1991

Maintaining Consistency of Interdependent Data in Multidatabase Systems

Amit P. Sheth
Yungho Leu
Ahmed K. Elmagarmid

Purdue University, ake@cs.purdue.edu

Report Number:
91-016

https://docs.lib.purdue.edu/cstech/865

This document has been made available through Purdue e-Pubs, a service of the Purdue University Libraries. Please contact epubspurdue.edu for additional information.
MAINTAINING CONSISTENCY OF INTERDEPENDENT DATA IN MULTIDATABASE SYSTEMS

Amit P. Sheth
Yungho Leu
Ahmed K. Elmagarmid

CSD-TR-91-016
March 1991
Maintaining Consistency of Interdependent Data in Multidatabase Systems*

Amit P. Sheth  
Bellcore  
RRC 1J-210, 444 Hoes Lane  
Piscataway, NJ 08854-4182

Yungho Leu and Ahmed K. Elmagarmid  
Department of Computer Science  
Purdue University  
West Lafayette, IN 47907

Abstract

Interdependent data are characterized by dependency constraints and mutual consistency requirements. Maintaining consistency of interdependent data that are managed by heterogeneous and autonomous DBMSs is a real problem faced in many practical computing environments. Supporting a mutual consistency criterion that is weaker than one copy serializability is often acceptable if better performance can be achieved and the autonomy of DBMSs is not sacrificed.

Updates to interdependent data have to be controlled in order to maintain their consistency. We propose a solution where at least one of the copies of each interdependent data, called current copy, is kept up-to-date in the system. Using the concept of update through current copy, we show how a weaker mutual consistency requirement, called eventual consistency, can be satisfied. The proposed approach requires writing only the local copy (as opposed to writing many or all copies) and hence preserves autonomy by not requiring synchronization of the local concurrency controllers. This approach exploits the semantics of both the interdependent data and transactions; thus, it is non-intrusive, flexible and efficient.

1 Introduction

In a vast majority of computing environments in industry, software is built and developed as applications designed around functional divisions. Often, each application has its own database, and such databases are managed by autonomous and heterogeneous DBMSs. Sharing of data is often done at the application level, and not directly supported and managed by the DBMSs. An example to illustrate interdependent data is that of multiple applications maintaining customer addresses, where the definition and details of the addresses may be different in different databases. However, if one application changes an address in its own database, but related addresses in other databases are not updated, other applications may be using old addresses. Many companies incur huge costs because of the use of inconsistent data. Manual reconciliation in absence of system

*Sponsored by the Indiana Corporation for Science and Technology (CST), a PYI Award from NSF under grant IRI-8857952, grants from the AT&T Foundation, Bellcore, Tektronix, SERC, Mobil Oil and Bell Northern Research.
support for managing consistency of these related data is tedious, error prone, slow, and also incur significant cost.

An interdependent data is a set of related data items, called members of interdependent data, in different databases. The relationships among the members can be characterized by the data dependency and the mutual consistency requirements. A related work is that of replica control in distributed database systems. An interesting distinction between an interdependent data and a replicated data (as used in literature) is that interdependent data typically exists in multidatabase systems, while data replication is induced to provide better reliability and availability in distributed database systems. The data dependency among replicas in a replicated database is that of identity (i.e., all related data/copies are identical). Furthermore, traditionally, mutual consistency over replicated data requires that replicated data should behave as if it were a single copy data, as far as a user can tell. This mutual consistency requirement is called immediate mutual consistency. The most often used immediate mutual consistency criterion in the literature is that of one copy serializability (or 1SR) [BHGS87].

Enforcing one copy serializability for managing immediate mutual consistency of interdependent data in a multidatabase system may be difficult and expensive because different member of interdependent data may be stored in autonomous and heterogeneous DBMSs. It has been shown in [DELO89] that it is very difficult to synchronize the activities of heterogeneous and autonomous DBMSs to ensure multidatabase consistency, including mutual consistency[1]. Hence, several weaker mutual consistency criteria for interdependent data have been proposed [SK89],[SR90],[ABGM90],[WQ90]. One such criterion, called eventual consistency [SK89] [SR90], allows the specification of mutual consistency ranging from immediate mutual consistency to no consistency.

One copy serializability is an application independent criterion. It may be unnecessarily strict for many applications. Eventual consistency, on the other hand, is an application dependent criterion that can be specified using three aspects [SR90] [RSK91]: time (e.g., by 5 P.M. every day), data state (e.g., more than 5% change on the data value), or operations (e.g., number of updates performed on the interdependent data)[SR90]. Different combinations over these three aspects model different degrees of mutual consistency required for the interdependent data.

In this paper, we propose an algorithm for maintaining the eventual consistency of a simple type of interdependent data and discuss extensions to manage some of the more complex types of interdependent data. The proposed approach is both non-intrusive and efficient (i.e., it incurs low cost in performing updates on interdependent data). We also propose a new update principle called update through current copy for updating interdependent data. It relies on keeping at least one member of a set of interdependent data current. By current we mean that the member reflects all updates that have been performed on the interdependent data at any given instance. We also present an algorithm for converging the members of an interdependent data to its current copy when the eventual consistency constraint of the interdependent data is violated.

This paper is organized as follows. In Section 2, we discuss a specification of interdependent data. In Section 3, we review the relevant work and the problems in using the previous solutions for

[1]SK89] discusses inappropriateness of 1SR for more practical environments in further detail.
managing interdependent data. In Section 4, we present the update through current copy principle and discuss its correctness criterion. In Section 5, we present an interdependent data update control algorithm which applies the update through current copy principle and preserves the correctness criterion. The update control algorithm ensures that every local database is in a consistent state and a correct current copy for each interdependent data exists. Then, we present an algorithm for converging the members of an interdependent data to its current copy according to the eventual consistency constraint associated with each interdependent data. In Section 6, we propose an efficient way to perform the copy operation of the update through current copy principle and discuss extensions to the update control algorithm to handle some of more complex types of interdependent data. In Section 7, we present concluding remarks.

2 A Specification of Interdependent Data

In this section, we will describe a simple specification of interdependent data that is adequate for this paper. An in-depth treatment of the specification of interdependent data appears in [RSK91].

Relationships among interdependent data can be characterized by two components, interdatabase dependency and eventual consistency requirement [SR90]. The interdatabase dependency has a structural and a control aspect. Examples of structural aspects include full replication, partial replication and fragmentation, overlapping data (e.g., some fields are common), and constraints (e.g., value of field X of relation R1 is less than that of field Y of R2). Control dependency allows capturing the restrictions on updating interdependent data and hence allows representing relationships such as derived data, primary-secondary copies and independently updatable data [SK89]. The eventual consistency requirement component allows the user to specify mutual consistency for members of an interdependent data.

Thus, an interdependent data D is a set of k member data items (or simply called members) denoted by {d1, d2, ..., dk}, where each member is in a different database. Associated with D is a dependency specification DEP(D) and an eventual consistency constraint CON(D).

A dependency specification DEP(D) indicates structural dependency that specifies the structural relationships among all members of interdependent data and a control dependency that specifies whether updates are allowed on a specific member. The latter is used for the specification of access control and update propagation. For most of this paper, we restrict the interdatabase dependency to the following:

A1: We assume that members of each interdependent data are fully replicated\(^2\) and that each member is independently updatable.

In Section 6, we will discuss partial relaxation of the above restriction.

CON(D) is a logical expression (consisting of conjunction and/or disjunction) of three aspects, data state (C.\(d(D)\)), operation (C.\(o(D)\)) and time (C.\(t(D)\)). For example, we can express the

\(^2\)However, the contents of their corresponding fields may be different when immediate mutual consistency is not required.
following eventual consistency constraint (similar to the one in [ABGM90]):

\[
CON(D) \leftarrow ((C \cdot D = 2 \text{ hour}) \land (C \cdot \omega(D) = 2))
\]

This constraint means that the members of \( CON(D) \) must converge after two hours since the previous convergence of the members if there have been at least two updates. A special case is to set \( CON(D) \leftarrow (C \cdot \omega(D) = 0) \) which implies that members of \( D \) should be made consistent immediately after an update on any member of \( D \) has been performed. The mutual consistency provided by immediate consistency is equivalent to that provided by the traditional criteria of one copy serializability. Another extreme case is to use predicate “FALSE” for \( CON(D) \), which states that the mutual consistency on members of \( D \) may be enforced at any convenient time or is not required.

Based on the associated eventual consistency constraint, a member of an interdependent data can be in one of three states:

- **mutually inconsistent**, if the associated eventual consistency constraint is violated (i.e., evaluated to false);
- **lagging**, if the eventual consistency constraint holds, but it does not reflect all the updates performed so far on members of the interdependent data;
- **current**, if this member reflects all the updates performed so far on (all members of) the interdependent data.

We distinguish between two kinds of transactions: transactions that update interdependent data, called **Interdependent Data update transactions** (or ID update transactions), and those that read but do not update interdependent data, called **Interdependent Data read-only transactions** (or ID read-only transactions). It is required that the ID update transactions read only from the current member of the interdependent data because they have direct effects on the consistency of the interdependent data. However, it is possible to provide two levels of consistency to ID read-only transactions\(^3\). A **strong read** requires that an ID read-only transaction reads only a current member of an interdependent data. A **weak read** allows reading a lagging member of an interdependent data. When the immediate consistency is supported for an interdependent data, the strong and weak read have the same effect—read from the current copy. However, when eventual consistency is supported, it is possible for an ID read-only transaction to specify the level of consistency desired by choosing between strong and weak reads.

### 3 Related Work

The main body of previous related work is on replicated data control in distributed database systems. In this section, we first review some well known replica control algorithms and discuss

\(^3\)A similar discussion on providing different degrees of consistency for read-only transactions in distributed database systems is presented in [GMW82].
the problems in applying them on interdependent data. Finally, we review more recent work on
extensions to correctness criteria for mutual consistency.

3.1 Replica Control Algorithms

Many replica control algorithms have been proposed (see [SD85] for an extensive survey). All well
known algorithms maintain immediate consistency of the replicas by ensuring one copy serializability
[BHG87]. Based on the way one copy serializability is ensured, we classify the replica control
algorithms into two categories—those requiring update propagation and those requiring synchronous
multiple updates.

The algorithms requiring update propagation are primary copy [AD76] [Sto79] and token [MW82]
approaches. In the primary copy approach, one copy of a logical data is designated as the primary
copy, other copies are called secondary copies. The primary copy assumes the responsibility of
synchronization of the read and write operations. All reads and writes are first performed at the
primary copy and synchronized there. Updates are then propagated to all secondary copies. The
token algorithm [MW82] is an improvement of the primary copy by allowing the role of the primary
copy to switch from one copy to another. The problems in applying update propagation approach
on interdependent data are as follows. First, for these algorithms to be correct, updates applied
on the primary copy (or the copy with the token) must also be applied in the same order on
the secondary copies. However, it is very difficult in multidatabase systems to enforce this order
due to the local autonomy [DEL089] [SK89]. Furthermore, an update may be rejected because it
does not satisfy the local constraints. Second, each propagated update has to be packaged in a
(sub)transaction and then submitted to the designated site for execution. One can envision that
when a data is replicated in many databases and the update propagation approach is used, many
subtransactions will have to be created for performing one physical update on a replica (or a
member of an interdependent data). Third, to ensure that the propagated updates are executed
atomically, distributed commit over the set of update subtransactions (for propagated updates)
is needed, which is both very difficult (almost impossible if a high level of autonomy needs to be
maintained) and very expensive (even if it can be implemented). In our opinion, update propagation
should be avoided, if possible, or limited, if not possible, to lower the overhead.

There are numerous replica control algorithms which require synchronous multiple updates for
a single update on a logical data [Gif79], [ASC85], [BG84]. Among them, we choose the quorum
consensus (QC) algorithm [Gif79] as a representative and explain the problems in applying it to
managing interdependent data. In the QC algorithm, each copy of a logical data is assigned a
non-negative weight. Furthermore, for a logical data x, a read threshold RT and write threshold
WT are defined. A read (or write) quorum of x is any set of copies of x with a weight of at least
RT (or WT). Every transaction must collect a read quorum to read a data, and a write quorum
to write a data.

There are two problems in applying these algorithms to interdependent data. First, similar

\footnote{Some optimization might be possible by packaging all the propagated updates of a transaction to the same site in a single subtransaction; still, many subtransactions may be needed for the propagate updates.}
to update propagation, the multiple writes (or reads) induce numerous subtransactions. Unlike update propagation, the original transaction cannot complete until the subtransactions complete. Second, besides the difficulty in scheduling the execution of the induced subtransactions, it will also result in significant performance penalty (note that in the primary copy approach, the transaction that updates the primary need not wait for the propagated updates to commit).

In summary, the existing replica control algorithms cannot be directly applied to the interdependent data to provide adequate performance, particularly in systems involving heterogeneous and autonomous DBMSs.

3.2 Quasi Copy and Materialized View

The notion of quasi copy [ABGM90] has been introduced for modeling the consistency requirement of secondary copies stored in the user's local storages, called quasi copies, with respect to the copy stored in a central site, called source copy. Two characteristics of a quasi copy have been addressed: a selection condition and a coherency condition. A selection condition specifies how a quasi copy can be derived from the source copy and a coherency condition specifies how much the quasi copy can deviate from the source copy. Several useful coherency conditions have been formally defined, which are also applicable to the interdependent data. However, unlike the interdependent data stored in different databases, the quasi copy approach assumes that all information is controlled at a central site and all updates are performed at the central site.

To apply the quasi copy approach for the interdependent data, multi-site update are needed. Furthermore, to maintain the database consistency constraints which relate multiple data items, the updates on the source copies should be applied in the same order at their corresponding quasi copies. This is difficult in an environment in which local sites retain their autonomy.

The work on the materialized views [SF90] [SP89] [BCL89] are also relevant. A materialized view is a stored relation whose data is derived from the base relations. Unlike synchronized replicated data, update transactions to the base relation do not update the materialized relation. After the transaction that updates a base relation is committed, update transactions to the materialized views may be generated. Two problems in updating the materialized views are "when" and "how" to update the materialized views. [SF90] discusses the "when" issue, while [BCL89] and [SP89] discuss the "how" issue. Similar to the quasi copy approach, the materialized views approach requires that only the base relations can be directly updated, which limits its applicability to managing the interdependent data.

3.3 Identity Connection

A concept called identity connection [WQ90] has been introduced for modeling the update propagation of replicated data within autonomous and distributed database systems. The identity connection model allows specifying replication, permissible delay in enforcing the consistency of copies and data derivation rules for replicated data\(^5\). An identity connection links attributes of two

\(^5\)Our specification of interdependent data can be viewed as an extension of this concept.
relations or fragments of relations (called copies hereafter) that may be located at different sites, and specifies the consistency requirement between them. A temporal constraint is specified with every identity connection to specify when an update on one copy (called primary copy) should be applied on the other copies (called secondary copies). To maintain an identity connection, whenever an update occurs to the primary copy, a transaction is posted at the sites of the secondary copies with the temporal constraint associated the identity connection. The transaction scheduler at a secondary site decides whether it can satisfy the temporal constraint based on a satisfiability test. If the scheduler can satisfy the temporal constraint, it will promise the primary site to execute the transaction in accordance with the temporal constraint. The primary site can commit as soon as promises are obtained from all secondary sites.

This approach may require changes to LDBSs to support the feature of giving promise. This may require relaxing the design autonomy of LDBSs. The problem of synchronizing the update transactions on the secondary copies required for maintaining database consistency (see Section 4.2) has also not been addressed.

4 An Update Principle

In view of the difficulties in performing synchronized multiple writes on interdependent data, we propose a new update principle called update through current copy. The new principle requires a copy operation for every update to a member of an interdependent data and circumvents the problem of performing and synchronizing multiple writes. At the same time, due to the reduced number of writes for each update, this approach incurs low overhead. These advantages are possible, however, only when the mutual consistency criteria is eventual consistency that is weaker than immediate consistency of interdependent data.

4.1 Update Through Current Copy

The rationale behind update through current copy principle is that in many existing applications (see [SK89] for examples) deviation of the members of an interdependent data is acceptable (i.e., the members may lag) within a pre-determined limit. In contrast, all replicas in a replicated database must remain current to be useful. Because deviation on members is allowed for interdependent data, it becomes unnecessary to keep all members current if doing so will incur high overhead. In the meantime, it is required that at least one member remains current for converging the members of the interdependent data when the consistency constraint on the interdependent is violated (i.e., the divergence exceeds the limit). In the following, we will say that a member of an interdependent data is local to a transaction if it is managed by the database system to which the transaction is submitted. We will also use the notation $cp(D)$, read as “current copy of $D$”, to denote the current member of $D$.

---

6A replica can be lagging with respect to other replicas due to the failure on its site. However, when a failed site recovers, all replicas on that site are made current.
To apply update through current copy principle, we require each update to a member of an interdependent data to follow the following rule:

**update rule:** For a transaction $T$ to update member $d_j$ of an interdependent data $D$, the following sequence of operations are performed:

- $T$ reads the value from current copy of $D$; the read operation is denoted by $r_T(cp(D))$.
- $T$ writes the value of the current copy into the local copy $d_j$; the write operation is denoted by $w_T(d_j)$.
- Based on the computation, $T$ generates a new value for $d_j$ and then writes this value into the local copy $d_j$. This operation is denoted by $w_T(d_j)$.

After applying the update according to the above rule, the local copy $d_j$ becomes the new current copy of $D$.

If the members are large, copying current copy could be expensive. A more efficient implementation of update rule that does not require copying current copy is discussed in Section 6.1.

### 4.2 Consistency among Interdependent Data

So far we have only discussed the constraints among the members of a single interdependent data (i.e., $CON(D)$ and $DEP(D)$). Let us call these constraints *interdatabase constraints* since the members of an interdependent data are managed by different databases. However, there may also be constraints among members of different interdependent data within a single database\(^7\). Traditionally, the constraints among multiple data items are called *consistency constraint* [EGLT76]. To distinguish them from the interdatabase constraints, we will call the consistency constraints among data items (including those data items that are members of various interdependent data) in the same database as *internal consistency constraints*. Besides satisfying the interdatabase constraints, the internal constraints of the interdependent data must also be satisfied when ID update transactions are executed. Traditionally, when interdependent data do not exist, the consistency of the databases (i.e., the validity of the internal consistency constraints) is maintained by the concurrency controller of the local database systems. However, when interdependent data exist and the update through current copy is employed, the local concurrency controllers are not sufficient for maintaining the internal consistency of the interdependent data. This can be illustrated by the following example.

**Example 4.1** Suppose that there are two interdependent data $X$ and $Y$. As shown in Figure 1, $X$ consists of two members, $x_1$ at site 1 and $x_2$ at site 2; $Y$ also consists of two members, $y_1$ at site 1 and $y_2$ at site 2. Let us assume that currently $cp(X)=x_1$ and $cp(Y)=y_1$, and the members of both interdependent data are initialized to 10. Let us further assume that the internal consistency constraints are $x_i + y_i = 20$, $i = 1, 2$. Transaction $T_1$ at site 1 and transaction $T_2$ at site 2 are executed concurrently.

\(^7\)This is similar to the consistency constraints among the quasi copies of different data items[ABGM90].
Figure 1: An execution of two ID update transactions

\[
\begin{align*}
T_1: & \quad x_1 = x_1 - 5; \quad y_1 = y_1 + 5 \\
T_2: & \quad y_2 = y_2 - 5; \quad x_2 = x_2 + 5
\end{align*}
\]

One possible execution schedule of these two transactions is shown in Figure 1.

The execution starts from a consistent state (i.e., satisfying internal consistency constraints \( x_i + y_i = 20, \ i = 1, 2; \) and \( cp(X) + cp(Y) = 20 \)) and ends up in an inconsistent state \( (x_i + y_i = 15, \ i = 1, 2; \) and \( cp(X) + cp(Y) = 20 \)). Even though the internal consistency constraint on the current copies of the two interdependent data is satisfied (i.e., \( cp(X) + cp(Y) = 20 \)), the above execution is not desirable because it violates the internal consistency constraints on the local copies. Notice that if we execute \( T_2 \) after \( T_1 \) has finished or vice versa, all internal consistency constraints will still remain satisfied. The problem in the above execution is that \( T_1 \) and \( T_2 \) interleave their updates on the interdependent data inappropriately. This example shows that even though \( T_1 \) and \( T_2 \) are
located at different sites, they interact with each other through updating different members of the same interdependent data.

In general, internal consistency constraints cannot all be explicitly enumerated [EGLT76]. Instead, serializability has been used as a sufficient condition for preventing concurrent transactions from violating the internal consistency constraints. In the next subsection, we will present a criterion for reasoning about the correctness of concurrent ID update transactions.

4.3 Current Copy Serializability

Because an ID update transaction may access both interdependent and non-interdependent data in a local database, it is important that both interdependent and non-interdependent data remain consistent. We assume that each site uses a concurrency controller which maintains serializability of transaction execution at its site. For convenience, we define an abstract operation called \textit{id.access operation} to represent the sequence of operations (given in Section 4.1) for updating a member of an interdependent data. For transaction $T_k$ that updates local member $d_j$ of interdependent data $D$, the corresponding id.access operation is defined as

$$a_{T_k}(d_j) = (r_{T_k}(cp(D)), w_{T_k}(d_j), w_{T_k}(d_j)).$$

We say that two id.access operations $id.access$ conflict if they belong to two different ID update transactions and access members (that may or may not be different) of the same interdependent data. We then say that two ID update transactions $id.access$ conflict if they contain conflicting id.access operations. The term \textit{conflict} (without \textit{id.access} in front of it) continues to refer to traditional (read/write) conflict used for serializability. To apply the update through current copy correctly, two conditions must hold. First, an id.access operation must be executed exclusively against any conflicting id.access operation. That is, when the id.access operation starts it read operation (its first constituent operation), no conflicting id.access operation can start its read operation until the id.access operation finishes. This is required, otherwise the two corresponding ID update transactions may read from the same current copy and generate their own current copies, which causes the current copy to diverge.

The second condition is that the concurrent execution of a set of ID update transactions must satisfy a property called \textit{current copy serializability} (abbreviated as CPSR) which is defined as follows:

\textbf{Definition 4.1 (CPSR)} The execution of a set of ID update transactions $T = \{T_1, T_2, \cdots, T_n\}$ is said to be current copy serializable if there exists a total order in $T$ such that

1. for each pair of conflicting id.access operations $a_i \in T_i$ and $a_j \in T_j$, $a_i$ precedes $a_j$\footnote{The order of conflicting id.access operations is well defined because they are executed exclusively.} if and only if $T_i$ precedes $T_j$;

2. if $T_i$ and $T_j$ are at the same site and $T_i$ precedes $T_j$ in the total order then $T_i$ is serialized before $T_j$ by the local concurrency controller at their site.
Condition 2 in the definition of CPSR requires that the relative order of ID update transactions defined by CPSR is abided by the concurrency controller at each site.

CPSR is a sufficient condition for preserving internal consistency of the interdependent data. This can be argued as follows. We assume that each $T_i \in T$ when executed alone preserves the internal consistency constraints of all interdependent data and the internal consistency constraints of the database at the site at which the transaction is executed. That is, by seeing a consistent interdependent data and its local database, an ID update transaction will transform both the interdependent data and its local database from a consistent state into another consistent state. Now assume that an equivalent total order (there may be more than one) determined by CPSR (condition 1) is $T_1 \rightarrow \cdots \rightarrow T_i \rightarrow T_{i+1} \rightarrow \cdots \rightarrow T_n$. Consider two consecutive ID update transactions $T_i$ and $T_{i+1}$ in the order. If $T_i$ and $T_{i+1}$ are at different sites, then $T_i$ can only have id_access conflict with $T_{i+1}$ through accessing the interdependent data. Since CPSR requires that $T_i$ executes the conflicting id_access operations of $T_i$ and $T_{i+1}$ before $T_{i+1}$ does, the concurrent execution of $T_i$ and $T_{i+1}$ is the same as that of $T_{i+1}$ executes after $T_i$ has finished. If $T_i$ and $T_{i+1}$ are at the same site\footnote{In this case, if $T_i$ id_access conflicts with $T_j$ then $T_i$ also (read/write) conflicts $T_j$ because they access the same member.}, $T_i$ and $T_{i+1}$ may also conflict through access of non-interdependent data. However, due to condition 2, $T_i$ must be serialized before $T_{i+1}$ by the local concurrency controller. According to serializability, this in effect is the same as executing $T_i$ and $T_{i+1}$ serially. In both cases, the concurrent execution of $T_i$ and $T_{i+1}$ is the same as that if we have executed $T_{i+1}$ after $T_i$ has finished. Applying the same argument to all pairs of consecutive ID update transactions in the equivalent total order, the effect of concurrent execution of $T$ is the same as that of executing them serially in the equivalent total order. Since each ID update transaction, when executed alone, transforms the interdependent data and its local database from one consistent state into another consistent state, a serial execution of a set of ID update transactions will transform both the interdependent data and the related local databases from an initially consistent state into another consistent state. Since the effect of a CPSR execution is the same as that of a serial execution, we conclude that a CPSR execution of $T$ will also transform both the interdependent data and the related local databases from an initially consistent state into another consistent state. In other words, a CPSR execution of $T$ preserves the internal consistency of both the interdependent data and the related local databases.

To validate the current copy serializability property, we can employ a graph called current copy serialization graph (CPSG) which is defined as follows:

**Definition 4.2** A current copy serialization graph (CPSG) over a set of ID update transactions $T = \{T_1, T_2, \cdots, T_n\}$ is a graph whose nodes are transactions in $T$ and whose edges are all $T_i \rightarrow T_j$ ($i \neq j$ and $T_i, T_j \in T$) if $\exists aT_i$ (of $T_i$) and $aT_j$ (of $T_j$), such that $aT_i$ and $aT_j$ conflict and $aT_i$ precedes $aT_j$.

**Theorem 4.1** The execution of a set of ID update transactions is current copy serializable if and only if its corresponding CPSG is acyclic.

**Proof:** The proof is similar to that of serializability theorem in [BHG87] and is omitted here. □
5 Algorithms

In this section, we present the model for interdependent data management system, an access control algorithm for maintaining CPSR over concurrent ID update transactions, and a convergence control algorithm for enforcing eventual consistency.

5.1 System Model

As shown in Figure 2, a system for interdependent data management consists of an ID lock manager and a set of local database systems (LDBSs), each with an interdependent data manager (IDM). An IDM is an interface to an LDBS which detects the access to the interdependent data and controls the access to satisfy CPSR. IDMs control access to the interdependent data by requesting locks (called ID locks) on the interdependent data. The ID locks are collectively controlled by the ID lock manager. For the interdependent data access control, subtransactions of distributed transactions are treated in the same way as the local transactions. Back-ups and distributed implementation of the ID local manager are possible for better availability but with higher complexity, just as for a lock manager for distributed transactions, but we do not discuss such alternatives in this paper.

There are two ways by which an IDM can detect access to interdependent data. In this paper, we take the approach that involves slight modification of the transactions to indicate the interdependent data they access. The modification uses Mark and UnMark operations described in the next subsection. The second approach is to make IDMs more intelligent and analyze the transactions to identify any accesses to interdependent data, and treat such access differently. Such an approach is currently being investigated [RS91].

Note that the system model does not sacrifice autonomy of the LDBSs since they (including their interfaces) do not change in any way. The system model also supports an easy evolution from a system that does not support the mutual consistency of interdependent data stored in multidatabases to the one that does. Only difference, besides adding an IDM over each LDBS is...
to slightly modify the transactions that access any interdependent data. This is a small penalty to pay if the access to local data items that are members of any interdependent data are to be supported with low overhead.

5.2 Access Control

To control access to the members of interdependent data, we have to ensure that the execution of id_access operations (discussed in subsection 4.3) are atomic and the concurrent execution of ID update transactions is CPSR. A lock based method is described below.

5.2.1 Locks on Interdependent Data

A transaction declares its intention to access interdependent data by issuing Mark/UnMark primitives. The IDM sends each primitive to the ID lock manager that manages the locks and ensures eventual consistency of interdependent data. Three types of primitives and corresponding lock types are used. W-Mark/W-UnMark primitives and W-locks are used by ID update transactions. Rs-Mark primitive and Rs-locks (for strong read locks) are used by ID read-only transactions that need to read current copy only. Rw-Mark primitive and Rw-locks (for weak read locks) are used by ID read-only transactions that may read lagging members of interdependent data. Once the ID lock manager succeeds in acquiring the locks, it succeeds the Mark primitive. The IDM then submits the transaction to the LDBS that executes the transaction by accessing local members of interdependent data and other data items that may not be a part of any interdependent data\textsuperscript{10}.

The compatibility matrix of the three lock types is given in Figure 3. A W-lock conflicts with W-lock or Rs-lock (on the same data). In addition to the locks, the ID lock manager maintains a current copy indicator for each interdependent data, which indicates the member that is the current copy at any given instance.

\textsuperscript{10}The distributed transaction execution mechanism ensures that a subtransaction is sent to the IDM of an LDBS that has a member of the interdependent data. The global execution mechanism does not need to know where all members of an interdependent data are. This is a matter of how much optimization is performed.
5.2.2 Primitives

The ID locks are set at the beginning of the transaction execution. There are two reasons for this. First, due to the local autonomy of LDBSs, we do not know when a transaction will actually need to access the interdependent data. Second, by locking the interdependent data in advance, we can prevent deadlock due to the locking on the interdependent data. Deadlock detection is a subtle problem even in traditional distributed database systems. In multidatabase systems, the problem is further aggravated by the local autonomy.

For ID update transactions submitted to IDM at site \( j \) that intends to update members of interdependent data \( X_1, X_2, \ldots, X_n \), we use the following set of Mark/UnMark primitives:

\[
W-Mark(x_{1j}, x_{2j}, \ldots, x_{nj}) \ (\text{where } x_{ij} \text{ is the local member of } X_i \text{ at site } j)
\]

when the ID lock manager receives this primitive, attempts to set \( W \)-locks on interdependent data \( X_1, X_2, \ldots, X_n \). If any conflict mode of lock has been set on a requested interdependent data, the ID lock manager will reject the primitive and reclaim all the locks granted to the requesting transaction. The rejected primitive will be delayed for a certain time (which blocks the execution of the requesting transaction), and then re-executed from the beginning (i.e., start requesting its first lock and so on). If all requesting locks are available, the lock manager ensures the mutual consistency of each interdependent data by invoking consistency enforcer discussed in Section 5.3. It then succeeds the primitive and returns to the IDM the values of the current copy indicators of the related interdependent data. The IDM first makes the local copies current (by issuing the read and first write in each corresponding id_access operation or by using the efficient update approach in subsection 6.1); then continues the execution of the transaction. The current copies of different interdependent data may be managed by different LDBSs.

If the execution of an ID update transaction fails, the IDM notifies the ID lock manager which unlocks the \( W \)-locks set by the failed ID update transaction. In this case, some of the local members may have written the up-to-date contents from the current copies; however, the current copy indicators remain unchanged.

\[
W-UnMark(x_{1j}, x_{2j}, \ldots, x_{nj})
\]

when the ID lock manager receives this primitive, it sets the currency indicators to indicate the local members \( x_{1j}, x_{2j}, \ldots, x_{nj} \) as the new current copies and release (by issuing the \( W \)-unlocks) the \( W \)-locks on \( X_1, X_2, \ldots, X_n \).

The above set of primitives are used as follows:

**ID Update transaction:**

\[
\text{Begin_Transaction;}
\]

\[
W-Mark;
\]

\( (\text{original}) \text{ transaction code;} \)

\[
W-UnMark;
\]

\[
\text{End_Transaction.}
\]

If a transaction already existed for a system that did not manage interdependent data, it remains unchanged, except for enveloping it with the set of primitives (this is indicated by "(original)"

14
An ID read-only transaction requiring strong reads is called **strong ID read-only transaction**. It uses the following Mark primitive.

\[ R_s\text{-Mark}(x_{1j}, x_{2j}, \ldots, x_{nj}) : \]

when the ID lock manager receives this primitive, it will attempt to set \( R_s \)-locks on \( X_1, X_2, \ldots, X_n \). If any conflicting lock has been set on any of \( X_1, X_2, \ldots X_n \), the ID lock manager will reject the primitive by delaying it and executing it again from the beginning. Otherwise, the consistency enforcer (Section 5.3) is invoked, and then the \( R_s \)-Mark succeeds. Similar to the \( W \)-Mark primitive, the \( R_s \)-Mark primitive will first make the local copies current and then continues the execution of the transaction. Unlike the \( W \)-Mark, the ID lock manager will release the \( R_s \)-locks on \( X_1, X_2, \ldots, X_n \) immediately after the primitive succeeds.

A strong ID read-only transaction is formatted as follows:

**Strong ID read-only transaction**:

\[
\text{Begin\_Transaction} \\
R_s\text{-Mark;} \\
(\text{original) transaction code;} \\
\text{End\_Transaction}
\]

The ID read-only transaction which needs only the values of the local members of interdependent data (which may be lagging) is called **weak ID read-only transaction** and uses the following Mark primitive.

\[ R_w\text{-Mark}(x_{1j}, x_{2j}, \ldots, x_{nj}) : \]

when the ID lock manager receives this primitive, it will not set any lock because the weak read lock does not conflict with any other lock. The purpose of this primitive is to invoke the consistency enforcer (Subsection 5.3) to enforce the eventual consistency of the interdependent data.

A weak ID read-only transaction is formatted as follows:

**Weak ID read-only transaction**:

\[
\text{Begin\_Transaction} \\
R_w\text{-Mark;} \\
(\text{original) transaction code;} \\
\text{End\_Transaction}
\]

So far we have only discussed the access control for local transactions and subtransactions. For global (i.e., distributed) transactions that update any interdependent data, the \( W \)-Mark and \( W \)-UnMark should be applied with each of its subtransactions. For global transactions that only read interdependent data, users have the flexibility to specify which subtransactions need current data, and which do not, by using the \( R_s \)-Mark and \( R_w \)-Mark primitives, respectively.
5.2.3 Correctness

To show the correctness of the proposed algorithm, we make the following observations. First, the id..access operation is executed atomically as required. This is because the corresponding W-Mark primitive will set a W-lock on each intended interdependent data before an ID update transaction starts its execution, and the W-lock will not be released (using the W-UnMark primitive) until it commits. Therefore, no conflicting id..access operations can be executed concurrently. Second, the current copy serializability is maintained. This is intuitively true since an ID update transaction, say $T_i$, will not start its execution before acquiring all W-locks it needs on the interdependent data it intends to update and will not release the acquired W-locks until it terminates (commit or abort). This precludes the possibility that other ID update transactions concurrently execute any id..access operations which id..access conflict with those of $T_i$. In other words, ID update transactions are executed serially with respect to the conflicting id..access operations.

To prove the above more formally, we have to show that the two conditions of CPSR hold. We first assume that condition 1 of CPSR does not hold. Then there must exist a cycle in the CPSG. Assume that the cycle consists of $T_1 \rightarrow T_2, \ldots, \rightarrow T_k \rightarrow T_1$, where $T_1, \ldots, T_k$ are some ID update transactions in the system. Since for $T_1 \rightarrow T_2$, there must exist $a_1$ in $T_1$ and $a_2$ in $T_2$ such that $a_1$ id..access conflicts with and precedes $a_2$. Due to the proposed W-Mark and W-UnMark primitives, $T_1$ will not release the lock for $a_1$ until it terminates. Therefore, $T_2$ has to wait for $T_1$ before it can execute $a_2$. Reasoning in the same way, we have $T_2$ waiting for $T_1$, $T_3$ waiting for $T_2$, \ldots, $T_k$ waiting for $T_{k-1}$ and $T_1$ waiting for $T_k$. This implies a deadlock among $T_1, \ldots, T_k$. However, due to the no waiting policy (if some lock is not available, the requesting primitive is restarted), a deadlock cannot exist; therefore, no cycle in CPSG can exist. This concludes that condition 1 holds. To show that condition 2 of CPSR also holds, we assume that $T_i$ and $T_j$ are two ID update transactions at the same site and the relative order determined by CPSR is $T_i \rightarrow T_j$. Then there must exist a set (may be empty, in this case either $T_i$ id..access conflicts with $T_j$ or the relative order is undefined) of ID update transactions $T_{r_1}, T_{r_2}, \ldots, T_{r_n}$ such that $T_i$ id..access conflicts with and precedes $T_{r_1}$ which id..access conflicts with and precedes $T_{r_2}$ which id..access conflicts with and precedes $\ldots$ $T_{r_n}$ which id..access conflicts with and precedes $T_j$. According to the access control algorithm, $T_{r_1}$ cannot start its execution until $T_i$ has finished. $T_{r_2}$ cannot start its execution until $T_{r_1}$ has finished and so on. Finally, $T_j$ cannot start its execution until $T_{r_n}$ has finished. This implies that $T_j$ cannot start its execution until $T_i$ has finished. $T_i$, therefore, is effectively serialized before $T_j$.\[\Box\]

5.3 Ensuring Eventual Consistency

In this subsection, we present an algorithm for ensuring the eventual consistency constraint of the interdependent data.

\[\text{\footnote{This may not be true for some local concurrency control algorithms; however, it is true for many important concurrency control algorithms [BGRS90].}}\]
5.3.1 General Approach

Two issues need to be addressed to maintain the eventual consistency on interdependent data. First, we need to accurately detect when an eventual consistency is violated. Second, once a violation is detected, we have to promptly restore the consistency of the interdependent data such that the inconsistent interdependent data will not be used by any application (i.e., concurrent transactions).

We observe that data inconsistency (due to the violation of mutual consistency constraint) is harmless if access to inconsistent data is prevented until they are made consistent. We use the Mark primitives to acquire the locks (to ensure, using CPSR, that update on interdependent data do not violate local consistency of data) as well as ensure the mutual consistency of interdependent data. After acquiring the appropriate locks, the ID lock manager validates and enforces the eventual consistency constraint by invoking a procedure called \textit{consistency enforcer}.

5.3.2 A Consistency Enforcer

The consistency enforcer presented here supports a limited type of eventual consistency constraint, consisting of time (C\textsubscript{t}) and operation (C\textsubscript{o}) aspects, and their conjunction and disjunction. The eventual consistency constraints are stored in a \textit{constraints table}, indexed by its corresponding interdependent data. This table is maintained by the ID lock manager. An entry in the table contains the type of the constraint and two parameters specified in an eventual consistency constraint, \textit{Interval} and \textit{Update_limit}. \textit{Interval} is the time interval since the last convergence within which the members of the interdependent data should converge. \textit{Update_limit} is the maximum number of updates allowed since last convergence after which the members have to converge (before any additional update is performed). Members of an interdependent data converge by performing a \textit{convergence action} that makes all members current up to the previous update on that interdependent data. More complex types of \textit{convergence action} are discussed in [RSK91]. The consistency enforcer uses two more parameters: \textit{Convergence time} and \textit{Count}. We assume that both of them are kept in the constraint table. \textit{Convergence time} indicates the next expected convergence time which is initialized to the time when the interdependent data is first introduced to the system plus the value of \textit{Interval}. \textit{Count} keeps track of the number of updates performed on the interdependent data since the last convergence action, and is initialized to 0. Both parameters are initialized when an interdependent data is first introduced into the system and after every convergence action. The ID lock manager uses the following procedure call:

\begin{verbatim}
call consistency_enforcer( ID_i, lock_type, C_type, Convergence_time, Interval, Update_limit, Count)
\end{verbatim}

The \textit{ID} is the interdependent data identifier. The \textit{lock_type} is the type of lock request (i.e. \textit{Rw}-lock, \textit{Rs}-lock or \textit{W}-lock) or unlock request (\textit{W}-unlock). \textit{C_type} is one of the four types of eventual consistency constraints (\textit{C_o}, \textit{C_t}, \textit{C_o} \& \textit{C_t}, \textit{C_o} \lor \textit{C_t}). An algorithm for consistency enforcer, shown in Figure 4, validates the eventual consistency constraint according to the type of the constraint (\textit{C_type}). The rest of the parameters, discussed above, are initialized from the consistency table.
procedure consistency_enforcer(IDJ, lock_type, C_type, Convergence_time, Interval, Update_limit, Count);
begin
    case C_type of
    C.O:
        if (Count > Update_limit))
            then
                begin
                    call convergence.action;
                    Count ← 0;
                end
        else
            if (lock_type = W-unlock) then Count ← Count + 1; endif;
        endif;
    C.1:
        if ((actual_time() >= Convergence_time))
            then
                begin
                    call convergence.action;
                    Convergence.time ← actual_time() + Interval;
                end
        else
            if (lock_type = W-unlock) then Count ← Count + 1; endif;
        endif;
    C.D ∧ C.1:
        if ((actual_time() >= Convergence.time) AND (Count > Update_limit))
            then
                begin
                    call convergence.action;
                    Consistency.time ← actual_time() + Interval;
                    Count ← 0;
                end
        else
            if (lock_type = W-unlock) then Count ← Count + 1; endif;
        endif;
    C.D ∨ C.1:
        if ((actual_time() >= Convergence.time) OR (Count > Update_limit))
            then
                begin
                    call convergence.action;
                    Consistency.time ← actual_time() + Interval;
                    Count ← 0;
                end
        else
            if (lock_type = W-unlock) then Count ← Count + 1; endif;
        endif;
end /*end of case*/
write Count and Convergence.time back to the consistency table;
end

Figure 4: A consistency enforcer
For a consistency constraint which contains only the operation aspect \((C_0)\), if the locking request is \(W\)-lock (from ID update transaction) and the number equals \(Update\_limit\) (statement C.1 in Figure 4.1), granting the respective request will invalidate the constraint; therefore, convergence action is performed (by calling the \(convergence\_action\)) before the request can be granted. After the convergence action is completed, \(Count\) is set to 0. Whenever a constraint contains the operation aspect, an unlock request \(W\)-unlock (not \(W\)-lock) will cause the \(Count\) to be increased by one (statement C.2). This is because only when an ID update transaction issues \(W\)-unlock we know for sure that the corresponding update has been performed. Granting of \(W\)-lock request does not guarantee that the corresponding update will be performed (the corresponding \(W\)-Mark may still be rejected). For a consistency constraint which contains only the time aspect \((C.T)\), if the current time passes the next expected convergence time, a convergence action is performed, and the next convergence time is set to the current time (obtained by calling procedure \(actual\_time()\)) plus the value of \(T\_interval\) (statement C.3). Figure 4 also shows how the consistency constraint types that include conjunction and disjunction of the operation and time aspects are handled (statement C.5 and C.6). After the consistency enforcer finishes checking the eventual consistency, it will write the updated values of \(Count\) and \(Convergence\_time\) back to the consistency table (statement C.7). Note that statement C.4 keeps the number of updates performed since last convergence action. This information is not used for enforcing the \(C.T\) type eventual consistency constraint; however, it helps the design of an efficient update through current copy principle (in Section 6).

The immediate consistency on an interdependent data can be easily implemented by setting the \(C\_type = C_0\) and \(Update\_limit = 0\). Then, a successful update (signaled by a \(W\)-unlock) will set \(Count\) to 1; subsequently, any access (by issuing \(W\)-lock, \(R_w\)-lock or \(R_u\)-lock) to the interdependent data will satisfy condition \((Count > Update\_limit)\) and therefore invoke the \(convergence\_action\) which in turn makes the values of all members of the interdependent equal. As a result, the effect of any update on the interdependent data is immediately visible to the on-coming read and write operations.

Our approach of performing convergence after the mutual consistency criteria is violated (but before any one else can access members of interdependent data among which the specified mutual consistency is violated) can be characterized as a lazy approach. It is possible to modify this to a more eager approach in which the convergence may be performed (perhaps in background or when no lock/unlock request is being processed) even before the eventual consistency is violated. This approach could reduce the time to process some lock requests but may result in more updates on members than in the lazy approach.

6 Extensions

In subsection 6.1, we discuss a more efficient approach to implement the update rule than copying the current copy. In the previous sections, we have assumed that each member of an interdependent data has the same consistency requirement. In subsections 6.2 and 6.3, we discuss some simple relaxations of this condition. In subsection 6.4, we discuss a special type of interdependent data called derived data. The corresponding extensions to the algorithms incur minor cost and may be
useful in some multidatabase systems.

6.1 An Efficient Update Approach

According to the update through current copy principle, in order to update a member of an inter­
dependent an ID update transaction has to copy the value of the current copy into its local copy. This operation may be very costly when the members are large (e.g., a relation). This could be avoided as follows

For each interdependent data, the ID lock manager keeps an order list, called Update List (see Figure 5), of all updates performed on the interdependent data since the last convergence. Each update in the list is indexed by a number called version number. The IDM at each site keeps for each of its local member of every interdependent data a local version number corresponding to the last update since the convergence performed on that member. The version number \( i \) in the Update List implies that if we apply the updates indexed 1 through \( i \) on the last converged version (or copy), we will get the \( i \)th version of the interdependent data. If the local version number of a member is \( j < i \) where \( i \) is the version number of the last update in the Update List, then it can be made current by executing updates \( j+1 \) through \( i \). After an ID transaction is successfully updated, the local version number is incremented and version number at the lock manager are incremented and the update is added to the update list.

The advantages of this approach include that no copy operation needs to be performed for updating a member of an interdependent data and that there is no need to keep the current copy information. The disadvantages is that we have to keep the Update List, and, in order to do so, we have to detect and log the update statement performed by the ID update transaction. This can be done by requiring the IDM to perform some analysis on ID update transactions before they are actually executed.
6.2 Immediate Consistency for Some Members

In some situations, all members of an interdependent data may not be equal. In practice, some members require that updates on any other members have to reflect on them immediately. These members are called immediate members. It is very easy to support the immediate members by our approach by keeping an immediate list for every such interdependent data. The list records the names (or locations) of the immediate members. We require the ID lock manager to update the members in the list when a related update succeeds (signaled by an $W$-unlock request) to the content of the current copy (and subsequently release the write locks). The immediate members can be used to enhance the reliability of the proposed algorithm. Since the immediate members contain the same contents as the current copy, when the site containing the current copy fails, we can choose any immediate member as the new current copy. We do not address this issue further.

6.3 Read-only Members

Another useful extension is to support access control. We can make some members of an interdependent data to be read-only to the user. The read-only members can only be updated by convergence action. This feature is useful since in a multidatabase system, some LDBSs may be endowed with the right to control (and therefore to update) an interdependent data\footnote{The LDBS that controls an interdependent data is said to be a steward of that data.}, while some may not. This feature can be used to control the sharing of information among the LDBSs. To support this feature, we can keep a prohibit list for each interdependent data. When a $W$-lock request is received, the ID lock manager first refers to the prohibit list; if the member from which the locking request is issued is in the list, the request is rejected.

6.4 Derived Data

A derived data is derived by performing aggregation on other data, called source data (similar to materialized views). Typically, no update can be directly applied to the derived data. Instead, updates performed on the source data should be reflected on the derived data when a prespecified condition is satisfied. In general, we model a derived data $d$ as a 3-tuple

$$(\{S\}, E, C)$$

where $S$ is the set of source data, $E$ is an expression on $S$ which derives $d$ from $S$, and $C$ is the associated eventual consistency constraint. No update control is required for the derived data since it is not updated directly. To satisfy $C$, we use the proposed consistency enforcer. We can use the read operation on the derived data to "trigger" the consistency checking of the consistency enforcer. When $C$ is violated, a read operation will be blocked by the consistency enforcer until $E$ has been applied to derive new value for the derived data.
Most industrial computing environments have heterogeneous and autonomous databases. These databases have many interdependent (also called redundant) data, whose mutual consistency must be maintained without requiring a total consolidation or integration of these databases. Managing such interdependent data is different from the replica control in (homogeneous) distributed database systems in several ways, including environmental differences (heterogeneity and autonomy), type of dependencies among related data (complex interdependencies vs identical copies), the reasons for the existence of related data (managing the consistency among existing related data vs inducing replica to improve availability and/or read access time), and the desired or required level of consistency among related data.

The immediate mutual consistency requirement such as one copy serializability is unnecessary and almost impossible to provide with reasonable efficiency and without sacrificing autonomy. Instead, we recognize and exploit two types of information for managing consistency of interdependent data. First, it is not necessary to have immediate consistency of all interdependent data. Instead we support the flexible mutual consistency criterion of eventual consistency. Eventual consistency allows specification of different application dependent consistency requirement (that can be varied from immediate consistency to no consistency) on different interdependent data. Second, it is not necessary for some interdependent data read-only transactions to read the most up-to-date copy of the data. This is supported by the distinction between strong and weak reads.

To our knowledge, we propose the first complete approach for managing eventual consistency of interdependent data in a multidatabase system. The approach includes the principle of update through current copy, the access control algorithm that ensures local consistency of the interdependent data, and the consistency enforcer for guaranteeing eventual consistency of interdependent data. The update through current copy leads to significantly fewer writes as opposed to the previous replica control algorithms when immediate consistency is not required. This is because only one current copy need to be kept. Furthermore, distributed commit is not required for every update to an interdependent data. Using the consistency enforcer, different degrees of eventual consistency can be supported for different interdependent data. However, if most interdependent data require immediate consistency, the performance of the algorithm may not be good due to the overhead of the consistency enforcer.

An alternative approach that uses a declarative specification of interdependent data (including dependent and mutual consistency specifications) to automatically generate updates to other members of interdependent data when any member is updated, is being investigated [RS91]. The fault tolerance and recovery aspects need to be investigated in more detail. We are also investigating the complexity of interdependency (including the types of eventual consistency requirements) in real world databases supporting real industrial applications.
References


