthermore assume, that uncommitted write operations of a transaction are not visible to (the integrity checks of) other transactions. We distinguish *local transactions*, that, including their integrity checks, access only a single database system, from *global transactions*, that access multiple database systems.

As mentioned above, our transaction model does *not* require a specific strategy (e.g. locking or validation) or a specific data model (e.g. relational, object oriented or XML) or a specific access granularity (e.g. objects or XML fragments) or a specific query language to define integrity constraints (e.g. tuple relational calculus or OQL). Note that only for simplicity reasons, we present our approach as an extension of a two-phase lock protocol and use formulas of tuple relational calculus to express integrity constraints throughout the discussion of our approach. However, the results can be equally applied to other data models and database systems, other languages for integrity constraints and other synchronization protocols.

In the presented lock protocol, all locks needed for a specific operation are acquired before this operation and are released after commit and after the operation is completed. This transaction model is more formally defined in Section 4 and will be the basis of the correctness proof presented later in the paper.

2.2 Conflict definition for integrity checks and write operations, explained using an example

We will use and extend the following example throughout the paper: We have one production division, say in London, using a database called DB_1 which contains a relation called R_1 and another production division, say in Paris, using a database DB_2 which contains a relation R_2. Within the headquarters, a database DB_3 contains a relation R_3 which summarizes some of the information stored in R_1 in London and in R_2 in Paris. For our example, we require the following two global integrity constraints to hold:

1. "for every object o_3 in relation R_3 in the headquarters there exists an object o_1 in R_1 with the same number (nr) or there exists an object o_2 in R_2 with the same number (nr)". This global cross database integrity constraint IC_1 can be written as formula in the tuple relational calculus as follows:

$$\mathbf{IC_1: \quad \forall\, o_3 \in R_3 \quad (\ \exists\, o_1 \in R_1\ (\, o_1.nr = o_3.nr\,) \lor \exists\, o_2 \in R_2\ (\, o_2.nr = o_3.nr\,)\ ).}$$

2. "for every object o_1 in relation R_1 located in DB_1 in London, there is a corresponding object o_3 in the summarizing relation R_3 in DB_3 in the headquarters with the same number (nr)". For this integrity constraint IC_2, we get the following formula written in tuple relational calculus:

$$\mathbf{IC_2: \quad \forall\, o_1 \in R_1 \quad \exists\, o_3 \in R_3\ (\, o_1.nr = o_3.nr\,)\ \ .}$$

Furthermore, we have three (global) transactions T_1, T_2 and T_3 each modifying only a single local database:

T_1: { delete o_1 from R_1 where $o_1.nr < 4$;
do time consuming operations on local database; }
T_2: { delete o_2 from R_2 where $o_2.nr < 4$; }
T_3: { delete o_3 from R_3 where $o_3.nr = 2$; }

We say an integrity constraint is *violated,* iff the truth value of its formula is changed from *TRUE* to *FALSE.* Note that the delete operation of T_1 could violate the integrity constraint[1] IC_1, because R_1 occurs existentially quantified in IC_1. However the delete operation of T_3 could never violate IC_1, because R_3 occurs universally quantified in IC_1, and therefore the only possible change of the truth value is from FALSE to TRUE [2]. Therefore T_1 and T_2 respectively have to check IC_1, whereas T_3 does not need to check IC_1. On the other hand, T_3 which deletes an object from R_3 may violate IC_2, because R_3 occurs existentially quantified in IC_2. This can be generalized as follows.

We say, a transaction *violates* an integrity constraint, iff its write operations (delete, insert, update) change the truth value of the Boolean-valued query (or formula) associated with the integrity constraint from TRUE to FALSE [31,32]. Whether an insert (a delete) operation into (from) a relation R_i could violate an integrity constraint, depends on the positive (or negative) occurrence of R_i in the formula of the integrity constraint. An occurrence of R_i in a formula is said to be *positive* (*negative*), iff it occurs in the syntactic scope of an even (odd) number of negations and universal quantifications [17,33]. Within IC_1, R_1 and R_2 occur positively and R_3 occurs negatively, whereas within IC_2, R_1 occurs negatively and R_3 occurs positively.

The following table summarizes the possible changes of the truth values of integrity check formulas IC by a following insert (R_i+o) or delete (R_i-o) operation:

	R_i occurs positively In the formula of IC	R_i occurs negatively in the formula of IC
$R_i + o$	From FALSE to TRUE	from TRUE to FALSE
$R_i - o$	From TRUE to FALSE	from FALSE to TRUE

This can be taken as the basis for conflict definitions. An integrity check IC and a following write operation on a relation R_i are called *in conflict,* iff the

---

[1] Read "could violate the integrity constraint" as: there exists a possible global database state (i.e. there may be a possible combination of objects in the databases) in which the integrity constraint is violated.

[2] A change of the truth value from FALSE to TRUE can be ignored by integrity checks of the current transaction, because it is assumed that all integrity constraints are TRUE after the completion of previous transactions.

write operation on R_i may change the truth value of the formula of IC from TRUE to FALSE. Note that it is not useful to extend the conflict definition to write operations that modify an integrity constraint's truth value from FALSE to TRUE, because a transaction is aborted when an integrity constraint check yields FALSE and aborted transactions are not considered in serialization graphs [7].

2.3 Problem description

The problem description consists of two parts: first, checking global integrity constraints in multi-database systems needs synchronization; second, the usual treatment of integrity checks as queries tends to block more concurrent transactions than necessary from using large parts of the involved databases.

We extend the above example and assume, that initially the integrity constraint is valid and all three relations R_1, R_2, and R_3 contain at least one object o_1, o_2, and o_3 respectively with $o_1.nr = 2$, $o_2.nr = 2$, and $o_3.nr = 2$.

In order to demonstrate that transactions T_1 and T_2 have to synchronize their integrity checks although both perform write operations on different relations in different databases, we look at the following history, that without synchronization could cause a violation of the integrity constraint IC_1:

1. T_1 performs the delete operation on R_1 on a storage, that is not visible to T_2.
2. T_2 performs the delete operation on R_2 on a storage, that is not visible to T_1.
3. T_1 checks IC_1, but does not see the changes (by the delete operation) of T_2.
4. T_2 checks IC_1, but does not see the changes (by the delete operation) of T_1.
5. T_1 commits, since its integrity check of IC_1 was successful.
6. T_2 commits, since its integrity check of IC_1 was successful.
7. Both transactions make their changes visible to other transactions, i.e., the object o_3 with $o_3.nr=2$ does neither have a corresponding object o_1 in R_1 nor an object o_2 in R_2.

Note that after these 7 steps, IC_1 is violated, although both transactions checked the constraint.

The usual way to avoid this history is to use a query that checks the integrity constraint, say IC_1, [17,31,32,33,34] and to synchronize this query using ordinary read locks. In this case, T_1 would require a write lock on (a part of) R_1 and read locks on R_2 and R_3. Since T_2 requires a write lock on R_2 too, and both locks on R_2 are not granted at the same time, the history is avoided.

However, the disadvantage of the usual treatment of integrity checks as queries can be shown, when we consider transaction T_3 listed above. When T_1 checks IC_1 using a conventional query, the read lock required by T_1 on R_3 blocks transaction T_3 (which needs a write lock on R_3), although T_3 can never violate the integrity constraint IC_1, as shown above. Therefore, treating integrity checks as queries blocks more concurrent transactions than necessary [8].

As the previous example shows, it is necessary to block some but not all write operations of concurrent transactions on the data which is read for an integrity check, because a modification of the truth value from TRUE to FALSE has to be prevented, whereas truth value modifications from FALSE to TRUE can be ignored, because in this case the violating transaction is aborted. However, this is different for ordinary Boolean-valued queries, where every truth value modification by concurrent transactions has to be prevented.

Because of this difference, integrity checks allow for more concurrent transactions than other queries [8]. Now the problem can be stated as follows. ”Find a simple and efficient lock protocol that correctly synchronizes global integrity checks but reduces unnecessary conflicts of these integrity checks with write operations of concurrent transactions”.

3. OUR SOLUTION TO THE PROBLEM: THE LOCK PROTOCOL FOR INTEGRITY CONSTRAINTS

3.1 The basic idea: different synchronization for integrity checks and other queries

As mentioned before, the traditional way to synchronize integrity checks against write operations of concurrent transactions is to synchronize them the same way as queries are synchronized against write operations, i.e., to read lock the objects (or tuples or relations or XML fragments) which are read in order to perform the integrity check. In contrast to that, we distinguish between integrity constraint checks and other queries w.r.t. synchronization. While other queries use read locks as usual, an integrity constraint check does not use a read lock for the data which is read for the constraint check. Therefore, ordinary read operations on a data item will block all write operations of concurrent transactions on that data item. However, an integrity constraint check (using our protocol) does not block every write operation of parallel transactions on data which is read for the integrity check.

Note that we do not change the synchronization of conflicts between other queries and write operations, but use our protocol as an add-on protocol for conflicts between integrity checks and write operations.

3.2 Our solution: the integrity constraint as lockable object

Our new approach to concurrent global integrity control avoids the incorrect history in the given example by locking the integrity constraint itself, i.e., each transaction has to lock those integrity constraints which it has to check. As with locks for other objects, the lock for an integrity constraint has to be obtained, before the integrity check can be performed, i.e., before the truth value of it can be read. And the lock of an integrity constraint is released, after the write operations of the transaction are committed and visible to other transactions.

By forcing transactions to lock each integrity constraint before they check it, the scheduler allows only one transaction at a time to check that integrity constraint and perform those of its write operations that might violate the integrity constraint. Other transactions that want to check the same integrity constraint are forced to wait with their integrity check, until the first transaction is either committed and has completed its write operations, or is aborted. Thereby, the previous history of transactions T_1 and T_2 is avoided, because transactions T_1 and T_2 both have to acquire a lock on integrity constraint IC_1 - which is only granted to one transaction at a time.

On the other hand, transaction T_3 does not need such a lock on IC_1. Therefore T_3 can run in parallel with transaction T_1, e.g., while T_1 performs its time consuming work on local data.

In order to give a more detailed insight in the idea, let us look at transactions T_3 and T_1 and integrity constraint IC_2 from Section 2.2 . Since R_3 occurs existentially quantified in IC_2 , a deletion of objects from R_3 may violate IC_2. Therefore, T_3 needs a lock on IC_2, which allows it to check IC_2. However, a lock on IC_2 is not needed for T_1, because R_1 occurs (only) universally quantified in IC_2, and therefore a deletion from R_1 may not violate IC_2.

Note, that under our protocol it is still possible to run T_1 and T_3 in parallel, whereas the usual treatment of IC_2 as query would prohibit the parallel execution of T_1 and T_3, because T_1 modifies relation R_1 and T_3 reads R_1 for its integrity check.

In general, transactions need to lock only those integrity constraints which they might violate.

Furthermore, no transaction needs any read lock in order to check IC_1 or any other integrity constraint. The lock on the integrity constraint itself will be sufficient. Note that this is a significant simplification compared to a previous approach [8] that distinguishes insert-locks and delete-locks and blocks concurrent insert (delete) operations on positive (negative) occurrences of a relation R occurring in an integrity constraint check IC.

To summarize, our lock protocol requires to make a difference between integrity constraint checks and other queries. While other queries are synchronized by read locks as usual and integrity checks are not synchronized by read locks, our lock protocol can be considered as an add-on for integrity checks, allowing for increased concurrency of transactions.

3.3 Implementation of the global scheduler for cross database integrity constraint checks

A global scheduler for integrity constraints can be added in the following way to multiple database systems. Transactions T_1, T_2, and T_3 can be synchronized locally in their databases DB_1, DB_2, and DB_3 respectively, with the following extension. Whenever a transaction has to check a global integrity constraint (i.e., a constraint referring to data in a different database), it has to ask the global scheduler for a lock on that global integrity constraint (and release that lock after its commit and completing its write operations). Note that global synchronization is only needed for global constraints, i.e., checks for local integrity constraints could be synchronized locally on one database.

Cross-database integrity checking can be implemented using a single global lock array for integrity constraints and a replicated table containing necessary checks for global integrity constraints.

The single global lock array contains one entry for each global cross database integrity constraint and is used by the global integrity lock scheduler. The entries in the lock array change, when a transaction acquires or releases a lock on a global integrity constraint. A snapshot of the single global lock array might look like this.

	IC_1	IC_2	...	IC_n
locked	not locked	locked (by T_3 in DB_3)		not locked

3.4 The replicated table of necessary cross-database integrity constraint checks

The replicated table of integrity constraint checks contains the information of which insert or delete operations might violate which integrity constraints, and of which optimized queries can be submitted for integrity constraint checking. This table is never modified (unless a new integrity constraint is defined) and can therefore be replicated on all databases.

Each column of the table represents one cross-database integrity constraint. For each relation R_i occurring in at least one cross database integrity constraint, there are two rows in the table, one (**R_i+o**) for the insertion of objects **o** into the relation **R_i**, and one (**R_i-o**) for the deletion of objects **o** from relation **R_i**.

	IC_1	**IC_2**	**...**	**IC_n**
R_1+o	O.K.	Opt.Query$_{21}$+(o)		...
R_1-o	Opt.Query$_{11}$-(o)	O.K.		...
R_2+o	O.K.	O.K.		...
R_2-o	Opt.Query$_{12}$-(o)	O.K.		...
...				
...				
R_m+o	...	...		...
R_m-o	...	...		...

The fields of the table contain optimized queries which are needed for integrity checking. An **O.K.** entry in the field for R_1+o and IC_1 means that no integrity check of IC_1 is needed for insertions into relation R_1.

The entry **Opt.Query$_{21}$+(o)** contains the optimized query, which is necessary in order to check IC_2 for the insertion of an object o into relation R_1. The inserted object o is an input parameter of that optimized query. Since R_1 occurs universally quantified in IC_2, and we assume that IC_2 was valid before the insertion operation of o into R_1, the integrity constraint has to be checked only for the new object o. Hence, the integrity constraint check of IC_2 could be simplified to the following optimized Boolean query function with the inserted object o as parameter [17, 31, 32, 33]:

Boolean Opt.Query$_{21}$+(o) { return $\exists\, o_3 \in R_3\, (\, o_3.nr = o.nr\,)$; }.

Each transaction which wants to insert an object o into R_1 can provide the value for o.nr, and thereafter the optimized query can be applied to R_3 which is stored in database DB_3.

Note that this optimized query can be executed on database DB_3 without any synchronization with concurrent write operations of other transactions running on DB_3, i.e., locking of the integrity constraint is sufficient for correct synchronization, since it prevents other concurrent transactions from modifying the result of the (optimized) integrity check. Therefore, the optimized query for the integrity check does not need any read locks on the data read in database DB_3. This can be easily implemented in DB_3 by using a lower degree of isolation (e.g. allow to read uncommitted data) for the integrity checks.

3.5 Concurrency of global integrity checks and local transactions

We call a transaction *local,* if it needs to access only a single database including all necessary integrity constraint checks. Given the two integrity constraints IC_1 and IC_2 as before, a transaction running on database DB_2

T_4: { insert o_2 into R_2 ; }

is a local transaction for the following reason. It cannot violate IC_1, because an insert into R_2 can never change the truth value of IC_1 from TRUE to FALSE. Since T_4 does not have to check any global integrity constraint, there is no need to access another database, i.e. T_4 can be executed locally[3]. Note that the traditional approach to treat integrity checks as queries would require a read lock on R_2 for the check of IC_1, i.e. it would forbid to execute the check of the global integrity constraint IC_1 concurrently with the local transaction T_4. However, our approach allows to run T_4 concurrently with the check of the global integrity constraint IC_1, because T_4 cannot violate a successful check of the global integrity constraint IC_1. More generally, our approach allows all global integrity constraints to be checked concurrently with all local transactions.[4]

3.6 Comparison with the conventional synchronization of integrity checks

If we compare our protocol to the conventional treatment and synchronization of integrity checks as queries, our protocol needs exactly one additional lock for each (global) integrity constraint check of a transaction, but it does not need a single read lock for the (global) integrity checks in any of the databases.

However, when cross database integrity constraint checks are treated and synchronized as ordinary queries, a lock for each object (or page or relation or XML fragment) occurring in the optimized integrity check has to be acquired. Since in global cross-database integrity constraints at least two relations in different databases are involved, after optimization at least one remote relation is accessed. In the case of fine-grained lock operations, usually multiple objects will have to be locked, hence, our protocol will need much fewer locks. On the other hand, the coarser-grained the locks are, the more likely an ordinary query used for an integrity check will unnecessarily

---

[3] Note that T_4 is still a local transaction, when it has to check a local integrity constraint, e.g. a local constraint that allows inserts into R_2 only until a limit of n objects in R_2 is reached.

[4] The reason is that a transaction that may violate a global integrity constraint check, has to check that global integrity constraint itself, and therefore can not be a local transaction.

block insert (or delete) operations of concurrent transactions, i.e., the more will our protocol allow for increased concurrency of global integrity checks.

But even in the case of fine-grained locks, our protocol allows for increased concurrency compared to the treatment of integrity checks like queries, as can be seen in the above example of transactions T_1 and T_3.

Whenever global integrity checks tend to read large parts of the involved databases, the conventional treatment of global integrity checks as queries tends to block (or to be blocked by) a large number of concurrent writing transactions. This includes not only global transactions, but also local transactions writing only into a single database. Therefore, our protocol optimizing the synchronization of these global integrity checks will significantly contribute to increased parallelism, not only for global transactions, but also for local transactions. Therefore, we consider our protocol to be an important improvement to global consistency and increased application transaction parallelism.

4. THE LOCK PROTOCOL AND ITS CORRECTNESS

4.1 Lock protocol definition

The correctness criterion for a lock protocol is whether (or not) it guarantees serializability. The definition of serializability is usually based on a conflict definition for pairs of operations [7]. Conflicting operations op_1 and op_2 of different transactions T_1 and T_2 define a dependency (a directed edge in the dependency graph), formally written $T_1 \leftarrow T_2$, iff op_1 is executed before op_2. A history of concurrent transactions is serializable, iff the dependency graph of the committed transactions is acyclic.

Formally, a transaction T_i is *legal*, if it obeys the following precedence rules for its operations:

Transaction T_i waits for a read lock *lock-r(T_i,o_j)* for any data item o_j accessed by an ordinary query of T_i before the transaction reads the data item o_j (i.e. performs *read(T_i,oj))* , and the transaction releases this read lock *unlock-r(T_i,o_j)* after transaction commit *c(T_i)* . Hence, for these operations the lock protocol guarantees the following precedence relation < :

$$\text{lock-r}(T_i,o_j) < \text{read}(T_i,o_j) < c(T_i) < \text{unlock-r}(T_i,o_j)\ .$$

Transaction T_i waits for an integrity constraint lock *lock-i(T_i,Ic)* for any integrity constraint I_c that has to be checked by transaction T_i before the transaction checks the integrity constraint I_c, and the transaction releases this lock *unlock-i(T_i,Ic)* after transaction commit *c(T_i)* and after each of its write operations (e.g. *write(Ti,oj))* is completed. Furthermore, write operations are performed after all integrity constraints are successfully checked. Hence, for arbitrary operations *lock-i*(T_i,I_c), *write*(T_i,o_j), *unlock-i*(T_i,I_c) of committed transactions the lock protocol guarantees the following precedence relations < :

$$\text{lock-i}(T_i,I_c) < \text{write}(T_i,o_j) < \text{unlock-i}(T_i,I_c) \text{ and}$$

$$\text{lock-i}(T_i,I_c) < c(T_i) < \text{unlock-i}(T_i,I_c) \; .$$

Finally, every transaction T_i waits for a write lock *lock-w*(T_i,o_j) for any data item o_j written by T_i before the transaction writes the data item o_j *write*(T_i,o_j) and the transaction releases this write lock *unlock-w*(T_i,o_j) after the completion of the write operation and after transaction commit *c*(T_i). Hence, for these operations the lock protocol guarantees the following precedence relation < :

$$\text{lock-w}(T_i,o_j) < \text{write}(T_i,o_j) < \text{unlock-w}(T_i,o_j) \text{ and}$$

$$\text{lock-w}(T_i,o_j) < c(T_i) < \text{unlock-w}(T_i,o_j) \; .$$

The compatibility rules for locks are as follows:

Write locks on the same object o_j cannot be held by different transactions T_h and T_i at the same time, i.e., if both transactions get write locks on o_j, then the following lock rule holds:

$$\text{unlock-w}(T_h,o_j) < \text{lock-w}(T_i,o_j) \text{ or } \text{unlock-w}(T_i,o_j) < \text{lock-w}(T_h,o_j) \; .$$

An equal lock rule states that locks on the same integrity constraint I_c cannot be held by different transactions T_h and T_i at the same time, i.e., if both transactions get locks on I_c, then

$$\text{unlock-i}(T_h,I_c) < \text{lock-i}(T_i,I_c) \text{ or } \text{unlock-i}(T_i,I_c) < \text{lock-i}(T_h,I_c) \; .$$

A similar rule holds for read locks lock-r(T_h,o_j) and write locks lock-w(T_i,o_j) on the same object o_j.

$$\text{unlock-r}(T_h,o_j) < \text{lock-w}(T_i,o_j) \text{ or } \text{unlock-w}(T_i,o_j) < \text{lock-r}(T_h,o_j) \; .$$

The 2-phase rule states for each transaction T_i, that all of its lock operations must precede all of its unlock operations:

Let *lock*(T_i,*X*) be any lock operation of T_i, i.e. *lock-r*(T_i,o_j) or *lock-i*(T_i,I_c) or *lock-w*(T_i,o_j) on an object o_j or an integrity constraint I_c , and let *unlock*(T_i,*Y*) be any unlock operation of T_i on the same or a different object o_b or integrity constraint I_d , i.e. *unlock-r*(T_i,o_b) or *unlock-i*(T_i,I_d) or *unlock-w*(T_i,o_b) , then the 2-phase rule guarantees the following precedence:

$$\text{lock}(T_i,X) \; < \; \text{unlock}(T_i,Y)$$

4.2 Sketch of the correctness proof

Legality of transactions, together with 2-phase locking and the compatibility rules for locks guarantee serializability. The serializability proof is identical to that given for ordinary 2-phase locking (e.g. in [7]), except that it is extended to the additional locks for integrity constraints and the compatibility rules for integrity constraint locks.

If there is a dependency $T_i \leftarrow T_j$, then there are conflicting operations o_i of T_i and o_j of T_j, and both transactions must have acquired conflicting locks for these operations. Since the dependency $T_i \leftarrow T_j$ requires $o_i < o_j$, the lock rule and the legality rules allow only one possible order for the lock and unlock operations handling this conflict: the unlock operation of T_i must precede the lock operation of T_j, i.e. $unlock(T_i,X) < lock(T_j,X)$ [5].

Since each transaction T_i is two phased, it does all its lock operations before this unlock operation which occurs before the lock operation of T_j. This can be extended by induction to arbitrary long paths in the serialization graph, i.e. if there is a path $T_i \leftarrow T_j \leftarrow T_k \leftarrow \ldots \leftarrow T_h \leftarrow T_i$, then T_i must execute an unlock before T_j executes a lock ($unlock(T_i,X) < lock(T_j,X)$) and T_j executes this lock ($lock(T_j,X)$) before T_j executes an unlock conflicting with the lock operation of T_k ($unlock(T_j,Y) < lock(T_k,Y)$) ... and so on ... and this is before T_i executes a lock operation which conflicts with the unlock operation of T_h.

To summarize: T_i performs an unlock operation before it performs a lock operation, but this contradicts the assumption that all transactions are two phased. Therefore each history accepted by the lock protocol must have an acyclic serialization graph and must therefore be serializable.

5. SUMMARY AND CONCLUSIONS

We present a technique to guarantee global integrity constraints that cross the border of a single database system. The key idea, to use no read locks at all for integrity constraint checks on the underlying database systems, but to lock the integrity constraint itself, has the following advantages. It allows to perform global integrity checks parallel to all local transactions, i.e. it increases application parallelism. Furthermore, it allows a higher degree of concurrency of global integrity checks with global transactions. Finally, our lock protocol can be implemented in a very compact way (i.e., it needs only one single lock operation for each integrity constraint that a transaction has to check), and it is compatible with run time query optimization strategies proposed for integrity checks (e.g. [17,26]).

The presented lock protocol does not require the local database system to use a specific granularity of locks for read-write and write-write conflict synchronization. Note that the key idea of our protocol, to lock the integrity constraint itself, is independent of the lock granularity (e.g. object, page, or XML fragment) and compatible with arbitrary lock protocols obeying the underlying transaction model.

We have presented our contribution to global integrity control as an extension to two-phase locking schedulers, in order to keep the discussion

---

[5] $unlock(T_j,X) < lock(T_i,X)$ is not possible, because we could deduce $o_j < unlock(T_j,X) < lock(T_i,X) < o_i$ which contradicts $o_i < o_j$.

and the correctness proof simple. However, the idea of accessing the constraint itself does not depend on a specific synchronization strategy (i.e. two-phase locking). As long as the transactions obey the transaction model, i.e. make their changes visible to other transactions after commit, our protocol can be used in combination with other synchronization strategies too. For example, extending optimistic schedulers with our approach would result in the same distinction between integrity checks and queries, the same kind of conflict definitions, the same kind of optimizations and a similar extension to the correctness proof for optimistic schedulers. Therefore, the result seems to be applicable to optimistic schedulers too.

Furthermore, our lock protocol can be combined with local schedulers or global schedulers of multi-database systems. For example, an addition of our scheduler to [28] that itself is already an improvement of the ticket technique [13], would additionally allow the parallel execution of write transactions with global integrity checks, as long as different global integrity constraints are involved.

Finally, there seems to be a much broader spectrum of application areas, which may profit from our key idea, to lock the constraint itself instead of locking the data needed to check the constraint. For example, when a production plan that fulfills several "global" constraints is modified by parallel transactions, it seems to be advantageous too, to lock each constraint that has to be checked, instead of locking the data, which has to be read in order to check the constraint.

REFERENCES

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

[2] 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.

[3] R.K. Batra, D. Georgakopoulos, N. Rusinkiewicz: A Decentralized Deadlock-Free Concurrency Control Method for Multidatabase Transactions. Proc. 12th Int. Conf. on Distributed Systems, Yokohama, 1992.

[4] M. Benedikt, T. Griffin, L. Libkin: Verifiable Properties of Database Transactions. PoDS 1996.

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

[6] P. Bernstein, N. Goodman: Concurrency Control in Distributed Database Systems, Computing Surveys, 13 (2), 1981.

[7] P. Bernstein, V. Hadzilacos, N. Goodman: Concurrency Control and Recovery in Database Systems. Addison-Wesley, 1987.

[8] S. Böttcher: Improving the Concurrency of Integrity Checks and Write Operations. ICDT 1990.

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

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

[11] A. Deacon, H.-J. Schek, G. Weikum: Semantics-Based Multilevel Transaction Management in Federated Systems. ICDE 1994.

[12] W. Du, A.K. Elmagarmid: Quasi Serializability: a Correctness Criterion for Global Concurrency Control in InterBase, VLDB, Amsterdam, 1989.

[13] 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), 1994.

[14] J. Gray, A. Reuter: Transaction Processing, Concepts and Techniques, Morgan Kaufmann, 1993.

[15] A. Gupta, Y. Sagiv, J.D. Ullman, J. Widom: Constraint Checking with Partial Information. PODS 1994.

[16] A. Helal, Y. Kim, Marion Nodine, Ahmed K. Elmagarmid, Abdelsalam Heddaya: Transaction Optimization Techniques. In: Sushil Jajodia, Larry Kerschberg (Eds.): Advanced Transaction Models and Architectures. Kluwer 1997.

[17] A. Hsu, T. Imielinski: Integrity checking for multiple updates. ACM SIGMOD, 1985.

[18] N.B. Idris, W.A. Gray, R.F. Churchhouse: Providing Dynamic Security Control in a Federated Database. VLDB 1994.

[19] A. Kawaguchi, D.F. Lieuwen, Inderpal Singh Mumick, Dallan Quass, Kenneth A. Ross: Concurrency Control Theory for Deferred Materialized Views. ICDT 1997.

[20] Y.-S. Kim, A. Helal, A.K. Elmagarmid: Transaction Optimization. International Workshop on Advanced Transaction Models and Architectures, Goa, India , 1996.

[21] J. Klingemann, T. Tesch, J. Waesch: Semantics-Based Transaction Management for Cooperative Applications. International Workshop on Advanced Transaction Models and Architectures, Goa, India , 1996.

[22] F. Llirbat, E. Simon, D. Tombroff: Using Versions in Update Transactions : Application to Integrity Checking. VLDB 1997, Athens.

[23] S. Mehrotra, R. Rastogi, H.F. Korth, A. Silberschatz: The Concurrency Control Problem in Multidatabases: Characteristics and Solutions. ACM-SIGMOD Conf., San Diego, 1992.

[24] S. Mehrotra, R. Rastogi, H.F. Korth, A. Silberschatz: Ensuring Consistency in Multidatabases by Preserving Two-Level Serializability, ACM ToDS, 23, 2, 1998.

[25] J.M. Nicolas: Logic for Improving Integrity Checking in Relational Data Bases, Acta Informatica 18 (3), 1982.

[26] M.T. Özsu, P. Valduriez: *Principles of Distributed Database Systems*. 2nd Edition, Prentice Hall, 1999.

[27] 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.

[28] R. Schenkel, G. Weikum: Taming the Tiger: How to Cope with Real Database Product in Transactional Federations for Internet Applications. GI-Workshop Internet-Datenbanken 2000.

[29] 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.

[30] T. Sheard, D. Stemple: Automatic Verification of Database Transaction Safety. ACM ToDS 14 (3), 1989.

[31] E. Simon, P.Valduriez: Efficient Algorithms for Integrity Control in a Database Machine. NBS Conf. , Gaithersburg, 1984.

[32] E. Simon, P.Valduriez: Integrity Control in Distributed Database Systems. In Proc. 19th Hawaii Conf. On System Sciences, 1986.

[33] E. Simon, P.Valduriez: Design and Analysis of a Relational Integrity Subsystem. MCC Technical Report DB-015-87, 1987.

[34] P.J. Stuckey, S. Sudarshan: Compiling Query Constraints. PODS 1994.

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