CONSTRAINT CHECKING FOR NON-BLOCKING TRANSACTION PROCESSING IN MOBILE AD-HOC NETWORKS

Sebastian Obermeier[1] and Stefan Böttcher[2]
[1]*ABB Corporate Research, Segelhofstr 1K, 5405 Baden Daettwil, Switzerland*
[2]*University of Paderborn, Fürstenallee 11, 33102 Paderborn, Germany*

Keywords: Mobile databases, Mobile transaction processing, Bi-state-termination.

Abstract: Whenever business transactions involve databases located on different mobile devices in a mobile ad-hoc network, transaction processing should guarantee the following: atomic commitment and isolation of distributed transactions and data consisteny across different mobile devices. However, a major problem of distributed atomic commit protocols in mobile network scenarios is infinite transaction blocking, which occurs when a local sub-transaction that has voted for commit cannot be completed due to the loss of commit messages and due to network partitioning. For such scenarios, Bi-State-Termination has been recently suggested to terminate pending and blocked transactions, which allows to overcome the infinite locking problem. However, if the data distributed on different mobile devices has to be consistent according to some local or global database consistency constraints, Bi-State-Termination has not been able to check for the validity of these consistency constraints on a database state involving the data of different mobile devices.
Within this paper, we extend the concept of Bi-State-Termination to arbitrary read operations. We show how to handle several types of database consistency constraints, and experimentally evaluate our constraint checker using the TPC-C benchmark.

1 INTRODUCTION

With increasing capabilities regarding processing power and connectivity of mobile devices and a growing interest in mobile ad-hoc networks, the combination of database technology with mobile devices becomes an interesting and important challenge. We consider an ad-hoc network of mobile devices, each of which equipped with a local database. Within this network, distributed transactions should be processed and executed in an atomic fashion.

1.1 Problem Description

Whenever distributed databases must be accessed and all of their operations must be executed or none of them, traditional transaction processing would employ atomic commit protocols (ACPs) like 2-Phase-Commit (2PC, (Gray, 1978)(Reddy and Kitsuregawa, 2003)), 3-Phase-Commit (3PC, (Skeen, 1981)), or consensus based protocols (Paxos Consensus, (Gray and Lamport, 2006)) to guarantee the atomicity of distributed transactions. However, during the execution of an atomic commit protocol, each device suffers from *transaction blocking*:

Definition 1. A *transaction* T is *blocked*, after a database proposed to execute T (e.g. by sending a voteCommit message) and waits for the final commit decision, but is not allowed to abort or commit T unilaterally on its own.

Note that transaction blocking summarizes the unilateral impossibility of a database to commit or abort a transaction, but does *not* mean that a transaction U waits to obtain locks held by a concurrent transaction T, since in this case U can be aborted by the database itself.

Even time-out based approaches (e.g. "my commit vote is valid until 3:23:34") cannot solve the problem of transaction blocking, since this would fulfill the requirements of the coordinated attack scenario (Gray, 1978), in which a commit decision is not possible under the assumption of message loss.

The problem of *infinite transaction blocking* for a transaction T occurs, if a local database has sent a voteCommit message on T, but will never receive the final commit decision, e.g. due to disconnection, movement, or network partitioning. Whenever such a

Obermeier S. and Böttcher S. (2010).
CONSTRAINT CHECKING FOR NON-BLOCKING TRANSACTION PROCESSING IN MOBILE AD-HOC NETWORKS.
In *Proceedings of the 12th International Conference on Enterprise Information Systems - Databases and Information Systems Integration*, pages 166-175
DOI: 10.5220/0002889301660175

database is able to communicate, e.g. with the user, the blocked data of T will prevent concurrent and conflicting transactions U from being processed.

Bi-State-Termination (Obermeier and Böttcher, 2007) has been suggested to overcome the infinite transaction blocking problem by obeying both outcomes – commit *and* abort – of the distributed transaction. Note that Bi-State-Termination is optional for pending transactions; the database can choose for each transaction whether to wait as yet, or to use Bi-State-Termination, which is a promising approach in case the pending transaction modifies only a small set of data tuples.

However, when consistency constraints are imposed to the database, traditional consistency checks cannot be applied as the current database state is unclear.

1.2 Contributions

This paper extends the technique of Bi-State-Termination (BST) (Obermeier and Böttcher, 2007), a transaction termination mechanism for mobile transactions in unreliable environments that solves the infinite transaction blocking problem. Beyond the previous publication on BST (Obermeier and Böttcher, 2007), this paper further

- extends the BST concept to define and check consistency constraints.
- describes how insert, update, delete, commit and abort operations and arbitrary relational algebra queries as set union, projection, cartesian product, and set difference can be performed on the BST implementation by introducing a query rewrite system.
- shows experimental results comparing the performance of different implementations of consistency checks using a modified TPC-C benchmark.

2 BST MODEL

Bi-State-Termination is based on the following observation: whenever *transaction blocking* occurs, the database does not know whether a transaction T waiting for the commit decision will be aborted or committed. However, only if the transaction is committed, the database state changes. Let S_0 denote the database state before T was executed. Although the database does not know the commit decision for T, it knows that depending on the commitment of T, either S_0 or $S_T := T(S_0)$, i.e. the state reached when T is applied to S_0, is the correct database state. With this knowledge, the database can try to execute a concurrent conflicting transaction U on both states S_0 and S_T. Whenever the two executions of U on S_0 and on S_T return the same results to the Initiator, i.e. $\mathsf{Result}_U(S_0) = \mathsf{Result}_U(S_T)$, U can be committed regardless of T, even though they are conflicting. Otherwise it is the application's choice whether it handles two possible transaction results.

2.1 Bi-state-termination

Let $\Sigma = \{S_0, \ldots, S_k\}$ be the set of all legal possible database states for a database D. A *traditional transaction* T is a function $T : \Sigma \mapsto \Sigma$, $S_a \rightarrow S_b$, which means the resulting state S_b of T depends only on the state S_a on which T is executed on.

A *Bi-State-Terminated transaction* T is a function $BST : 2^{\Sigma} \mapsto 2^{\Sigma}$,

$$\underbrace{\{S_i, \ldots, S_j\}}_{\text{Initial States}} \rightarrow \underbrace{\{S_i, \ldots, S_j\}}_{T \text{ aborts}} \cup \underbrace{\{T(S_i), \ldots, T(S_j)\}}_{T \text{ commits}}$$

that maps a set $\Sigma_{\text{Initial}} \subseteq \Sigma$ of Initial States to a super set $\Sigma_{\text{Initial}} \cup \{T(S_x) | S_x \in \Sigma_{\text{Initial}}\}$ of new states, where $T(S_x)$ is the state that is reached when T is applied to S_x.

3 BST REWRITE RULES

Whether or not some tuples finally belong to a database relation depends on *conditions*, i.e. on the commit or abort decisions of pending transactions. In contrast to (Obermeier and Böttcher, 2007), our new BST rewrite system extends each database relation with a single extra column *Conditions* to store these conditions, and rewrites all queries, write operations, integrity constraint checks and data definition statements in such a way that they reflect these conditions. Furthermore, we add a single table *Rules* to the database that contains rules that relate these conditions to each other. BST is implemented by the following rewrite rules.

3.1 Creating Database Tables

The "create table" command for database relations is modified by the following rewrite rule:

```
create table R ( <column definitions> )
  ⇒ create table R'( <column definitions, string conditions>)
```

3.2 Status without Active Transactions

When all transactions are completed either by commit or by abort, the column "Conditions" contains the truth value "true" for each tuple in each relation of the database. The truth value "true" represents the fact that the tuple belongs to the relation without any further condition on the commit status of an active transaction.

3.3 Write Operations on the BST Model

Each time a write operation must be executed, it is rewritten according to the following rules.

3.3.1 Insertion

Whenever a tuple $t = (\text{value}_1, \ldots, \text{value}_N)$ is inserted into a relation R by a transaction with transaction identifier T_i, we implement this by inserting $t' = (\text{value}_1, \ldots, \text{value}_N, T_i)$ into the relation R', i.e. the database system implementation applies a rewrite rule:

insert into R values (value$_1$,...,value$_N$)
$\Rightarrow$ insert into R' values (value$_1$,...,value$_N$, T_i).

The idea behind the value T_i stored in the condition column of R' is to show that the tuple t belongs to the database relation R if and only if transaction T_i will be committed.

3.3.2 Deletion

Whenever a tuple $t = (\text{value}_1, \ldots, \text{value}_N)$ is deleted from a relation R by a transaction with transaction identifier T_i, we look up the tuple $t' = (\text{value}_1, \ldots, \text{value}_N, C) \in R'$ representing the tuple $t \in R$, where C is the condition under which t belongs to the database relation R.

We implement the deletion of t from R by the transaction T_i by replacing the condition C found in t' with a condition C_2 and by adding a logical rule to the table *Rules* stating that C_2 is true if and only if C is true and T_i is aborted. For this purpose, the database system applies the following rewrite rule, where $A_1, \ldots, A_N$ denote the values $(\text{value}_1, \ldots, \text{value}_N)$ for the attributes of R:

delete t from R where t.A$_1$=value$_1$,...,t.A$_N$=value$_N$
$\Rightarrow$ update t' in R' where t.A$_1$=value$_1$,...,t.A$_N$=value$_N$ set condition=C$_2$;
insert into Rules values (C$_2$, C$_1$ and not T_i)

The idea behind this rewriting is the following. (not T_i) represents the condition that transaction T_i will be aborted. The inserted rule states that C_2 is true if C_1 is true and T_i will be aborted. After the update operation, we have a tuple $t' = (\text{value}_1, \ldots, \text{value}_N, C_2)$ in R' which represents the fact that t belongs to R if and only if C_2 is true, i.e. if C_1 is true and T_i is aborted.

3.3.3 Update

An update of a single tuple is simply executed as a delete operation followed by an insert operation.

3.3.4 Set-oriented Write Operations

When a transaction inserts, updates, or deletes multiple tuples within a single operation, this can be implemented by a set of individual insert, update, or delete operations.

3.3.5 Completion of a Transaction

When transaction T_i is completed with commit, the condition T_i is replaced with true in each rule in the Rules table and in each value found in the column "Conditions" of a relation R'. However, when T_i is completed with abort, T_i is replaced with *false* in each rule found in the Rules table, and each tuple of R' containing the value T_i in the column "Conditions" is deleted.

Furthermore, rules that contain the truth value *true* or *false* are simplified. Whenever this results in a rule (C, *true*) or in a rule (C, *false*), then C itself is replaced with the value "true" or "false" respectively. Other rules that contain C are simplified as well. Furthermore, all tuples t' in which C occurs are treated as follows. If the rule is (C, *true*), the value C is replaced with true in each tuple t' in which C occurs in the column "Conditions". However, if the rule is (C, *false*), each tuple t' in which C occurs in the column "Conditions" is deleted. Finally, rules (C, *true*) or (C, *false*) are deleted from the relation *Rules*.

Example 1. Consider a database that executes the following transactions on a relation R that only consists of the attribute "Name" in the sequence $T_1 < T_2 < T_3$:

T_1: insert "Miller"
T_2: delete "Mitch"
T_3: change "M" to "R" in each name

Line 1 of Table 1 represents the initial database state S_0 of R', lines 2-3 show the content of R' after BST of T_1, lines 4-5 represent the table content after BST of T_1 and T_2, while lines 6-9 show the table after BST of T_1, T_2, and T_3. The conditions C_i in the column "Condition" of Table 1 are linked to the "Rules" column of Table 2. Table 2 defines for each condition

Table 1: Content after Bi-State-Terminating T_1, T_2, and T_3.

Line	Name	Condition	Comment
1	Mitch		Initial
2	Mitch		Content after BST of T_1
3	Miller	C_1	
4	Mitch	C_2	Content after BST of T_1, T_2
5	Miller	C_1	
6	Mitch	C_3	Content after BST of T_1, T_2, T_3
7	Miller	C_4	
8	Ritch	C_5	
9	Riller	C_6	

C_i by a boolean formula composed of other conditions and/or elementary conditions T_j, $\overline{T_k}$, where T_j in the column "Definition" of Table 2 represents that transaction T_j will commit and $\overline{T_k}$ represents that T_k will abort. When C_i is valid, the row (<t>, C_i) of Table 1 represents that the tuple <t> is in R. The condition C_4, for example, is fulfilled when T_1 commits and T_3 aborts. In this case, line 7 of Table 1 becomes valid.

Table 2: Rules Table after Bi-State-Terminating T_1, T_2, and T_3.

Condition	Definition	Comment
–	–	Initial
C_1	T_1	Content after BST of T_1
C_1	T_1	Content after BST of T_1, T_2
C_2	$\overline{T_2}$	
C_3	$\overline{T_2}\overline{T_3}$	Content after BST of T_1, T_2, T_3
C_4	$T_1\overline{T_3}$	
C_5	$\overline{T_2}T_3$	
C_6	T_1T_3	

3.4 Read Operations on the BST Model

Whenever a read operation on R is implemented by a read operation on R', the conditions are kept as part of the result. The relational algebra operations are implemented as follows.

3.4.1 Selection

Each selection with selection condition SC that a query applies to a relation R, will be applied to R', i.e. the database system applies the following rewrite rule to each selection:

SC(R) ⇒ SC(R')

3.4.2 Duplicate Elimination

Duplicate elimination is an operation that is used to implement projection and union. When duplicates occur, their conditions are combined with the logical OR operator. That is, given the relation R' contains two tuples $t'_1 = (\text{value}_1, \ldots, \text{value}_N, C_1)$ and $t'_2 = (\text{value}_1, \ldots, \text{value}_N, C_2)$ these two tuples are deleted and a single tuple $t' = (\text{value}_1, \ldots, \text{value}_N, C_{C12})$ is inserted into R, and a rule (C_{C12}, C_1 or C_2) is inserted into the Rules table.

3.4.3 Set Union

Set union of two relations R_1 and R_2 is implemented by applying duplicate elimination to the set union of R'_1 and R'_2. The database system applies the following rewrite rule:

$R_1 \cup R_2 \Rightarrow$ removeDuplicates($R_1' \cup R_2'$)

3.4.4 Projection

Projection of a relation R_1 on its attributes $A_1, \ldots, A_N$ is implemented by applying duplicate elimination to the result of applying the projection to R'_1 including the column "Conditions". The database system applies the following rewrite rule:

P($A_1, \ldots, A_n$) (R_1) ⇒ removeDuplicates(P($A_1, \ldots, A_n$, conditions) (R'_1))

3.4.5 Cartesian Product

Whenever the cartesian product $R_1 \times R_2$ of two relations R_1 and R_2 must be computed, this is implemented using R'_1 and R'_2 as follows. For each pair (t'_1, t'_2) of tuples $t'_1 = (\text{value}_1, \ldots, \text{value}_N, C_1)$ of R'_1 and $t'_1 = (\text{value2}_1, \ldots, \text{value2}_N, C_2)$ of R'_2, a tuple $t'_{12} = (\text{value}_1, \ldots, \text{value}_N, \text{value2}_1, \ldots, \text{value2}_N, C_{C12})$ is constructed and stored in $(R_1 \times R_2)'$. The database system applies the following rewrite rule:

$R_1 \times R_2 \Rightarrow (R_1 \times R_2)'$
where $(R_1 \times R_2)'$ can be derived by computing the set
$\{ (t_1, t_2, C_{C12}) \mid (t_1, C_1) \in R_1' \text{ and } (t_2, C_2) \in R_2' \}$
and by adding a rule (C_{C12}, C_1 and C_2) for each pair of C_1 and C_2 to the Rules table.

3.4.6 Set Difference

Whenever the set difference $R_1 - R_2$ of two relations R_1 and R_2 must be computed, this is implemented using R'_1 and R'_2 as follows. The set difference contains all tuples $t'_1 = (\text{value}_1, \ldots, \text{value}_N, C_1)$ of R'_1 for which no tuple $t'_2 = (\text{value2}_1, \ldots, \text{value2}_N, C_2)$ of R'_2 exists, and furthermore, it contains a tuple

$t'_{12} = (\text{value}_1, \ldots, \text{value}_N, C_{C12})$ for each tuple $t'_1 = (\text{value}_1, \ldots, \text{value}_N, C_1)$ of R'_1 for which a tuple $t'_2 = (\text{value2}_1, \ldots, \text{value2}_N, C_2)$, $C_2 \neq C_1$, of R'_2 exists. The condition C_{C12} is *true* if and only if (C_1 and not C_2) is *true*. The database system applies the following rewrite rule:

R_1 - $R_2 \Rightarrow R_1'$ - R_2'
where (R_1' - R_2') can be derived by computing the union of the following sets S_1 and S_2:
S_1 = { (t_1,C_1) | exists (t_1,C_1) $\in R_1'$ and not exists C_2 such that (t_1,C_2)$\in R_2'$ }
S_2 = { (t_1,C_{C34}) | exists (t_1,C_3)$\in R_1'$ and exists (t_1, C_4)$\in R_2'$ such that $C_3 \neq C_4$}
and by adding a rule (C_{C34}, C_3 and not C_4) for each pair of C_3 and C_4 used in S_2 to the Rules table.

3.4.7 Other Algebra Operations

Other operations of the relational algebra like join, intersection, etc. can be constructed by combining the implementation of the basic operations. Query optimization of operations like join etc. is also possible.

4 DATABASE CONSTRAINTS

We explain how the following types of database constraints are checked for a transaction T_{check} on a database that uses Bi-State-Termination. We assume that the database is in consistent state before T_{check} has been executed, thus only the effects of T_{check} can violate the database's consistency.

4.1 Domain Constraints

Domain constraints restrict attribute values to a given set. These constraints can be tested on each tuple individually.

Example 2. *A flight booking can only reserve a positive, integer number of seats.*

4.1.1 BST Check

Whenever a database contains Bi-State-Terminated transactions, the following steps are required in order to check that each given domain constraint D_c holds:

D_c is checked for only those tuples that have been inserted by T_{check}, i.e. tuples that are associated with a condition C_i that contains the string T_{check} in its definition. Tuples that are going to be deleted do not need to be checked since they already exist in the (consistent) database, thus their validity regarding D_c has been checked before.

4.2 Referential Integrity

Let R and S be database realtions, and let R_i be a referential integrity constraint R_i of the following form.

$$R_i := \forall x \in R\ \exists y \in S : (x.a_1 = y.a_1 \wedge \ldots \wedge x.a_n = y.a_n)$$

Then, our algorithm first eliminates all duplicates by using the removeDuplicates() function explained in Section 3.4.2 on a projection of the attributes $a_1, \ldots, a_n$ and the condition attribute C.:

$$\begin{aligned} R' &:= \text{removeDuplicates}(\Pi_{a_1,\ldots,a_n,C}(R)) \\ S' &:= \text{removeDuplicates}(\Pi_{a_1,\ldots,a_n,C}(S)) \end{aligned}$$

Thereafter, the following two sets are computed:

$$\begin{aligned} \text{RICheck}_1 := \quad & \{(C_1) \,|\, (t_1,C_1) \in R' \wedge (t_1,C_2) \notin S' \\ & \wedge\ \nexists (t_1,C_2) \in S\} \\ \text{RICheck}_2 := \quad & \{(C_1 \wedge \overline{C_2}) \,|\, (t_1,C_1) \in R' \wedge (t_2,C_2) \in S' \\ & \wedge t_1.a_1 = t_2.a_1 \wedge \ldots \wedge t_1.a_n = t_2.a_n\} \end{aligned}$$

Both sets RICheck_1 and RICheck_2 describe conditions that are associated with tuples that violate the referential integrity constraint if their conditions become true. RICheck_1 consists of all conditions of tuples that may be present in R', but have no reference in S'. RICheck_2 identifies tuples t_1 of R that are referencing tuples t_2 in S that contain a condition C_2. In this case, when t_1 becomes present in R and t_2 becomes invalid in S, the referential integrity constraint would be violated. Thus, RICheck_2 is composed of conditions that would violate the referential integrity when they would be fulfilled.

After both sets have been computed, the union of both set is checked for satisfiability:

$$\begin{aligned} & \exists c \in \{\text{RICheck}_1 \cup\ \text{RICheck}_2\} : c \text{ is satisfiable} \\ & \Leftrightarrow \text{check failed} \end{aligned}$$

When at least one condition in $\{\text{RICheck}_1 \cup\ \text{RICheck}_2\}$ is satisfiable, the referential consistency constraint can be violated. Thus, the check must fail. Otherwise, when the set does not contain any condition that is satisfiable, the referential consistency cannot be violated by T_{check}.

4.3 Functional Dependencies

Let R denote a relation and α and β be sets of attributes in R. Further, let $F_i : \alpha \Rightarrow \beta$ be a functional dependency, i.e.,

$$F_i := \forall t_1, t_2 \in R : (t_1[\alpha] = t_2[\alpha] \Rightarrow t_1[\beta] = t_2[\beta])$$

In order to check F_i, we first eliminate all duplicates by using the removeDuplicates() function:

$$R' := \text{removeDuplicates}(\Pi_{\alpha,\beta,C}(R))$$

Further, let FDCheck denote the following set:

$$\text{FDCheck} := \{(C_1 \wedge C_2) \mid (t_1, C_1) \in R' \wedge (t_2, C_2) \in R' \wedge t_1[\alpha] = t_2[\alpha] \wedge t_1[\beta] \neq t_2[\beta]\}$$

The set FDCheck comprises of conditions for tuples that may violate the functional dependency when their conditions become true, i.e. tuples that have the same value for their α attributes but different values for their β attributes.

Whenever a condition exists in FDCheck that may be satisfiable, the functional dependency may get violated. Thus, the check of functional dependencies must fail:

$$\exists c \in \{\text{FDCheck}\} : c \text{ is satisfiable} \Leftrightarrow \text{check failed}$$

4.4 Multiple Tuple Constraints

A different type of constraints are *multiple tuple constraints* (MTC) that apply to a set of tuples. Thus, an MTC check comprises several or all tuples of a relation and cannot be checked individually for each data tuple instead. We focus on a constraint that defines a limit on the sum of attribute values of a given attribute of a relation. These types of constraints are practically relevant, e.g. for specifying a maximum amount of seats, a limit on bank account transfers, or a maximum number of costs.

4.4.1 BST Check

As with other integrity constraints, we allow a transaction T_{check} that has inserted or deleted tuples from a relation R for which an MTC exists to vote for commit only if we can guarantee that the MTC evaluates to true independent of the commit decisions of the Bi-State-terminated transactions. A principal way to test this is the following check based on combinations of transaction decision and on combinations of tuples violating a constraint.

All combinations of transaction IDs that occur in the Conditions definition of R are computed, and all possible combinations of transaction decisions "commit" and "abort" of these transactions are created. This number grows exponentially in the number of Bi-State-Terminated transactions. Each combination of tuples that violates the MTC is checked for satisfiability of the combination of conditions. Whenever each tuple combination violating the constraint has an insatisfiable combination of transaction decisions, the MTC cannot be violated. Otherwise, T_{check} must be aborted.

As the number of combinations and checks grows exponentially, we present two optimizations for the MTC check.

4.4.2 Bounded MTC Check

We check a worst case approximation of the MTC, called *Bounded MTC Check* instead of checking MTC itself on all combinations of commit decisions of Bi-State terminated transactions. Whenever an MTC has the form $\sum \text{attribute}_1 < x$, i.e. the sum of all attribute values must not be greater than or equal to x, the bounded MTC check optimizes the check as follows: Only those values v of the attribute_1 are summed up that are greater than 0, regardless of their associated condition.

Whenever the sum exceeds x, there may be a violation of the MTC and the check fails. This check corresponds to the worst-case scenario for the inequality constraint, where all transactions that add negative values abort, and all transactions that add positive values commit. If even this worst-case scenario does not violate the inequality constraint, the transaction results in a valid database state and the voteCommit can be sent.

Whenever an MTC has the form $\sum \text{attribute}_2 > x$, i.e. the sum of all attribute values must not be smaller than or equal to x, the bounded MTC check optimizes the check as follows: Only those values v of the attribute_2 are summed up that are smaller than 0, regardless of their associated condition. Whenever the sum is equal to or less than x, there may be a violation of the MTC and the check fails. This check corresponds to the worst-case scenario for the inequality constraint, where all transactions that add positive values abort, and all transactions that add negative values commit.

However, the bounded MTC check can lead to false positives as a failure of the bounded MTC check includes impossible combinations of values and commit decisions of transaction that can violate the bounded MTC.

Therefore, this check is extremely fast (only one summation), but may produce unnecessary aborts.

4.4.3 Optimized MTC Check

The optimized MTC check combines the bounded MTC check with the regular check. For all values except the n greatest values, the sum is calculated according to the bounded MTC check. Then, $n+1$ combinations of the greatest values and the bounded MTC sum are evaluated regarding their satisfiability.

5 EXPERIMENTAL EVALUATION

We have evaluated the runtime for the consistency checks explained in Section 4 depending on the number of Bi-State-Terminated transactions. We have used a modified version of the online transaction processing benchmark TPC-C (Kohler et al., 1991), which contains additional consistency constraints. The TPC-C simulates an online-shop-like environment in which users execute order transactions against a database. The transactions additionally include recording payments, checking the status of orders, and monitoring the level of stock at the warehouses. Each TPC-C run consists of 1,000 transactions. In order to simulate Bi-State-Termination with n pending transactions, we delayed the outcome of the atomic commit protocol for n Bi-State-Terminated transactions, such that the n transactions are Bi-State-Terminated. During the simulation, the number of Bi-State-Terminated transactions remains constant, but the transactions itself vary. For our measurements, we have repeated each simulation run 10 times and measured the time that was needed for each consistency check.

Furthermore, we have implemented two versions, called *internal* and *external* implementation, of the consistency checking algorithms. The external implementation uses a separate definition table where the conditions under which each row becomes valid are stored.

The internal implementation does not use a separate condition definition table anymore. Instead, it directly adds the condition's definition to each data row. Thus, conditions need not be derived from the separate Conditions table, instead each tuple contains its conditions within the "Condition" column of the relation. Thus, Rules table lookups to derive the conditions under which a tuple becomes valid become superfluous in the internal implementation. This speeds up write operations that operate on many tuples for the following reason: The database does not need to generate and associate unique Condition IDs to replaced conditions, instead it can update the "Condition" column in one pass by concatenating its value with the transaction ID.

Figure 1 shows the experimental results for domain constraints. The $x-$axis indicates whether the test run was done with the internal implementation ("Int"), or with the external implementation ("Ext"). Furthermore, the number n of Bi-Sate-Terminated transactions is shown under each box. The $y-$axis shows the running time in msec. As the time for a single consistency check depends on the transaction, the consistency definition, and the database state, we have

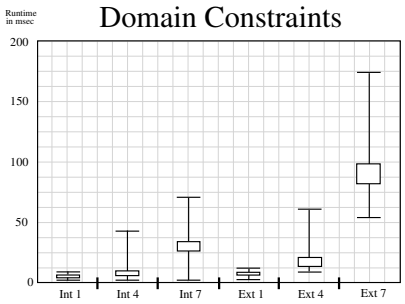

Figure 1: Domain constraint evaluation.

repeated our experiments 10 times and have used box plots to display the range of the runtimes by boxes. Each box consists of two horizontal lines indicating the mininum and maximum value. These horizontal lines are connected by a centered vertical line to the *box*. The box comprises 50% of all values, in detail it consists of all values within the second and the third quartile. Thus, 25% of all values are smaller and greater than the box, respectively.

A box plot gives an impression of how the values are spread. For our TPC-C measurement, this allows us to compare the runtimes for the internal and external implementation, and the absolute runtime ranges for the different types of consistency checks.

The box plot of Figure 1 shows that the average runtime of the domain constraint check is significantly greater for the external implementation than for the internal implementation. Furthermore, the runtime increases with the number of Bi-State-Terminated transactions.

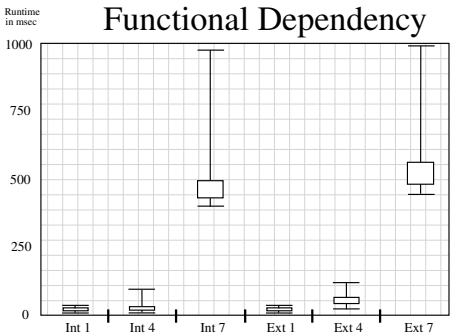

Figure 2: Functional dependency evaluation.

Figure 2 illustrates the runtimes for the functional dependency checks. For this kind of consistency check, the external implementation needs only slightly more runtime than the internal implementation. However, the runtime of the functional dependency check increases significantly with the number of Bi-State-Terminated transactions, as the check con-

tains more value comparisons than the domain constraint check.

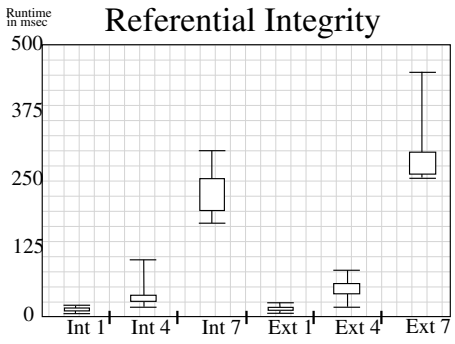

Figure 3: Referential integrity evaluation.

The referential integrity check experiments are visualized by Figure 3. Again, the internal implementation is slightly faster. Although the overall runtime for this kind of test is smaller than for the functional dependency check, the growth of runtime motivates a restriction of the number of Bi-State-Terminated transaction by the database. Instead of Bi-State-Terminating a blocked transaction, the database can wait and block resources as traditional transaction processing does.

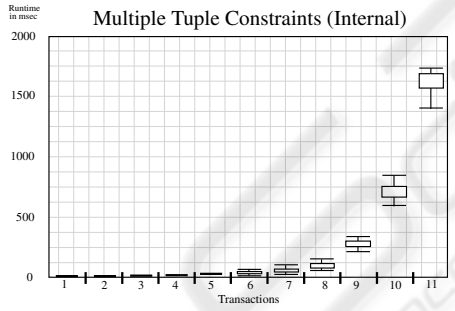

Figure 4: MTC evaluation – internal implementation.

Figure 4 shows the results for the internal implementation of the MTC checks. As the external implementation shows almost the same behavior, we have omitted to show these graphs. However, we have extended the number of Bi-State-Terminated transactions to 11, in order to show the exponential growth of runtime. However, our boxes indicate nicely that the time ranges in which the runtimes for each check fall are quite small. This allows the database to get quite exact estimations on the runtime of the consistency checks, and thus allows using Bi-State-Termination of a transaction as an option that can be chosen depending on the current database load situation.

Figure 5 uses the same setup as the MTC experiments shown by Figure 4, but uses the bounded MTC

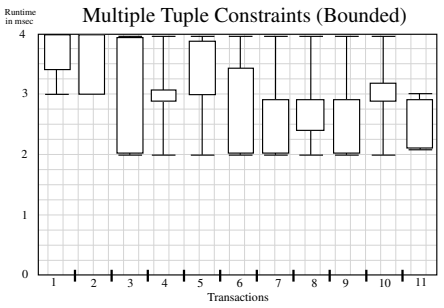

Figure 5: MTC evaluation – bounded optimization.

check instead. The runtime for this check is – in theory – linear in the number of Bi-State-Terminated transactions. However, as the bounded MTC check requires only one additional operation per Bi-State-Terminated transaction, our experiments show an almost constant runtime behavior.

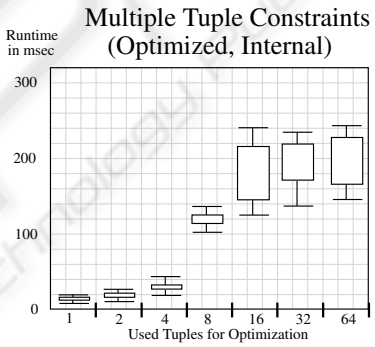

Figure 6: Optimized MTC evaluation for 8 Bi-state-terminated transactions.

The optimized MTC check shown in Figure 6 shows results for 8 Bi-State-Terminated transactions. The number m of used tuples for the optimization corresponds to the number of the m greatest tuple values. These m tuples are combined as in the regular MTC check, while the remaining tuples are summed up as in the bounded MTC check. Thus, the optimized MTC check combines the accuracy of the regular MTC check with the improved performance of the bounded MTC check.

In order to compare the quality of both optimizations, the bounded MTC check and the optimized MTC check, we have additionally measured the number of unnecessary aborts caused by the optimization. These results are shown in Figure 7. The gray parts of the bars show the amount of transactions where the MTC was violated. As we can see, all MTC checks work correct as they recognize this inconsistency. However, the extremely fast bounded MTC check classifies the outcome of 20% of the transac-

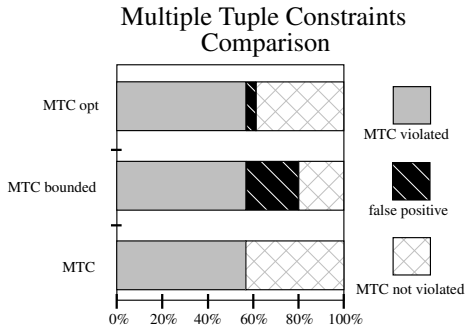

Figure 7: MTC optimization comparison.

tions that do not violate the MTC as inconsistent. As this results in an unnecessary abort, the bounded MTC check should only be used in situations in which the check must be very fast. However, the optimized MTC check using the eight largest tuples caused only 5% of unnecessary aborts by reducing the runtime compared to the regular MTC check by 50%. Thus, we consider the optimized MTC check as a good tradeoff between time and accuracy.

6 RELATED WORK

Our proposed solution relates to four ideas that are used in different contexts: Bi-State-Termination (Obermeier and Böttcher, 2007), Escrow locks (Gray and Reuter, 1993), and Speculative locking (Reddy and Kitsuregawa, 2004), multiversion databases (Katz, 1990), (Cellary and Jomier, 1990), (Chen et al., 1996).

Bi-State-Termination has been recently proposed in (Obermeier and Böttcher, 2007), but only for insert, update, and delete operations. We extend this concept to be able to deal with any relational expression including union, projection, cartesian product, and set difference. Furthermore, our approach allows the definition of database constraints like domain constraints, functional dependency constraints, referential integrity constraints, and multiple tuple constraints. We have experimentally evaluated several consistency constraints and have proposed an optimization to multiple tuple constraint checks.

Escrow locks are mainly used in environments with high transaction load within certain data hotspots. Instead of locking an entire data item, the escrow lock calculates an interval $[i, j]$ for each attribute a that is updated by a transaction T_1, which corresponds to an upper and lower bound of the attribute. Whenever a concurrent transaction T_2 checks a precondition P for a, this check is evaluated on the interval $[i, j]$ instead of trying to evaluate P on the data item a that is locked by T_1. In comparison the escrow locking technique, Bi-State-Termination does neither rely on numerical values, nor does it assume that an attribute may lie in an interval. Furthermore, all relational expressions can be evaluated by BST.

Speculative Locking (SL) (Reddy and Kitsuregawa, 2004) is another related technology. SL was proposed to speed up transaction processing in environments with high message delivery times by spawning multiple parallel executions of a transaction that waits for the acquisition of required locks. Like BST, SL also allows the access to after-images of a transaction U while U is still waiting for its commit decision. However, SL does not allow the commit of T before the final commit decision for U has been received. This means, SL cannot successfully terminate T while the commit vote for U is missing, which is possible with BST.

Multiversion database systems (Katz, 1990), (Cellary and Jomier, 1990), (Chen et al., 1996) are used to support different expressions of a data object and used for CAD modeling, and versioning systems. However, compared to BST, multiversion database systems allow multiple versions to be concurrently valid, while BST allows only one valid version, but lacks the knowledge which of the multiple versions is valid due to the atomic commit protocol. Furthermore, multiversion database systems are mostly central embedded databases that are not designed to deal with distributed transactions. Instead, the user specifies on which version he wants to work.

Other approaches rely on compensation of transactions. (Kumar et al., 2002), for instance, proposes a timeout-based protocol especially for mobile networks, which requires a compensation of transactions. However, inconsistencies may occur when some databases do not immediately receive the compensation decision or when the coordination process fails.

7 SUMMARY AND CONCLUSIONS

A major challenge for the integration of mobile databases into transaction processing is to guarantee atomic commit and isolation of distributed transactions and global database consistency, even in the case of communication failures. We have presented and extended Bi-State-Termination, a technique to handle the case that a participant does not receive the coordinator's commit decision for a long period of time. Bi-State-Termination allows continuing the processing of transactions U, even in the case that conflicting

transactions T that hold locks on resources required by U are blocked, by obeying both possible outcomes of the blocked transactions T. This allows transactions U to commit even if they conflict with pending transactions T.

We have focused on database table definitions, on arbitrary read- and write-operations, and we proposed a rewrite rule system that allows all these kinds of operations to be executed on a database that uses Bi-State-Termination. Furthermore, we presented a technique for checking typical integrity constraints even in situations where a database using Bi-State-Termination is not sure about its current state as pending transactions may commit or abort. Our proposed technique for consistency constraint checking allows checking whether or not a transaction may violate given constraints. The experimental evaluation has shown the feasibility of our constraints checker and has proposed an optimization for checking time consuming multiple tuples constraints.

Altogether, the constraint checking technique proposed in this paper is feasible and efficient, and it can be done in combination with Bi-State-Termination in mobile databases, even in case of network failures. This is why we consider the constraint checking technique as a very important addition to Bi-State-Termination, which is useful to integrate mobile databases into business transactions.

REFERENCES

Cellary, W. and Jomier, G. (1990). Consistency of versions in object-oriented databases. In McLeod, D., Sacks-Davis, R., and Schek, H.-J., editors, *16th International Conference on Very Large Data Bases, August 13-16, 1990, Brisbane, Queensland, Australia, Proceedings*, pages 432–441. Morgan Kaufmann.

Chen, I.-M. A., Markowitz, V. M., Letovsky, S., Li, P., and Fasman, K. H. (1996). Version management for scientific databases. In Apers, P. M. G., Bouzeghoub, M., and Gardarin, G., editors, *Advances in Database Technology - EDBT'96, 5th International Conference on Extending Database Technology, Avignon, France, March 25-29, 1996, Proceedings*, volume 1057 of *Lecture Notes in Computer Science*, pages 289–303. Springer.

Gray, J. (1978). Notes on data base operating systems. In Flynn, M. J., Gray, J., Jones, A. K., et al., editors, *Advanced Course: Operating Systems*, volume 60 of *Lecture Notes in Computer Science*, pages 393–481.

Gray, J. and Lamport, L. (2006). Consensus on transaction commit. *ACM Trans. Database Syst.*, 31(1):133–160.

Gray, J. and Reuter, A. (1993). *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann.

Katz, R. H. (1990). Toward a unified framework for version modeling in engineering databases. *ACM Comput. Surv.*, 22(4):375–409.

Kohler, W., Shah, A., and Raab, F. (1991). Overview of TPC Benchmark C: The Order-Entry Benchmark. Technical report, http://www.tpc.org, Transaction Processing Performance Council.

Kumar, V., Prabhu, N., Dunham, M. H., and Seydim, A. Y. (2002). Tcot-a timeout-based mobile transaction commitment protocol. *IEEE Trans. Com.*, 51(10):1212–1218.

Obermeier, S. and Böttcher, S. (2007). Avoiding infinite blocking of mobile transactions. In *Proceedings of the 11th International Database Engineering & Applications Symposium (IDEAS), Banff, Canada*.

Reddy, P. K. and Kitsuregawa, M. (2003). Reducing the blocking in two-phase commit with backup sites. *Inf. Process. Lett.*, 86(1):39–47.

Reddy, P. K. and Kitsuregawa, M. (2004). Speculative locking protocols to improve performance for distributed database systems. *IEEE Transactions on Knowledge and Data Engineering*, 16(2):154–169.

Skeen, D. (1981). Nonblocking commit protocols. In Lien, Y. E., editor, *Proceedings of the 1981 ACM SIGMOD International Conference on Management of Data, Ann Arbor, Michigan*, pages 133–142. ACM Press.