Concurrency control for Read-Only in Mobile Nested Transactions

Luis A. Gama-Moreno\textsuperscript{1}, Matías Alvarado\textsuperscript{2,1}
\texttt{lgama@sagitario.cic.ipn.mx, matiasa@imp.mx}

\textsuperscript{1}Centro de Investigación en Computación
Instituto Politécnico Nacional
J. Batiz esq. O. De Mendizabal s/n. C.P. 07738, México, DF,
\textsuperscript{2}PIMAyC, Instituto Mexicano del Petróleo
Eje Central L. Cárdenas 152, CP 07730, México, DF

Abstract
This paper introduces a concurrency control mechanism for the Mobile Nested Transactions model (MNT), this approach extends the 2PL mechanism for read-only operations. Under Nested Transactions (NT) 2PL, every transaction must wait for its ancestor to hold (by inheritance) the control over the requested lock. This assumption reduces the performance of applications running concurrent transactions. With the proposed lock scheme, a better performance is achieved in read-only concurrent transactions running over mobile environments, consistency and cascade aborts are not compromised. Transactions can accomplish despite of failures originated from: frequent disconnections, limited energy sources or insufficient bandwidth. The proposed approach facilitates transaction processing at oil extraction facilities platforms. These are places at open sea with difficult access to wired networks or the Internet.

Keywords
Concurrency control, Mobile Transactions, Nested Transactions, Read-Only Transactions, Serializability.

1. Introduction
In order to preserve data consistency, concurrency control is used in Database Management Systems (DBMS) [Gray and Reuter, 1994]. The objective is to ensure transactions isolation whenever there are concurrent operations requesting access to the same object; these must be coordinated in order to prevent inconsistencies [Härder and Rothermel, 1993; Lu and Satyanarayan, 1994], in such a way that each user perceives he/she is the only one with access to the objects (data) [Gama and Alvarado, 2002]. This way, the DBMS is required to guarantee the isolation property [ANSI, 1992; Adya et al., 2000; Lu and Satyanarayan, 1994], by transaction serialization. That is, when a set of operations among transactions (better known as histories) does not cause inconsistent values, as it happens with a serial execution [Koulimitch and Sheremetov, 1998; Dogdu, 1998; Berenson et al., 1995; Bertino et al., 1998]. Other important properties that must be ensured are: Atomicity, Consistency and Durability [ANSI, 1992].

Different protocols for transaction isolation have been proposed. The most traditional are: Two-Phase Lock (2PL), TimeStamp Ordering (TSO) [Gray and Reuter, 1994; Dogdu, 1998; Bertino et al., 1998] and multiversion [Bernstein and Goodman, 1983]. Likewise, extensions to the concurrency control schemes for flat transactions have also been proposed, these are: an alternative model based on the multiversion approach [Park and Park, 1997], nested transactions [Moss, 1985], nested transactions for real-time databases [Pavlova and Nekrestyanov, 1997], improved locks control in nested transactions [Reddy and Kitsuregawa, 2000] and optimistic protocols [Kung and Robinson, 1981].

In this paper, we present an alternative approach for the concurrency control model proposed in [Moss, 1985], where the main goal is:

\begin{itemize}
  \item to extend shared locks (s-lock) accessibility and visibility (for read-only) at any level from the transaction tree, thus improving applications performance.
\end{itemize}

In this schema, a transaction $T_i$ can request a s-lock, for read-only, in such a way that the locked object can be available for \textit{any} other transaction $T_j$, that is, $T_i$ does not need to wait for:

\begin{itemize}
  \item $T_i$ to inherit its s-lock control to its direct ancestor, and
  \item the s-lock control not only depends from $T_i$ direct ancestors.
\end{itemize}
If a given transaction \( T_j \) requests to s-lock an object previously s-locked by \( T_i \), then \( T_j \) does not have to wait until \( T_i \) finishes, even when they share the same ancestor. Our approach is based on shared locks compatibility, in this way, transactions do not wait for the result of any other transactions that previously s-locked the concurrently accessed objects. Because of the operations read-only nature, a s-locked object holding between two transactions \( T_i \) and \( T_j \) does not imply that \( T_j \) aborts as consequence of \( T_i \) abortion and vice versa. This avoids cascade aborts, more common in writing operations on dependent transactions [Gray and Reuter, 1994; Dogdu, 1998].

The rest of the paper is organized as follows. Section 2 introduces concurrency control protocols currently implemented in commercial DBMSs and describes the 2PL on NT. Section 3 presents an extension to the NT concurrency control with the purpose of improving processing performance on MNT. Section 4 describes the advantages of using ROT under MNT model. Finally, Section 5 presents the conclusions.

2. Concurrency Control Protocols

This section describes the operation of the two most important concurrency control protocols in transaction processing. Concurrency control (CC) is an important functionality required in a DBMS. The purpose of CC mechanisms is to ensure DB consistency during concurrent access from different users [Gray and Reuter, 1994; Berenson et al., 1995; Bertino et al., 1998]. Transactions must remain isolated in order to avoid inconsistent values. However, a tight isolation mechanism decreases application performance and increases the possibility of deadlocks [Bertino et al., 1998]; on the other hand, a loose isolation level increases the possibility of inconsistencies and cascade aborts [Adya et al., 2000; ANSI, 1992; Berenson et al., 1995; Bertino et al., 1998]. Different protocols have been proposed in order to ensure concurrent transactions serialization. The most known are: Two-Phase Locking (2PL) protocol and Timestamp Ordering (TSO) protocol [Gray and Reuter, 1994; Dogdu, 1998; Bertino et al., 1998]. These are explained below.

2.1. Two-Phase Locking Protocol (2PL)

This protocol delays operation execution among concurrent transactions that can conflict, as it may happen with two write attempts. For instance, \( w_i(x) \), \( w_j(x) \) \( \forall i \neq j \) where \( w_i(x) \) denotes a transaction \( T_i \) writing over the x object, \( w_j(x) \) denotes writing over the x object by a transaction \( T_j \). These operations can result in inconsistent values between the involved transactions. Therefore, transactions lock objects in order to prevent inconsistencies.

2PL defines two phases. In the first phase, the transaction locks each object that will be accessed. In the second phase, the transaction confirms results (commit) by writing all the modified data to persistent storage, thus releasing the acquired locks. In this protocol, two type of locks are considered:

- **Shared (s-lock):** if a transaction \( T \) gets a shared lock over an object \( Q \), then \( T \) can read but not write over \( Q \).
- **Exclusive (x-lock):** if a transaction \( T \) gets an exclusive lock over an object \( Q \), then \( T \) can both read and write over \( Q \).

Depending on the type of operation to be executed over \( Q \), a shared or exclusive lock is acquired. Suppose that \( A \) and \( B \) be an arbitrary locking modes representation. Given a locking modes set, a compatibility function can be defined between them. If \( T_i \) asks for an A locking mode over the object \( Q \), and there exists a \( T_j \) \( (i \neq j) \) which currently holds a B locking mode over the same object. If \( T_i \) can get the requested lock over \( Q \), then it can be said that A and B are compatible locking modes. Such compatibility function can be represented by a matrix \( M \) where \( M(i,j)=true, \) if \( i \) locking mode is compatible with \( j \) locking mode. Table 1 shows the compatibility matrix \( M \) between different locking modes.

<table>
<thead>
<tr>
<th>Lock</th>
<th>Shared</th>
<th>Exclusive</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shared</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Exclusive</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

A different way for serializing operations is through a progressive order selection between transactions. This approach is implemented with timestamps ordering, as it shown below.

2.2. Time Stamping Ordering Protocol (TSO)

In this protocol, each transaction is assigned an unique timestamp denoted by \( TS(T_i) \). This tag is assigned by the DBMS before the transaction begins. If a timestamp (TS) is assigned to \( T \), and a new transaction \( T_j \) gets started in the system, then \( T_j \)'s timestamp is the lower than \( T_i \)'s timestamp: \( TS(T_j) < TS(T_i) \). In order to implement this tagging scheme as timestamps, one of the following methods can be used: 1.) Using the system clock. Transaction’s TS is the clock’s value at the time the transaction begins. 2.) An internal counter that gets increased each time a TS is assigned. The transaction’s TS is the counter’s value at the time the transaction begins.

Timestaps define a serialization order. In this way, if \( TS(T_i) < TS(T_j) \), the system must ensure that the generated schedule is equivalent to the serialized one, where \( T_j \) appears before \( T_i \). In order to implement this scheme, on of two timestamp types is associated to each object \( Q \) according to the performed operation:
• W-TS(Q) represents the greatest TS among the timestamps from all the transactions that succeeded at writing over Q.
• R-TS(Q) represents the greater TS among the timestamps from all the transactions that succeeded at reading over Q.

These values are updated after new readings or writings over Q. TSO protocol guarantees that every read/write operations with possibility of conflict, can be executed according to the defined order. This protocol is organized as follows:

1. Suppose that a transaction $T_i$ needs to read over the object Q,
   a) If $TS(T_i) < W-\text{TS}(Q)$, it means that $T_i$ will read a value from Q that has already been overwritten. Then the reading operation must be rejected and transaction $T_i$ must be aborted as well.
   b) If $TS(T_i) \geq W-\text{TS}(Q)$, it means that the reading operation over Q can be accomplished and, R-TS(Q) gets set the greatest value between R-TS(Q) and TS($T_i$).

2. Suppose that a transaction $T_i$ needs to write over the object Q,
   a) If $TS(T_i) < R-\text{TS}(Q)$, it means that the value of Q is currently modified by $T_i$ but, it also has been previously read by another transaction, then it can not be updated.
   b) If $TS(T_i) < W-\text{TS}(Q)$, it means that $T_i$ is attempting to write an obsolete value from Q.

The write operation is then ignored and transaction $T_i$ must abort.

c) If any from the previous cases apply, then the write operation is executed and W-TS(Q) assigns the greater value between W-TS(Q) and TS($T_i$).

Each aborted transaction, gets tagged with a new timestamp and is restarted. TSO protocol guarantees serialization since conflicting operations are processed under a timestamp order.

Both protocols serialized concurrent transactions, however, 2PL can fall into deadlocks problems, whilst TSO has the possibility to present cascade aborts.

2.3 Two-Phase Locking for Nested Transactions

Moss’s concurrency control proposal for NT, use 2PL protocol in order to guarantee the concurrent transactions serialization [Moss, 1985; Gama and Alvarado, 2003]. If T is a transaction that requires an A mode lock over the object Q, then the next situations arise:

Rules for getting locks:
L1. T can get a lock in exclusive mode over object Q if:
   1. there is no other transaction holding a shared or exclusive lock over Q or,
   2. every transaction currently holding a shared or exclusive lock over Q is a T’s predecessor.

L2. T can get a shared lock over the object Q if:
   1. no other transaction is currently holding an exclusive lock over object Q or
   2. every transaction currently holding a shared or

---

Figure 1 T2 waits for grant a s-lock over q.
exclusive lock over Q is a T’s predecessor.

Rules for termination (commit/abort)
T1. If a transaction T commits, its predecessor inherits all the released locks,

T2. If a transaction T aborts, the held blocks are released and its predecessor inherits them.

2PL protocols for NT lock objects during transactions execution; if another transaction Tx requires a lock over a currently locked object by a transaction Ty, then Tx must wait until such lock gets controlled by a Ty’s predecessor in order to acquire the lock. This restriction guarantees transaction isolation at the leafs level from each nested transaction. However, this reduces the application performance since Ty must wait until the lock is carried to its predecessor. In Figure 1-(a), transaction T122 gets a s-lock over the object q. In Figure 1-(b), the transaction T2 belonging to a different branch than T122, requests a s-lock over the same object q. Due to the L2 rule (see Section 2.3), T122 must wait for one of its predecessors to hold that lock (T for this case). Figures 1-(c), (d), (e) show the termination (commit/abort) of T122 predecessor transactions, they inherit the lock over q at each level until it reaches the T2’s predecessor. In Figure 1-(f) T2 gets the lock according the L2 rule.

3. Concurrency control for MNT

The disadvantage for the approach introduced in Section 2 is overcome, thus allowing shared locks to be carried up to any level from the transactions tree. This also allows that different branches are candidates to inherit the lock, thus improving application performance. Objects locks by read-only operations (ROO) is informed to any ancestor requesting a ROO over a previously locked object in a shared mode. This is the typical scenario found at centralized or distributed processing systems in wired networks. However, mobile environments entail additional difficulties due to frequent disconnections. In [Gama and Alvarado, 2003] a mobile nested transaction (MNT) model is proposed. There, failures are captured under a nested transactions context. Top-level transaction commitment is reached despite of failures in sub-transactions.

3.1. Mobile Nested Transactions

The MNT model [Gama and Alvarado, 2003] is based on the classic nested transactions model. It features an execution mechanism in logical units – each unit is assigned to a mobile device – and better control over concurrency and recovering in contrast to flat transactions. The MNT model extends that functionality from hierarchical control in the NT model [Moss, 1985]. It is implemented over a group of mobile devices thus allowing networking in contexts with difficult access to wired networks or the Internet.

The main MNT advantage is the possibility of handling sub-transactions perceived as logical work units in mobile environments. MNT ensures the successful accomplishment of the full transaction despite faulty sub-transactions. This concurrency control scheme is feasible to implement in environments exposed to frequent disconnections.

3.2. Read-Only Transactions

Read-only operations need to be protected from shared locks (s-lock). This operations do not generate new versions of the modified objects, that is, they do not change the object state. This feature has arisen the concept of read-only transactions (ROT) [Han et al., 2000; Park et al., 1997; Lee et al., 2002]. A ROT is a transaction that only reads the current state of some object (for instance, reading the value of a data field) without trying to modify it. With the use of ROT, the number of conflicting operations among update transactions is reduced. As a consequence, an application can process a larger number of transactions, thus improving the concurrency level and application performance.

Mobile computing features add more restrictions in transaction processing. Idle time between concurrent transactions is the result of the time objects remain locked. This situation gets worse with failures in device operation such as: disconnections, asymmetric bandwidth, limited batteries, and so on. Then, transactions performance can be even more affected by all those additional restrictions.

Given such reason, the interest of this work is based on framing read-only operations. It must be allowed that any other sub-transaction (this includes sub-transactions belonging to the same predecessor or to a different one), can get a lock over a previously locked object in a shared mode. Under such context, write operations requesting exclusive locks are not considered. These can continue under the traditional concurrency control approach in MNT.

Concurrency control extension levels up shared locks visibility and availability. Application performance gets improved while avoiding idle time in transactions that require shared locks over objects in use.

Figure 2 shows this approach. In Figure 2(a), transaction T122 gets a shared lock over object q. In figure 2(b), T2 requires a s-lock over the object holding by T122. T2 does not need to wait for the transaction T122 to finish since the s-lock is automatically inherited to all the predecessors. This in turn, allows T2 to get such s-lock as it is shown in Figure 2(c). This feature improves concurrency control and application performance. With this approach, transactions requesting s-locks are not supposed to experiment idle time since all the s-locks are available, even to transactions from different branches.

3.3. ROO effects over the commit/abort

ROT do only feature reading operations to different objects [Han et al., 2000]. If and object q is locked in shared mode, any other transaction can not get an exclusive lock over the
same object until it gets released. Given such reason, if two or more transactions hold a shared lock over the same object, confirmation or cancellation from one of them does not entail inconsistencies. Cascade aborts are neither triggered because of the cancellation of an involved transaction. This is due to the fact that, if two transactions $T_x$ and $T_y$ accomplished only read operations over the object $q$, an abort does not generate inconsistencies.

4. ROT advantages on MNT

In our approach, shared locks (for ROO) are inherited at any level within the transaction tree hierarchy, even before the transaction finishes. This represents a major advantage since transactions waiting for previously s-locked objects, can get their locks in advance. Once a read-only lock is ensured, the transaction cannot try to execute write operations (with exclusive locks). Suppose that $T_j$ is a transaction that needs to execute both read and write operations within the same transaction; then two possibilities may arise:

1. $T_j$ locks the object $q$ in an exclusive mode, thus preventing a transaction $T_j$ to access the same object in a shared way. Therefore, $T_j$ will have to wait until $T_i$ releases the object $q$, besides it will have to wait until the lock gets inherited by one of its ancestors (lock rule L1 from Section 2.3).
2. $T_j$ locks the object $q$ in a shared mode and performs the intended operations. Later, $T_j$ requests an unlock over object $q$ follows by an exclusive lock (x-lock) in order to perform a write operation over $q$. After the write operation is accomplished, $q$ gets released (unlock). However, this option does not comply with the 2PL rules. The reason is that it can not be requested a new lock over some object during unlocking operations. Therefore, this option does not ensure, neither serialization in concurrent transactions, nor isolation; thus, application performance is lowered.

Given such reasons, with ROT (for instance: balances query over a group of bank accounts, sales briefings over a given time frame, sale products reports, and so on) this approach can improve the performance between ROT, thus avoiding deadlocks or cascade aborts situations.

Different efforts for synchronizing nested transactions execution have been proposed. In [Resende, 1994], a new theory framework for testing serialization of NT synchronization protocols is introduced. In [Härder and Rothermel, 1993], there is a descendent inheritances concept applied in order to increase parallelism in NT. In [Madria et al., 1997], pre-write operations are introduced in order to increase concurrency in NT processing; this model allows certain sub-transactions to release their locks before their parents confirm. This allows that other sub-transactions can get locks without having to enter an idle state. However, once the sub-transaction pre-wrote (pre-commit) a value, it can not longer abort.

With respect to update transactions, different approaches have been proposed in order to improve concurrency control in nested transactions. In [Pavlova and Nekrestyanov, 1997], a hybrid algorithm for real-time transactions is presented. It allows certain branches from the tree to be controlled with optimistic or pessimistic perspectives. In [Lee and Lam, 1999], a new variant of optimistic concurrency control is applied over broadcast environments. There, read-only transactions can be processed locally at the mobile clients. Only those transactions featuring updates are sent to the servers for a final validation.

5. Conclusions

Concurrency control is a crucial mechanism for guaranteeing data consistency during transactions
concurrent execution. This paper extended the NT locking mechanism under the 2PL protocol for the MNT model. A visibility extension for objects locking under read-only operations (ROO) was proposed. Advantages from this extension are: 1) improvements in the ROT performance. 2) Idle time and the number of cascade aborts are reduced. 3) Deadlocks between transactions are avoided. 4) If two transactions $T_i$ and $T_j$ share a read-only lock, $T_i$ abortion does not trigger $T_j$ abortion and vice versa. Future research is concerned with extending and improving the concurrency control scheme for update (write) operations in mobile environments.

References


