Correctness of Parallel Executions in Multidatabase Systems Ruled by Strict 2 Phase Locking


*University of Versailles, **University of Montpellier II, PRiSM Laboratory, LSI Laboratory, 45, avenue des Etats-Unis, 860, rue de St Priest, 78000 Versailles (France) 34090 Montpellier (France) e-mail: <Lastname>@prism.uvsq.fr e-mail: <Lastname>@lsi.fr

Abstract

This paper addresses the problem of transaction management in multidatabase systems where the participating local DBMSs serialize transactions by 2 Phase-Locking (2PL) and synchronize their commit actions through a 2 Phase-Commit protocol (2PC). These DBMSs represent the majority of commercial relational and object-oriented DBMSs. We demonstrate that if local DBMSs support strict 2PL optimization (i.e., relax the read locks during the prepare phase of the 2PC protocol), then schedules of parallel global transactions may become non-globally serializable. X/Open DTP protocol avoids the problem by adding a blocking synchronization phase at transaction end, thereby loosing the whole benefit of strict 2PL. This paper proposes two strategies to preserve the benefit of strict 2PL while ensuring correct executions.

Keywords: multidatabase, strict 2PL, degrees of isolation.

1. Introduction

A growing number of applications require access to data located in multiple DBMSs, motivating research efforts in multidatabase transaction management. In a multidatabase system (MDBS), local transactions are executed under the control of pre-existing and autonomous local database management systems (DBMSs) while global transactions accessing several DBMSs are under the control of a Global Transaction Manager (GTM). A global transaction is split into subtransactions, each one being considered as a local transaction by the local DBMS. Various protocols have been proposed to guarantee the ACID properties [13] of global transactions when the accessed local DBMSs support different serialization and validation policies [7]. Surprisingly, less attention has been paid by the research community to MDBS in which local DBMSs implement the same well known policies. Typically, most relational and object-oriented commercial DBMSs serialize local transactions by 2 Phase-Locking and support the XA interface defined in the X/Open DTP model [21] to participate in a standardized 2PC protocol. The small interest devoted by researchers to this important class of MDBSs can be explained by the fact that all transaction schedules are globally serializable if (i) all local DBMSs are rigorous (e.g., implement 2PL) and (ii) all local DBMSs participate in a 2PC protocol [6]. Unfortunately, this ignores the fact that some local DBMSs exploit the strict 2PL optimization [2] (i.e., relax the read locks during the prepare phase of the 2PC protocol) to increase inter-transaction parallelism. DBMSs supporting both strict 2PL and 2PC protocols are called strict-2PLPC DBMSs [4].

In this paper, we show that the use of strict 2PL may lead to non-serializable schedules when global transactions issue operations to the MDBS asynchronously. This problem must be considered carefully since window-based applications (e.g., CASE, CAD, OIS) as well as transactional workflows and parallel transactional programs are likely to submit asynchronous operations to a multidatabase. The X/Open DTP model [21] avoids the problem by adding a blocking synchronization phase for all subtransactions of a same global transaction before issuing the first prepare-to-commit demand. However, this protocol no longer benefits from strict 2PL. This limitation is severe in the case of applications accessing a large amount of resources in read mode. This paper proposes two different strategies to preserve the value of strict 2PL with parallel global transactions, while maintaining correct schedules.

The first strategy, called Global Serializability Strategy, re-establishes global serializability by exploiting the properties of “serialization points” [7]. The basic idea is to use either the “prepare-to-commit” or the “ready-to-commit” messages to ensure that subtransactions are serialized in the same order on all sites. This strategy relies on fairly simple rules which can be added in any GTM or TP-Monitor. However, this strategy may lead to reject some global serializable schedules.

The second strategy, called Value Dependency Stability, avoids this problem by relaxing the constraint of global serializability. The basis of this strategy is that uncontrolled executions on strict-2PLPC DBMSs lead to the same isolation problems as transactions running under degree 2 isolation [13] in centralized DBMSs. In the latter case, applications are responsible for preserving the consistency of the database by declaring SQL cursors to maintain a complete isolation between correlated SQL statements (e.g., updates depending on previous reads). This concept is commonly called Cursor Stability. We introduce the concept of Value Dependency Stability to maintain the isolation of correlated operations when these operations are involved in global integrity constraints (i.e., inter-site constraints). The advantage of this strategy is that it preserves the benefit of strict 2PL, and increases parallelism between global and local transactions by accepting correct but non-globally serializable schedules. As with degree 2
to the widespread situation in which local DBMSs are the theoretical scope of this study, this assumption corresponds to the widespread situation in which local DBMSs are not globally serializable. Section 3 and Section 4 respectively present the Global Serialization and the Value Dependency Stability strategies. For each strategy, we evaluate to what extent the benefit of strict 2PL is preserved. Finally, Section 5 summarizes the contribution and outlines our future research actions.

2. Motivations of the study

2.1 Multidatabase systems context

A multidatabase system is built from pre-existing and independent DBMSs, located at different sites, in a manner that allows global transactions (GTs) to access objects residing at these DBMSs. It also enables global integrity constraints to be defined, joining objects at different DBMSs. It is based on the existence of a global transaction manager (GTM). Each global transaction GTi is decomposed by the GTM into global subtransactions, one per site (GSTik is executed on site Sik). There is no restriction on parallel or sequential processing by the GTM, of the operations belonging to a GT.

We suppose that each DBMS guarantees, by means of a local transaction manager (LTM), serializability and fault tolerance of the local transactions (LTs) and the GSTs accessing the objects of the DBMS. It thus preserves the ACID properties and the integrity constraints involving local objects. As for the GTM, it manages the GTs in order to ensure the correctness of the schedule resulting from the execution of all the transactions, while respecting as much as possible the autonomy of sites.

2.2 2PL and 2PC Assumption

In this paper, we consider multidatabase systems in which local DBMSs serialize transactions by means of a two phase locking (2PL) protocol and provide standardized interface (e.g., X/Open Xa interface[21]) to be able to participate in a two phase commit (2PC) protocol. While limiting the theoretical scope of this study, this assumption corresponds to the widespread situation in which local DBMSs are relational and object-oriented commercial DBMSs implementing these protocols. We recall below the basis of these protocols and introduce some notations used all along the paper.

2PL protocols

Transaction Ti can read (r) or write (w) an object x, only after having locked it in the corresponding mode (rli[x] or wli[x]). The object is unlocked (ru[x] or wu[x]) when it ceases to be used. Ti is blocked when locking x, if Tj already holds a lock on x in a conflicting mode; that is to say by noting < the local execution order of operations when : ( o'lj[x] < ol[x] ) and (o or o' ∈ {w}). According to the nature of the operations involved, classical conflicts RW, WW and WR are considered and we note Oo> the resulting dependency between transactions Ti and Tj.

Locking is called “two phase” (2PL) when Ti begins to release some locks only after having obtained all its locks (maximum locked point):

\[
2PL \equiv \forall T_i, \forall o \in \{r, w\}: (ol_i[x] < ol[x] < ou_i[x]) \quad \text{and} \quad (\forall x, y : ol_i[x] < ou_i[y]).
\]

Protocol is called rigorous [6] when, in addition to 2PL, all locks are released only after the commit (commiti) or the reject (aborti) of the transaction:

\[
\text{rigorous } 2PL \equiv 2PL \quad \text{and} \quad (\forall T_i, \forall o \in \{r, w\}: \text{commit}_i < ou[x] \quad \text{or} \quad \text{abort}_i < ou[x]).
\]

Protocol is called strict [2], when only write locks are released after the termination:

\[
\text{strict } 2PL \equiv 2PL \quad \text{and} \quad (\forall T_i, \text{commit}_i < wu_i[x] \quad \text{or} \quad \text{abort}_i < wu_i[x]).
\]

2PL protocol maintains serializability of (LTs and GSTs) transactions on each site. The serialization order between transactions, noted --->, is thus:

\[
T_i \xrightarrow{---}\ T_j \equiv \forall x \in \text{Object}(T_i) \cap \text{Object}(T_j), \text{ol}_i[x] < \text{o'}l_j[x] \quad \text{and} \quad (o \text{ or } o' \in \{w\}).
\]

The interest of the strict and rigorous 2PL protocols is to generate recoverable and cascadless abort schedules. The obvious advantage of strict 2PL over rigorous 2PL is that it increases concurrency on each site, by allowing transactions to access the objects, read unlocked by Ti, without having to wait Ti’s termination. We expect this advantage to be preserved in a multidatabase context.

2PC protocol

The 2PC protocol is based on a coordinator (the GTM) and participants (the LTMs). This protocol ensures a GT is either committed on all sites or aborted on all ones. Briefly GTi’s commit works as follows:

![Architecture of a multidatabase system](image)
a) The GTM asks every LTMk concerned whether they are ready to commit by sending the prepareik message.

b) Each LTMk puts GSTik in the "ready to commit" state (pik) and returns the readyik message or, if GSTik is locally rejected, it returns the rejectedik message.

c) The GTM considers GTi to be in the state commitablei if (i.e., \( \forall k, readyik \) is received). In this case, the GTM commits GTi's commit by sending to every LTMk the commitik message. Otherwise GTi is aborted by sending to every LTMk the abortik message.

d) When receiving commitik (resp. abortik), every LTMk commits (resp. aborts) GSTik.

By noting \( \preceq_{\text{GTM}} \) the order of the events observed by the GTM, it follows:

\[
(\forall \text{ site } k : \text{send(prepareik)} \preceq_{\text{GTM}} \text{receive(readyik)}) \preceq_{\text{GTM}} \text{commitablei} \preceq_{\text{GTM}} (\forall \text{ site } k : \text{send(commitik)} \text{or send(abortik)})
\]

**Rigorous-2PLPC protocol**

Using 2PC associated to local rigorous 2PL on each LTM ensures global serializability [6]. It is sufficient for the GTM to confirm the commitment as soon as GTi is in a "commitable" state. Since all locks are only released on every site after committing GTi, this serves to enforce global rigorous 2PL for GTi.

**Strict-2PLPC protocol**

The use of strict 2PL by LTMs allows any GST to release its read locks without having to wait for commit. Though a transaction can theoretically unlock from the maximum locked point, GSTik can be sure that this point is reached only when it has received prepareik and terminated all its operations. In this way, with the strict 2PL-2PC protocol that we call strict-2PLPC [4], read locks are released as soon as GSTik has become "ready to commit" and sent readyik:

\[
\text{strict-2PLPC} \equiv \forall \text{ GSTik} \in \text{ site } k, \forall o \in \{r, w\}:
\]

\[
(\text{olik}[x] < pik \text{ and } (pik < \text{send(readyik)} < \text{ruik}[x]) \text{ and } (\text{abortik} < \text{wuik}[x] \text{ or } \text{commitik} < \text{wuik}[x])
\]

It thus follows that a transaction T (either local or global), blocked by a GST because of a RW conflict (GST\(_{RW \rightarrow T}\)), will be able to continue, as soon as GST is in the ready to commit state, without having to wait for the end of commit as is the case with rigorous-2PLPC. This means that when there is RW conflict, it is possible to both increase concurrency between transactions and to preserve LTs autonomy.

2.3 Problem position

Numerous multidatabase applications (CAD, CASE, OIS, transactional workflows,…) perform parallel and asynchronous computations. Through a window-based interface or a parallel programming language, a client can submit tasks (e.g., a source code compilation) to the GTM (e.g., a TP-monitor) and continues its work without waiting for an answer. The GTM enqueues the client demands and can potentially execute them in a deferred mode. If the client issues its commit to the GTM while some tasks are always being processed, the 2PC protocol guarantees that the commit is effective only if all the tasks succeed. The GTM on its own can trigger parallel treatments at transaction commit, for example to maintain the consistency of replicated data or to integrate in public databases the work done in a private one. In such situation, the GTM can ask to some LTMs to prepare the commit of a GST without having to wait for the end of all the triggered actions. We feel that parallelism will be a key feature of most future transactional applications. However, parallel transactions have not yet been deeply studied, especially in the multidatabase context.

In the following, we consider parallel GTs composed of different tasks or operations run concurrently over several local DBMSs. We assume the GSTik to be equally potentially parallel. Some operations can be submitted asynchronously (i.e., the sender does not wait for an answer). Unlike most of the studies conducted on MDBSs [7], we do not impose that a LTM acknowledges to the GTM the execution of all operations submitted to it. Consequently, the GSTs terminations can be asynchronous. On demand of a GT, the GTM can prepare to commit when some GSTs are still executing operations whilst other GSTs have finished theirs. If LTMk receives prepareik when GSTik is still executing operations, readyik will be sent only after having finished GSTik local processing.

Parallel GTs are excellent candidates to fully exploit the benefit of strict-2PLPC protocol. Suppose a global transaction GTi accessing sites Sk and Sj. If GSTik and GSTij have been sent asynchronously and GTi issues its commit to the GTM, then GSTik can relax its read locks before the end of GSTij (assuming pik < pij). On the other hand, GTi may not be 2PL anymore. Indeed GSTik releases read locks on site Sk while GSTij may continue to obtain some other locks on site Sj (Fig. 2). Only write locks are held until GTi commits. In this way, a GT is globally 2PL only with respect to write locks.
It follows that strict-2PLPC can lead, in some cases, to global non-serializable schedules. These uncontrolled situations can lead to some inconsistent views for global read transactions and to inconsistent states for DBMSs due to violation of global integrity constraints. Let us consider a MDBS located at two sites: $S_1$ with object $a$ and $S_2$ with object $b$. $GT_1$ and $GT_2$ have direct conflicts in reverse order on $S_1$ and $S_2$. For the sake of clarity, we use simplified notations\(^1\) where $r_{ik}[x]$ summarizes ($r_{ik}[x]$), $r_{ik}[x]$ and $p_i(x)$ summarizes ($receive(prepare_{ik})$, $p_i(x)$, send($ready_{ik}$), unlock read locks held by $GST_{ik}$). The schedule on each site is as follows:

\[
S_1 : r_{11}[a] \; p_{11} \; w_{21}[a] \; ...
\]

\[
S_2 : r_{22}[b] \; p_{22} \; w_{12}[b] \; ...
\]

At site $S_1$ the dependency is therefore $GST_{11}^{RW} \rightarrow GST_{21}$, and at site $S_2$ $GST_{22}^{RW} \rightarrow GST_{12}$. The transactions $GT_1$ and $GT_2$ are not serializable even though the GSTs have not yet been locally committed, whatever their chronological commit order. \(\square\)

Using standard X/Open DTP interface of the 2PC protocol [21] restricts the asynchrony of the GSTs, by imposing the GTM to send $prepare_{ik}$ only after having made sure that all the GSTs are terminated. For this, the GTM sends $xa-end_{ik}$ to every $LTM_k$ that reply $xa-ok_{ik}$ only when $GST_{ik}$ terminates its local processing. The DTP protocol is as follows:

\[
X/Open \; DTP \equiv [\forall k : \text{send}(xa-end_{ik}) <_{\text{GTM}} \text{receive}(xa-ok_{ik})] <_{\text{GTM}} [\forall k : \text{send}(prepare_{ik})] ...
\]

By synchronizing GSTs termination at the beginning of commit, the X/Open DTP protocol forces a GT to be 2PL with respect to all its locks and therefore prevents global non-serializable schedules to occur. The drawback is that read unlocks are delayed, and thus the interest of strict 2PL is lost.

In the sequel, we propose three different strategies to properly control the execution of parallel and asynchronous GSTs, while preserving the benefit of strict 2PL on each site.

3. Global serializability

In this section, we first characterize the non-serializable schedules produced by uncontrolled executions of parallel global transactions over strict-2PLPC DBMSs. Then we show that strict-2PLPC DBMSs satisfy the serialization point property defined in [7]. Finally, we propose two strategies exploiting this property to re-establish the global serializability of transaction schedules.

3.1 Characterization of non-serializable schedules

Under the assumption that local DBMSs serialize the local executions, a schedule of global transactions is not globally serializable iff global transactions are not serialized in the same order on all accessed sites. The local serialization order $LSO_k$ on a site $S_k$ is induced by the conflicts between all global subtransactions. A conflict between two subtransactions $GST_{ik}$ and $GST_{jk}$ can be either direct (e.g., $GST_{ik}^{*} \rightarrow GST_{jk}$) or indirect (e.g., $GST_{ik}^{*} \rightarrow LT_{lk}^{*} \rightarrow GST_{jk}$, where $LT_{lk}$ denotes a local transaction on site $S_k$). Let us note $GST_{ik}^{*} \rightarrow GST_{jk}$ any dependency -either direct or indirect- existing on site $S_k$ between two subtransactions. This local dependency generates a global dependency of the form $GT_i^{*} \rightarrow GT_j$. A globally non-serializable schedule (i.e., $\exists S_k S_l / LSO_k \neq LSO_l$) is characterized by a dependency cycle between GTs.

When coupling 2PC with rigorous 2PL, a globally non-serializable schedule leads to a global deadlock. Consequently, GTs are always serialized in compatible orders. However, as shown in Section 2.3, some non-serializable global schedules produced over strict-2PLPC DBMSs do not generate global deadlocks.

3.2 Serialization point property of strict-2PLPC DBMS

A serialization point (sp) of a transaction is a distinguished action that determines the serialization order of the transaction in a local schedule [18]. For instance, in a timestamp based concurrency control scheme, the serialization point corresponds to the assignment of a timestamp to a transaction. If $sp(GST_{jk})$ precedes $sp(GST_{ik})$ in the local schedule of site $S_k$, then no dependency of the form $GST_{jk}^{*} \rightarrow GST_{ik}$ can occur. If all local DBMSs satisfy the serialization point property, the global serializability of any schedule is guaranteed if the serialization point of GTs are executed in the same order at all sites. This condition can be refined by maintaining a site-graph [7]. A site-graph is a bipartite graph connecting each global transaction with all sites it accessed during its execution. If two global transactions do not participate in a

---

1 For the sake of clarity, these simplified notations are used in all figures of the paper.
cycle in the site-graph, no cyclic dependencies can occur between them and then, they can be run concurrently. Strict-2PLPC DBMSs satisfy the following local property:

\[ \text{P1: } \text{GSTik} \rightarrow \text{GSTjk} \implies pik < pjk \]

Indeed, on each site, either local and global transactions are ruled by 2PL. Consequently, the serialization point of a transaction corresponds to its maximum locked point. Since local DBMSs exploit strict-2PL instead of rigorous 2PL, the serialization point of a global subtransaction GSTik corresponds to pik. Let us note SG the site-graph and C the set of cycles in SG, maintaining the global serializability of transaction schedules requires to satisfy the following rule:

\[ \text{P2: } \exists GT_i, GT_j \in SG, \exists C_l \in C / GT_i \in C_l \text{ and } GT_j \in C_l \implies (\forall S_k \in C_l, pik < pjk) \text{ or } (\forall S_k \in C_l, pjk < pik) \]

### 3.3 Pessimistic commit protocol

A first way to satisfy rule P2 is to exploit the commit-graph algorithm proposed in [7]. In this algorithm, all global transactions are executed concurrently except during their commit phase. The commit phases of transactions that are involved in a cycle in the site-graph are executed serially. This algorithm can be optimized in the particular case of strict-2PLPC DBMSs. Assume transactions GTi and GTj participating in a cycle Cl, satisfying rule P2 just imposes to the GTM to schedule the prepare and ready messages in the following order:

\[ \forall S_k \in C_l \text{ receive}(\text{readyik}) < \text{GTM send}(\text{preparejk}) \]

Compared to the commit-graph algorithm, with this optimization the prepare phase of GSTjk is delayed only until readyik is received by the GTM instead of being delayed until the readyik are received for all sites accessed by GTj.

The main drawback of this algorithm is to abort global transactions if the order in which they enter in their commit phase do not correspond to the local dependencies order. Assume GTi enters in its commit phase before GTj and there exists a dependency of the form GSTjk \( \rightarrow \) GSTik on site Sk. The GTM will never receive the response of prepareik and will abort transaction GTi after a given timeout, even in the absence of cyclic dependencies.

### 3.4 Optimistic commit protocol

Instead of synchronizing global transactions during their commit phase, the optimistic commit protocol accepts concurrent commits for all global transactions. The basic idea of the algorithm is to check a posteriori that rule P2 is satisfied. To this end, the GTM must be aware of the LSOk for each site Sk involved in a cycle of the site-graph. Under the assumption that all messages ready are sent by a site Sk to the GTM through the same communication channel, the order in which these messages are received by the GTM corresponds to LSOk. From the GTM point of view, sp(GSTik) corresponds to the reception of readyik. This leads to the following global property:

\[ \text{P3: } pik < pjk \implies \text{receive}(\text{readyik}) < \text{GTM receive}(\text{readyjk}) \]

The GTM builds a precedence-graph with the transactions participating in a cycle of the site-graph as nodes. If readyik is received before readyjk, an arc from GTi to GTj is added in the graph. If a cycle is introduced in the precedence-graph, one of the participating transactions is aborted. This protocol guarantees the serializability of all global schedules. However, it may reject some serializable schedules because pik < pjk \( \implies \) GSTik \( \rightarrow \) GSTjk (property P1 is not bijective). Assume GTi and GTj both access sites Sk and Sl and assume pik < pjk and pjl < pil. Either GTi or GTj will be aborted even if no local conflict occurs between these two transactions. Avoiding these unjustified rejections requires a stronger cooperation of the LTMs to make the GTM aware of the local conflicts. We did not explore this issue which runs against site autonomy.

#### 3.5 Comparison with X/Open DTP

This section evaluates the ability of the pessimistic and optimistic commit protocols to preserve the benefit of strict 2PL, compared to X/Open DTP protocol. We introduce below two basic parameters to properly evaluate the behaviour of each protocol. Although important, we neglect in these comparisons the messages transmission cost on the network because this cost has the same influence on all protocols.

\[ \Delta \text{GSTik} : \text{time required on site Sk to complete the execution of GSTik at the time transaction GTi issues its commit to the GTM (i.e., time to complete the execution of GSTik ongoing operations).} \]

\[ \Delta \text{Lijk} : \text{latency time needed by GSTik to reach the state \text{GSTik} on site Sk after all its operations have been completed (i.e., among others, time to log GSTik updates).} \]

As mentioned in Section 2, the DTP protocol broadcasts the prepare messages to all sites accessed by a global transaction once all of its subtransactions have completed their execution (i.e., have answered to the xa-end message). The benefit of strict-2PL is then lost but none condition is imposed on the concurrent validation of distinct global transactions. Conversely, the optimistic commit protocol fully preserve the benefit of strict-2PL since the prepare
messages are sent as soon as the GTM receives the commit demand of a global transaction. Compared to the optimistic commit protocol, the DTP protocol delays the read unlocks of a subtransaction GST\textsubscript{ij} of (Max\textsubscript{k} (ΔGST\textsubscript{ik}) - ΔGST\textsubscript{ij}). Indeed, each GST has to wait for the completion of the longest GST of the same GT before relaxing its read locks. In the sequel of the paper, this difference will be called the benefit of strict 2PL.

The behaviour of the pessimistic commit protocol is more difficult to evaluate. Clearly, the benefit of strict 2PL is fully preserved for all subtransactions which do not participate in a cycle of the site-graph. Assume now that transactions GT\textsubscript{i} and GT\textsubscript{j} participate in a cycle. For each site S\textsubscript{k} involved in the cycle, the prepare\textsubscript{jk} is delayed until the ready\textsubscript{ik} is received by the GTM. Roughly speaking, the prepare phase of GST\textsubscript{jk} is delayed by (ΔGST\textsubscript{ik} - ΔGST\textsubscript{jk}) + L\textsubscript{ik}. Depending on the ΔGST’s, this delay can be worse for some subtransactions than the one imposed by DTP.

As a conclusion, the pessimistic and optimistic commit protocols guarantee the serializability of global schedules while preserving the benefit of strict 2PL in most cases, at the price of aborting some serializable transactions. Sections 4 and 5 present more permissive protocols which relax the global serializability constraint.

### 4. Value Dependency Stability

The strategies presented in Section 3 preserve the benefit of strict 2PL at the cost of increasing the ratio of aborted transactions. This drawback can be avoided by accepting non-globally serializable schedules under the control of the application. The strategy proposed in this section extends the cursor stability notion [13] introduced in centralized DBMSs to properly control non-serializable executions.

#### 4.1 Cursor Stability Requirement

As demonstrated in Section 2, with the strict-2PLPC protocol RW conflicts can lead to opposite serialization orders on different sites, in presence of parallel global transactions. This may lead to lost updates as well as to inconsistent updates with regard to global integrity constraints for transactions containing value dependencies. A value dependency is defined as follows:

**Value Dependency**

A global transaction GT\textsubscript{i} contains a value dependency denoted by GST\textsubscript{ik}\textsubscript{a} \rightarrow GST\textsubscript{ij} iff operation w\textsubscript{ij}([b]) performed by GST\textsubscript{ij} on Site S\textsubscript{j} depends on the result of operation r\textsubscript{ik}([a]) performed by GST\textsubscript{ik} on Site S\textsubscript{k}.

Assume transaction GT\textsubscript{i} willing to copy the value of object \textit{a} into object \textit{b} while transaction GT\textsubscript{j} is adding the value \textit{v} to \textit{a} and \textit{b}. Objects \textit{a} and \textit{b} are respectively located in sites S\textsubscript{1} and S\textsubscript{2} and a global integrity constraint has been declared to enforce the constraint \textit{a} = \textit{b}. If GT\textsubscript{i} performs the update on \textit{b} asynchronously, the schedule given in Figure 3 may occur and will be responsible for both lost update and inconsistent update. Indeed, site S\textsubscript{1} can receive the message prepare\textsubscript{i1} from the GTM while GST\textsubscript{i2} is still being processed.

\[
\begin{align*}
S_1: &\text{ prepare}_i(a)\text{ p}_i1 & w_j(a) \text{ p}_j1 \\
S_2: &\text{ w}_j(b) \text{ p}_j2 & w_j[b] \text{ p}_j2
\end{align*}
\]

*Figure 3.* Schedule involving lost update and global integrity constraint violation

The same situation may occur in centralized DBMSs if some transactions do not run in complete isolation. To gain parallelism between transactions, most relational systems provide the opportunity to ignore WR and/or RW dependencies. These options are generally called *degrees of isolations*. They have been formalized in [13] as follows:

- **Degree 0**: Lock protocol is well-formed (i.e., all actions are covered by locks) with respect to writes (ignores all dependencies).
- **Degree 1**: Lock protocol is 2PL with respect to write locks and well-formed with respect to writes (ignores WR and RW dependencies).
- **Degree 2**: Lock protocol is 2PL with respect to write locks and well-formed (ignores RW dependencies).
- **Degree 3**: Lock protocol is 2PL and well-formed (preserves all dependencies).

While degree 0 and degree 1 are generally restricted to read-only transactions, updates are allowed for transactions running in degree 2 or degree 3 isolation. Degree 2 isolation is the default in most SQL systems which automatically release read locks after a record or table read. Performing updates in degree 2 isolation may lead to the same schedule as the one presented in Figure 3, even if objects \textit{a} and \textit{b} are located on the same site. To avoid lost updates, SQL systems introduce a better form of isolation than pure degree 2, called *cursor stability*. The cursor stability protocol prevents lost updates by keeping a read lock on the record currently addressed by the SQL cursor. The application is responsible for the correct management of RW dependencies by properly using SQL cursors. Only degree 3 isolation discharges the application from this task.

An analogy can be made between the isolation provided by the strict-2PLPC protocol with parallel transactions and degree 2 isolation in centralized SQL systems. The main difference is derived from the fact that the locking protocol remains 2PL on each site in the former case. Consequently, the *Repeatable Read* property (i.e., reading the same object twice gives the same result) is preserved whereas this
property is lost in degree 2 isolation. Thus degree 2 isolation is more permissive than the strict-2PLPC protocol, itself being more permissive than degree 3 isolation. Therefore, a mechanism similar to the cursor stability protocol extended to multilocal transactions should solve lost updates and global integrity constraint violation problems. The following subsections present two solutions for correctly treating RW conflicts in the strict-2PLPC protocols. Both solutions have to enforce the isolation of value dependency defined below.

**Value Dependency Isolation**

A value dependency of the form

\[ \text{GST}_i \text{r}_a[a|w_i[b]] \rightarrow \text{GST}_j \]

is isolated iff \( w_i[b] < r_u[a] \).

### 4.2 Acknowledge-based protocol

As in [7], we assume here that each local DBMS acknowledges to the GTM, using \( \text{ack}_j \) messages, the execution of all operations submitted to it, whether operations are sent synchronously or asynchronously. From the GTM point of view, maintaining the value dependency isolation for

\[ \text{GST}_i \text{r}_a[a|w_i[b]] \rightarrow \text{GST}_j \]

under this assumption means to guarantee:

\[ \text{receive}(\text{ack}(w_i[b])) < \text{GTM}(\text{send}(\text{prepare}_i)) \]

A straightforward solution to enforce this equation is for the GTM to block the execution of the transaction \( \text{GST}_i \) until it receives the message \( \text{ack}(w_i[b]) \). Roughly speaking, this transforms asynchronous calls in synchronous calls. The drawback of this solution is to enforce

\[ \text{receive}(\text{ack}(w_i[b])) < \text{GTM}(\text{send}(\text{next GST}_i)) \]

which is a stronger condition than is required. This solution prevents the execution of following \( \text{GST}_i \) operations in parallel with \( \alpha_2 \). A more permissive solution is for the GTM to set a barrier at the time the value dependency is declared. This barrier delays the message \( \text{prepare}_i \) until \( \text{ack}(w_i[b]) \) is received.

### 4.3 One-way request based protocol

Acknowledge-based protocol using barriers guarantees an optimal treatment of value dependencies. However, we presume that most of the asynchronous calls are not acknowledged by the callee. As an example, the usual way to send asynchronous Remote Procedure Calls (RPC) is not to wait for an answer. Under this assumption, the GTM cannot infer the end of any asynchronous operation \( \alpha_0 \) sent to site \( S_k \) before having received the message \( \text{ready}_i \). The condition which has to be enforced in order to guarantee the value dependency isolation for

\[ \text{GST}_i \text{r}_a[a|w_i[b]] \rightarrow \text{GST}_j \]

becomes:

\[ \text{receive}(\text{ready}_i) < \text{GTM}(\text{send}(\text{prepare}_i)) \]

Towards this end, the GTM registers the value dependency declarations of each transaction in a graph called Value Dependency Graph (VDG). The VDG of a transaction is defined as follows.

**Value Dependency Graph (VDG)**

The Value Dependency Graph of transaction \( \text{GST}_1 \) is a valued directed graph \( \text{VDG}_i(S, E) \), where \( S \) denotes the set of nodes and \( E \) denotes the set of edges of \( \text{VDG}_i \). A node \( S_k \in S \) iff site \( S_k \) has been accessed by transaction \( \text{GST}_i \). An edge \( e = (S_1, S_k, \text{label}(e)=r_{ik}[a|w_i[b]]) \in E \) iff a value dependency of the form

\[ \text{GST}_i \text{r}_a[a|w_i[b]] \rightarrow \text{GST}_k \]

has been declared by \( \text{GST}_i \).

At the time transaction \( \text{GST}_i \) issues its commit to the GTM, the way the \( \text{prepare} \) phase is ruled by the GTM depends on whether or not \( \text{VDG}_i \) is acyclic. In the former case, the \( \text{prepare} \) phase is processed as follows. For each pending node \( S_k \in S \) (i.e., node without out-edges), the GTM sends \( \text{prepare}_k \) to site \( S_k \). Once the GTM receives \( \text{ready}_k \), it removes \( S_k \) and all edges referencing \( S_k \) from \( \text{VDG}_i \). This process is repeated recursively until \( S=\emptyset \). This algorithm guarantees that sites relax their read locks in an order reverse to the value dependency order. Unfortunately, this algorithm gets into trouble if \( \text{VDG}_i \) contains cycles.

Breaking a cycle of the form \((S_1, S_2, r_{i1}[a|w_2[b]]) (S_2, S_3, r_{j2}[c|w_3[d]]) (S_3, S_1, r_{j3}[e|w_1[f]])\) can be done by upgrading the lock held on either \( a \), \( c \) or \( e \) from read mode to write mode. Assume read lock on \( a \) has been selected for upgrade, the corresponding write lock will not be released on site \( S_1 \) before receiving \( \text{validate}_1 \) or \( \text{reject}_1 \). The value dependency \((S_1, S_2, r_{i1}[a|w_2[b]])\) can therefore be removed from \( \text{VDG}_i \). To upgrade the read lock on \( a \), the GTM issues to \( S_1 \) the request \( w_1[a]\) (this may lead to read object \( a \) again). Figure 4 illustrates the behaviour of this algorithm.

![Figure 4. Management of cyclic value dependencies](image)

**4.4 Value Dependency Stability vs X/Open DTP**

Assume a transaction \( \text{GST}_i \) accessing \( n \) sites. We will compare the behaviour of the DTP protocol with both Value Dependency Stability protocols (VDSack for the acknowledge-based protocol and VDSowr for the one-way request protocol) in the following situations: (1) \( \text{GST}_i \) does not contain any value dependency and (2) \( \text{GST}_i \) contains a value dependency of the form

\[ \text{GST}_i \text{r}_a[a|w_i[b]] \rightarrow \text{GST}_j \]
For each situation the GTM enforces the following schedules of events:

\[
\begin{align*}
\text{DTP (1)(2):} & \quad [\forall k : \text{send}(xa-end_{ik}) <_{\text{GTM}} \text{receive}(xa-ok_{ik}) <_{\text{GTM}} \forall k : \text{send}(prepare_{ik}) <_{\text{GTM}} \text{receive}(xa-ready_{ik})] \\
\text{VDS (1) (VDSack and VDSowr):} & \quad [\forall k : \text{send}(prepare_{ik}) <_{\text{GTM}} \text{receive}(ready_{ik})] \\
\text{VDSack (2):} & \quad \text{VDS(1) and } \text{receive}(\text{ack}(w_{il}[b])) \text{send}(prepare_{ik}) <_{\text{GTM}} \\
\text{VDSowr (2):} & \quad \text{VDS(1) and } \text{receive}(\text{ready}_{il}) <_{\text{GTM}} \text{send}(prepare_{ik})
\end{align*}
\]

As already mentioned, the DTP protocol never preserves the benefit of strict 2PL since the prepare phase on each site is postponed until the end of all GTi subt :rnsactions. VDSack(1) and VDSowr(1) fully exploit strict 2PL for all subtransactions. In the worst case, VDSack(2) delays the prepare phase of GST_{ik} by the time required to compute \( w_{il}[b] \) on \( S_i \) if GST_{ik} finishes its work before the end of \( w_{il}[b] \). The benefit of strict 2PL is preserved for all other subtransactions.

We now evaluate the behaviour of VDSowr(2). To this end, we reuse the notations \( \Delta GST_{ik} \) (time required to complete GST_{ik} at GTi commit time) and \( L_{ik} \) (latency time for GST_{ik}) introduced in section 3.5. Roughly speaking, the prepare phase of GST_{ik} is delayed by \( (\Delta GST_{il} - \Delta GST_{ik}) + L_{il} \). More generally, if a subtransaction GST_{ik} is involved in a cycle of value dependencies, the delay before it enters into its prepare phase can roughly be estimated by \( ((\text{Max} \Delta GST_{il} + \sum L_{il}) - \Delta GST_{ik}) \), \( \forall GST_{il} \) preceding GST_{ik} in the cycle. Again, the benefit of strict 2PL is preserved for all subtransactions which are not involved in a value dependency.

In addition to this comparison, note that the Value Dependency Stability protocols support correct but non-globally serializable schedules, thereby increasing parallelism between global transactions as well as between global and local ones.

5. Conclusion

This paper addresses the practical problem of transaction management in multidatabase systems where the participating local DBMSs are strict-2PLPC DBMSs. In this context, we have shown that parallel global transactions loose the 2PL property, thereby producing non-globally serializable schedules. X/Open DTP protocol avoids the problem by adding a blocking synchronization phase at transaction end, which leads to loosing the whole benefit of strict 2PL.

We propose two different strategies to preserve the benefit of strict 2PL while ensuring correct executions. The Global Serial strategy enforces global serializability at the cost of aborting global transactions when non-serializable executions are presumed. The accepted schedules fully exploit strict 2PL for each subtransaction. Compared to the Global Serial strategy, the Value Dependency Stability strategy avoids unnecessary aborts. In addition, it increases parallelism between global transactions as well as between global and local ones by relaxing the global serializability constraint. However, the preservation of global integrity constraints involving value dependencies is the responsibility of the application. The rules on which rely these two strategies can be integrated quite easily in any GTM. We insist on the implementation feasibility of these strategies because we believe that parallelism will become a key feature of most transactional applications.

We are currently defining a complete analytical model to more accurately evaluate the gain of the proposed strategies compared to the X/Open DTP standard. This could help to integrate optimization in X/Open DTP to take advantage of strict 2PLPC DBMSs. In addition, we are determining a third type of strategies using strict-2PLPC, based on two-level serializability.

Bibliography


