MAINTAINING INTEGRITY IN DISTRIBUTED AND HETEROGENEOUS DATABASE SYSTEMS

Deborah L. Goldsmith
The MITRE Corporation
San Diego, CA

Bhavani M. Thuraisingham
The MITRE Corporation
Bedford, MA

ABSTRACT

Within many organizations there are a number of computerized databases scattered across several sites. Efficient access to the information contained in these databases as well as sharing it has become an urgent need. As a result, the increasing number of heterogeneous database systems need to be interconnected. For many applications including C4I, consistent interoperability of distributed and heterogeneous database systems is essential. Furthermore, for such applications, it is also important that the data in the distributed database is accurate. In this paper, we provide an overview of maintaining integrity in distributed and heterogeneous database systems. We first define various types of integrity and then focus on concurrency control, recovery, and enforcing integrity constraints in distributed and heterogeneous database systems.

1. INTRODUCTION

The rapid growth of the networking and database technologies has had a major impact on the information processing requirements of organizations. Information has become the most critical resource in many organizations and therefore efficient access to information as well as sharing it has become an urgent need. As a result, many efforts have been reported on interconnecting the increasing number of databases scattered across several sites. In order to reconcile the contrasting requirements of the different database management systems (DBMSs), tools which enable users of one system to use other systems are being developed. Furthermore, efficient solutions for interconnecting different database systems as well as administering them are also being investigated.

Maintaining the integrity of the distributed and heterogeneous database systems is critical for various applications including C4I. It must be ensured that the consistency of the distributed and heterogeneous databases must be maintained. For example, the databases must be consistent with the real-world that they represent. In addition, the databases must be brought back to a consistent state after the system has recovered from failures. Consistency must also be maintained in the midst of concurrent updates. In addition to the traditional notion of integrity in database systems such as application independent and application specific integrity constraints, concurrency control and recovery techniques, integrity also includes ensuring the quality of the data as well as preventing the data from unauthorized modifications. This paper describes integrity in distributed and heterogeneous databases. First, various definitions of integrity are discussed. Then the focus will be on traditional aspects of integrity in distributed and heterogeneous database systems. In particular, enforcing application specific and application independent integrity constraints, concurrency control, and recovery are discussed.

The organization of this paper is as follows. In section 2, various definitions of integrity are discussed. Maintaining integrity in distributed database management systems (DDBMS), such as enforcing integrity constraints, concurrency control, and recovery, is discussed in section 3. Handling heterogeneity is discussed in section 4. The paper is summarized in section 5.

2. DEFINITIONS OF INTEGRITY

As stated in section 1, there are various definitions of integrity. In traditional database systems research, integrity techniques include concurrency control techniques, recovery techniques, and techniques to enforce application independent and application specific integrity constraints (see for example [DATE90, BERN87]). Concurrency control techniques must ensure that the database is consistent when multiple users access it at the same time. Recovery techniques must ensure that the database is brought back to a consistent state in the case of failures. In general, concurrency control and recovery are closely associated with transaction execution. A transaction is a program unit consisting of a series of query and update requests. In the case of multiple transactions executing at the same time, it must be ensured that each transaction satisfies the ACID (Atomicity, Consistency, Isolation, and
Durability) properties. Atomicity means that either all the instructions of a transaction are executed or none are executed. Consistency means that a transaction must transform one consistent state of the database to another. Isolation means that a transaction does not access values that are not committed by other transactions. Durability means that when a transaction commits, its actions must be made permanent in the database.\footnote{A concise overview of traditional transaction model is given in \cite{AGRA92}.}

Application independent integrity constraints are usually those associated with the data model utilized. For example, in a relational model, it must be ensured that there are no two tuples with the same primary key. Application dependent integrity constraints are those that ensure that the database is consistent with the real world. For example, in an employee database, one could enforce a constraint that the salary value of an employee must be positive or the age of an employee must increase with time.

While database researchers have focused on developing concurrency control, recovery, and integrity constraint enforcement techniques, security researchers have focused on a type of integrity which is concerned with preventing unauthorized modifications to the data. Various models of integrity have been proposed. Notable among them are the Biba model [Biba77] and the Clark-Wilson model [CLAR87]. Biba model assigns integrity levels to users and data and specifies rules for users at different levels to access data at different levels. Clark-Wilson model describes users executing transactions depending on their roles. Here, transactions are considered to be activities to be carried out. Another type of integrity, which has received more attention recently, is data accuracy/quality. Much of the research in this area is being carried out by those in management information systems. The main issue here is ensuring the accuracy of the data. For example, when operators enter large amounts of data, how can one ensure that the number of errors are minimized? Closely associated with data quality is the validity of the source of the input. How accurate is the source? What sort of confidence level can be assigned? Accuracy of the data on output is also an issue. For a discussion of this type of integrity we refer to \cite{WANG93}.

While all types of integrity discussed in this section are essential for maintaining the accuracy of the data as well as the consistency of the database, this paper addresses traditional database integrity. We focus on distributed and heterogeneous environments. We will first discuss integrity in a DDBMS and then focus on the impact of handling heterogeneity.

3. INTEGRITY ISSUES IN DISTRIBUTED DATABASE SYSTEMS

This section describes integrity issues in a distributed database management system.\footnote{Much of the information in this section has been obtained from our earlier paper \cite{SMAL93}.} We first provide an overview of distributed database systems in section 3.1. Certain aspects of integrity are discussed in sections 3.2 and 3.3. In particular transaction processing as well as handling integrity constraints are discussed.

3.1 OVERVIEW

The definition of a DDBMS that we have considered is the following one given by Ceri and Pelagatti \cite{CERI84}:

“A distributed database is a collection of data which is distributed over different computers of a computer network. Each site of the network has autonomous processing capability and can perform local applications. Each site also participates in the execution of at least one global application, which requires accessing data at several sites using a communication subsystem. A distributed database management system supports the creation and maintenance of a distributed database.”

Figure 1 illustrates an architecture that conforms to this definition of a DDBMS. In this architecture, the nodes are connected via a communication subsystem. This subsystem could be any network such as a Local Area Network or a Long Haul Network. Each node has its own local DBMS which is capable of handling the local applications. In addition, each node is also involved in at least one global application.

The module that is responsible for data distribution, distributed query processing, distributed transaction management, distributed metadata management, and enforcing integrity constraints at the global level is the
Distributed Processor. It augments each local DBMS. The Distributed Processors at various nodes communicate with each other to carry out a particular task. The components of the Distributed Processor are illustrated in figure 2.

![Diagram of Distributed Database System (DDBMS)](image)

**Figure 1. Architecture for a DDBMS**

![Modules of the Distributed Processor](image)

**Figure 2. Modules of the Distributed Processor**

In section 3, we focus on just two aspects of integrity. One is to maintain the consistency of the distributed database in the midst of concurrent user access and ensure that the consistency is maintained in the case of system and site failures. The other is to enforce appropriate application independent and application specific integrity constraints. In section 3.1, we discuss maintaining consistency during distributed transaction management. The component of the Distributed Processor that is responsible for transaction management is the Distributed Transaction Manager (DTM). The DTM is responsible for maintaining consistency when multiple transactions execute concurrently and also ensuring that the distributed database is recovered to a consistent state in the case of a failure. In section 3.2, we discuss application independent and application specific integrity issues. The module that is responsible for enforcing integrity constraints is the Distributed Integrity Manager.

### 3.2 Maintaining Integrity During Transaction Management in a Distributed Environment

#### 3.2.1 Distributed Transaction Management

Transaction management in a DDBMS involves the handling of distributed transactions. By a distributed transaction we mean a transaction which executes at multiple sites. The portion of the transaction which executes at a particular site is a subtransaction associated with that site.

---

3 Security is not within the scope of this paper. For a discussion of security for distributed and heterogeneous environments we refer to [THUR91].
Two types of log files are maintained at each node; one is the local log file which records the subtransaction execution and the other is the global file which records the actions of the coordinator of all of the subtransactions. When a transaction commits, either all of the local subtransactions must commit or none should commit in which case the transaction aborts. The technique used to ensure this requirement is the two-phase commit protocol.

If transactions are executed serially, then there will be a performance bottleneck. Therefore, transactions usually execute concurrently. As a result, it must be ensured that the multiple transactions maintain the consistency of the distributed database. Concurrency control techniques ensure the consistency of the distributed database when transactions execute concurrently. There are additional problems in a DDBMS because of the replication of files. It has to be ensured that all of the copies of a file are consistent. The techniques of locking, time-stamping, and validation have been proposed for concurrency control in a DDBMS.

An additional problem of transaction management in a distributed environment is ensuring the consistency of the data in the presence of network partitions. Network partitions occur when certain nodes and/or links fail.

We will first describe the various concurrency control techniques used in a DDBMS and then discuss the recovery and reliability issues.

3.2.2 Concurrency Control

In order to ensure consistency, the transaction has to be serializable. We first discuss serializability for distributed transactions. Then we discuss three popular concurrency control techniques.4

Serializability: A schedule is a sequence of operations performed by multiple transactions. Two schedules are equivalent if their outcomes are the same. A serial schedule is a schedule where no two transactions execute concurrently. An objective in transaction management is to ensure that any schedule is equivalent to a serial schedule. Such a schedule is called a serializable schedule. Various conditions for testing the serializability of a schedule have been formulated for a DDBMS.

Three popular concurrency control techniques which ensure the serializability of schedules are locking, time-stamping, and validation.

Locking: Two-phase locking is used to provide concurrency control (i.e., a transaction acquires all necessary locks first before releasing any). Two types of locks may be obtained. They are shared locks and exclusive locks. A shared lock for a data item is requested by a transaction for the read operation. The request is granted if no other transaction has a write lock for the same data item. An exclusive lock is requested by a transaction for the write operation. The request is granted when no other transaction has either a shared lock or an exclusive lock for the same data item.

In a distributed environment, due to multiple copies of the data, it may be necessary to obtain locks on more than one copy. Three of the schemes to obtain locks are:

(i) Exclusive locks are obtained for all copies of the data item. Shared locks are obtained for only one copy.
(ii) Both shared and exclusive locks are obtained for a majority of copies of the data item.
(iii) Only the primary copy of the data item is locked.

Locking can cause deadlocks. A deadlock detection graph is used to detect deadlocks. The graph is called a "distributed wait-for graph." The algorithm attempts to detect cycles in the graph.

Time-stamping: In the time-stamp technique, each transaction is assigned a time-stamp when it begins execution. Similarly a data item is also assigned a read stamp and a write stamp which are the time-stamps of the most recent transaction which read or updated that data item. When a transaction requests to read or write a data item, certain checks are made between the time-stamp of the transaction and the read/write stamps of the data item. If the checks are satisfied, then a transaction continues; otherwise the transaction aborts and restarts with a new time-stamp.

4For a detailed discussion we refer the reader to [CERI84].
In a distributed database system there are some additional considerations. For example, the time-stamps assigned to the transactions are global time-stamps. Since it is difficult to keep the system clocks at each node synchronized, other techniques to assign global time stamps have been devised.

Validation: In the validation technique (also called optimistic concurrency control technique) each local transaction is validated first. During this validation certain checks are performed. If it does not pass the validation test, the subtransaction is aborted. This in turn will cause the global transaction to be aborted. If a subtransaction passes the validation test, then it enters a global validation phase. If it fails the global validation, then it is aborted. Otherwise, a message is sent to the coordinator that it has passed the global validation. If all the subtransactions have passed the global validation, then the transaction is committed.

3.2.3 Recovery

A popular recovery technique for distributed transaction is the two-phase commit protocol. In this protocol, one site (usually the site of origin) acts as the coordinator and other sites where the transaction executes act as participants. The coordinator as well as the participants maintain log records and write all their actions on the records. First the coordinator sends a message to the participants to prepare to commit. Each participant sends a message to the coordinator as to whether it is ready to commit. If all of the participants have answered positively, then the coordinator sends a commit message to all participants. Otherwise, the coordinator sends an abort message to the participants. If a participant receives a commit message, then the subtransaction at that site is committed. If an abort message is received, the subtransaction is aborted.

The two-phase commit protocol has been extended (e.g., three-phase commit, voting protocols) to accommodate various types of failures such as site failures and network partitions. Details are given in [CERI84].

3.3 ENFORCING INTEGRITY CONSTRAINTS IN A DISTRIBUTED ENVIRONMENT

Two types of integrity constraints are enforced in DBMSs. These are application independent integrity constraints and application specific integrity constraints. In this section we discuss both types of constraints in a centralized environment and then discuss issues for a DDBMS. Our focus will be on the relational data model [CODD70].

Application independent integrity constraints include the primary key constraint, the entity integrity rule, referential integrity constraint, and the various functional dependencies involved in the normalization process [DATE90]. We discuss some of the integrity rules and associated definitions below.

Primary Key: A primary key uniquely identifies an entity in a relation. It is possible to have more than one unique identifier; each such key is a candidate key. Candidate key which is not a primary key is an alternate key.

Uniqueness: at any given time, no two tuples of a relation have the same value for a candidate key.

Minimality: if candidate key K is composite (consists of two or more attributes), then no component of K can be eliminated without destroying the uniqueness property.

Entity integrity: no component of the primary key of a base relation is allowed to accept null values

Foreign key: Foreign key in the relational model is defined as follows [DATE90]. Attribute FK (possibly composite) of base relation R2 is a foreign key if and only if the following two properties are satisfied: (i) Each value of FK either has all of its components null or all of its components nonnull. (ii) There is a base relation R1 with primary key PK such that each nonnull value of FK is identical to the value of PK in some tuple of R1.

Referential Integrity: The rule associated with foreign keys is the referential integrity rule. This rule states that the database must not contain any unmatched foreign key values.

We illustrate the essential points with an example. Figure 3 illustrates a relational database with relations EMP and DEPT. EMP has attributes E#, Ename, Salary, and D#. DEPT has attributes D#, Dname, and Mgr. E# is the primary key of EMP and D# is the primary key of DEPT. D# is a foreign key in DEPT. The integrity rules must ensure that each E# in EMP and D# in DEPT is unique. Any D# value in EMP must be referenced in DEPT.
Application specific integrity constraints are those constraints that are specific to an application. Examples include "An employee’s salary cannot decrease" and "No manager can manage more than two departments".

Integrity issues in a DDBMS include enforcing integrity constraints across multiple DBMSs. For example, the EMP relation could be at site 1 and DEPT relation could be at site 2. If referential integrity is to be maintained, then when a D# value is inserted into EMP, the system must query the remote DBMS to ensure that the D# value is referenced in DEPT. In the case of ensuring the uniqueness of primary keys, if a relation is horizontally fragmented across the sites (for example, some of the tuples of the EMP relation could be at site 1 and the rest could be at site 2) then when a tuple is inserted into a fragment, the remote DBMS must be queried as to whether a tuple with the same primary key already exists.

In the case of application specific integrity constraints, they could be enforced across the multiple databases. For example, consider the constraint "the total salary of the employees cannot exceed two million". If the relation EMP is fragmented, then whenever an employee tuple is entered, the remote DBMSs must be queried to obtain the salaries of the employees. Another issue with application specific constraints is distribution. That is, should the constraints be distributed across sites, should they be replicated, or should they be stored in one location. One could treat the application specific integrity constraints as part of the metadata and therefore the approach taken to store the metadata could be used for such constraints.

<table>
<thead>
<tr>
<th>EMP</th>
<th>DEPT</th>
</tr>
</thead>
<tbody>
<tr>
<td>E#</td>
<td>Name</td>
</tr>
<tr>
<td>1</td>
<td>John</td>
</tr>
<tr>
<td>2</td>
<td>Paul</td>
</tr>
<tr>
<td>3</td>
<td>Mary</td>
</tr>
</tbody>
</table>

Figure 3. Relational Database Satisfying Application Independent Integrity Rules

4. HANDLING HETEROGENEITY

Although research on interconnecting different DBMSs has been underway for over a decade, it is only recently that many of the difficult problems are being handled. In section 4.1 we provide an overview of various types of heterogeneity. Then we focus on certain aspects of integrity in sections 4.2 and 4.3. In particular, transaction processing as well as handling integrity constraints in heterogeneous environments are discussed.

4.1 OVERVIEW

Scheuerman [SCH90] et al have discussed various types of heterogeneity which are listed below.

(i) Schema (or data model) Heterogeneity: Not all of the databases in a heterogeneous architecture are represented by the same data model. Therefore, the different conceptual schemas have to be integrated. In order to do this, translators which transform the constructs of one data model into those of another are being developed.

(ii) Transaction Processing Heterogeneity: Different DBMSs may utilize different algorithms for transaction processing. Work is being directed toward integrating the various transaction processing mechanisms. For example, techniques which integrate locking, timestamping, and validation mechanisms are being developed.

(iii) Query Processing Heterogeneity: Different DBMSs utilize different query processing and optimization strategies. One of the research areas here is to develop a global cost model for distributed query optimization.
(iv) Query Language Heterogeneity: Different DBMSs will utilize different query languages. Even if the DBMSs are based on the relational model, one could use SQL and the other Relational Calculus. Standardization efforts are under way to develop a uniform interface language.

(v) Constraint Heterogeneity: Different DBMSs enforce different integrity constraints which are often inconsistent. For example, one DBMS could enforce a constraint that all employees must work at least 40 hours while another DBMS may not enforce such a constraint.

(vi) Semantic Heterogeneity: Data may be interpreted differently at different components. For example, the entity address could mean just the country at one component while another component could interpret it to be the number, street name, city name and country. It has been recognized that semantic heterogeneity is very difficult to handle [CERC90].

Researchers are providing solutions to handling various types of heterogeneity (see for example [IEEE91, IEEE93]). The problems become even more complex if there are several types of heterogeneity to be handled. For example, the various systems could not only utilize different data models but also different transaction processing algorithms.

Closely associated with heterogeneous database systems is the notion of autonomy. This is because in a heterogeneous environment, different database systems exist and each system could possibly be used for a specific application. Therefore, each system would desire some degree of autonomy. The goal of a heterogeneous distributed database system is for the individual database systems to cooperate with one another but at the same time have some degree of autonomy.

Various types of autonomy have been discussed in the literature [SHET90]. These include: Design autonomy, Communication autonomy, Execution autonomy, and Association autonomy. We discuss each type.

Design Autonomy: Total design autonomy would imply that each component will have the ability to choose its own design. For example, it could determine (i) the data to be managed, (ii) the policies to be enforced, (iii) the query processing and transaction management algorithms used, and (iv) the semantic interpretation of data.

Communication Autonomy: Total communication autonomy implies that a component will determine whom it wishes to communicate with.

Execution Autonomy: Total execution autonomy implies that the local operations of a component are not affected by the global users in any way.

Association Autonomy: Total association autonomy implies that a component can decide when and what information to share with the others.

In section 4 we discuss integrity issues for a distributed database management system where the local database systems are heterogeneous and have some degree of autonomy. Transaction processing in heterogeneous and autonomous environments are discussed in section 4.2. Enforcing integrity constraints is discussed in section 4.3.

4.2 MAINTAINING INTEGRITY DURING TRANSACTION MANAGEMENT IN HETEROGENEOUS AND AUTONOMOUS ENVIRONMENTS

This section examines the impact of handling heterogeneity and autonomy during transaction management. While research on transaction processing in heterogeneous and autonomous environments is fairly recent, several efforts have now been reported. Much of the focus has been on investigating serializability for autonomous environments. In addition, work on integrating different concurrency control protocols such as locking and timestamping has also begun.

To handle the new types of application environments, advanced transaction models have been proposed [ELMA92]. As stated by Elmagarmid et al, in an autonomous environment, each database system may often process the local requests first before the global requests. Therefore, the global transactions could be of long duration. In addition, since each local system is autonomous, it may be difficult to detect and resolve conflicts between global transactions and, as a result, serializability conditions would be difficult to meet. Due to these problems, the traditional transaction model, where each transaction satisfies the ACID properties, is not suitable for heterogeneous and
autonomous environments. New models such as the DOM transaction model [ALEJ92], Flex transaction model [KUHN92], S-Transaction model [VEIJ92], Polytransactions [SHET92], and Cooperative transaction hierarchy [NODI92] have been proposed. We provide a brief overview of these models.5

In the DOM (Distributed Object Management) model [BUCH92], the heterogeneous system is considered to be a collection of objects which interact with each other. Therefore entities such as a nodes, DBMSs, databases, and applications are all treated as objects. The model distinguishes between two types of objects, the native objects which are part of the DOM and also attached objects which may also include database systems and applications. The DOM transaction model adapts itself to the application requirements from flat transaction model to a nested model. The correctness criteria depends on the component systems of the DOM system.

The Flex transaction model [KUHN92] is a generalization of the traditional transaction model. Its features are function replication, mixed transaction, and value function. Function replication feature enables a transaction to have the flexibility to determine where a particular function has to be executed. The mixed transaction feature enables the actions of a subtransaction to be compensated by that of another. This will enable a subtransaction to be committed before the parent transaction. The value function feature associates value functions with transactions as well as subtransactions. Execution may be determined by the value function. Each application executed by flexible transactions determines to what extent the ACID properties are satisfied.

The S-Transaction model [VEIJ92] was developed to handle different types of autonomy which are organizational, design, management, communication, and execution. One of the goals in developing this model is for a local system not to distinguish between the local transactions and the subtransactions of a global transaction which access local data. Also, to handle the arbitrary and complex interactions between the sites to cooperatively carry out a task, the model includes the notion of a spanning tree which can be dynamically generated. This tree is the execution structure of a transaction. Variation of the ACID properties are proposed.

The polytransaction model [SHET92] was developed to maintain the consistency constraints between interdependent data. It is argued that some of the other models, such as the flex transaction model, assume that the relationships between interdependent data are known to the designer in advance which may not be the case in reality. In the polytransaction model, a polytransaction is specified as a tree and dependency requirements between the data are used to structure the tree dynamically. User defines the correctness criteria.

The models proposed for long-lived transactions are being investigated for autonomous environments as the global transaction may have a long duration. The Cooperative transaction hierarchy [NODI92], which was proposed for CAD/CAM applications where the transactions are long-lived, is one such example. In this model, the transactions are designed hierarchically to reflect the structure of the design environments. Details of subtransactions can be isolated from the parent transactions. Furthermore, the subtransactions of a parent transaction can run concurrently. In a heterogeneous and autonomous environment, the parent transaction would be the global transaction with the subtransactions executing at multiple nodes. Instead of atomicity being the correctness criteria, user specified correctness criteria is proposed.

4.3 ENFORCING INTEGRITY CONSTRAINTS IN HETEROGENEOUS AND AUTONOMOUS ENVIRONMENTS

Enforcing integrity constraints in heterogeneous and autonomous environments has also received much attention recently. There are several aspects to consider and we briefly discuss them in this section.

First of all, constraints may be specified in different languages. One system may use the structured query language to express constraints while another may use first order logic clauses. A solution to this problem is either to specify transformations between one representation to another or mandate the use of a standard language such as SQL.

In the case of data model heterogeneity, different data models may be utilized by the different systems. This means that the specification of the integrity constraints may vary with the model. For example, in a relational DBMS, one could enforce the constraint that "the salary attribute of the employee relation must be positive". In an object-oriented DBMS, one could enforce this same constraint as "the salary instance variable of the employee class must be positive". Techniques to handle data model heterogeneity must ensure that the constraints are transformed correctly.

5Several other models have also been proposed in the literature. For a detailed discussion we refer to [ELMA92].
The problem is more complex if the constraints themselves are heterogeneous regardless of the model. That is, different DBMSs may enforce different integrity constraints which are often inconsistent. For example, one DBMS could enforce a constraint that all employees must work at least 40 hours while another DBMS may enforce a constraint where all employees must work 50 hours. If an employee relation is updated, then both constraints are triggered. The administrators of the two systems may have to resolve the conflict possibly through negotiation.

Semantic heterogeneity, where data may be interpreted differently at different components could cause integrity problems. For example, the same attribute, say salary of an employee, may be interpreted differently by different DBMSs. One system may consider the salary to be the hourly wage of an employee while another system might interpret it to be the weekly wage. Another example of heterogeneity is the use of different names to denote the same object. The word "aircraft" may be interpreted as "plane" or "aviation vehicle". One approach to handling semantic heterogeneity is to agree upon standard notions. This is often infeasible as different groups may not agree on common definitions. Therefore, an alternative approach will be to accept semantic heterogeneity, but to have a directory which specifies all the names that may be used to denote an object and to check this directory to handle conflicts.

Enforcing integrity constraints in an autonomous environment is difficult if the information about a remote database is not known. For example, there could be a constraint that the total salary of an organization cannot exceed one million. If the organization consists of different groups, each having its own database, then enforcing such a constraint is difficult if a remote group does not make its data available. In such a situation, the system has to reason with partial information making some guesses possibly based on previous experience. Some techniques for enforcing integrity constraints in heterogeneous and autonomous environments are given in [RAM93].

5. SUMMARY AND FUTURE CONSIDERATIONS

Maintaining integrity in distributed and heterogeneous environments is critical for many applications. In this paper, we first defined various types of integrity. These include traditional database integrity such as concurrency control recovery, and enforcing integrity constraints, as well as other types of integrity such as data accuracy and quality. Then we focused on traditional database integrity for distributed and heterogeneous database systems. We discussed transaction management and enforcing integrity constraints in a distributed environment and then described the impact of heterogeneity and autonomy. The purpose of this paper was to provide an overview of integrity in distributed and heterogeneous database systems. For a discussion of the details, we refer the reader to the references listed.

Several areas need further work. One is to implement the various transaction processing techniques for heterogeneous and autonomous environments in commercial as well as operational systems. Another area is to design and implement techniques for enforcing integrity constraints in heterogeneous and autonomous environments. Finally, techniques for enforcing other types of integrity, such as data quality and accuracy, need to be developed.

REFERENCES


6Much of the development has been in the form of prototypes.


