Improving the Concurrency of Integrity Checks and Write Operations

Stefan Böttcher [1]
IBM Deutschland GmbH
Scientific Center
Institute for Knowledge Based Systems
P.O.Box 80 08 80
D-7000 Stuttgart 80
West Germany

Abstract

Transaction synchronization and integrity control have the goal to preserve correctness of the database. Transactions which intend to modify the database perform integrity checks, which can be considered as a specific kind of read operations. These integrity checks (like other read operations) have to be synchronized with write operations of concurrent transactions. Since integrity checks often access large parts of the database, the synchronization of integrity checks with write operations is a major bottle-neck of transaction synchronization. We show that the synchronization of integrity checks with write operations of concurrent transactions can be substantially improved so that it allows for more parallelism.

The key idea of the improvement is that the scheduler uses the knowledge of whether or not a read operation is used for integrity checking, and if so, then the scheduler allows for more parallelism with write operations of concurrent transactions.

The improvement presented achieves a higher transaction concurrency and can be combined with other integrity check optimization techniques. Furthermore, the improvement is adaptable to various synchronization techniques, e.g. physical and predicative locking and validation. A scheduler using the presented improvement for both predicative locking and predicative validation is implemented within the DBPL database system which was developed at the University of Frankfurt.

[1]This work has been partially done at the University of Frankfurt and has been supported by the Deutsche Forschungsgemeinschaft under Grant-No SCHM350/3-1.

1 Introduction

During the last decade optimization of transaction concurrency and optimization of integrity checking have been discussed independently of each other, e.g. [Bernstein *et al.*, 1987], [Papadimitriou, 1986] and [Simon and Valduriez, 1987], [Nicolas, 1982], [Bernstein and Blaustein, 1982]. While concurrency control strategies aim at higher parallelism of transactions, integrity control optimization reduces the number of integrity checks of a single transaction or their query complexity. Nevertheless, integrity checking still remains a major bottle-neck in database transaction processing.

In this paper, we argue that transaction concurrency can be substantially further improved, if the scheduler uses the knowledg of whether or not a query is used for integrity checking. For this purpose, we investigate for which pairs of write operations and integrity constraints synchronization is necessary and for which pairs it is not. We get the result that less synchronization is needed for the synchronization of integrity checks with write operations than for the synchronization of other read operations with write operations. This will be the basis on which we improve the synchronization of integrity checks with write operations. The presented optimization requires that the following basic assumptions hold:

1. Every transaction that violates an integrity constraint is aborted.
2. Before a transaction writes into the database, i.e. before it makes the effect of its write operations visible to other transactions, the transaction checks all its integrity constraints and performs all its other read operations.

The presented optimization for the synchronization of integrity checks and write operations can be combined with the integrity control optimization methods proposed e.g. in [Simon and Valduriez, 1987], [Nicolas, 1982] and [Bernstein and Blaustein, 1982]. Furthermore, the presented optimization is applicable to various synchronization strategies. For example, a scheduler using the presented optimization for two synchronization strategies, predicative locking [Eswaran *et al.*, 1976] and predicative validation [Reimer, 1983], is implemented in the DBPL database system [Böttcher *et al.*, 1986], [Böttcher, 1989]. Nevertheless, the presented optimization is not only applicable to predicative synchronization, but also to physical synchronization.

The rest of the paper is organized as follows. The next section presents some examples motivating why integrity checks and write operations have to be synchronized and why their synchronization should differ from the synchronization of other queries with write operations. The third section presents the optimization result, namely, for which pairs of operations consisting of an integrity check and a write operation the execution order is relevant and for which pairs of operations it is irrelevant. It further outlines how this optimization can be integrated into a scheduler based on two-phase locking.

2 Motivating examples

Section 2.1 presents an example in order to demonstrate that certain pairs of integrity checks and write operations need not be synchronized with each other, and therefore the synchronization of integrity checks with write operations can allow for more parallelism than the synchronization of other read operations with write operations. In section 2.2 we modify the example and show that other pairs of integrity checks and write operations have to be synchronized with each other.

2.1 A first example

The example contains two transactions, T1 and T2, writing on two relations, R1 and R2 respectively, and the following integrity constraint (all examples are written in the tuple relational calculus of DBPL [Eckhardt *et al.*, 1985], [Schmidt *et al.*, 1988], [Schmidt and Matthes, 1990]):

IC1 "For every element e1 in relation R1 and for every element e2 in relation R2 the attribute value a1 of e1 is different from the attribute value a2 of e2":

ALL e1 IN R1 ALL e2 IN R2 (e1.a1 $\neq$ e2.a2) .

Transaction T1 wants to insert a tuple t1 into the relation R1. Therefore T1 has to check whether the integrity constraint will hold for the new element t1. Hence, the integrity check can be simplified to:

IC1' ALL e2 IN R2 (t1.a1 $\neq$ e2.a2) .

Transaction T2 deletes an element from R2 after transaction T1 executed its integrity check. In this case, the execution order of the integrity check of T1 and the delete operation of T2 is irrelevant (w.r.t. serializability). We distinguish two cases in order to give the reason:

If the integrity check was successful, i.e. if transaction T1 evaluated the integrity check of IC1' to *true*, then the integrity check will remain *true* after transaction T2 deletes an element from R2. Hence, in this case the result of the integrity check is not influenced by the succeeding write operation. Therefore, the scheduler can allow for the write operation at any time after the integrity check. Furthermore, the scheduler can allow for the write operation at any time before the integrity check, because this corresponds with the execution order of both operations in an equivalent serial history (c.f. the proposition in section 3.2). Hence, in this case the execution order of both operations is irrelevant.

On the other hand, if the integrity check of transaction T1 is not successful, then T1 is aborted (this was our basic assumption 1 in the first section), and again, the

execution order of both operations is irrelevant (w.r.t. serializability). This is because serializability only depends on the order of operations of committed transactions (see e.g. [Bernstein *et al.*, 1987]). Hence, in both cases the order of the integrity check and the write operation is irrelevant.

Note however, that this is different if we have a boolean query and a write operation instead of an integrity check and a write operation. The execution order of a boolean query of transaction T1

BQ1 ALL e2 IN R2 (t1.a1 $\neq$ e2.a2)

and a succeeding delete operation of transaction T2 (deleting an element from R2) is relevant. The reason is as follows. The query result may be *false* before the delete operation is executed and may be changed to *true* by the delete operation. If no integrity constraint is violated, both transactions are committed. Therefore, the execution order of both operations is relevant, if transaction T1 submits a boolean query, which is not an integrity check.

To summarize: The execution order of both operations is relevant, if the query of transaction T1 is not an integrity check. However, the execution order of both operations is irrelevant, if the scheduler knows that the query of T1 is an integrity check. Thus, a scheduler can allow for more parallelism for the synchronization of integrity checks with write operations than for the synchronization of other (boolean) queries with write operations.

2.2 Modifying the first example

In the last example, a delete operation on relation R2 either changes the truth value of the integrity constraint (i.e. the result of the integrity check) from *false* to *true* or does not change the truth value of the integrity constraint. But this delete operation can never change the truth value of the integrity constraint from *true* to *false* (c.f. the lemma in section 3.1).

On the other hand, if we substitute the integrity constraint in the first example with a referential integrity constraint

IC2 "For every element e1 in relation R1 there exists an element e2 in relation R2 so that the attribute value a1 of e1 is equal to the attribute value a2 of e2":

ALL e1 IN R1 SOME e2 IN R2 (e1.a1 = e2.a2)

then a delete operation on R2 can never change the truth value of this integrity constraint from *false* to *true*. But the delete operation may change the truth value of the integrity constraint from *true* to *false*. That is the reason why an integrity check of the

referential constraint IC2 has to be synchronized with the delete operation on relation R2.

Note that the truth value of the integrity constraint IC1 can never be changed from *true* to *false* by a delete operation on R2, whereas the truth value of the integrity constraint IC2 can be changed from *true* to *false* by a delete operation on R2. That is the reason why the integrity check of constraint IC1 needs not be synchronized with the delete operation, whereas the integrity check of constraint IC2 has to be synchronized with the delete operation.

Now let us modify both examples in order to get two further examples. We replace the delete operation of transaction T2 with an insert operation of transaction T2 inserting an element into the relation R2.

In this case the insert operation either does not change the truth value of the integrity constraint IC1 or it changes the truth value of the integrity constraint IC1 from *true* to *false*. However, the insert operation can never change the truth value of the integrity constraint IC1 from *false* to *true* (c.f. the lemma in section 3.1).

Again, the truth value of the integrity constraint IC2 is modified vice versa by the insert operation. In this case the insert operation inserting an element into relation R2 either does not change the truth value of the integrity constraint IC2 or it changes the truth value of this integrity constraint from *false* to *true*. However, the insert operation can never change the truth value of the integrity constraint IC2 from *true* to *false*.

The following table summarizes how the truth value of both integrity constraints may be changed by succeeding insert or delete operations. (Remember: Transaction T1 has to perform the integrity check IC1', because it wants to insert a tuple into relation R1.[2] The important aspect here is whether or not an integrity check of one transaction has to be synchronized with a write of another transaction.)

Possible modifications of the integrity checks' truth values by succeeding write operations of other transactions on relation R2		
	delete operation	insert operation
constraint IC1	from *false* to *true*	from *true* to *false*
constraint IC2	from *true* to *false*	from *false* to *true*

Hence, whether or not the truth value of an integrity constraint can be changed from *true* to *false* by a succeeding write operation depends on both the quantification of the modified relation R2 in the integrity constraint (ALL or SOME quantification) and the

[2]Of course, if transaction T1 would perform a delete operation on R1 instead of an insert operation, it would not have to check any of the integrity constraints at all (c.f. e.g. [Simon and Valduriez, 1987]). This kind of optimization is also implemented in the DBPL database system [Böttcher, 1989], but it is not the topic of this paper.

kind of the write operation (insert or delete) modifying this relation (c.f. the lemma in section 3.1).

Whenever a succeeding write operation can change the truth value of an integrity check of a concurrent transaction from *true* to *false*, both operations have to be synchronized in order to prevent the write operation from violating the integrity check. However, both operations need not be synchronized, if the succeeding write operation can only change the truth value of the integrity check from *false* to *true*, because a transaction which evaluates an integrity check to *false* is aborted (according to basic assumption 1 outlined in the first section), and the execution order of operations of aborted transactions is irrelevant (e.g. [Bernstein *et al.*, 1987]). This is summarized in the following table.

The following modifications of the integrity checks' truth values by succeeding write operations on R2 have to be synchronized:		
	delete operation	insert operation
constraint IC1	—	from *true* to *false*
constraint IC2	from *true* to *false*	—
"—" means that in these cases synchronization is not necessary.		

However, this is different for other boolean queries and succeeding write operations. Other boolean queries have to be synchronized with write operations, because the transaction executing the query may be committed independent of the truth value of the boolean query.

To summarize: We have given four examples in order to show two things: First, integrity checks have to be synchronized with write operations. Second, the synchronization of integrity checks and write operations can allow for more parallelism than the synchronization of other boolean queries and write operations. Whether or not the synchronization optimization can be applied to an integrity check and a write operation depends on the kind of the write operation and on the quantification of the modified relation in the integrity constraint.

Traditional integrity check optimization methods (e.g. [Simon and Valduriez, 1987]) look only at single transactions and reduce their query complexity (e.g. by checking IC1' instead of IC1) or their number of integrity checks. In this paper, however, we describe how transaction concurrency can be improved (for the remaining integrity checks and write operations of concurrent transactions). Therefore, the suggested concurrency improvement can be combined with the traditional integrity check optimization methods.

3 The optimization

In order to generalize from the specific formulas and operations given in the examples above, section 3.1 presents a further differentiation of read and write operations. This will be the basis on which section 3.2 summarizes the generalized optimization approach. Furthermore, section 3.3 outlines how this generalized optimization is integrated into a scheduler based on two-phase locking.

3.1 Further differentiation of read and write operations

In this subsection, we characterize the difference between the two integrity constraints regarding to the modification of their truth values by succeeding write operations. We call the occurrence of the relation R2 within the integrity constraint IC2 *positive*, whereas the occurrence of R2 within the integrity constraint IC1 is called *negative*. More generally, depending on the quantification of a relation variable occurring in a formula f written in tuple relational calculus, we distinguish positive and negative occurrences of the relation variable.

Definition

> A *positive* occurrence of a relation variable R in a formula f written in tuple relational calculus is either a SOME quantified occurrence of R occurring in the scope of an even number of NOTs or an ALL quantified occurrence of R occurring in the scope of an odd number of NOTs. Similarly, a *negative* occurrence of a relation variable R in a formula f written in tuple relational calculus is either an ALL quantified occurrence of R occurring in the scope of an even number of NOTs or a SOME quantified occurrence of R occurring in the scope of an odd number of NOTs.

For example, the relation variable R2 occurs negative in integrity constraint IC1, but positive in constraint IC2, and it also occurs positive in the following constraint

IC3 ALL e1 IN R1 (NOT ALL e2 IN R2 (NOT (e1.a1 = e2.a2)))

which is logically equivalent to constraint IC2.

According to a relation R, we divide the read operations on R into the following three groups:

$ic^{R+}(f)$: integrity checks with R occurring only positive within the formula f
$ic^{R-}(f)$: integrity checks with R occurring only negative within the formula f
$r^{R*}(f)$: any other read operation with R occurring within the formula f

Similarly, we divide the write operations on R into the following three groups:

$w^{R:+}$: inserts into R
$w^{R:-}$: deletes from R
$w^{R:*}$: any other write operation on R

The following lemma formalizes the observation that a true formula f will remain true after the execution of an insert operation $w^{R:+}$ inserting elements into some relation R, if R occurs only positive in f. Similarly, a true formula f will remain true after the execution of a delete operation $w^{R:-}$ deleting elements from a relation R, if R occurs only negative in f. For a proof of the following lemma see e.g. [Simon and Valduriez, 1987].

Lemma

Let f_1 and f_2 be formulas and let $ic^{R+}(f_1)$ and $ic^{R-}(f_2)$ be integrity checks so that the relation R occurs only positive in f_1 and R occurs only negative in f_2. Further, let s be an arbitrary database state, $w^{R:+}$ a write operation inserting elements into R and $w^{R:-}$ a write operation deleting elements from R. Finally, let $w^{R:+}(s)$ (and $w^{R:-}(s)$ respectively) denote the database state reached after applying the operation $w^{R:+}$ ($w^{R:-}$) to the database state s, and let $I(ic(f),s)$ be the truth value received for the integrity check ic(f) submitted to the database state s. Then the following holds:

1. If $I(ic^{R+}(f_1),s) = true$, then $I(ic^{R+}(f_1),w^{R:+}(s)) = true$.
2. If $I(ic^{R-}(f_2),s) = true$, then $I(ic^{R-}(f_2),w^{R:-}(s)) = true$.

Whereas a number of researchers (e.g. [Simon and Valduriez, 1987], [Nicolas, 1982], [Bernstein and Blaustein, 1982]) have outlined that successful integrity checks can not be violated by special kinds of succeeding write operations, the important new aspect of this paper is to use this knowledge in order to improve the synchronization of integrity checks and write operations. We will discuss this aspect in the next subsection.

3.2 Generalization of the optimization idea

Serializability proofs for a history synchronized by two-phase locking or by validation (e.g. [Bernstein *et al.*, 1987], [Böttcher, 1989]) do construct an equivalent serial history as follows. Increasing transaction numbers are assigned to the transactions according to the time they execute a specific operation. For an arbitrary history produced by a scheduler based on two-phase locking or validation, we can choose the following operation of each transaction as specific operation in order to determine the transaction number. The specific operation of a transaction is the first write operation, if the transaction writes into the database, and the last operation of the transaction otherwise.

The serializability proofs furthermore show that the following holds for every history. If two operations o1 of transaction T1 and o2 of transaction T2 are in conflict (i.e. if their execution order is relevant), and both transactions commit, and the transaction number of T1 is less than the transaction number of T2, then o1 preceeds o2 in the given history.

Having shown this, the serializability proofs conclude that the equivalent serial history is the sequence of committed transactions ordered by their transaction numbers.

Therefore, the following proposition directly follows from the serializability proofs given e.g. in [Bernstein *et al.*, 1987]:

Proposition

> Assume that transaction numbers are assigned to the transactions as mentioned above. Then a history is serializable, if for all pairs of operations o1 of T1 and o2 of T2 the following holds: If the transaction number of T1 is less than the transaction number of T2, then o1 preceeds o2.

This proposition means that serializability can only be violated, if there is a pair of operations o1 of T1 and o2 of T2 so that the transaction number of T1 is less than the transaction number of T2 and o2 preceeds o1.

Furthermore, if o2 is a write operation and o1 is a read operation and o2 preceeds o1, then basic assumption 2 of the first section requires that the specific operation of T2 preceeds the specific operation of T1, and therefore the transaction number of T2 is less than the transaction number of T1. That is why you can interpret the proposition as:

> Serializability of a history can not be violated by such conflicts where a write operation preceeds a read operation.

We now outline the general idea of the optimization.

Let us look at an arbitrary history (produced by a scheduler based on two-phase locking or validation) and two operations of different transactions occurring in this history, where one operation is an integrity check ic^{R+} (and ic^{R-} respectively) and the other is a write operation $w^{R:+}$ ($w^{R:-}$). Then the basic optimization idea can be stated as follows:

> The serializability of the history is independent of the order of the integrity check and the write operation (see the theorem below). *Therefore, the integrity check and the write operation need not be synchronized with each other.*

Theorem

> The serializability of histories is independent of the order between ic^{R+} operations and $w^{R:+}$ operations of concurrent transactions and is independent of the order between ic^{R-} operations and $w^{R:-}$ operations of concurrent transactions.

The complete proof as given in [Böttcher, 1989] needs a lot of formal definitions. Instead, we give a sketch of the proof idea in the appendix.

The advantage of this approach is the following. Since pairs of write operations and integrity checks as mentioned above need not be synchronized, a scheduler can allow for more parallelism. For example, a scheduler based on two-phase locking needs not delay a transaction because of conflicts between these operations. Furthermore, a scheduler based on predicative validation needs not restart a transaction because of conflicts between these operations.

Since integrity checks often access a large part of the database, conflicts between integrity checks and write operations occur rather frequently. Furthermore, integrity checks are expensive (c.f. [Simon and Valduriez, 1987]). Therefore, the approach presented in this paper optimizes a major synchronization bottle-neck.

The following subsection describes how to apply this synchronization optimization to a scheduler based on two-phase locking. However, the optimization is applicable to other synchronization strategies as well, e.g. to predicative validation [Böttcher, 1989].

3.3 A lock protocol for improved synchronization

In this subsection we define a lock protocol which uses the advantages of the improved synchronization. It separates the synchronization of integrity checks with write operations from the synchronization of other read operations with write operations. A special instance of this lock protocol is implemented within the scheduler of the DBPL database system [Böttcher, 1989].

A history H is called *legal*, if every transaction uses the following (improved) lock protocol. Every transaction requires a sufficient lock on R before it performs an operation on R, and it releases the lock at the end of transaction.

The improved scheduler distinguishes six types of locks (ic^{R+}-lock, ic^{R-}-lock, r^{R*}-lock, $w^{R:+}$-lock, $w^{R:-}$-lock, $w^{R:*}$-lock) for the six types of operations (ic^{R+}, ic^{R-}, r^{R*}, $w^{R:+}$, $w^{R:-}$, $w^{R:*}$). Furthermore, any write lock on a part of the database is sufficient for any read operation on this part of the database, and a general write lock ($w^{R:*}$-lock) is sufficient for insert and delete operations, and a general read lock (r^{R*}-lock) is sufficient for any read operation or integrity check. Hence, the following table summarizes which locks are sufficient for which kinds of operations:

operation	each of the following locks is sufficient for the operation		
$ic^{R+}(f)$	ic^{R+}-lock , r^{R*}-lock ,	$w^{R:+}$-lock , $w^{R:-}$-lock , $w^{R:*}$-lock	
$ic^{R-}(f)$	ic^{R-}-lock , r^{R*}-lock ,	$w^{R:+}$-lock , $w^{R:-}$-lock , $w^{R:*}$-lock	
$r^{R*}(f)$	r^{R*}-lock ,	$w^{R:+}$-lock , $w^{R:-}$-lock , $w^{R:*}$-lock	
$w^{R:+}$		$w^{R:+}$-lock , $w^{R:*}$-lock	
$w^{R:-}$		$w^{R:-}$-lock , $w^{R:*}$-lock	
$w^{R:*}$		$w^{R:*}$-lock	

The scheduler only grants a lock required by a transaction, if no other transaction keeps an incompatible lock on an overlapping part of the database. The following lock compatibility table summarizes which pairs of locks on the same relation are compatible (comp.) and which are not compatible (not comp.).

	ic^{R+}-lock	ic^{R-}-lock	r^{R*}-lock	$w^{R:+}$-lock	$w^{R:-}$-lock	$w^{R:*}$-lock
ic^{R+}-lock	comp.	comp.	comp.	**comp.**	not comp.	not comp.
ic^{R-}-lock	comp.	comp.	comp.	not comp.	**comp.**	not comp.
r^{R*}-lock	comp.	comp.	comp.	not comp.	not comp.	not comp.
$w^{R:+}$-lock	**comp.**	not comp.	not comp.	not comp.	not comp.	not comp.
$w^{R:-}$-lock	not comp.	**comp.**	not comp.	not comp.	not comp.	not comp.
$w^{R:*}$-lock	not comp.	not comp.	not comp.	not comp.	not comp.	not comp.

The difference between this lock compatibility matrix and a lock compatibility matrix distinguishing only between read locks and write locks can be summarized as follows:

	ic^{R+}-lock	ic^{R-}-lock	r^{R*}-lock
$w^{R:+}$-lock	**comp.**	not comp.	not comp.
$w^{R:-}$-lock	not comp.	**comp.**	not comp.
$w^{R:*}$-lock	not comp.	not comp.	not comp.

	read-lock
write-lock	not comp.

While read locks and write locks on the same relation are not compatible in general, they are compatible in the special cases summarized in the left table. If the scheduler uses these kinds of locks instead of an ordinary lock protocol distinguishing only between read and write locks, then it can increase concurrency, because it allows for concurrency of ic^{R+} operations with $w^{R:+}$ operations and for concurrency of ic^{R-} operations with $w^{R:-}$ operations.

The following theorem states the correctness of this lock protocol which allows for increased concurrency.

Theorem

Every legal history H is serializable.

The full proof of this theorem is given in [Böttcher, 1989]. It is similar to other serializability proofs for histories of transactions synchronized by two-phase locking (e.g. in [Bernstein *et al.*, 1987]). The difference between this serializability proof and other serializability proofs (i.e. in [Bernstein *et al.*, 1987]) can be summarized as follows:

Proof outline:

Let Ti and Tj be two committed transactions in a given history, and let ic_j^{R+} be an integrity check of transaction Tj and $w_i^{R:+}$ a write operation of transaction Ti. On the one hand, we do not need to synchronize ic_j^{R+} operations with succeeding $w_i^{R:+}$ operations for the following reason. Since Tj commits, the integrity check yields the truth value *true* in the database state s where it is checked. From the lemma in section 3.1 we know that ic_j^{R+} operations can not be violated by a succeeding $w_i^{R:+}$ operation. Hence, we conclude that the ic_j^{R+} operation of transaction Tj needs not be synchronized with a succeeding $w_i^{R:+}$ operation of a concurrent transaction Ti. For the same reason, we do not need to synchronize ic_j^{R-} operations of Tj with succeeding $w_i^{R:-}$ operations of Ti.

On the other hand, if a write operation $w_i^{R:*}$ of Ti preceeds an integrity check (ic_j^{R+} or ic_j^{R-}) of Tj, then this corresponds to the transaction order of the equivalent serial history in which the transaction Ti preceeds the transaction Tj (c.f. the proposition in section 3.2). □

More details of the serializability proofs can be found in [Bernstein *et al.*, 1987] or [Böttcher, 1989].

4 Summary and Conclusion

Integrity checking is a major bottle-neck of transaction processing. We showed that the synchronization of integrity checks with write operations can be substantially improved, because the synchronization of integrity checks with write operations allows for more parallelism than the synchronization of other read operations with write operations. In order to increase parallelism, the scheduler has to use the knowledge of whether or not a read operation is used for an integrity check. The main result is that serializability can be achieved without synchronizing insert operations on a relation R with integrity checks in which R occurs only positive and without synchronizing delete operations on a relation R with integrity checks in which R occurs only negative.

The improvement presented has been combined with other optimizations which reduce the number or the query complexity of integrity checks [Böttcher, 1989].

We furthermore presented an extended lock protocol extending two-phase locking with the presented optimization. This lock protocol has been implemented within the predicative scheduler of the DBPL database system [Böttcher *et al.*, 1986], [Böttcher, 1989].

However, the optimization is not only applicable to two-phase locking, but also to other synchronization strategies, e.g. predicative validation.

Furthermore, the presented improvement is not restricted to relational database systems, but can also be applied to database systems supporting a more general data model, e.g. to database systems supporting attribute inheritance [Böttcher, 1990].

Since the synchronization of integrity checks with write operations of concurrent transactions is a major bottle-neck of transaction synchronization, the proposed optimization seems to be an important improvement of transaction synchronization.

Acknowledgement

I would like to thank J.W. Schmidt and the DBPL group who developed the programming language DBPL and the DBPL database system into which the described techniques could be easily integrated.

References

[Bernstein and Blaustein, 1982] P.A. Bernstein and B. Blaustein. Fast methods for testing quantified relational calculus assertions. In *Proceedings of the ACM SIGMOD International Conference*, 1982.

[Bernstein *et al.*, 1987] P.A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1987.

[Böttcher, 1989] S. Böttcher. *Prädikative Selektion als Grundlage für Transaktionssynchronisation und Datenintegrität.* PhD thesis, FB Informatik, Univ. Frankfurt, 1989.

[Böttcher, 1990] S. Böttcher. Attribute inheritance implemented on top of a relational database system. In M. Liu, editor, *Proc. 6th International Conference on Data Engineering*, Los Angeles, California, USA, 1990.

[Böttcher *et al.*, 1986] S. Böttcher, M. Jarke, and J.W. Schmidt. Adaptive predicate managers in database systems. In *Proceedings of the 12th International Conference on Very Large Data Bases*, Kyoto, Japan, 1986.

[Eckhardt *et al.*, 1985] H. Eckhardt, J. Edelmann, J. Koch, M. Mall, and J. W. Schmidt. *Draft Report on the Database Programming Language DBPL.* DBPL-Memo 091-85, Univ. Frankfurt, 1985.

[Eswaran *et al.*, 1976] K.P. Eswaran, J.N. Gray, R.A. Lorie, and I.L. Traiger. The notions of consistency and predicate locks in a database system. *Communications of the ACM*, 19(11), 1976.

[Nicolas, 1982] J.M. Nicolas. Logic for improving integrity checking in relational data bases. *Acta Informatica*, 18:227–253, 1982.

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

[Reimer, 1983] M. Reimer. Solving the phantom problem by predicative optimistic concurrency control. In *Proceedings of the 9th International Conference on Very Large Data Bases*, Firence, Italy, 1983.

[Schmidt and Matthes, 1990] J.W. Schmidt and F. Matthes. *DBPL Language and System Manual.* In. Document, Univ. Hamburg, 1990.

[Schmidt *et al.*, 1988] J.W. Schmidt, H. Eckhardt, and F. Matthes. *DBPL Report.* DBPL-Memo 111-88, Univ. Frankfurt, 1988.

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

Appendix

In this appendix, we sketch the idea of the proof of the theorem of section 3.2, since the complete proof as given in [Böttcher, 1989] needs a lot of formal definitions.

Theorem

> The serializability of histories is independent of the order between ic^{R+} operations and $w^{R:+}$ operations of concurrent transactions and independent of the order between ic^{R-} operations and $w^{R:-}$ operations of concurrent transactions.

Sketch of the proof idea:

> We show the proof idea for an integrity check ic^{R+} and a write operation $w^{R:+}$. The proof for pairs of ic^{R-} and $w^{R:-}$ operations can be given correspondingly.
>
> We distinguish two cases in order to show that the serializability of the history is independent of the order of the integrity check and the write operation.
>
> On the one hand, the basic assumption 2 outlined in the first section guarantees that the integrity check ic^{R+} of transaction T1 preceeds the specific operation so1 of transaction T1
>
> $ic^{R+} < so1$

and that the specific operation so2 of transaction T2 preceeds or is equal to the write operation $w^{R:+}$ of T2

$so2 \leq w^{R:+}$.

Hence, if the write operation $w^{R:+}$ of T2 preceeds the integrity check ic^{R+} of T1

$w^{R:+} < ic^{R+}$,

then we have

$so2 \leq w^{R:+} < ic^{R+} < so1$,

i.e. so2 (the specific operation of T2) preceeds so1 (the specific operation of T1). Therefore, the order of the write operation and the integrity check is as required by the construction of an equivalent serial history: the order corresponds to the transaction order of an equivalent serial history in which the transaction executing the write operation preceeds the transaction executing the integrity check. The proposition in section 3.2 states that this pair of operations can not violate serializability.

On the other hand, if the integrity check preceeds the write operation, we distinguish two cases: If the integrity check is successful, then the following holds: Since successful integrity checks of the type $ic^{R+}(f)$ can not be violated by succeeding $w^{R:+}$ operations, the execution order of $ic^{R+}(f)$ and succeeding $w^{R:+}$ operations is irrelevant. However, if the integrity check is not successful, then its transaction is aborted (basic assumption 1 of the first section guarantees this), and therefore the execution order of both operations is irrelevant (serializability only depends on the execution order of operations of committed transactions [Bernstein *et al.*, 1987]).

For the same reason the execution order of $ic^{R-}(f)$ and succeeding $w^{R:-}$ operations is irrelevant.

Since the execution order of both operations is either the order required by the corresponding equivalent serial history or the execution order of both operations is irrelevant, these operations need not be synchronized.

□