Using Versions in Update Transactions: Application to Integrity Checking

Francois Llirbat
INRIA, France
Francois.llirbat@inria.fr

Eric Simon
INRIA, France
Eric.simon@inria.fr

Dimitri Tombroff
Chorus Systems. France
dimi@chorus.fr

Abstract

This paper proposes an extension of the multiversion two phase locking protocol, called EMV2PL, which enables update transactions to use versions while guaranteeing the serializability of all transactions. The use of the protocol is restricted to transactions, called *write-then-read* transactions that consist of two consecutive parts: a write part containing both read and write operations in some arbitrary order, and an abusively called read part, containing read operations or write operations on data items already locked in the write part of the transaction. With EMV2PL, read operations in the read part use versions and read locks acquired in the write part can be released just before entering the read part. We prove the correctness of our protocol, and show that its implementation requires very few changes to classical implementations of MV2PL. After presenting various methods used by application developers to implement integrity checking, we show how EMV2PL can be effectively used to optimize the processing of update transactions that perform integrity checks. Finally, performance studies show the benefits of our protocol compared to a (strict) two phase locking protocol.

1 Introduction

Constraint checking is a key issue of many modern applications, which is acknowledged by recent evolutions of the SQL standard to accommodate a larger class of assertions (SQL2) and triggers (SQL3 [Mel93]). To verify integrity constraints, update transactions may have to perform many additional read operations.

**Proceedings of the 23rd VLDB Conference
Athens, Greece, 1997**

Update transactions that terminate by issuing many read operations expose to specific performance problems that are illustrated bellow.

Example 1.1 Consider a stock-market application where a broker is responsible for ordering shares for her client's portfolios. The following relations are used:

```
Order (o_key, portfolio_id, share_name, stockmarket,
       price)
MaxRisk(stockmarket, value)
MinRisk(stockmarket, value)
```

Share orders are registered in the `Order` relation. The `MaxRisk` and `MinRisk` relations respectively give for each stock-market the minimal and maximal risk that the broker should take to make benefits. These relations are frequently updated by transactions that analyse the activity of the various stock-markets. The application enforces the `Risk` constraint which forbids any insertion of a share order if the corresponding risk is not contained in the risk threshold interval indicated by relations `MaxRisk` and `MinRisk`.

Take a transaction program, *entry_order*(p) that inserts tuples into relation `Order` for a given portfolio p. To enforce the `Risk` constraint the transaction has to perform additional read operations before committing. Checking the `Risk` constraint requires to read for each new inserted share order the corresponding items in relations `MaxRisk` and `MinRisk`.

Suppose that an *entry_order*(p) transaction runs in isolation degree 3 and obeys the strict two phase locking policy (S2PL) [BHG87]. Whenever *entry_order* executes, the reads incurred by constraint checking may be conflicting with concurrent executions of update transactions on relations `MaxRisk`, `MinRisk`. When a conflict occurs between two transactions, one transaction is blocked and waits that the other one releases its locks (when committing or aborting). Thus, running instances of *entry_order* augment the lock contention and impede the transactional traffic in the database system. This situation may lead to performance thrashing as showed in [Tho91].

Our main contribution is to propose an extension of the multiversion two phase locking (MV2PL) protocol, called EMV2PL, which enables update transactions to use versions and guarantees the serializability of all transactions. The use of our protocol is restricted to a particular class of update transactions called *write-then-read* transactions

(henceforth noted W|R transactions). A transaction in this class consists of two consecutive parts: the first part (called the write part) contains both read and write operations in some arbitrary order, and the second part (abusively called the read part) only contains read operations or write operations on data items already written in the first part of the transaction. In our protocol, the W part of a W|R transaction T is executed under the S2PL protocol. Then, when T reaches the end of its W part, T gets a *Lockpoint_Timestamp* and releases all its read locks. From that point, T does not take locks anymore and a *read(x)* operation accesses the most recent committed version of x that precedes T's *Lockpoint_Timestamp*.

Example 1.2 Let `T1` be an *entry_order* transaction that inserts a share order o on the stock-market m such that $risk(o) = 35$, and checks the *Risk* constraint. Let $t = (m, 20)$ (resp. $u = (m, 30)$) be the tuple of `MinRisk` (resp. `MaxRisk`) that contains the minimal (resp. maximal) advised risk for the stock-market m. Suppose that a transaction $T2$ increments the *Value* attribute of tuples t and u by 20. Then executing $T1$ and $T2$ concurrently under EMV2PL may yield the following history that would not have been accepted by the S2PL protocol :

```
H1: W1(o) R1(t) W2(t) W2(u) c(T2) R1(u) c(T1)
```

$T1$ gets a *Lockpoint_Timestamp* $ts1$ after inserting o and before reading tuples t and u, while $T2$ gets a *Lockpoint_Timestamp* $ts2$ after writing u. Thus, since $ts1 < ts2$, $T1$ does not see the version of u created by $T2$ and reads a risk interval of $[20, 30]$. Hence, $T1$ is serialized before $T2$ and the `Risk` constraint violation is detected. □

Our protocol increases concurrency by allowing W|R transactions to take advantage of versions in two ways: (i) they release their read locks before executing their read part, and (ii) they execute their read part without taking any lock, as read-only transactions do with MV2PL. All the read operations of the read part are performed using the same lockpoint-timestamp and thus guarantees that no phantom occurs during the execution of the read part. Moreover, like S2PL, EMV2PL uses a "pessimistic" approach to concurrency control which allowed us to fairly compare its performance with an S2PL protocol, and demonstrate that EMV2PL brings a significant increase in concurrency and reduces the probability of deadlocks. Finally, a notable feature of our protocol is its simplicity, as attest the few modifications to a classical MV2PL implementation that are required to implement it(see [LST97]). We consider it as a virtue since our intention in this research is not to invent yet another new concurrency control protocol but rather to enhance existing implementations to better match application needs.

A second important contribution of this paper is to show how EMV2PL can be effectively used to optimize an application's transaction throughput when update transactions execute triggers or check integrity constraints. We examine various methods, procedural and declarative, used by application developers, to implement integrity checking in transactions and analyse the consequence of each method on the pattern of transactions. Finally, we propose tuning rules that indicate how to design transactions with integrity checking under EMV2PL to achieve the better performance throughput.

1.1 Paper Outline

The remaining of the paper is structured as follows. Section 2 formally defines the EMV2PL protocol, proves its correctness and briefly explains how it can be implemented. Section 3 discusses the potential of EMV2PL to optimize transactions that perform integrity checking. Section 4 presents our performance study and provides simple tuning rules. for designing transactions with constraint checking under EMV2PL. Section 5 presents related work, and Section 6 concludes the paper.

2 Extended Multiversion Two Phase Lock Protocol

In this section we formally define the EMV2PL protocol and prove its correctness, we explain its behavior with respect to deadlocks and external consistency.

2.1 The EMV2PL Protocol

We now present the EMV2PL protocol. First, R and W transactions are processed as with MV2PL. An R transaction first obtains a *start number* noted sn from TM. Then every read(x) gets the most recent version of x having a timestamp less than or equal to sn. Reads in a W transaction follow the usual S2PL protocol, whereas a write(x) creates a new version of x (if x is written for the first time). Before committing, a W transaction obtains its *transaction number* (noted tn), associates this number to each of its versions, and releases all its locks.

Figure 1 shows how the operations issued by a W|R transactions are processed. The write part of the transaction is processed as a W transaction. When the end of the write part is reached, the transaction signals it reached a *lockpoint*[1] to the TM and receives a transaction number tn. The transaction then releases all the S locks it has acquired so far. After that point, read and write operations are processed as follows. A read(x) operation invokes a function *check_read*(x) that checks if there is an uncommitted version[2] of x created by another transaction whose number is smaller than the caller's tn. In that case, *check_read* waits until that transaction commits. After that, the W|R transaction reads the most recent version of x with timestamp smaller than or equal to tn. A write(x) operation only modifies a version already created in the write part. Before committing, a W|R transaction associates its tn to each version it created and releases all its locks.

To maintain the tn's, the TM uses a monotonically increasing counter. Since W and W|R transactions obtain their tn after they acquired their last locks and before committing, tn's are lockpoints. For R transactions, the TM simply guarantees that their sn is smaller than the tn of

[1] A lock point of a transaction is any point in time between the last lock acquired and the first lock released

[2] i.e., a version created by a still active transaction

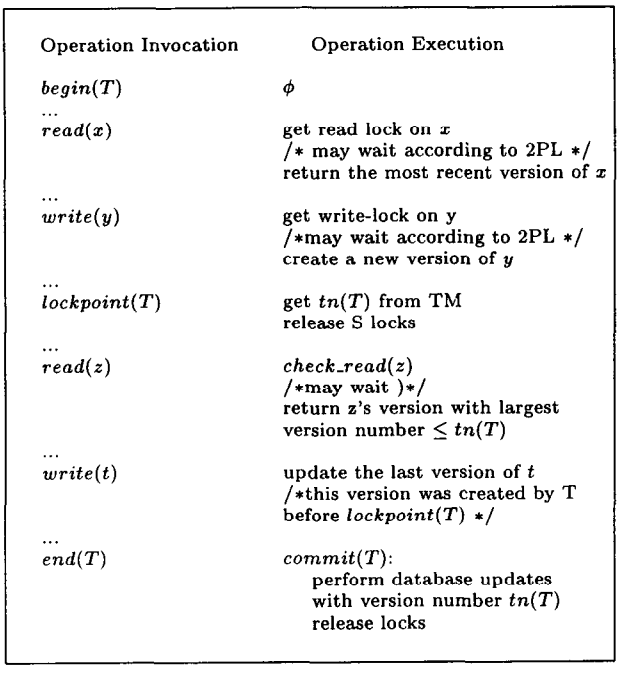

Figure 1: Execution of W|R transaction

any active or forthcoming transaction. Thus, an R transaction reads only versions of committed transactions.

Theorem : The EMV2PL protocol guarantees serializability of all transactions (see the proof in [LST97]).

2.2 Deadlocks

Clearly, EMV2PL suffers from deadlocks since it uses S2PL for serializing W transactions and the write parts of W|R transactions. However, once a W|R transaction starts executing its read part, it may not be involved anymore in deadlocks (see the proof in [LST97]).

2.3 External Consistency

Although EMV2PL guarantees serializability, it does not preserve *external consistency*. That is, the order in which transactions commit may differ from their serialization order as shows the following example:

Example 2.1 Let T1 be a W|R *entry_order* transaction that inserts a share order o on the stock-market m such that $risk(o) = 35$ and checks the `Risk` constraint in its read part. Let $t = (m, 20)$ (resp. $u = (m, 30)$) be the tuple of `MinRisk` (resp. `MaxRisk`) that contains the minimal (resp. maximal) advised risk for the stock-market m. Suppose a transaction $T2$ increments the *Value* attribute of tuple u by 10. Then executing $T1$ and $T2$ concurrently under EMV2PL may yield the following history:

```
H1:  W1(o)  R1(t)  W2(u)  c(T2)  R1(u)  Abort(T1)
```

Since $T1$ reaches its lockpoint before $T2$ then it is serialized before $T2$ by EMV2PL and thus $T1$ does not see that transaction $T2$ made (and committed) a larger risk interval and that the order could have been accepted. □

	procedural	declarative
R|W	check-before-write	none
W|R	write-then-check	deferred mode: assertions and safe RCA triggers
(W|R)*	- immediate checking - others	- immediate mode: assertions and triggers - others

Table 1: patterns of transactions and integrity checking methods

Such consistency "faults" are likely to occur if the read part W|R transactions are long. Should external consistency be critical, it may help to show the value of transaction lockpoint-timestamps to users, instead of showing the transaction commit time, since these timestamps reflect the serialization order. Intuitively, the lockpoint-timestamp of a W|R transaction indicates at which time the decision to commit or abort was taken (even though the system committed or aborted the transactions at some later time).

3 Application to Integrity Checking

We showed through the examples of section 1 that EMV2PL allows to avoid constraint anomalies. However, the applicability of EMV2PL suffer from the following limitations: (i)the transactions must be write-then-read transactions and, (ii) the lockpoint (i.e., the end of the write part) must be detected. In this section, we discuss the applicability of EMV2PL to constraint checking. We consider different methods for programming integrity checks and analyze consequences of each method on the pattern of transactions. All the resulting patterns are summarized in table 1. Moreover, We show how the lockpoint can be automatically detected (i.e., without knowing the transactions in advance) in the case of deferred declarative triggers and assertions.

Procedural Approach: The vast majority of database applications implement integrity constraints using a procedural approach whereby integrity checks are embedded into application programs. We distinguish three classes of integrity checking methods:

- the *write-then-check* method consists in checking constraints at the end of transaction. Such method is sometimes mandatory because some temporary inconsistent state is unavoidable during the execution of the transaction, or the interactive effects between two or more updates have to be controlled afterwards, or the integrity checks depend on the logic of the transaction program (especially when some conditional branching is used).

- the *check-before-write* method consists in checking constraints at the beginning of transactions. This method is expected to bring the following advantages: (i) exclusive locks on the updated data items are held for a shorter time if the updates occur at the end of the transaction, and (ii) less work is possibly wasted when

the transaction violates data integrity since there are no unnecessary writes.

- the *immediate checking* method consists in checking constraints just before or just after update operations. Such a method allows to provide intermediate consistent states during the execution of the transaction.

All the resulting patterns are shown in Table 1. Among these methods, only the *write-then-check* method yields W|R transactions. Hence, provided that the programmer has the ability to manually insert a lockpoint (e.g., using a specific command in the transaction program) at the end of the write part, a transaction's execution can take advantage of our EMV2PL protocol.

Declarative Assertions: Declarative assertions include two forms of constraints: *check constraints*[3] and *referential constraints*. Referential constraints and check constraints can be checked either immediately (*immediate* mode) after an SQL statement or at the end of the transaction (*deferred* mode). The execution of immediate referential and check constraints yields (W|R)* transactions. The execution of deferred referential and check constraints yields W|R transactions provided that the referential constraints with "cascade" or "set null" action are executed first. In this case, a lockpoint can be dynamically detected by the system (i.e., without knowing in advance the transactions) when all the remaining constraints are with " no action".

Declarative Triggers: Another way to enforce integrity constraints is to use triggers. Triggers can be executed either immediately before or after the triggering SQL statement (*immediate* mode) or at the end of the transaction (*deferred* mode). But unlike deferred constraints, deferred triggers do not necessarily yield W|R transactions since the action of triggers can perform database updates. More precisely, the problem is the following: given a transaction ready to commit and a set of deferred triggers activated by the transaction, how can the database system statically detect a transaction lockpoint[4]?

A simple method that can be used by the rule manager to detect a lockpoint when triggers execute at the end of transactions, consists in detecting a specific class of triggers called *RCA safe* triggers. An *RCA trigger* is a trigger whose action part does not acquire new exclusive locks. *RCA* stands for *Rollback, Compensative, Alerter* trigger. Indeed, the action part of triggers that do not acquire new exclusive lock typically (i) performs a rollback, (ii) overwrites database items that have been already inserted, updated or deleted by the triggering transaction or (iii) raises an alert. An *RCA* trigger is *safe* if it cannot transitively trigger a non-RCA trigger. When the rule manager receives the "end-of-transaction" signal from a transaction, the S set of triggers that have been activated is computed. Then, the rule manager recursively selects a trigger r from S, executes r and recomputes S. The rule manager signals the lockpoint to the TM when all the triggers in S are RCA and safe.

Remark about *RCA* and *safe* triggers: Let us note that *RCA safe* triggers can be detected at the time triggers are defined. The *safe* property can be determined using a *triggering action graph* (*TAG*), as defined in [Mel93]. The *RCA* property may require a complicated code analysis (except for evident cases as alerters or rollback triggers), however.

[3]SQL-92 distinguishes *table check constraints* and *assertions*: A table check constraint is attached to one table and is used to express a condition that must be true for every tuple in the table. An assertion is a stand-alone check constraint in a schema and is normally used to specify a condition that affects more than one table.

[4]In fact, the problem is more complicated because check constraints, referential constraints, and triggers can be mixed together. However, considering the general framework requires to have a precise description of an execution model for deferred triggers and assertions, a still open problem which is largely out of the scope of this paper.

4 Performance Study

In this section we evaluate the performance of S2PL and EMV2PL under various transaction workloads in a centralized database. To compare the relative benefits of S2PL and EMV2PL for various transaction patterns and in a wide range of operating conditions, we have implemented a simulation model [5]. In our experiments, we consider the case of workloads where some initial W transactions are lengthened by the execution of additional read operations (e.g., implied by the execution of decision support procedures or constraint checking) producing either R|W, W|R or (W|R)* transaction patterns (see table 1).

Moreover, as we will see, performance analysis of different workloads and operating situations authorizes a kind of feedback tuning approach which given a set of W transactions and a set of additional reads operations, suggests a suitable transaction pattern (R|W, W|R or (W|R)*) to profitably execute these new read operations under EMV2PL. In particular, our tuning guide is useful in the context of constraint checking in order to select a performant constraint checking method between the procedural check-before-write or write-then-check, and the declarative immediate or deferred methods.

4.1 The Simulation Model

Our simulation model is strongly derived from [BC92] and [SLSV95]. It has two parts: the *system model* simulates the behavior of the various operating system and DBMS components, while the *application model* simulates the database items and the transactional workload.

4.1.1 The System Model

In our simulation, we model the concurrent execution of transactions on a single site database. To keep the simulator simple, we simulate page-level locking. This allows us not to simulate indexes and index locking, and transactions access records randomly. The system model is divided into four main components: a Transaction Manager (TM), a Concurrency Control Manager (CCM), a Data Manager (DM) and a Log Manager (LM). The TM is responsible for issuing concurrency control requests and

[5]Note that our simulation model only reflects the relative benefits and costs but not the exact numbers.

name	Description	Value
buf_size	Nb of pages in the buffer pool	600
k	Resource Unit (kCPUs and 2k Disks)	1, 2, 3
rec_cpu	CPU time for accessing a record	1 ms
p_io	I/O time for accessing a page	7 ms
log_io	time for issuing a I/O log access	7 ms
log_io_w	I/O time for sequentially writing 1 page on log disk	1 ms
com_cpu	cpu time for executing a commit	1 ms
ab_cpu	cpu time for executing an abort	1 ms
restart	restart delay of an aborted transaction	5 ms
cpu_cc	cpu time for servicing one cc request	1 ms

Table 2: System Parameters Definitions and Values

name	Description	Value
num_rec	Nb of records in the database	150000
nb_rec_p	Nb of cached records per page	3
mpl	Nb of terminals	1 to 70
U_size	mean size of Update part of tx	30
p_write_U	fraction of write in the Update part of tx	25%
RperW	Nb of additional reads per write in the Update part of tx	1 to 13
P_Roll	Probability that RperW read operations generate a rollback	0 to 10
P_type	pattern of the extended tx	W|R, R|W, (W|R)*

Table 3: Workload Parameters Definitions

their corresponding database operations. It also assures the durability property by flushing all log records of committed transactions to durable memory. The CCM schedules the concurrency control requests according to either the S2PL or EMV2PL protocol. The LM provides read and insert-flush interfaces to the log table. The DM is responsible for granting access to the physical data objects and executing the database operations.

The DM encapsulates the details of a LRU Buffer Manager. The number of pages in the buffer cache is *buf_size*. These pages are shared by the main segment and the version pool. When a dirty version pool is chosen for replacement by the LRU algorithm, the DM first checks if it contains needed versions. If not (i.e., if it contains only obsolete versions), the page is considered non-dirty and simply discarded. Otherwise, it is written on disk. We have chosen to simulate the on-page version caching technique [BC92] because it is one of the most efficient technique for maintaining version and processing transactions: first, versions are maintained for records (instead of pages), second, a small portion of each page is used for caching previous versions of records. As a result, readers may find the adequate version without performing any additional I/Os. Also, these versions may sometimes be eliminated while still in the page and thus have not to be appended into the version pool at all.

The physical queuing model consists in k resource units, each containing one CPU server and two I/O servers. The requests to the CPU queue and I/O queues are serviced FCFS (first come, first serve). Parameter *rec_cpu* is the amount of CPU time for accessing a record in a page. Parameter *p_io* is the amount of I/O time associated with accessing a data page from the disk. We added one separate I/O server dedicated to the log file. The parameter *log_io* represents the fixed I/O time overhead associated with issuing the I/O. Parameter *log_io_w* is the amount of I/O time associated with writing a log record on the Log disk in sequential order. Parameter *com_cpu* is the amount of CPU time associated with executing the commit (releasing locks, etc). Parameter *ab_cpu* is the amount of CPU time associated with executing the abort statement (executing undo operations, releasing locks etc). Table 2 summarizes the parameters of the system model and their values for the experiments.

4.1.2 The Application Model

The database contains *num_rec* records. With S2PL, 18 records fit in one page (this corresponds to pages of 8K containing records of 454 bytes). With EMV2PL the records are assumed to contain an additional 50 bytes to store the timestamp and version pointer. As a result, only 15 records fit in one page, whose *nb_rec_p* records are used to cache previous versions.

A transaction workload contains transactions that consist of an *update part* that is extented with additional read operations. There are *mpl* terminals executing transactions. Parameter *U_size* is the average number of operations executed by the update part of each transaction (without the additional reads). Among these operations, *p_write_U* are write operations. Moreover, a transaction executes *RperW* additional read operations per write operations occurring in its update part. Parameter *P_type* represents the obtained pattern of transactions. They can be R|W, W|R or (W|R)* transactions. In a R|W (resp. W|R) transaction then all the additional read operations are executed at the beginning (resp. at the end) of the transaction. If the transaction is (W|R)* then *RperW* read operations are executed "on the fly" just after each write operation. We also vary the probability *P_Roll* of executing a rollback after *RperW* additional reads. When additional read operations consist of integrity checks, this enables to simulate the detection of an integrity violation that leads to reject the transaction. Let us recall that, in our experiments, workloads only contain transactions of the same pattern. All the parameters are summarized in Table 3 (where "transaction" is abbreviated "tx").

Regarding the measurements, each simulation consisted of 3 to 5 repetitions, each consisting of 2000 seconds of simulation time. These numbers were chosen in order to achieve more than 90 percent confidence intervals for our results.

4.2 Experiment 1 : Benefits of EMV2PL

The goal of this experiment is to show the value of EMV2PL for applications containing W|R transactions. The workload contains only W|R transactions (*P_type* = W|R). In Figure 11(a), we vary the multi-programming level *mpl* from 1 to 70. *RperW* is fixed to 6. We measure the throughput (number of transactions per second) of concurrent W|R transactions running under S2PL (curve

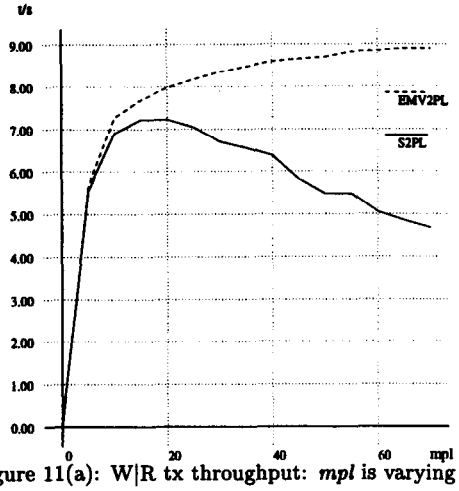

Figure 11(a): W|R tx throughput: *mpl* is varying

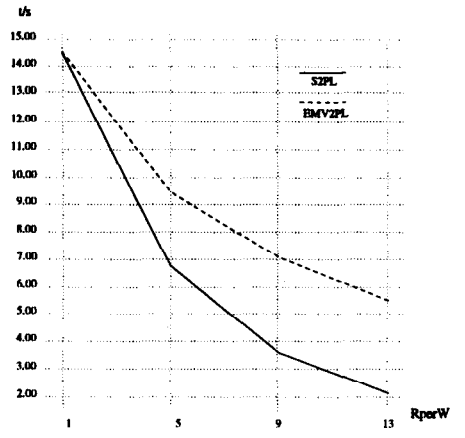

Figure 11(b) W|R tx throughput:*RperW* is varying

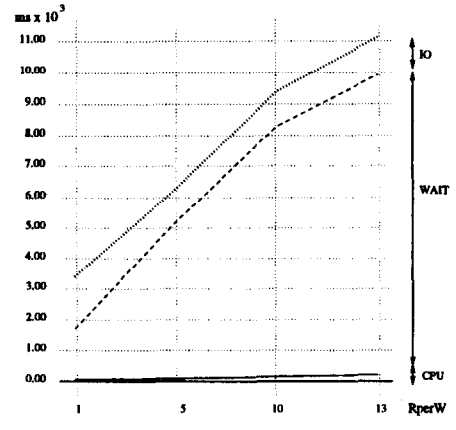

Figure 12(a): W|R tx response time (S2PL)

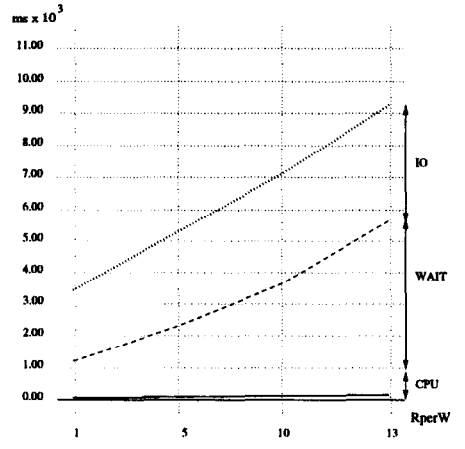

Figure 12(b): W|R tx response time (EMV2PL)

S2PL) or EMV2PL (curve *EMV2PL*). Figure 11(a) shows that EMV2PL always gives the best performance. With large multi-programming levels ($mpl \geq 40$), the throughput of W|R transactions under S2PL reaches a thrashing situation. As observed and explained in several studies [Tho91] [CKL90], the thrashing situation is caused by system under-utilization due to transaction blocking and wasted processing caused by transaction aborts. The curve *EMV2PL* shows that executing W|R transactions under EMV2PL avoids such thrashing situation. This is because EMV2PL eliminates the read-write lock conflicts due to the read operations executed in the read part of the transactions and reduces deadlocks among W|R transactions. Indeed, with $mpl = 60$, the mean number of waits per transaction is 2.21 under S2PL and only 1.01 under EMV2PL. With $mpl = 60$, the rate of aborts is 20% under S2PL and only 1% under EMV2PL.

In Figures 11(b), 12(a) and 12(b) we fixed *mpl* to 50 and varied the number of additional reads per write operation from 1 to 13. Figure 11(b) shows the throughput of W|R transactions. The throughput of W|R transactions is always better under EMV2PL than under S2PL. Moreover, the longer are the W|R transactions, the bigger is the gain in performance for EMV2PL. Figure 12(a)(resp. 12(b)) shows the response time of W|R transactions and how it is divided into CPU, wait and I/O times under S2PL (resp. EMV2PL). EMV2PL significantly reduces the wait time of W|R transactions while the contention on disk servers increases because transactions execute operations at a faster rate. The number k of resource units is thus an important parameter. Figure 13 shows the gain in throughput of W|R transactions under EMV2PL relative to S2PL with one, two or three resource units and under various multiprogramming levels. It shows that EMV2PL is more efficient as there are more resource units since the gain in concurrency is less affected by a higher contention on disk servers.

4.3 Experiment 2 : Application to Integrity Checking

In the following experiments, we evaluate workloads of transactions that perform integrity checking. We assume that one single integrity check is performed per write operation occurring in the transactions. We also assume that each integrity check requires a fixed number of read operations indicated by the *RperW* parameter. The *P_Roll* parameter represents the probability that an integrity check detects a constraint violation and rollbacks the transaction. In these experiments, we compare the performance of transactions executed with various integrity checking methods. As shown in Table 1, the choice of an integrity

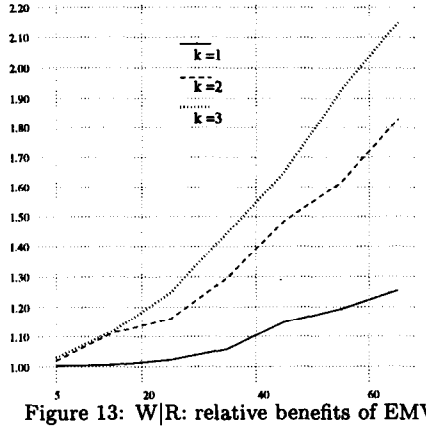

Figure 13: W|R: relative benefits of EMV2PL/S2PL

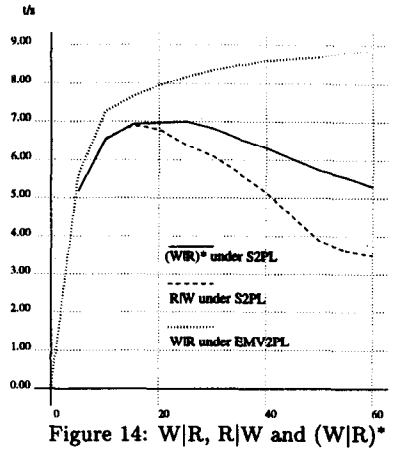

Figure 14: W|R, R|W and (W|R)* throughputs

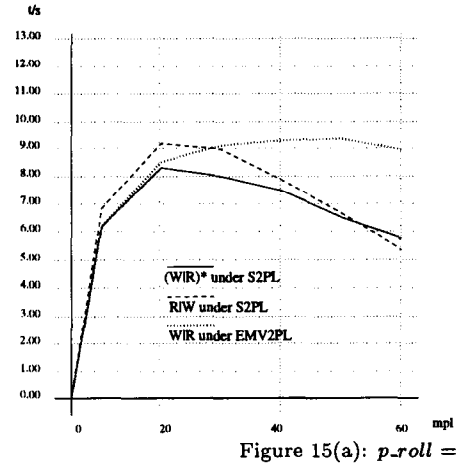

Figure 15(a): $p\_roll = 5\%$

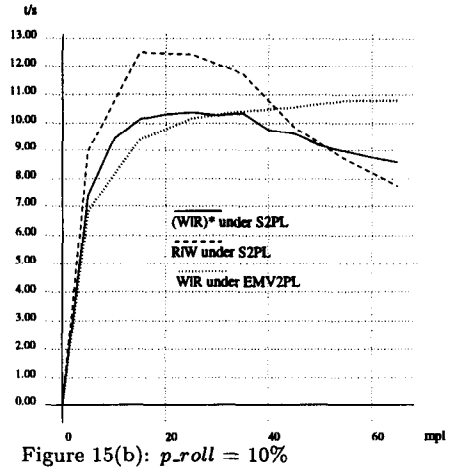

Figure 15(b): $p\_roll = 10\%$

checking method has an impact on the pattern of a transactions. Procedural check-before-write policy for transactions granularity yields R|W transactions. Declarative deferred checking and procedural write-then-check for transactions granularity yields W|R transactions. Other policies yield (W|R)* transactions. In the following experiments, we fixed the *RperW* parameter to 6. We execute W|R transactions under EMV2PL. R|W and (W|R)* are executed under S2PL.

Figure 14 shows the total throughput when no integrity check can issue a rollback (p_roll =0). Curves R|W and (W|R)* show that with large multiprogramming levels the throughput of R|W or (W|R)* reaches a thrashing situation caused by the additional read locks taken during constraint checking. Curves W|R shows that checking the constraints at the end of the transaction under EMV2PL avoids the thrashing situation. This is because EMV2PL eliminates all the read-write conflicts and deadlocks due to constraint checking.

Effect of rollbacks Figures 15(a) and 15(b) shows the influence of transaction rollbacks caused by integrity checks. In Figure 15(a) (resp Figure 15(b)) the probability *p_roll* that a constraint is violated and produces a rollback is set to 5% (resp. 10%). The curves show that R|W transactions provide the best troughput when the multiprogramming level is small. Indeed, checking constraints at the beginning of transactions allows to avoid unnecessary operations when the constraint is violated. Of course, higher is the probability of rollbacks, better is the throughput of R|W transactions (compare the curves R|W in Figures 15(a) and 15(b)). However, W|R transaction under EMV2PL outperform R|W transactions when the multiprogramming becomes larger. Indeed, W|R transaction under EMV2PL outperform R|W transactions when $mpl > 30$ if $p\_roll = 5\%$ (see Figure 15(a)) and when $mpl > 40$ if $p\_roll = 10\%$ (see Figure 15(b)). This is explained as follows. When the multiprogramming level is large the read-write lock contention is high. Since EMV2PL eliminates these locks conflicts the effect of EMV2PL becomes predominant.

Effect of read-write lock conflicts To show the effect of read-write lock conflicts we divided the database into two parts DB1 and DB2. Transactions perform only operations on DB1. Constraint checking produces only read operations on DB2. In these experiments *RperW* is fixed to 12 and $p\_roll = 0$. Figure 16(a) shows the throughput of W|R, R|W and (W|R)* transactions. This figure shows that checking constraint at the beginning of transactions gives always the best performance. This is explained as follows. In W|R transactions, exclusive locks taken in the write part are held for a longer time than with R|W transactions. Indeed, W|R transactions keep the exclusive locks until the whole read part is executed while R|W transactions release their exclusive locks at the end of the write part. Moreover, since constraints are only executed

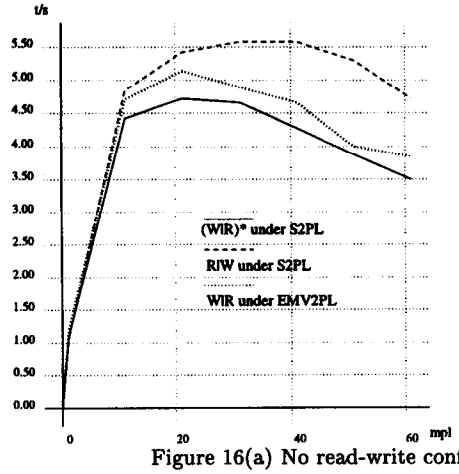

Figure 16(a) No read-write conflicts

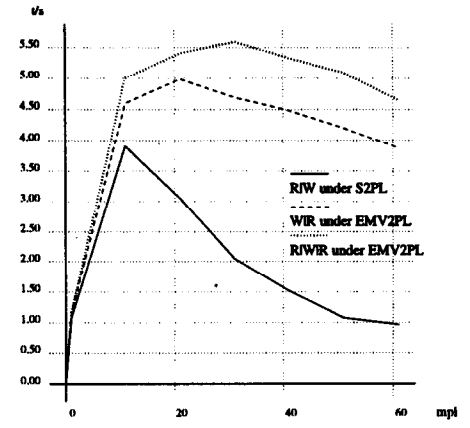

Figure 16(b): 50% read-write conflicts

on DB2, they never conflict with the write operations performed by the transactions on DB1. Thus, EMV2PL does not eliminate any read-write conflicts and its good effect becomes insignificant. Finally, Figure 16(a) also shows that executing W|R transactions under EMV2PL gives better performance than executing (W|R)* under S2PL. This is because EMV2PL allows transactions to release their read locks before executing their read part. Thus, in average, read locks on DB1 are held for a shorter time under EMV2PL.

In Figure 16(b), we consider a workload where 50% of the constraints perform operations on DB1 and 50% on DB2. We consider here three constraint checking policies. The check-before-write policy that yields R|W transactions, the write-then-check (or deferred checking) policy that yields W|R transactions and a mixed policy that consists in checking constraints on DB2 at the beginning of transactions and checking constraints on DB1 at the end of transactions. This policy yields R|W|R transactions. R|W transactions are executed under S2PL. W|R and R|W|R transactions are executed under EMV2PL. Let us note that, under EMV2PL, R|W|R transactions take locks until the end of their W part, that is, the first read part of these transactions is executed as under S2PL and the second read part is executed using versions. Figure 16(b) shows the resulting throughputs under various multi-programming levels. It shows that the mixed policy (R|W|R transactions) gives the best performance. This is explained as follows. First, by executing the constraints that perform read operations on DB1 at the end of transactions under EMV2PL, we eliminate all the additional read-write conflicts and avoid a trashing situation (only the R|W curve shows a thrashing behavior.) Second, by executing constraints that performs read operations on DB2 at the beginning of transactions we do not add any read-write lock conflict and shorten the write lock holding time of the transactions (compare the W|R curve with the R|W|R curve).

4.4 Rule-of-Thumb Lessons from these Experiments

The simulation results show that, when the workload contains W|R transactions, EMV2PL allows to increase the performance by eliminating read-write lock conflicts due to the R part of the W|R transactions. In particular, the results show that executing additional read operations in W|R transactions under EMV2PL allows to prevent lock contention thrashing possibly caused by these additional reads. This performance improvement is reduced by the increased resource contention caused by versions.

Moreover, the simulation results show that executing read operations for constraint checking at the end of transactions is usually the best solution under EMV2PL to improve the performance except in the following situations:

(1) *There is resource contention.* Once again, adding read operations at the end of transactions intensifies the use of versions and thus increases the resource contention.

(2) *The read operations are not involved in any read-write lock conflict.* In such a case, executing the read operations using versions does not eliminate any lock conflict. Moreover, exclusive locks taken in the Write part of the transaction are held for a longer time (until all the additional read operations are executed). Thus, the best solution is to execute the additional read operations at the beginning of transaction.

(3) *the read operations consist of integrity checks with a "high" probability of rollbacks.* In such a case, checking the constraint at the end of the transaction will possibly waste unnecessary operations. Once again, a best solution is to perform the checks with a high probability of rollback at the beginning of transactions.

These considerations lead to a simple feedback method to improve the pattern of transactions that execute a set of constraints.

Guideline for tuning constraint checking under EMV2PL

(1) Evaluate the lock contention

(2) Evaluate the system load

(3) If the system is under-utilized because of lock contention then:

  (a) Find the constraints that are involved in a lot of lock conflicts. If they are not often violated, try to execute them in a deferred mode.

(b) Find the constraints that are involved in very few lock conflicts, and try to execute them at the beginning of transactions.

(4) If lock contention is low, find the constraints that are very often violated and try to execute them at the beginning of transactions.

5 Related Work

An extensive literature addresses the problem of designing concurrency control algorithms that augment the performance of concurrent transactions. Our work directly builds on previous work on MV2PL protocols ([CFL+82], [BHG87], [AK91], [MPL92]), but differs from those by focusing on the problem of optimizing read operations in update transactions.

To avoid thrashing due to read operations a first solution is to use S2PL with a "read committed" locking level (also called locking isolation level 2). A transaction running under this level uses short duration read locks. Unfortunately, such a consistency level may lead to constraint violation anomalies as demonstrated in [BBG+95]. EMV2PL avoids these anomalies by ensuring serializability of transactions.

Another solution is to use versions to increase concurrency between transactions is with the *Snapshot Isolation* multiversion protocol [BBG+95]. In this protocol, a transaction T executing a read(x) operation always reads the most recent version of x that has been committed before the beginning of T, later called *Start_Timestamp* of T. Therefore, T reads a *snapshot* of the database as of the time it started. Updates performed by transactions that are active after T's *start_Timestamp* are invisible to T. When T is ready to commit, it gets a *Commit_Timestamp* greater than any existing *Start_Timestamp* or *Commit_Timestamp*. Then, T successfully commits only if no other transaction with a *Commit_Timestamp* belonging to T's interval: [*Start_Timestamp*, *Commit_Timestamp*] wrote data that T also wrote. Otherwise, T aborts. When T commits, its changes become visible to all transactions whose *Start_Timestamp* are larger than T's *Commit_Timestamp*. The *Snapshot Isolation* admits a simple implementation modeled on the work of Reed [Ree81]. Unfortunately, *snapshot Isolation* does not guarantee serializability and may lead to constraint violation anomalies as demonstrated in in [BBG+95].

The 2V2PL multiversion protocol authorizes the use of versions in update transactions, and guarantees serializability of transactions. In this protocol, there are three modes of locks associated to each lock unit: read lock (r), write lock (w) and certification locks (c). The corresponding compatibility matrix is the following:

	r	w	c
r	compatible	compatible	not compatible
w	compatible	not compatible	not compatible
c	not compatible	not compatible	not compatible

After each read (resp. write) operation, a r lock (resp. w lock) is taken. A *read(x)* operation always accesses the last committed version of x. At the end of the transaction, w locks are converted into c locks, and the transaction commits if and only if all the w locks have been successfully converted into c locks. To compare EMV2PL with 2V2PL, we compare the set of serializable histories that they admit. In fact , EMV2PL accepts histories that are not accepted by 2V2PL and conversely as shown by the following examples.

Example 5.1 Suppose we have two W|R transactions T1 and T2, and consider the following history:

```
w1(x) r1(t) w2(y) r1(y) r2(y) c2 r1(z) c1
```

If we assume that the read part of T1 contains the $r1(t)$, $r1(y)$ and $r1(z)$ operations and the read part of T2 contains $r2(y)$, then EMV2PL accepts this history. Indeed, T2 is never blocked since T2 does not access an item already locked by another transaction and can commit as soon as all its operations are performed. Moreover, since T1 reaches its lockpoint before T2, T1 cannot be blocked by the locks taken by T2 on y. 2V2PL does not accept this history: before committing, T2 must convert its w lock on y into a c lock. To do this, T2 must wait that T1 releases its r lock on y. Thus T2 must wait until T1 commits.

Consider now two transactions T3 and T4 and the following history:

```
w3(x)  r4(x) w4(y) c4 w3(z) c3
```

2V2PL accepts this history. Indeed, T4 can commit since it takes a w lock on a item that is not accessed by another transaction. On the contrary, EMV2PL does not accept this history because T4 is not a W|R transaction and hence the $r4(x)$ operation must take a lock on x; Since x is locked by T3, T4 must wait until T3 commits. □

So far, the largest body of work on the enforcement of semantic integrity constraints has focused on efficient algorithms to detect if a constraint is violated. The problem of optimizing the execution of multiple integrity checks within a transaction has been first addressed in [BP79]. This paper compares the performance of different constraint checking policies (including the check-before-write and the write-then-check methods) and show that the check-before-write method is the most efficient (in terms of transaction response time) because it avoids redundant computations and expensive rollbacks. However, all these works do not consider the possible concurrency between transactions.

A very few research papers have addressed the problem of optimizing the throughput of concurrent transactions that perform integrity checks. The *Commit_LSN* method, proposed in [Moh90], is used (among other things) to avoid taking a lock when checking a referential integrity constraint. More precisely, no read lock is acquired on data items involved in a "no action" referential integrity constraint [6] if (i) the constraint is satisfied, and (ii) a property of the *Commit_LSN* is verified. In contrast, our proposal is not limited to referential constraints (verified or not) but can be applied to any integrity check as long as it does not require new write locks. However our method only applies to W|R transactions.

[6] immediate or deferred

6 Conclusions

EMV2PL is a simple yet efficient extension of MV2PL which enables a W|R transaction that has acquired all its write locks to (i) release its read locks, and (ii) execute new read operations on versions without taking locks. In [LST97], we proved the correctness of this protocol, and showed that its implementation only requires a few changes with respect to an existing implementation of MV2PL. Performance studies show that for workloads containing W|R transactions, EMV2PL can significantly improve the overall throughput of transactions (i.e., W|R, and W transactions), with a relatively small utilization of versions.

We then presented a specific, yet important, application of our protocol to the problem of integrity checking. We described various possible methods for implementing integrity checking. For "read-only" integrity checks, we showed that: if the probability that a transaction violates integrity is small, then checking integrity at the end of transactions run under EMV2PL, is the method that often achieves the best total transaction throughput. Hence, (declarative) deferred checking generally offers better performance that immediate checking for "read-only" integrity checks, which in our view relaunches the interest of implementing deferred assertions and deferred triggers in relational database systems.

We foresee two directions of future work. One is to extend our simulation experiments to handle non uniform lock conflicts between data items. An interesting application of this is given by "summary tables", which consist of materialized views computed from base relations. Relations *MaxRisk* and *MinRisk* in the example of Section 2 are two examples of summary tables. These tables are quite frequent in decision support applications and for integrity checking. We plan to investigate the performance of EMV2PL in application scenarios where summary tables are read at the end of transactions. Another direction is to compare more in depth the performance of immediate versus deferred checking by taking into account the respective overheads associated with these two methods.

Acknowledgments

We are grateful to Anthony Tomasic for his detailed comments that enabled to improve this paper. We also thank Françoise Fabret, Angelika Kotz-Dittrich, C. Mohan, and Dennis Shasha for constructive discussions about the paper.

References

[AK91] D. Agrawal and V. Krishnaswamy. Using multiversion data for non-interfering execution of write-only transactions. *Proc. ACM SIGMOD Int. Conf. on Management of Data, Denver, Colorado*, 20:98–107, May 1991.

[BBG+95] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ansi sql isolation levels. *Proc. of the ACM SIGMOD Int. Conf. on Management of Data, San Jose, California*, pages 1–8, May 1995.

[BC92] P. M. Bober and M. J. Carey. On mixing queries and transactions via multiversion locking. *Proc. Int. Conf. on Data Engineering, Tempe, Arizona*, pages 535–545, February 1992.

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

[BP79] D.Z. Badal and G.J. Popek. Cost and performance analysis of semantic integrity validation methods. *Proc. ACM SIGMOD Int. Conf. on management of Data, Boston, Mass.*, pages 109–115, 1979.

[CFL+82] A. Chan, S. Fox, W.K. Lin, A. Nori, and D.R. Ries. The implementation of an integrated concurrency control and recovery scheme. *Proc. ACM SIGMOD Int. Conf. on Management of Data, Orlando, Florida*, pages 184–191, June 1982.

[CKL90] M.J. Carey, S. Krishnamurthy, and M. Livny. Load control for locking : the half and hall approach. *Proc. ACM Symposium on the Principles of Database Systems*, 1990. voir lck.Carey.89.

[LST97] F. Llirbat, E. Simon, and D. Tombroff. Using versions in update transactions: Application to integrity checking. Technical report, INRIA, 1997. extended version, available at http//rodin.inria/personnes/francois.llirbat.

[Mel93] J. Melton, editor. *(ISO/ANSI Working Draft) Database Language SQL3.* Number ANSI X3H2-90-412 and ISO DBL-YOK 003. February 1993.

[Moh90] C. Mohan. Commit_lsn: a novel and simple method for reducing locking and latching in transaction processing systems. *Proc. of the 16th Int. Conf. on Very Large Data Bases, Brisbane, Australia*, pages 406–418, August 1990.

[MPL92] C. Mohan, H. Pirahesh, and R. Lorie. Efficient and flexible methods for transient versioning of records to avoid locking by read-only transactions. *Proc. ACM SIGMOD Int. Conf. on Management of Data, San Diego, California*, pages 124–133, June 1992.

[Ree81] D. Reed. Implementing atomic actions decentralized data. *ACM TOCS*, 1981.

[SLSV95] D. Shasha, F. Llirbat, E. Simon, and P. Valduriez. Transaction chopping: Algorithms and performance studies. *ACM Transactions on Database Systems*, 20(3), December 1995.

[Tho91] A. Thomasian. Performance limits of two-phase locking. *Proc. Int. Conf. on Data Engineering*, pages 426–435, April 1991.