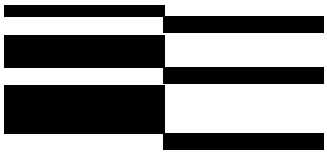
14



Using Polytransactions to Manage Interdependent Data

Amit P. Sheth

Bellcore, RRC 1J210
444 Hoes Lane
Piscataway, NJ 08854-4182
amit@ctt.bellcore.com

Marek Rusinkiewicz, George Karabatis

Department of Computer Science
University of Houston
Houston, TX 77204-3475
<marek,george>@cs.uh.edu

14.1 Introduction

Many large companies use multiple databases to serve the needs of various application systems. One of the significant problems in managing these databases is maintaining the consistency of inter-related data in an environment consisting of multiple semi-autonomous and heterogeneous systems. We use the term *interdependent data* to imply that two or more data items stored in different databases are related through an integrity constraint that specifies the data dependency and the consistency requirements between these data items. Management of such data implies that a certain degree of mutual consistency among the interdependent data is maintained. Therefore, the manipulation (including concurrent updates) of the interdependent data must be controlled.

In the majority of existing applications, the mutual consistency requirements among multiple databases are either ignored, or the consistency of data is maintained by the application programs that perform related updates to all relevant databases. This can be accomplished using various techniques. For example, a message may be sent to another database system managing related data, so that a complementary transaction will be submitted there, or a replica of the data is sent to another system, either electronically or physically (e.g., a tape). However, these approaches have several disadvantages. First, they rely on the application programmer to enforce integrity constraints and to maintain mutual consistency of data, which is not acceptable if the programmer has incomplete knowledge of constraints to be enforced. Secondly, a modification of a part of an application requires changing of other parts of the same or another application to maintain integrity and consistency. Since integrity requirements are specified within an application, they are not written in a declarative way. If we need to identify these requirements, we must extract them from the code, which is a tedious and error prone task.

Alternative approaches to the problem, are based on system supported maintenance of mutual consistency. A possible solution is to enhance the techniques of preserving integrity that were proposed for distributed databases [SV86]. The main limitation of these techniques is that they assume that the consistency between the related data must be restored immediately. However, in loosely-coupled environments we may need to temporarily tolerate inconsistencies among related data. Active databases [MD89] address this problem by allowing evaluation of time constraints, in addition to data-value constraints. They use object oriented techniques to encapsulate the maintenance of con-

sistency inside the methods.

With the current emphasis on data as a corporate asset, whose integrity is of basic importance [Mil89], the management of interdatabase consistency is receiving more attention. Distributed transaction technologies can be used to address some of these problems. Transaction management systems, based on the traditional transaction concept [Gra81, HR83], or its extensions (nested transactions [Mos85], sagas [GMS87], multidatabase transactions [Geo90, GRS91, Reu89]), can be used to preserve database consistency. Transaction technology can assure that all actions needed to preserve the interdatabase dependencies are executed in an atomic way. However, the specification of these actions and, hence, the correctness of the multidatabase updates still rests with the application programmer, who is responsible for the design of the transaction. Also, distributed concurrency control and commitment present serious problems when long-lived transactions span across systems with vastly different capabilities.

In this chapter, we discuss a framework for maintaining consistency among interdependent data stored in multiple databases. Two important features of the proposed framework are: (a) declarative specification of interdependent data and their mutual consistency requirements, and (b) use of the specification to automatically generate related update transactions that manage interdependent data.

The separation of the constraints from the application programs, facilitates the maintenance of data consistency requirements and allows flexibility in their implementation. It also allows investigation of various mechanisms for enforcing the constraints, independently of the application programs. Thus, changes in the application programs can be made independently of the constraints, and vice versa. By grouping the constraints together, we can check their completeness and discover possible contradictions among them.

The chapter is organized as follows. In Section 14.2, we define the data dependency descriptors which are stored in the *Interdatabase Dependency Schema (IDS)*. In Section 14.3, we introduce the concept of polytransactions. We describe a possible system architecture for management of interdependent data and we specify how polytransactions can be derived from the dependency schema. Section 14.4 provides a more detailed description of the various components of data dependency descriptors and discusses some aspects of correctness of the *IDS*. In Section 14.5 we classify different degrees of consistency among interdependent data. Section 14.6 provides the conclusions.

14.2 Specification of Interdatabase Dependencies

In this section we will discuss briefly the specifications of interdatabase dependencies[1]. They can be used to perform updates in a way that allows us to overcome some of the limitations imposed by traditional transaction management systems. For example, we may allow data to be temporarily inconsistent, within specified limits.

Our framework for specifying interdatabase dependencies consists of three components: data dependency information, mutual consistency requirements, and consistency restoration procedures [RSK91]. While these components have been addressed in the literature separately, in our opinion they represent facets of a single problem that should be considered together. Data dependency conditions are similar to integrity constraints [SV86], although they do not require that full integrity must be maintained at all times. In addition to *immediate consistency*, mutual consistency requirements can be used to specify more relaxed criteria of *eventual and lagging consistency* that have been introduced recently [SK89, RSK91]. Other weak mutual consistency criteria have also been defined (see Section 14.4). Actions to restore consistency between interdependent data were used extensively in active databases [DBB+88, HLM88].

We use *Data Dependency Descriptors* (D^3) to specify the interdatabase dependencies [RSK91]. They can be viewed as an extension of the *identity connection* proposed by Wiederhold and Qian [WQ87]. Each D^3 consists of identification of related objects and a directional relationship defined in terms of three components discussed earlier. A D^3 is a 5-tuple:

$$D^3 =< \mathcal{S}, U, P, C, \mathcal{A} >$$

where:

$\mathcal{S}$ is the set of *source data objects*,

U is the *target data object*,

[1] Before we can specify the interdatabase dependencies, we must eliminate incompatibilities that may exist among related data items in different databases. The corresponding data items in different databases, may have different names, and may be defined using different data types and/or units [DH84, DAT87]. We will not address the problem of resolving data incompatibility in this chapter. Rather, we assume that there is a method [RCE+88], possibly involving the use of a dictionary, thesaurus, knowledge base, and/or an auxiliary database, which may be used to identify the correspondences between the related items and to resolve the incompatibilities among them.

P is a boolean-valued predicate called *interdatabase dependency predicate* (dependency component). It specifies a relationship between the source and target data objects, and evaluates to true if this relationship is satisfied.

C is a boolean-valued predicate, called *mutual consistency predicate* (consistency component). It specifies consistency requirements and defines when P must be satisfied.

$\mathcal{A}$ is a collection of *consistency restoration procedures* (action component). Each procedure, specifies actions that must be taken to restore consistency and to ensure that P is satisfied.

Interdatabase dependency descriptors are uni-directional, from the set of source objects to the target object. While the objects specified in $\mathcal{S}$ and U may belong to the same database, we are particularly interested in those dependencies in which the objects belong to different databases and are managed by independent database management systems.

An operation on either source or target data objects specified in a D^3 may require additional actions to maintain mutual consistency of interdependent data. Consistency predicates involving only operations performed on the source data objects, can be referred to as **push constraints**. In this case, an operation applied to one or more source data objects propagates its effects to the target data object. If a consistency predicate involves only operations on the target data object, we refer to it as **pull constraint**. In this case, the results of earlier updates (if any) to the source data objects are propagated to the target data objects, before the operation specified in the consistency predicate is performed. The implications of push and pull constraints are further discussed in the next section.

D^3s are specified in a declarative fashion and can be viewed as separate schema entities. An *Interdatabase Dependency Schema (IDS)* is a set of all D^3s to be enforced in a multidatabase system. A more detailed discussion of the structure of dependency descriptors is presented in Section 14.4.

14.3 Polytransactions for Managing Interdependent Data

As mentioned earlier, multidatabase transactions can be used to preserve

consistency of multiple databases. Multidatabase transaction management addresses the issues of *global* and *local* consistency in such an environment. However, one problem with the concepts of nested transactions [Mos85] and multidatabase transactions (e.g., "contracts" [Reu89], "flexible transactions" [ELLR90], and "multitransaction activities" [GMGK+90]) is the assumption that all subtransactions, as well as all execution dependencies between them are known in advance. While such assumptions may be realistic in some cases, they limit the applicability of these concepts in environments consisting of interdependent data. This is because the information about all relationships between data stored in multiple data repositories may not be known in advance to the transaction designer. Therefore, in this section we will introduce the concept of *polytransactions* and discuss their use to manage interdependent data. The objective of a polytransaction is to preserve the *mutual* consistency requirements between interdependent data defined in the IDS.

14.3.1 System Architecture

A typical multidatabase system architecture assumed in the literature consists of two levels. The local databases comprise the lower level and the global transaction manager acts as a coordinator of global transactions. A multidatabase transaction is decomposed into a collection of subtransactions submitted to the local systems. When all the subtransactions successfully commit, the global transaction succeeds. In contrast, in this chapter we are concerned with the effects that an update operation submitted to a single database may have on the related data. In general, whenever a change occurs in data stored in one database, the interdependent data may have to change. Hence, we view transactions as means to satisfy the dependency and consistency predicates defined in the IDS.

A possible system architecture that can be used to maintain interdependent data objects is illustrated in Figure 14.1. Every database participating in a multidatabase environment is augmented with a *Dependency Subsystem (DS)* which acts as an interface between different databases. DSs at different sites can communicate with each other. When a transaction is submitted for execution, the DS consults the IDS to determine whether the data accessed by the transaction are dependent on data in other databases. The IDS itself can be either centralized, or distributed over the local sites. In the latter case, only those dependency descriptors that have their source data objects stored locally, may be kept in each fragment.

A transaction submitted to a DS, is analyzed before being executed by

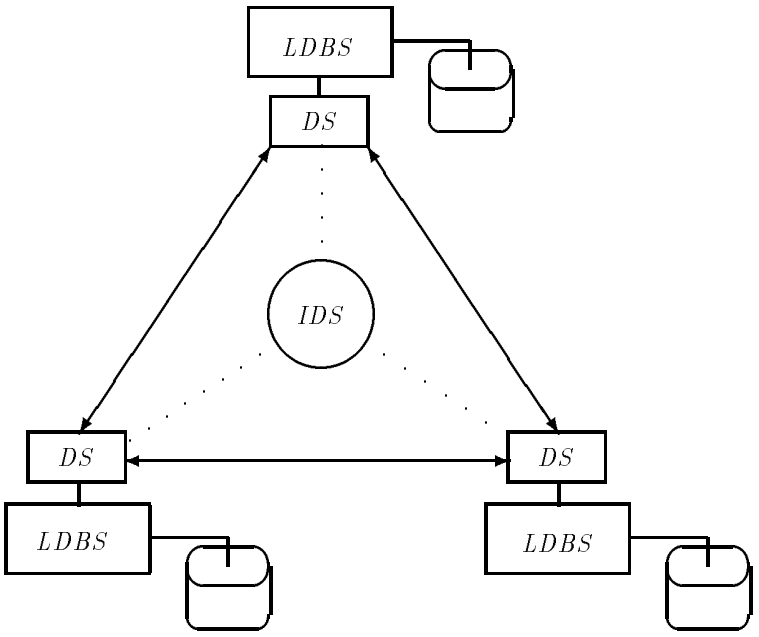


FIGURE 14.1
Architecture of a system for managing interdependent data

the local DBMS (*LDBS*). In particular, if the update involves data that are related to data in other databases, a series of related transactions may be scheduled for execution, to preserve mutual consistency of related data. The initially submitted transaction, and related transactions corresponding to restoration procedures, are submitted to their corresponding *LDBS*s. After the execution of a restoration procedure, the values of the relevant consistency predicates are updated.

14.3.2 The Concept and Properties of Polytransactions

Transactions we are interested in may not have all the *ACID* properties [HR83]. We require that a transaction is correct in a sense defined by the semantics of the application. Depending on the application, requirements other than correctness may optionally be imposed on updates performed on interdependent data.

One such optional properties is *atomicity* which may not be required if some of the operations are *compensable*. In such a case, if it is determined later that the operations should not have been executed, their effects can be "undone" by issuing compensating operations. For example the *flexible transactions* in [ELLR90] allow specification of alternative actions that can be invoked when some actions fail. A multidatabase transaction may accomplish its objectives even if some of its actions (or subtransactions) are not executed successfully.

Sometimes, the interactions of concurrent update transactions require transaction *isolation*. In many traditional applications, such as banking, this requirement is a very natural one. However, this requirement becomes completely impractical in long-lived transactions where an operation may last for several hours and involve multiple data items of large granularity. Enforcing isolation in such cases may mean that no concurrent activities are allowed. *Sagas* [GMS87] divide long-lived transactions into subtransactions, such that each subtransaction can be compensated, if necessary, and the isolation property of the transactions is relaxed. Flexible transactions [ELLR90] may also relax isolation by allowing both compensable and noncompensable subtransactions within a single global transaction.

Another frequent requirement on transactions is *durability*, which states that once the transaction is committed, its results must survive successive system failures. However, durability may not be required for transactions that do not manipulate persistent data.

Earlier in this section, we argued that in a multidatabase environment consisting of multiple autonomous systems, the concept of global (multidatabase) transaction that is composed of a well-defined set of subtransactions may be too restrictive. Therefore, we use the more flexible notion of a *polytransaction* to describe a sequence of related update activities.

A polytransaction (T^+) is a "transitive closure" of a transaction T submitted to an interdependent data management system. The transitive closure is computed with respect to the IDS. A polytransaction can be represented by a tree in which the nodes correspond to its component transactions and the edges define the "coupling" between the parent and children transactions. Given a transaction T, the tree representing its polytransaction T^+ can be determined as follows. For every data dependency descriptor D^3, such that the data item updated by T is among source objects of the D^3, we look at the the dependency and consistency predicates P and C. If they are satisfied, no further transactions will be scheduled. If they are violated, we create a new

node corresponding to a (system generated) new transaction T' (child of T) to update the target object of the D^3. T' will restore the consistency of the target object, so that the P and C predicates will be satisfied[2]. Specification of weaker mutual consistency criteria will result in infrequent violations of C. Hence, the restoration procedures (and the corresponding child transaction) will be scheduled less often.

T' corresponds to the restoration procedure (component A) that updates the related target data items specified in the D^3. The above actions correspond to *push* constraints specified in IDS. Similarly, we examine all D^3s, such that the data item read by T, is the target of D^3. If a *pull* constraint involving this data item exists, and P and C are violated, another child transaction is generated to update the affected data item, and a corresponding new node is created in the tree. This process is applied recursively to all children of T.

When a user submits a transaction that updates a data item that is related to other data items through a D^3, this transaction becomes the root of a polytransaction. Subsequently, the system responsible for the management of interdependent data uses the IDS to determine what descendent transactions should be generated and scheduled in order to preserve interdatabase dependency. Execution of a descendent transaction, in turn, can result in generating additional descendent transactions. This process continues until the consistency of the system is restored as specified in the IDS.

The ways by which a child transaction is related to its parent transaction within a polytransaction is specified in D^3, by the *execution mode* of the action component. This relationship is indicated as a label of the edge between each parent and its child in the polytransaction tree. A child transaction is *coupled* if the parent transaction must wait until the child transaction completes before proceeding further. It is *decoupled* if the parent transaction may schedule the execution of a child transaction and proceed without waiting for the child transaction to complete.

If the dependency schema requires immediate consistency, the nested transaction model may be used, in which the descendent transactions are treated as subtransactions which must complete before the parent transaction can commit. Two-phase commit protocol may be used in this case. A coupled transaction can be *vital* in which case the parent transaction must fail if the

[2] T' can be scheduled either immediately, or its execution can be delayed. The policy of generating and scheduling the child transaction will be discussed in more detail in Section 14.5.

child fails, or *non-vital* in which case the parent transaction may survive the failure of a child [GMGK+90].

Several new transaction paradigms have been proposed recently in the literature that are based on various degrees of decoupling of the spawned activities from the creator (e.g., [KR88]). Triggers used in active databases [DHL90] are probably the best known mechanism in this group. An application of this idea in the management of interdependent data was discussed in [Mil89]. The authors used a table driven approach to schedule complementary updates whenever a data item involved in a multi-system constraint was updated. The parent transaction would then terminate, without waiting for a chain of complementary actions to take place. The main problem with asynchronous triggers is that the parent transaction has no guarantee that the activity that was triggered will, in fact, complete in time to assure the consistency of the data.

To allow the parent transaction some degree of control over the execution of a child transaction, the concept of a VMS mailbox has been generalized in [GMGK+90]. Similar ideas have been presented in [BHM90], and in [HS90], where the notion of a "persistent pipe" has been introduced. Both generalized mailboxes and persistent pipes allow the parent transaction to send a message to a child process and know that the message will be eventually delivered. If such a guarantee is sufficient, the parent transaction may then commit, without waiting for the completion of the action that were requested. The parent or its descendant may check later if the message has been indeed received and take a complementary or corrective action.

14.3.3 Executing Polytransactions

Most of the work on multidatabase transaction management assume the existence of a *multidatabase management system* (MDBS) which is responsible for the processing of all global transactions. Such MDBS architectures have transactions that are either local (execute at a single database without passing through the MDBS interfaces) or are global. We propose an architecture with a distributed IDS that stores at each site the D^3s involving the interdependent data that can be updated at that site. Using the concept of polytransactions, many of the global transactions can be executed as collections of related single database transactions.

Two approaches to control the execution of multidatabase transactions have been discussed in the literature. Under the first approach, the MDBS controls the scheduling of all subtransactions of a transaction. As we dis-

cussed earlier, a disadvantage of this approach is that the set of all subtransactions and the precedence dependencies between them must be known in advance. The second approach is used in active databases and uses triggers to asynchronously schedule subtransactions based on some events, possibly in a decoupled fashion [DHL90].

We propose to schedule the polytransaction activities based on the information in the dependency schema and the database states. We will illustrate this approach using an example.

Example: Consider a collection of telecommunication databases used by applications for planning and establishing new services. Let us consider four databases as follows:

> $DB1$ contains information about each switch (an equipment that establishes circuits and routes telephone calls) and its contents (e.g., what each of its slots contains).
>
> $DB2$ contains summary information about the equipment installed in different switches, for use by a statistical application.
>
> $DB3$ is an operational database containing the status information about each switch.
>
> $DB4$ contains planning information about the switches whose capacities are close to being exhausted.

Figure 14.2 illustrates an example polytransaction on our hypothetical interdependent databases. Let us consider that transaction *T1* submitted to *DB1*, modifies the status of one of the slots in the switch as a result of adding an equipment to the switch. Let us suppose that the interdatabase dependency specifies that database *DB2* should be eventually updated to reflect the changes of the switches. Hence, transaction *T2* will be scheduled to make required changes in *DB2*. Note that due to the eventual consistency requirements, *T2* can be scheduled as a decoupled transaction. Let us further suppose that *DB3* must be updated immediately to reflect the change in the status of the switch. Therefore a transaction *T3* must be scheduled. Because of the immediate consistency requirement, *T3* should be a coupled transaction and *T1* cannot commit until *T3* does. To continue our example, if the change of the status of the switch brings its capacity above a threshold specified in the dependency schema, transaction *T4* will be scheduled to add

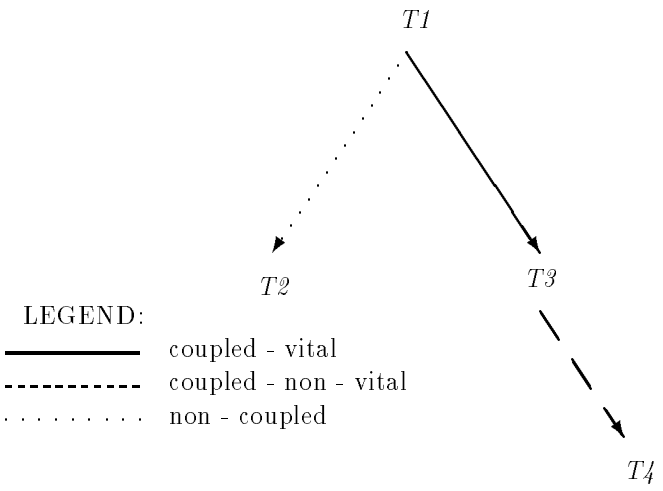


FIGURE 14.2
Generation of a Polytransaction

the relevant switch information to *DB4*. When all transactions resulting from *T1* complete, the polytransaction completes.

14.4 Interdatabase Dependency Schema

In this section we will provide a more detailed discussion of the main components of the dependency descriptors. A language or syntax is used for discussing each component. Our specification is extensible. Furthermore, an alternate language or syntax could be used for any of them.

14.4.1 Specification of the Dependency Predicate

The dependency predicate P is a boolean-valued expression specifying the relationship that should hold between the source and target data objects.

Dependency predicates can be specified, using operators of relational algebra (selection, projection, join, union, difference, intersection, etc.). Together with the basic operators, we also use the aggregate operator ξ [Klu82],

and the transitive closure operator α [Agr87]. The ξ operator allows specification of aggregate functions such as *sum* or *count* for the whole relation or for groups obtained by partitioning the relation according to the specified attribute. The α operator computes the transitive closure of a single relation R, assuming that the relation is transitive over its first two attributes. Aggregate and grouping operators can be used inside the α operator.

As an example, let us consider relation *EMP* containing the information about individual employees and their salaries that is stored in one database, and another relation *DEPT_SAL* with the summary information about the average salary in each department, that is stored in second database. The *DEPT_SAL* relation is derived by aggregation of the data from the *EMP* relation.

The requirements that each row of the *DEPT_SAL* relation contains the actual average of salaries of all employees in the department can be specified by the following predicate:

$\xi_{Dnum,Avg(Salary)}(EMP) = \Pi_{Dnum,Avg\_Sal}(DEPT\_SAL)$

14.4.2 Specification of Mutual Consistency Requirements

In this section, we will discuss the specification of mutual consistency requirements. They will be classified along two dimensions that are to a large degree orthogonal: time and state of data.

A mutual consistency requirement specifies the desired degree of consistency among related data. The requirement that is used usually in the literature is that of *immediate consistency* [SK89] which specifies that as soon as a transaction completes, all related data are also mutually consistent [SLE91]. The immediate consistency can be provided by multi-site transactions that commit updates at multiple sites together. Mutual consistency requirements using timing constraints were introduced in [WQ87, HLM88] and were further explored in [CB+89, SK89, WQ90, SR90]. Also related is the specification of coherency condition used to define "how far" primary copy and quasi-copies can diverge based on time, versions, and arithmetic conditions [ABGM90]. A similar criterion is offered in [PL91].

A mutual consistency criterion called *eventual consistency*, [SK89, RSK91] specifies that related database objects are made consistent at certain points of time specified by a condition, although they may not be consistent in the interim intervals. The condition is specified as a combination of time and data state (including events/operations). Eventual consistency allows the related

objects to diverge during some period, as long as they will be made consistent periodically. **Lagging consistency** [SK89] assumes that the data in one database may be most current while in the other ones the data may not be up-to-date. Updates applied to the first database are always propagated to the related databases. Hence, if all external updates are stopped, the databases will become consistent. Eventual consistency does imply that at some point in time, all databases will be consistent, while lagging consistency does not imply this, because some databases may always lag behind others[3].

The consistency requirement predicate, denoted by C, specifies when (in terms of time and/or data state), the related data must be consistent. Interdependent data objects may be allowed to be inconsistent within certain limits, determined by C. The specification of the consistency predicate can involve multiple boolean valued conditions, each referred to as *consistency terms* and denoted by c_i. Each c_i refers to a mutual consistency requirement involving either time or the state of a data object. Hence, C is a logical expression involving $\vee$, $\wedge$, and $\neg$ operators and consistency terms c_i.

Temporal Consistency Terms

To identify the point in time at which the related data objects must be consistent, we use the following notation. *DD-MMM-YYYY* is used to specify the date (day, month and year) and *hh:mm.ss* to denote time (hour, minute and second). The *on* operator is used in conjunction with date and the @ operator specifies time. The expression *on d* means "on date d", and the expression @*t* translates into "at time t". All time instants are referenced to the international accepted standard of Universal Coordinate Time (UTC). The following consistency requirements involving time may be defined:

> At a particular date and/or at a specific point in time. For example,
> *@ 9:00* means at 9.00 o' clock,
> *on 27-May-1991* means 'on 27^{th} of May, 1991'.
>
> Before or after a specific instant of time or date. We use the ! operator to denote predicates of this type. The position of the ! is used to distinguish between *before* and *after*. For example,
> *! 8:00* means before 8 a.m.
> *25-Aug-1991 !* means after August 25, 1991.

[3] Similar notions of consistency have also been explored within the context of Distributed Directory Services [MD82].

We can specify intervals of time or/and date using the Δ operator[4]. For example,
Δ *(9:00 - 17:00)* means during the whole interval between 9a.m. and 5 p.m.
Δ *(10-Jun-1991@17:00 - 11-Jun-1991@8:00)* specifies an overnight interval.

Periodically, when certain amount of time has elapsed. We use the expression ε(*period* @ t *on* d), to specify 'every *period* of time starting at time t and on date d'. The "@ t" and "*on* d" parts of this expression are optional. A period of time, can be specified using

A difference of two times, e.g., $t_2 - t_1$,

One of the keywords *year, month, day, hour, min, immediately*,

A duration of time.

For example,
ε*(day @17:00)* means 'every day at 5 p.m.'.

Data State Consistency Terms

The data state requirements determine "how far" the related data may be allowed to diverge (in terms of data values) before the mutual consistency must be restored [ABGM88, SR90, BGM90]. Consistency terms involving data values, can be specified in the following ways:

By specifying (or limiting) the maximum change in the value of data which is allowed before the consistency must be restored. As an example, let us consider relations *EMP* and *DEPT_SAL* introduced earlier. We may specify that the *DEPT_SAL* relation must be updated whenever salary of an employee is changed by more than 500, using the following condition:

Δ *EMP.Salary* > 500,
where Δ is used to denote the change of value.

[4] Intervals of time can also be specified by a combination of *before* and *after* operations. For example Δ *(9:00 - 12:00)* is equivalent to *9:00 ! $\wedge$! 12:00*

By specifying a condition involving data values or an aggregate function on the values of source data items. When this condition is violated, the consistency must be restored. For example, we may specify that the $DEPT\_SAL$ relation must be updated, whenever the average salary of all employees changes by more than 50, using the following condition:
Δ *AVG (EMP.Salary)* > 50

If a data object is modified by creating new versions, we can specify the maximum allowed discrepancy between version numbers of a given object that can exist in related databases. Once the difference between version numbers exceeds the allowed maximum, the data consistency must be restored. If we assume that relation EMP in database *D1* has a copy EMP' in database *D2* we can specify that EMP' can lag no more than 5 versions behind EMP, as follows:
Δ *5 versions (D1.EMP)*
When this requirement is violated, the consistency is restored by updating $D2.EMP'$ relation.

Mutual consistency requirements can be also related to some operations performed on data objects. Such requirements may indicate that consistency should be restored, when a particular operation[5] is performed. These operations can be applied to the source objects, the target object, or on both source and target objects. Several types of consistency terms involving operations can be defined:

We may allow a certain number of updates to be performed on a given source object, before the corresponding changes are made in the target object. As an example of a *push* constraint, the consistency term
10 updates on R_1 where $R_1 \in \mathcal{S}$
specifies that after 10 updates on a source relation R_1, the mutual consistency must be restored.

We may specify that a mutual consistency must be restored before an operation is performed on the target data object (*pull* constraint). This can be specified by the following consistency term:
read on U

[5] The term *operation* is used here to mean any user- or system-defined operation. For example, operation may refer to a particular transaction type defined in the system or an external event.

By specifying (or limiting) the number of operations allowed on the related data objects before the consistency is restored. For example, we may allow no more than 10 sales transactions before consistency must be restored, using the following term:
10 sales
where *sales* is the transaction name.

We can also request the restoration of mutual consistency, *before* or *after* a specific operation is performed. This is specified, by using the ! symbol placed before or after the operation. For example,
!calculate_payroll_checks
specifies that mutual consistency must be restored before the calculate_payroll_checks transaction is executed. We can also enforce consistency after the execution of an operation or a transaction. In this case, some other updates may be invoked leading to chained updates (triggers fall into this category). For example,
take_inventory!
specifies that after the inventory transaction has been completed, mutual consistency must be restored.

The framework discussed above is adequate for specifying many types of dependencies typically found among data in industry [SK89]. These dependencies can be characterized by the structure of data dependency and its control aspects. The *structural dependencies* may include full or partial replication, overlap of the informational content of the data, vertical or horizontal partitioning, value or existence constraints, etc. It is easy to see that the P predicate can be used to specify this aspect of interdependent data. The *control* aspect specifies the constraints on updating the interdependent data. For example, the derived data may always be extracted or aggregated from one database and stored in another one, that is not directly updatable. In the case of primary-secondary copies, the updates to the primary database are propagated to the secondary databases through a coordinator-subordinate relationship. The directionality of D^3, and the push and pull constraints discussed in this chapter support this aspect.

14.4.3 Specification of consistency restoration procedures

The action component $\mathcal{A}$ of a dependency descriptor is a set of one or more restoration procedures that can be invoked under certain conditions to

restore mutual consistency among interdependent data.

As mentioned before, whenever the dependency and consistency predicates, P and C, are not satisfied, inconsistencies between the source and target objects cannot be further tolerated. Hence, we assume the existence of a system that keeps track and controls the updates of all databases. In further discussion we will refer to such a system as the *manager of interdependent data* [SLE91] or the *multidatabase monitor* [Ris89, MD89].

Using the current value of each c_i, the monitor can calculate the value of the consistency predicate C. If the consistency is violated, appropriate actions must be taken to restore it. The action component $\mathcal{A}$, specifies conditional execution of one or more consistency restoration procedures. Conditions in $\mathcal{A}$, denoted by C_R (for restoration condition) can be, but need not be, the same as C. C_R is a logical expression involving $\vee, \wedge$, and $\neg$ operators and the consistency terms.

The syntax of the $\mathcal{A}$ component allows the specification of either single or multiple consistency restoration procedures. $\mathcal{A}$ with single consistency restoration procedure is specified as:

$\mathcal{A}$ *: procedure name* [**as** *execution mode*]

Since there may be several ways to restore consistency, more than one consistency restoration procedures can be specified in the action component $\mathcal{A}$. In this case we use the following notation:

when C_{R_1} **do** *procedure name* [**as** *execution mode*]
⋮
when C_{R_n} **do** *procedure name* [**as** *execution mode*]
otherwise *default procedure name* [**as** *execution mode*]

The above specification allows combining the C predicate with one or more of the consistency restoration procedures in a structure similar to a guarded command.

The descriptor D^3 specifies the name of the procedure to be invoked and optionally the *mode* in which it will run. The mode identifies the relationship between the restoration procedure (child) and the transaction that invoked it (parent). The restoration procedures are invoked by the multidatabase monitor.

14.4.4 Correctness of Dependency Specifications

The *IDS* contains semantic information describing the relationships between the participating database objects. In this subsection, we discuss the

correctness of dependency predicates, limiting our discussion to D^3s involving singleton source sets.

The IDS can be represented by a *dependency graph* consisting of nodes and edges. The nodes correspond to the source and target objects and the edges drawn from the source to the target data objects, are labeled with the corresponding D^3s. Below, we analyze the semantics of the cycles between D^3s, and examine their validity.

Example: Let us consider the following pair of D^3s:

$S_1 : o_i$
$U_1 : o_j$
$P_1 : o_j = o_i + 2$
$C_1 : immediately$
$A_1 : Update\_o_j$

$S_2 : o_j$
$U_2 : o_i$
$P_2 : o_i = o_j - 3$
$C_2 : immediately$
$A_2 : Update\_o_i$

In the example, a cycle exists between objects o_i and o_j. If an update occurs on o_i so that P_1 no longer holds, o_j becomes inconsistent, and must be made current by the restoration procedure $Update\_o_j$. After o_j has been updated, the P_2 predicate is violated. That means, o_i is inconsistent, and requires immediate execution of the $Update\_o_i$ procedure. This will be repeated indefinitely.

The above abnormal behavior is caused by the fact that each P_i predicate of the corresponding D^3s is not the *complement* of the other. If P_2 were $o_i = o_j - 2$ instead of $o_i = o_j - 3$ the cycle would be acceptable since after performing A_2, P_1 and P_2 are both satisfied. Therefore, no more updates will be performed due to restoration procedures. Cycles that do not cause indefinite updates are harmless and are referred to as *stable*.

One way to avoid unstable cycles is to disallow cycles in the IDS. This is undesirable, since there may be applications that require cyclic dependencies. Therefore, we will require that all cycles in the IDS are stable.

To generalize the previous discussion let us assume a cyclic dependency

graph as follows:

$$o_1 \xrightarrow{D_1^3} o_2 \xrightarrow{D_2^3} \cdots \xrightarrow{D_{k-1}^3} o_k \xrightarrow{D_k^3} o_1$$

All updates resulting from an update to o_1 can be propagated further, up to o_k. The last dependency D_k^3 introduces a cycle by linking objects o_k and o_1. Let P_i be the dependency predicate of $D_i^3, i = 1, 2 \ldots k$. In order for the cycle to be stable, the last predicate P_k, in general, must be the complement of the composition of all previous predicates, i.e., $P_k = (P_1 \circ P_2 \circ \cdots \circ P_{k-1})^C$. For example, if

$P_1 : o_2 = o_1 + 3,$
$P_2 : o_3 = o_2 - 1,$ then for a stable cycle we must have
$P_3 : o_1 = o_3 - 2.$

All this information can be inferred from the *IDS* and appropriate actions must be taken to ensure that it conforms with the above requirements, i.e., there are only stable cycles. We realize that the correctness of the *IDS* requirements, particularly involving consistency predicates and restoration procedures, is a difficult issue and needs to be explored further.

14.5 Consistency of Interdependent Data

Let us now explore the notion of consistency of interdependent data within the framework presented above.

14.5.1 Definition of Consistency of Interdependent Data

DEFINITION

At any instant of time, the target data object within a given D^3 is defined to be current *with respect to a given D^3, if both the consistency predicate C and the dependency predicate P evaluate to true. The data item is said to be* consistent *if the consistency predicate C evaluates to true, but the dependency predicate P does not. Hence, if a target item is current then its consistency is implied, but the opposite is not true: an object may be consistent and not current. If neither P nor C evaluate to true, the target data object is* inconsistent *and the corresponding D^3 is* violated.

A multidatabase with an IDS is defined to be in a consistent state *if every target data object, is either current or consistent, i.e.,* $\forall D^3 \in IDS\ D^3.C = true$ *(i.e., the C predicates of all descriptors must be satisfied).*

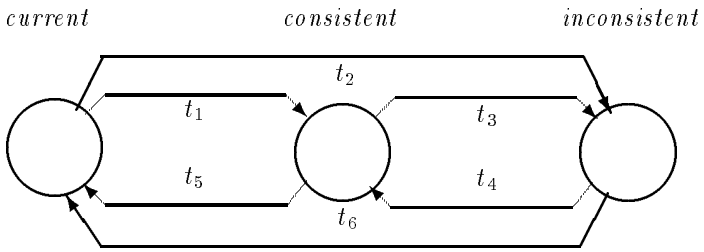


FIGURE 14.3
Transitions of a target data object

Figure 14.3 illustrates the transitions among the three states (current, consistent and inconsistent) of a target data object. These transitions occur as a result of the changes in values of the P and C predicates. The transitions are explained below:

Transition t_1: the state of the target data object changes from current to consistent, as a result of an update to a data item belonging to the source set $\mathcal{S}$. The consistency predicate C continues to be satisfied, although the values of the temporal or data state consistency terms may be changed.

Transition t_2: a current target data object becomes inconsistent as a result of an update on an object in the source set $\mathcal{S}$. Either the data state and/or the temporal terms violate the consistency predicate C.

Transition t_3: a target data object is transformed from the consistent to the inconsistent state, due to a change in the terms of the C predicate.

Transition t_4: the purpose of a restoration procedure is to change the state of a target data object, to either consistent or current. Transition t_4 occurs when we perform a *partial restoration*, so that an inconsistent data object becomes consistent, but not current. Various cost policies

may indicate that a partial restoration to a consistent state is more appropriate than a (sometimes more expensive) restoration to a current state.

Transition t_5: this transition occurs when the state of the target data object changes from consistent to current. When the target object is consistent, we have the choice of either doing nothing, or executing a restoration procedure to make the data object current. This choice can be made considering performance parameters, load balancing, etc. Execution of restoration procedures in this manner will be referred to as *eager restoration of current state.*

Transition t_6: an inconsistent data object becomes current, by invoking a restoration procedure in one of the following ways:

(a) the restoration procedure is executed when it is discovered that C is violated. This is referred to as *late restoration of current state.*

(b) the particular data object is marked as inconsistent but no action is taken until an access to the target object is attempted. Then, the restoration procedure is activated before the access is granted. This method is referred to as a *lazy restoration of current state.* This strategy is used in [SLE91].

The main purpose of the IDS is to define when a multidatabase system is inconsistent and how its consistency should be restored. The restoration procedures specify how to upgrade the state of a target data object to consistent or current. The consistency predicate C complemented with the rules specifying eager, late or lazy restoration, determine the policy that maintains the desired level of consistency. Polytransactions provide a possible mechanism for enforcing a desired policy.

14.6 Summary

In this chapter we discussed issues of managing interdependent data. We presented the concept of database dependency descriptor, D^3, that can be used to specify the relationships between data in multiple databases together

with the mutual consistency requirements and restoration procedures. Each dependency descriptor consists of three components: dependency predicate, mutual consistency predicate and a set of consistency restoration procedures. Some of the components of a D^3 have been described in the literature separately. However, we believe that the components of the dependency descriptor represent three aspects of a single problem, and must be considered together because of the interactions among them.

The descriptors in a multidatabase system constitute the Interdatabase Dependency Schema. The information stored in the *IDS* can be used to convert a transaction updating a data item in a single database into a polytransaction that spawns various activities needed to maintain the consistency of the interdependent data. The activities constituting a polytransaction, can be either coupled to the parent transaction, or decoupled from it. In the latter case, we can guarantee better response times if the weaker consistency guarantees (e.g., eventual consistency) are sufficient in a given application.

We believe that the polytransaction paradigm provides an attractive alternative to the traditional multitransaction models, since it provides the flexibility needed to support complex interdatabase consistency requirements. Unlike many multidatabase transaction models that require that the subtransactions are known in advance, dependent transactions in a polytransaction need not be known a priori. Like triggers in active databases, the dependent transactions are identified dynamically, using a rather extensive specification of the mutual consistency predicate.

We also discussed the issues of correctness of the dependency between interdependent data using the concept of the dependency graph. We classified various states of consistency of data objects, explained the transitions from one state to another, and identified possible policies that can be used by the polytransaction mechanism to maintain a desired level of consistency among interdependent data. We are currently investigating further the concepts presented in this chapter.

Bibliography

[ABGM88] Rafael Alonso, Daniel Barbara, and Hector Garcia-Molina. Quasi-Copies: Efficient Data Sharing for Information Retrieval Systems. *Proceedings of the Intl. Conf. on Extending Data Base Technology*, 1988.

[ABGM90] Rafael Alonso, Daniel Barbara, and Hector Garcia-Molina. Data Caching Issues in an Information Retrieval System. *ACM-TODS*, 15(3):359–384, September 1990.

[Agr87] R. Agrawal. Alpha: an extension of relational algebra to express a class of recursive queries. *Proceedings of the Third International Conference on Data Engineering*, pages 580–590, February 1987.

[BGM90] Daniel Barbara and Hector Garcia-Molina. The Case for Controlled Inconsistency in Replicated Data (Position Paper). In *Proceedings of the Workshop on the Management of Replicated Data*, Houston, TX, November 1990.

[BHM90] P. Bernstein, M. Hsu, and B. Mann. Implementing Recoverable Requests Using Queues. In *Proceedings of ACM SIGMOD Conference on Management of Data*, 1990.

[CB+89] S. Chakravarthy, B. Blaustein, et al. HiPAC: A Research Project in Active, Time-Constrained Database Management. Technical report, XEROX (XAIT), Cambridge, MA, July 1989.

[DAT87] S.M. Deen, R.R. Amin, and M.C. Taylor. Data Integration in Distributed Databases. *IEEE Transactions on Software Engineering*, SE-13(7):860–864, 1987.

[DBB+88] U. Dayal, B. Blaustein, A. Buchmann, U. Chakravarthy, M. Hsu, R. Ladin, D. McCarthy, A. Rosenthal, S. Sarin, M.J. Carey, M. Linvy, and R. Jauhari. The HiPAC Project: Combining Active Databases and Timing Constraints. *SIGMOD Record*, 17(1), March 1988.

[DH84] U. Dayal and H.Y. Hwang. View Definition and Generalization for Database Integration in a Multidatabase System. *IEEE Transactions on Software Engineering*, SE-10(6):628–644, 1984.

[DHL90] U. Dayal, M. Hsu, and R. Ladin. Organizing Long-Running Activities with Triggers and Transactions. In *Proceedings of ACM SIGMOD Conference on Management of Data*, 1990.

[ELLR90] A. Elmagarmid, Y. Leu, W. Litwin and M. Rusinkiewicz. A Multidatabase Transaction Model for InterBase. In *Proceedings*

of the 16th International Conference on Very Large Databases, August 1990.

[Geo90] D. Georgakopoulos. *Transaction Management in Multidatabase Systems.* PhD thesis, University of Houston, Department of Computer Science, 1990.

[GMGK+90] H. Garcia-Molina, D. Gawlick, J. Klein, K. Kleissner, and K. Salem. Coordinating Multi-Transaction Activities. Technical Report CS-TR-247-90, Princeton University, February 1990.

[GMS87] H. Garcia-Molina and K. Salem. SAGAS. In *Proceedings of ACM SIGMOD Conference on Management of Data*, 1987.

[Gra81] J.N. Gray. The transaction concept: Virtues and limitations. In *Proceedings of the 7th International Conference on VLDB*, September 1981.

[GRS91] D. Georgakopoulos, M. Rusinkiewicz, and A. Sheth. On Serializability of Multidatabase Transactions through Forced Local Conflicts. In *Proceedings of the 7^{th} IEEE International Conference of Data Engineering*, April 1991. Kobe, Japan.

[HLM88] M. Hsu, R. Ladin, and D. McCarthy. An Execution Model for Active Data Base Management Systems. In *Proceedings of the 3^{rd} International Conference on Data and Knowledge Bases*, June 1988.

[HR83] T. Haerder and A. Reuter. Principles of transaction-oriented database recovery. *ACM Computing Surveys*, 15(4), December 1983.

[HS90] M. Hsu and A. Silberschatz. Persistent Transmission and Unilateral Commit- A Position Paper. In *Workshop on Multidatabases and Semantic Interoperability*, Tulsa, OK, October 1990.

[Klu82] A. Klug. Equivalence of Relational Algebra and Relational Calculus Query Languages having Aggregate Functions. *Journal of the Association for Computing Machinery*, 29(3):699–717, July 1982.

[KR88] J. Klein and A. Reuter. Migrating Transactions. In *Future Trends in Distributed Computing Systems in the 90's*, Hong Kong, 1988.

[MD82] J.G. Mitchell and J. Dion. A Comparison of two Network-based File Servers. *Communications of the ACM*, 25(4), April 1982.

[MD89] D. McCarthy and U. Dayal. The Architecture of an Active Data Base Management System. *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 215–224, 1989.

[Mil89] J. Mills. Semantic Integrity of the Totality of Corporate Data. *Proceedings of the First International Conference on Systems Integration*, pages 482–491, April 1990.

[Mos85] J.E.B Moss. *Nested Transactions: An Approach to Reliable Distributed Computing.* PhD thesis, MIT Press, Cambridge, MA, 1985.

[PL91] C. Pu and A. Leff. Replica Control in Distributed Systems: An Asynchronous Approach. *Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data*, pages 377–386, Denver, May 1991.

[RCE+88] M. Rusinkiewicz, B. Czejdo, R. Elmasri, D. Georgakopoulos, A. Jamoussi, G. Karabatis, Y.Y. Li, and L. Loa. OMNIBASE: Design and Implementation of a Multidatabase System. *Distributed Processing TC Newsletter*, 10(2):20–28, November 1988.

[Reu89] A. Reuter. Contracts: A Means for Extending Control Beyond Transaction Boundaries. In *Third International Workshop on High Performance Systems*, September 1989.

[Ris89] T. Risch. Monitoring Database Objects. In *Proceedings of the 15th International Conference on Very Large Databases*, pages 445–453, 1989.

[RSK91] M. Rusinkiewicz, A. Sheth, and G. Karabatis. A Framework for Specifying Interdependent Data. Technical Report TM-STS-018609/1, Bellcore, May 1991.

[SK89] A. Sheth and P. Krishnamurthy. Redundant Data Management in Bellcore and BBC Databases. Technical Report TM-STS-015011/1, Bellcore, December 1989.

[SLE91] A. Sheth, Y. Leu, and A. Elmagarmid. Maintaining Consistency of Interdependent Data in Multidatabase Systems. Technical Report CSD-TR-91-016, Computer Sciences Department, Purdue University, March 1991.

[SR90] A. Sheth and M. Rusinkiewicz. Management of Interdependent Data: Specifying Dependency and Consistency Requirements. In *Proceedings of the Workshop on the Management of Replicated Data*, Houston, TX, November 1990.

[SV86] E. Simon and P. Valduriez. Integrity Control in Distributed Database Systems. In *Proceedings of the* 20^{th} *Hawaii International Conference on System Sciences*, 1986.

[WQ87] G. Wiederhold and X. Qian. Modeling Asynchrony in Distributed Databases. In *Intl. Conf. on Data Engineering*, February 1987.

[WQ90] G. Wiederhold and X. Qian. Consistency Control of Replicated Data in Federated Databases. In *Proceedings of the Workshop on the Management of Replicated Data*, Houston, TX, November 1990.