1990

Critical Issues in Multidatabase Systems

Ahmed K. Elmagarmid
Purdue University, ake@cs.purdue.edu

Marek Rusinkiewicz

Report Number:
90-966

CRITICAL ISSUE IN MULTIDATABASE SYSTEMS

Ahmed Elmagarmid
Marek Rusinkiewicz

CSD TR-966
March 1990
(Revised April 1990)

Critical Issues in Multidatabase Systems*

Ahmed K. Elmagarmid and Marek Rusinkiewicz †
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907
(317)-494-1998
ahmed@cs.purdue.edu, marek@cs.purdue.edu

Abstract

For many applications, a multidatabase approach constitutes an attractive alternative to a single, integrated database. In this paper we describe our experiences gained in the course of the development of experimental multidatabase systems: OMNIBASE, at the University of Houston and InterBase at Purdue University. We concentrate on the problems of query processing and multidatabase transaction management and discuss solutions that were developed within both projects. We identify the main issues related to supporting updates spanning across multiple autonomous databases and define an extended transaction model suitable for heterogeneous computing environments. Then we review the major outstanding research problems and discuss the directions of further research in the area of multidatabase systems.

Key words and phrases: heterogeneous/federated database systems, transaction management in heterogeneous distributed database systems, multi-system applications, systems integration.

*This work is supported by a PYI Award from NSF under grant IRI-8857952 and grants from AT&T Foundation, Tektronix, SERC and Mobil Oil.

†on leave from the University of Houston

1 Introduction

In many large organizations, multiple software systems exist (databases, application programs with their dedicated files, etc.) that are frequently not compatible with each other. The problem is further aggravated by the heterogeneity of the hardware and operating systems used by these systems. To protect the investments made in such systems, it is necessary to not only provide uniform access to all their data and resources, but also to allow them to cooperate by exchanging data and synchronizing their execution. This emerging need to provide organization-wide access to data and software resources is creating a demand to interconnect previously isolated software systems. An end-user in a heterogeneous computing environment should be able not only to invoke multiple existing software systems and hardware devices, but also should be able to coordinate their interactions. These systems may run autonomously on different computers, may be supported by different operating systems, may be designed for different purposes, and may use different data formats.

The problem of integrating programs and data from various application systems has been addressed, among others, in [Gat87], [BW86], [HN87] where several approaches based on a common task specification language have been proposed. As an example of a global application that needs to access and coordinate interactions of several software and hardware systems, consider the problem of preparing materials for a meeting in a company [Gat87]. Suppose that, in order to prepare the materials, data must be extracted from a DB-2 database residing on a mainframe computer, combined with additional data stored in a LOTUS file on a personal computer, formatted, and spooled to a printer to prepare transparencies for the meeting. Furthermore, the mail utility should be invoked to deliver a memo announcing the meeting, together with the materials, to each attendee's workstation.

Software systems capable of automatically executing such applications do not exist, and many issues introduced by interconnection of existing heterogeneous application systems have been virtually unexplored. In the *InterBase* project we are currently developing a prototype of a system that allows specification and execution of global applications in a heterogeneous computing environment. Global applications in *InterBase* may involve not only database systems but also knowledge bases, expert systems and other application software running on a wide range of mainframe, mini- and micro-computers. DOL, a distributed operation language [ROE90], is used in the system to specify tasks that need to be executed by an application and the interactions among them. The DOL is simple and concise enough to be used directly by end users.

In this paper we will first discuss the architecture of multidatabase systems and then we will concentrate on the problem of providing consistent updates across multiple databases.

The paper is organized as follows. In Section 2 we discuss a possible architecture of a multidatabase system, based on a concept of a distributed task

specification language. We also show how such a language can be used to specify global applications requiring access to multiple and autonomous software systems. In Section 3, we discuss some major problems related to the management of multidatabase transactions. In Section 4 we propose extensions to the transaction model which address the specific requirements of the multidatabase environment. In Section 5 we review some of the work we are currently carrying out in the InterBase project at Purdue University.

2 Architecture of a Multidatabase System

In this section, we will discuss possible approaches to the architecture of a mutlidatabase system using as examples the OMNIBASE and InterBase prototypes. OMNIBASE is an experimental prototype developed at the University of Houston [Rea88], while InterBase is a prototype system built in the InterBase Laboratory at Purdue University [ROE90]. InterBase is an experiment in Interoperable Heterogeneous Computing.

2.1 OMNIBASE Architecture

The main functional components of the multidatabase system OMNIBASE and their interactions during the evaluation of a global query are illustrated in Figure 1. Initially, the user (application program) presents a multidatabase query to the *global user interface*, which analyzes it and removes ambiguous references (if necessary) using the knowledge base. The query is then sent to the global *query parser* and *decomposer*, that decomposes it into a set of subqueries. Next, a query evaluation plan is formulated as a program expressed in a Distributed Operations Language (DOL). The use of DOL allows the specification of complex, multi-site data processing requests, that may involve not only multiple databases but also other software packages. The query evaluation plan is subsequently executed under the control of the *Query Evaluation Supervisor* (QES), which directs the subqueries to their respective local systems. For each subquery, a *Local Access Manager* (LAM) translates the query into the local database language and submits it to the Local Database Management System (LDBS). An intermediate result of a local query produced under the control of a LAM may be then directed to some other site. LAMs are used to protect local autonomy of LDBSs participating in a multidatabase system and act as local agents of the MDBS. The results of subqueries are then combined to produce the final answer to the global query.

Because of the lack of centralized control in the design of LDBSs, different databases may define data objects that logically belong to the same domain using different data types. Furthermore, data representation and precision may vary even for data of the same type. To accommodate the heterogeneity and possible inconsistency among databases a multidatabase dictionary service must

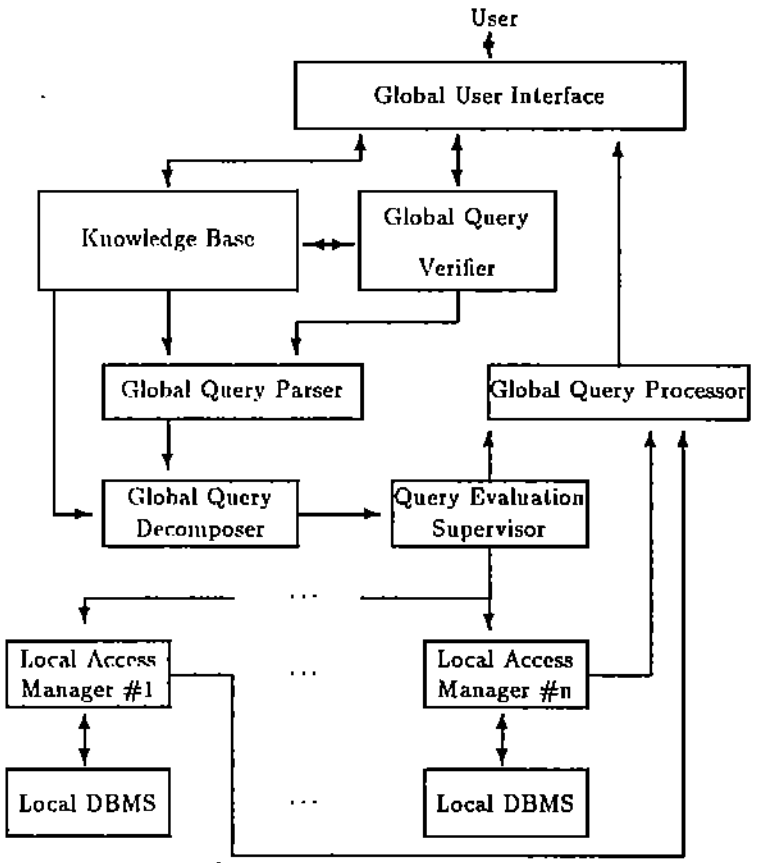

Figure 1: OMNIBASE Architecture

be provided either as a part of the MDBS or as independent directory server. Whenever a new LDBS joins the MDBS, or a member database is modified in a way affecting its export schema, the appropriate schema information must be made available to the MDBS.

OMNIBASE uses a knowledge base to resolve relation name ambiguity, DBMS incompatibility and inconsistencies in participating databases. The knowledge base contains information about local databases, relation attributes, attribute domains and necessary conversions. For all attributes, information about their names, aliases, types, measurement units, corresponding domains and conversions into canonical form is stored. This information is used whenever implicit or explicit data conversions are required by a global query. The domain information in the global dictionary includes domain name, canonical type, canonical units, allowed operations, precision and tolerance. Each attribute in an export schema is assigned a domain code and an attribute characteristic code. For example, the domain of an attribute pertaining to an entity of type person contains a code indicating whether the domain is an ID, Name,

Sex, Age, Address, Phone number, or some other descriptive characteristic. For domains corresponding to measurements, the domain code may indicate length, area, capacity, mass, time, money, score or quantity.

The system's knowledge base contains the information described above and a set of rules. The inference engine uses the rules stored in the knowledge base to automatically remove ambiguity from a global query and to derive an efficient query evaluation plan. Occasionally the system may have to consult the user in order to eliminate ambiguities from pending requests. [1] The query evaluation plan is created as a program in a Distributed Operation Language. In the context of multidatabase operations, DOL can be used to express various query processing algorithms taking into account different cost functions, network characteristics, load balancing, etc. Thus, in OMNIBASE query evaluation can be optimized by generating different DOL programs for each query.

2.2 InterBase Architecture

The InterBase prototype implementation environment includes various database systems (SYBASE, INGRES, dBase IV), expert systems (GURU), spreadsheets (LOTUS 123), etc [ROE90], running on a wide range of hardware (from mainframes to microcomputers) and using various operating systems and network software. The system uses the task specification language (DOL) to specify local or remote executions of member software systems. The DOL allows the user to specify all actions associated with a distributed application, the sequence of actions, logical dependencies, data paths, and the maximum allowed degree of concurrency. The details of communication between the various components of the system are transparent to the user. The architecture of the prototype implementation built in the InterBase Laboratory at Purdue is presented in Figure 2. The prototype consists of the following main components:

DOL Program Interpreter

> The *DOL Program Interpreter* reads a DOL program and performs the actions specified. It is responsible for managing the flow of control specified by the DOL program, contacting and opening connections to each service requested, setting up communication paths between services, monitoring the status of the individual services, and closing connections when the program is finished. It also enforces the semantic rules of the language such as insuring that services are opened before use, making sure that dependent batch processes do not run concurrently, and checking the requested connection modes against those supported by each service. It consults the service directory to determine how to contact the individual services, and which contact and data transfer modes they support.

[1] The query decomposition algorithm used by OMNIBASE is described in [RC87]. It is based on identifying connected sub-graphs of the query connection graph that are fully contained in a single database.

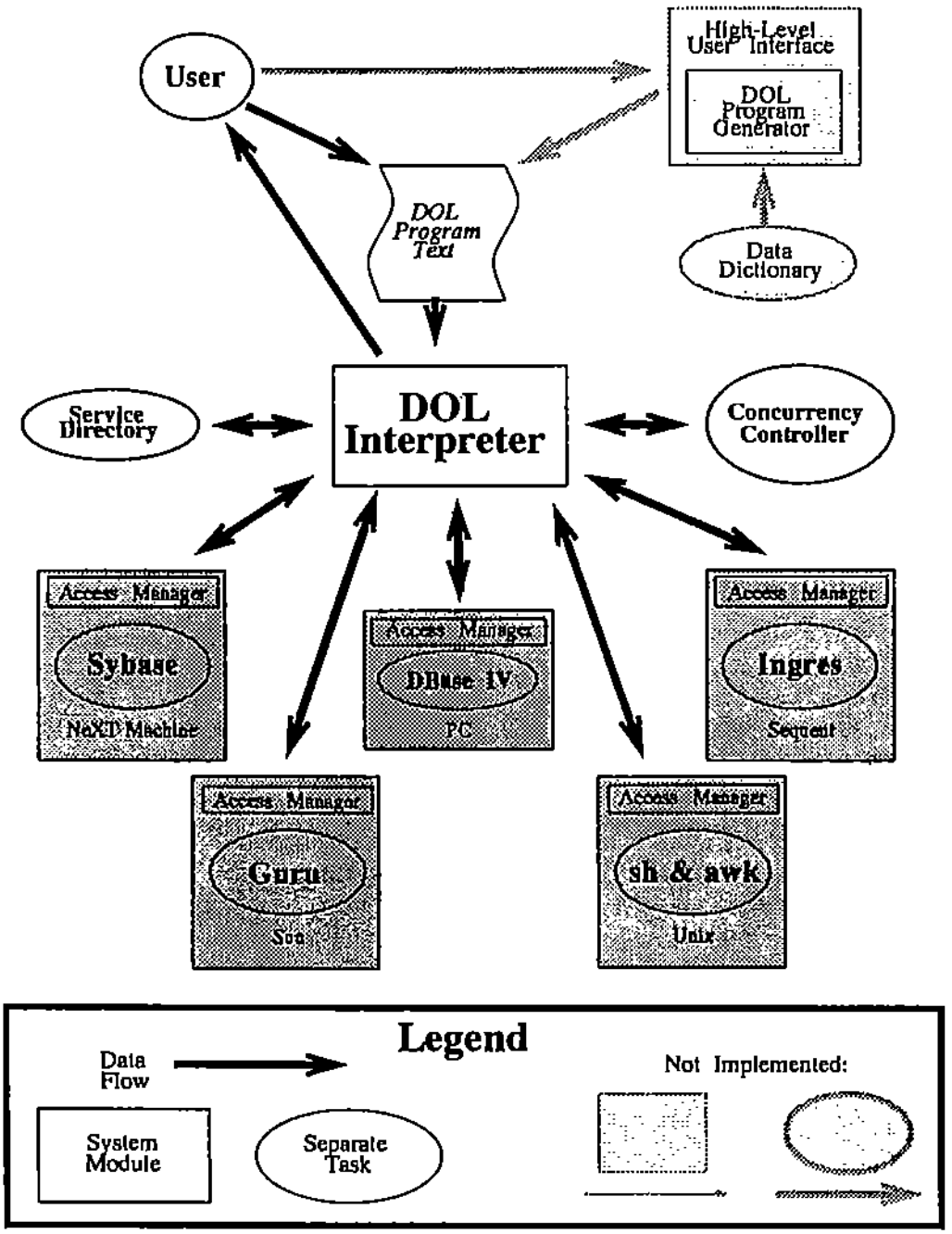

Figure 2: InterBase system Components

Local Access Managers

Local Access Managers (LAMs) are designed to preserve the autonomy of each member software system [EH88a]. A LAM acts as a proxy user for the software system it manages, encompassing it in a sort of logical shell. The only underlying software system interfaces available to the LAM are those that were originally designed to be available to the system's users.

Once a software system has been provided with a specially designed LAM, the DOL interpreter does not need to know anything about the underlying software system. This service abstraction is possible because the same communication protocol is used with all LAMs, regardless of what service they're providing. This greatly reduces the complexity of the overall system, in that adding new software systems does not require changes in the other modules of the system.

Service Directory

A software system which has been incorporated into our heterogeneous distributed environment acts as a *service* which can be used directly by the end-user client. The main function of the *Service Directory* is to provide *location and distribution transparency* by locating individual services for the end-user. To accomplish this, the *Service Directory* contains a list of all services available in the system, together with the information needed to access each of them. This information includes the physical channel to use for communication, the supported communication protocol, the connection protocol to use, and the data transfer methods supported. The physical channel information could specify a local area network and an address, a serial line and a device name, a local program and a path, etc. The communication protocol will usually be a function of the physical channel. For example, a service on a local area network might communicate via TCP/IP. In order to use a service, we must know how to contact it; some services are always running, some are started automatically by their host when requested, and some must be started explicitly before they can be used. Finally, since DOL allows data to be transferred in *batch* mode or in *pipe* mode, the *Service Directory* needs to know which modes are supported by each service, and in what manner the data exchange should take place.

Concurrency Controller

The concurrency controller is responsible for managing both the commitment and serialization of the local subtransactions. One of the objectives of the InterBase project is to investigate the applicability of various concurrency control approaches (e.g. such as the pre-specified orders and simulated prepared states of subtransactions explained in [ED90][LE90]) in a multidatabase environment.

High-Level User Interface

If a global application involves only database systems, a multidatabase language, such as MSQL [LITW89], can be used to specify queries and updates. We have recently started a project in which we are developing a MSQL query interface for end-users. We have successfully implemented a compiler for a subset of MSQL which translates MSQL queries into DOL programs. The compiler is tested and ready to be incorporated into the InterBase prototype. Work on graphical query and update interfaces to multidatabase systems is being carried out in the OMNIBASE project.

3 Concurrency Control in Multidatabase Systems

In a database system, several users may read and update information concurrently. Undesirable situations may arise if the operations of various user transactions are improperly interleaved. Concurrency control is an activity that coordinates concurrently executed operations so that they interfere with each other in an acceptable fashion. Some of the prototype implementations of multidatabase systems described in the literature [T^+83, T^+87] allow only retrieval operations in a heterogeneous environment, because updates present serious problems in areas such as concurrency control, logging and security. Recently, much attention has been focused on providing support for updates spanning across multiple autonomous database systems A key step in achieving this goal is global concurrency control, which has been discussed in [GL84] [GPZ86] [LS86] [AGMS87] [Vid87] [BS88a] [EH88a] [Pu88] [ED90] [LE90].

The problem of concurrency control in multidatabase environments is different from that in distributed database systems, and global concurrency control strategies developed in homogeneous distributed database environments do not work well in multidatabase environments. Furthermore, most efforts attempting to generalize the classical concurrency control strategies for multidatabase systems are only partially successful. For example, many concurrency control protocols proposed for MDBSs either violate local autonomy or do not maintain global serializability (see [DELO89a]).

Designing a concurrency control strategy for a heterogeneous database environment is more difficult than in its homogeneous counterpart, primarily because we must deal not only with the data distribution but also with heterogeneity and autonomy of underlying databases. In a tightly-coupled distributed database system, there is only one concurrency controller to certify and/or produce the schedules. The concurrency controller has access to all information it needs to produce and/or certify the schedules. In addition, it normally has control over all transactions running in the system. In contrast, in a multidatabase systems we must deal with the following problems caused by autonomy of the

local systems.

- Local concurrency controllers are designed in such a way that they are totally unaware of other LDBSs or of the integration process. This type of autonomy is defined as design autonomy and indicates that each of the LDBSs is free to use whatever algorithms it wishes. When this LDBS is incorporated into a federation, design autonomy specifies that we cannot retrofit its algorithms. The only way to remedy this problem is to use a hierarchical approach to concurrency control [LE90].

- The Global Concurrency Controller (GCC) needs information regarding local executions in order to maintain global database consistency. However, the GCC has no direct access to this information, and cannot force the Local Concurrency Controllers (LCCs) to supply it. This type of autonomy is defined as communication autonomy, which means that an LCC is allowed to make independent decisions as to what information to provide.

- LCCs make decisions regarding transaction commitments based entirely on their own considerations. LCCs do not know or care whether the commitment of a particular transaction will introduce global database inconsistency. In addition, a GCC has no control over LCCs at all. For example, a GCC cannot force an LCC to restart a local transaction even if the commitment of this local transaction will introduce global database inconsistency. We call this type of autonomy the execution autonomy; it says that each of the LCCs is free to commit or restart any transaction running under its control.

3.1 Hierarchical Concurrency Control

It has been argued that due to the autonomy of the local database systems, concurrency control in multidatabase systems should be performed using the hierarchical approach [GPZ86]. In this approach, the concurrency control responsibility is properly distributed among the GCC and the LCCs. However, due to the lack of a correctness proof, it is not clear that the general hierarchical approach is correct. As a result, the correctness of many algorithms [Pu88], [AGMS87] [Vid87] based on this approach has not been formally proven. As a matter of fact, it has been pointed out in [DELO89a] that in some cases these algorithms even produce non-serializable schedules.

It is shown in [LE90] that following two conditions are sufficient for the global schedules to be serializable:

Condition 1 All LCCs maintain the serializability of their local schedules and ensure that all the local transactions and subtransactions are serialized in their lifetime.

Condition 2 The serialization orders of the subtransactions of all committed global transactions are compatible.

3.2 Quasi-Serializability

Serializability has been generally used as the correctness criterion for the proposed concurrency control strategies. Unfortunately, serializability does not work well in heterogeneous distributed database environments. In [DELO89a], we discussed the difficulties of maintaining global serializability in heterogeneous distributed database environments. In our opinion, these difficulties result from the fact that serializability was originally introduced for centralized database environments and therefore is centralized in nature. Global concurrency control in heterogeneous distributed database environments, on the other hand, is hierarchical in nature due to the autonomy of the element databases. As a result, some of the proposed algorithms violate local autonomy (e.g., Sugihara's distributed cycle detection algorithm [Sug87]), some allow low degree of concurrency (e.g., Breitbart's site graph testing protocol [BS88a]), and others fail to maintain global serializability (e.g., [AGMS87], [Pu88], [EH88a] and [LEM88]).

The hierarchical nature of global concurrency control in MDBSs makes it very difficult to maintain global serializability. On the other hand, it relieves the global concurrency controller from some responsibilities (e.g., the correctness of local histories). This suggests that the correctness criteria for global concurrency control in HDDBSs should be based primarily on the behavior of global applications, with proper consideration of the effects of local applications on global applications.

More details on this work can be found in [DE89], where we have defined quasi serializability (QSR) as a possible correctness criterion for global concurrency control in multidatabase environments. We define an execution as quasi serializable if it is equivalent to a quasi serial execution in which global transactions are executed sequentially and local executions are all serializable.

Example: Let $E = \{E_1, E_2\}$ be an execution of transactions G_1, G_2, L_1 and L_2, where

$E_1 : w_{g_1}(a) r_{l_1}(a) w_{l_1}(b) r_{g_2}(b)$

$E_2 : r_{g_2}(c) w_{l_2}(d) r_{g_1}(d) w_{g_2}(e) r_{l_2}(e)$

E is quasi serializable. It is equivalent to the quasi serial execution $E' = \{E'_1, E'_2\}$, where

$E'_1 : w_{g_1}(a) r_{l_1}(a) w_{l_1}(b) r_{g_2}(b)$

$E'_2 : w_{l_2}(d) r_{g_1}(d) r_{g_2}(c) w_{g_2}(e) r_{l_2}(e)$

A significant difference between serializable executions and quasi serializable executions is that quasi serialization order of global transactions is determined by their execution order. In other words, if a global transaction G_1 was executed completely before another global transaction G_2, then G_1 also precedes G_2 in the quasi serialization order. Therefore, a specific quasi serialization order of global

transactions can be guaranteed at global level by controlling their submission order. This is obviously very useful in global concurrency control.

3.3 Dealing with local conflicts

When there are no direct inter-database dependencies, the quasi serializability constitutes a simple and relatively easy to enforce correctness criterion for concurrent execution of multidatabase transactions. However, when these assumptions are not satisfied, we have to use traditional criteria based on serializability. To assure multidatabase consistency in this case, the MDBS must deal with both direct and indirect conflicts. Direct conflicts involving only subtransactions of multidatabase transactions can be easily handled by the MDBS concurrency control mechanism. However, indirect local conflicts involving local transactions are extremely difficult to detect. Since the MDBS is not aware of local transactions and the indirect conflicts they may cause, it cannot determine if an execution of arbitrary global and local transactions is serializable.

In the early work in this area the above problem was misunderstood and the existence of indirect conflicts was ignored. Several solutions were proposed [GPZ85], [LEM88], [EH88a] that required conflicting multidatabase transactions to have the same relative serialization order at each of the local databases only in cases where they have a direct conflict. Du and Elmagarmid [DELO89a] have derived scenarios where the above paradigms are shown to violate global serializability.

When it became clear that indirect conflicts could not be ignored, several solutions were proposed in the literature which utilize information about the execution order of multidatabase transactions to either determine their serialization order or to prevent indirect conflicts. However, these attempts were only partially successful due to the following reasons:

- Observing the execution order of the multidatabase transactions at each LDBS is not enough to determine their relative serialization order. Even if a subtransaction of a global transaction G_2 is executed and committed before the subtransaction of another global transaction G_1 in some local database, G_1 may precede G_2 in the equivalent serialization order, because of indirect conflicts caused by the local transactions.

- Indirect conflicts between multidatabase subtransactions cannot be prevented by controlling their submission and execution order. In [DELO89a] it is shown that global serializability may be violated even when multidatabase transactions are submitted serially to their corresponding LDBS.

An alternative approach is to assume that direct conflicts between multidatabase transactions exist whenever they are possible. Breitbart and Silberchatz proved [BS88b] that in order to guarantee global consistency, multidatabase transactions must be serialized in the same way in all LDBS, even in

the absence of conflicts among them. This idea has been used by the Amoco Distributed Database Systems (ADDS) [BS88b].

Georgakopoulos and Rusinkiewicz [GR89] proposed an scheme under which the subtransactions of the global transactions are required to perform special data manipulation operations at each LDBS. This ensures that either the subtransactions of each multidatabase transaction have the same relative serialization order in all participating LDBSs or they are aborted. This method provides an answer to two complementary questions:

1. How can the MDBS obtain the information about the relative serialization order of subtransactions of global transactions at each site, in the presence of local transactions, whose existence and behavior is unknown to the MDBS, and

2. How can the MDBS guarantee that the serialization orders of subtransactions at each site are consistent with a global serialization order for multidatabase transactions?

We have argued that it is very difficult to determine the serialization order of the subtransactions without modifying the local database management systems. Instead of requiring the local systems to report their serialization orders to the MDBS, we incorporate additional operations in all subtransactions of global transactions which create direct conflicts between them at each LDBS. The execution order of the incorporated operations can be observed by the MDBS. The LDBS concurrency control mechanism will then guarantee that either the execution order of the incorporated operations is consistent with the serialization order of the subtransaction they belong to or, if not, the conflict will be resolved by the local concurrency control mechanism. Hence, the LDBSs will prevent subtransactions from being executed if the execution order observed by the MDBSs is to be inconsistent with their local serialization orders.

To achieve these objectives, we associate with each subtransaction a timestamp generated locally at each LDBS that is guaranteed to reflect its serialization order. To distinguish the timestamps used to specify subtransaction serialization orders from the timestamps (possibly) used by the LDBS concurrency control mechanisms, we call the first *tickets* and we use the expression *Take-A-Ticket* to refer to the process of obtaining such a timestamp. To ensure that the ticket value obtained by each subtransaction reflects its serialization order, we store the current value of the ticket in each LDBS. Hence the value of a ticket becomes a regular data item and all operations on tickets are subject to normal database constraints. Furthermore, we require the actions to read and increment the ticket to be part of each subtransaction.

It can be shown that the tickets obtained by the subtransactions of multidatabase transactions determine their relative serialization order. If there is an indirect conflict involving multidatabase transactions which is inconsistent

with the order in which the subtransactions obtain their tickets, the local execution becomes non-serializable and it is not allowed by the LDBS concurrency control. Therefore, indirect conflicts can be resolved by the local concurrency control even if the MDBS cannot detect their existence. The tickets can be used to maintain the global consistency by validating global transactions using the *Global Serialization Graph*. To guarantee global serializability we allow global transactions to commit only if their relative serialization order is the same in all participating LDBS.

4 Extending the Transaction Model

Since it has been first introduced, the concept of a transaction has become one of the fundamental abstractions used in the design and analysis of information systems that provide concurrent access to shared information. The basic idea of transactions is to divide application programs into well defined units that provide semantically correct transitions between consistent states of the information system. The fundamental properties of transactions, as defined by Gray [Gra81], namely atomicity, isolation and durability turned out to be very useful in the modeling of real world applications. In particular, most work on concurrency control, commitment and recovery has been performed using transactions as a basic unit of work.

However, as the new computing environments encompassing heterogeneous and autonomous information systems began to emerge, it became increasingly clear that the limitations of the traditional transaction concept began to outweigh its virtues. In the multidatabase systems transactions must access multiple autonomous (and frequently heterogeneous) database systems, in order to accomplish their objectives. The main source of difficulty in applying the traditional transaction management techniques to these environments is the requirement of *local autonomy* discussed above. Another problem is the potential for *long lived* transactions which make many basic techniques developed in the context of centralized databases (strict locking, two-phase commitment, etc), totally inapplicable in these environments.

There have been several attempts to overcome the inadequacy of the traditional transaction concepts and the limitations of serializability as a basic correctness criterion for the concurrent execution of transactions spanning across multiple and autonomous systems. Gray proposed to divide a global transaction into relatively independent subtransactions. He proposed to associate with each subtransaction a *compensating transaction* that can "undo" the effects of a committed subtransaction if required by the global transaction. This idea was further extended by Garcia-Molina who developed a notion of *sagas* [GMS87] which reject serializability as a basic correctness criterion. Another idea that has received attention as a means for overcoming the above mentioned difficulties is the concept of nested transactions [Mos81]. However the notion of nested trans-

actions as proposed by Moss does not address the autonomy of local systems at all.

In this section we outline an extended transaction model, which we believe, is much more suitable for computing environments consisting of autonomous systems. The proposed model allows us to utilize knowledge of the semantics of the application that is to be modeled by a transaction. The model allows composition of flexible transactions consisting of mutually dependent subtransactions. The execution of these subtransactions may depend on the success of previous subtransactions, and alternative sources of information may be specified. This approach requires redefinition of the notion of successful execution of transactions, their scheduling and commitment.

Some of the ideas incorporated in the proposed extended transaction model have been introduced elsewhere. For example, the concepts of functional equivalence of subtransactions and conditional execution of subtransactions have been formalized in [LER89]. Similarly, the idea of using time in the specification of transaction execution can be found in [WQ87] and [LT88]. Here, we review briefly the main ideas related to extensions of the transaction model for a multidatabase environment.

4.1 Specification of Multidatabase Transactions

Under the traditional model, a multidatabase transaction is defined by providing the set of subtransactions composing it and a partial ordering among them. We propose to extend this definition by specifying execution dependencies among subtransactions, the acceptable execution states and temporal value of the completion. The main concepts will be explained intuitively below, the more formal treatment is presented in [RELL90] [LER89].

Subtransaction Set

With every subtransaction from the set S we associate the following descriptors:
Transaction type:

- C - transaction is compensatable (i.e. can be undone)
- NC - transaction performs real actions
- R - transaction is repeatable (i.e. can be executed in accordance with the "at least once" semantics)
- NR - transaction is nonrepeatable (i.e. must be executed in accordance with the "at most once" semantics)

Commitment date (t_c): Date after which the transaction is considered to be implicitly committed (optional)
[2]

[2]The idea of value date can be traced to the early paper by Wiederhold et al. [WQ87]; it has been formally defined by Litwin [LT88].

Expiration date (t_e) : Date after which the transaction is considered to be implicitly aborted (optional).

temporal predicate (tp) : this will indicate that a transaction can be executed only when a particular predicate related to time is true (e.g. between 9 and 5 GMT).

Execution Dependencies

D - is a set of predicates that define the dependencies among subtransactions of a (global) transaction. The following types of dependencies can be defined [LER89]:

- *Positive execution dependency* determines a (partial) ordering of subtransactions. (Value dependency constitutes a special case of positive dependency.) We say that a subtransaction T_i is positively dependent on subtransaction T_j, if T_i cannot be executed until T_j completes successfully.
- *Negative execution dependency* exists between subtransactions T_i and T_j if T_i cannot be executed until T_j has been scheduled and failed [3].
- *Alternative dependency* is defined by providing equivalence classes for transactions that are functionally equivalent. With every class we may associate a preference ordering relation defined over all subtransactions in the class. Two subtransactions are *functionally equivalent* if they accomplish the same function in achieving the transaction's objective. If there are several functionally equivalent transactions, typically only one of them will need to be executed.
- *Compensating execution dependency* exists between subtransactions T_i and T_j if T_j "undoes" the effects of T_i when executed after the successful completion of T_i.

A subtransaction may be in one of four states: {not executed, executing, executed successfully, executed and failed}. A *transaction execution state* is a vector of execution states for all its subtransactions. The transaction execution state is modified whenever a scheduled subtransaction completes (commits or aborts).

At any point in time there exists a set ST of *schedulable transactions*. A subtransaction T_i is *schedulable* if the current execution state of the transaction satisfies all prerequisite dependencies of T_i and temporal constraints of T_i.

[3] We say that a subtransaction fails if it is executed but does not achieve its objectives (logical failure) or if it can not be successfully completed because of a physical failure.

Acceptable Execution State

The execution state of a transaction is acceptable if the objectives of the transaction have been accomplished. The set of all acceptable states of a transaction constitutes the *acceptable state set* which is denoted by A. Once a transaction execution reaches an acceptable state, no new subtransactions are scheduled.

The extended transactions are submitted to the scheduler which attempts to execute them by scheduling subtransactions in accordance with their execution dependencies. A transaction completes successfully when it reaches an acceptable state; it must be aborted when its execution state is not acceptable and there is no schedulable subtransaction.

Completion Value of Transaction

With every transaction we associate its (relative) value as a function of time. The value function, V, reflects the fact that the completion of a transaction may represent a different value (utility) to the user depending on the completion time. Typically, the completion value of a transaction will decrease with time.

Examples:

We will illustrate the concepts introduced above with some intuitive examples using banking and travel agent examples.

- A deposit(100, Acct_No) may be **compensatable**. A withdrawal(100, Acct_No) may be not compensatable, if it is accompanied by a real action.
- A read(fileid, Key_value) from an indexed file is **repeatable**. A read(fileid, next) from a sequential file is **not repeatable**.
- **Expiration date**: Let reservation transaction abort automatically after 24 hours.
- **Value date**: Let the check withdrawal commit automatically on December 2, 1989.
- **Positive dependency**: Make hotel reservation in a destination city, if you made a flight reservation.
- **Negative dependency**: Get reservations on American, if your attempt to get reservations on Continental failed.
- **Compensating transaction**: Cancel car rental in NYC for December 24th, 1989.

- **Alternative transactions**: Rent a room in Chicago on December 11, 1989, at Hilton, Sheraton or Ramada, in this order of preference.

- **Execution state, acceptable state**: Let us consider the following transaction:

  make flight reservation {American, Delta, UsAir}
  rent a car {Avis, National, Dollar}
  make hotel reservation {Hilton, Sheraton}.

  and the following execution state:

  (failed, successful, not executed,
  successful, not executed, not executed,
  failed, not executed)

  In this case, the set of schedulable subtransactions consists of one transaction: {make hotel reservation (Sheraton)} If this subtransaction executes successfully, then the global transaction reaches an acceptable state (please notice that this occurs despite the fact that some of the subtransactions have failed and some were never executed). Otherwise the global transaction fails, since the set of schedulable transactions becomes empty.

- **Value function**: Buy a ticket to Chicago for departure on December 21st, before closing today (at a special fare X), or before next Monday (at a normal fare Y), or before the departure date (at a "full coach" fare).

4.2 Transaction Processing

With the addition of the time value function the problem of scheduling multidatabase transactions can be formulated as an optimization problem.

The objective is to maximize the total value of committed transactions within a particular period of time, while observing the intra- and inter-transaction dependencies. The intra-transaction dependencies are specified explicitly in each (global) transaction. The inter-transaction dependencies are determined by the correctness criteria used by the multidatabase systems.

For example, if we assume that no dependencies exist between local database systems involved in a multidatabase transaction, *quasi serializability* (See Section 3) can be used as an inter-transaction correctness criterion. In this case, local executions are serializable and there is an order of global transactions. Another possibility is to use *value-date safe* executions as defined by the value date paradigm [LT88]. Under this proposal a data item can be accessed "safely", only if the current time is greater than the value date associated with this data item.

In general, the addition of the value function for completion may alleviate some problems with the pure "value date" approach that are shared with other algorithms based on deadline scheduling.

The notion of commitment will have to be re-examined in the context of the extended transaction model. We have already mentioned that a transaction may complete successfully, even if some of its subtransactions failed or were not executed. We have assumed that subtransactions are scheduled in accordance with their execution dependencies. When an acceptable state is reached we may need to perform the following actions:

- All subtransactions needed to achieve the objectives of the global transaction are committed.
- All subtransactions that are currently executing are aborted, and
- Compensating subtransactions are scheduled for all those completed subtransactions that violate the alternative dependencies.

For example, let us consider our travel agent transaction. Let us suppose that the three airline reservation subtransactions were scheduled concurrently and the last two of them have completed successfully. Then, one of the "rent a car" subtransactions has been successful. Let us assume that currently the two hotel reservation subtransactions are in progress. As soon as the first completes successfully, the objective of the global transaction is achieved. The global transaction can be now committed as follows: The second hotel subtransaction in progress, is aborted, the second airline reservation subtransaction must be compensated, and the remaining completed subtransactions are committed.

Of course, the commitment process as described above, may not always be applicable. In such cases, we can take advantage of the mechanisms for implicit commitment or expiration of subtransactions that are provided in the model. By choosing the appropriate commitment and expiration dates for the subtransaction, we may *implicitly commit or abort* extended transactions.

5 Further Work

In this paper we have addressed several problems related to the development of applications involving multiple and autonomous databases.

We described the possible architecture of such a system, using as examples experimental prototypes we have developed. We have also discussed the problems of development of applications involving multiple heterogeneous hardware and software systems. The approach based on the Distributed Operation Language has proven successful in the integration of data and knowledge bases. We believe that it will be useful in distributed and heterogeneous computing environments where current network services, such as file transfer or remote login are no longer sufficient.

The next part of the paper addressed the difficult problems in multidatabase transaction management that are caused mainly by the heterogeneity and autonomy of local systems participating in a federative environment. We then proposed a new transaction model which allows specification of complex transactions that are capable of capturing more semantics of the applications. A given objective can frequently be accomplished in more than one way. The most important elements of this proposal are:

- Clear recognition of the fact that global transactions accessing multiple autonomous systems cannot be efficiently processed according to the traditional transaction model using serializability as a correctness criterion. Hence, a global transaction must be treated as a set of subtransactions with complex intra-transaction dependencies, whose execution may not always be completely controlled by a centralized scheduler.

- A transaction may complete successfully even if some of its subtransactions were never executed or failed. In this case some of the scheduled subtransactions may need to be aborted and some of the committed subtransactions may have to be compensated (if applicable).

- Since the same function can frequently be accomplished in more than one system, we need to have a mechanism to define the functional equivalence of subtransactions and to specify their alternative scheduling.

- Transaction models proposed so far, either completely ignored the completion time of a transaction (serializability), or treated it as a hard deadline that a transaction must meet in order to complete successfully (real time transactions, value dates). Here, we propose a much more realistic paradigm under which a transaction's completion value varies as a function of time. Hence, if a transaction deadline cannot be guaranteed we may still want to complete it, recognizing that the utility of such completion may be lower, but not necessarily zero.

Many ideas presented in this paper are only preliminary, and we are currently working on their development in the InterBase project. The main problems currently under investigation include:

- Development of an execution environment for a multidatabase language such as MSQL.

- Development of a transaction specification language capable of expressing the concepts defined in the extended transaction model.

- Study of the extensions to the task specification language DOL, that are needed in order to support execution of multidatabase transactions.

- Study of the appropriateness (i.e., strength and weakness) of quasi serializability as the correctness criterion in MDBS and the problem of maintaining MDBS consistency using quasi serializable executions.

6 Acknowledgements

The authors would like to acknowledge the contributions made by all the researchers of the Interbase project at Purdue University and the OMNIBASE project at the University of Houston. The work reported here was done jointly with W. Du, D. Georgakopoulos, Y. Leu, K. Loa, and S. Ostermann. Other members of these projects have also contributed to the content and presentation of this paper.

References

[LE90] Y. Leu and A. Elmagarmid. A Hierarchical Approach to Concurrency Control for Multidatabase Systems. In *Second International Symposium on Databases in Parallel and Distributed Systems*, July, 1990, Ireland.

[AGMS87] R. Alonso, H. Garcia-Molina, and K. Salem. Concurrency control and recovery for global procedures in federated database systems. In *IEEE Data Engineering*, pages 5–11, September 1987.

[BS88a] Y. Breitbart and A. Silberschatz. Multidatabase systems with a decentralized concurrency control scheme. In *Distributed Processing Technical Committee-NEWSLETTER*, pages 35–41, November 1988.

[BS88b] Y. Breitbart and A. Silberschatz. Multidatabase update issues. In *Proceedings of the 4th International Conference on Data Engineering*, February 1988.

[Day83] U. Dayal. Processing queries over generalization hierarchies in a multidatabase. In *Proceedings of The Ninth International Conference on Very Large Data Bases*, October 1983. Florence.

[DE89] W. Du and A. Elmagarmid. Quasi serializability: a correctness criterion for global concurrency control in InterBase. In *Proceedings of the International Conference on Very Large Data Bases*, Amsterdam, The Netherlands, August 1989.

[DELO89a] W. Du, A. Elmagarmid, Y. Leu, and S. Osterman. Effects of autonomy on maintaining global serializability in heterogeneous distributed database systems. In *Proceedings of the Second International Conference on Data and Knowledge Systems for Manufacturing and Engineering*, October 1989.

[DER82] C. Devor, R. Elmasri, and S. Rahimi. Design of DDTS: a reliable distributed database testbed system. In *Proceeding of the 2nd IEEE Symposium on Reliability in Distributed Systems*, July 1982.

[ECR87] D. Embley, B. Czejdo, and M. Rusinkiewicz. An approach to schema integration and query formulation in federated database systems. In *Proceedings of the Third International Conference on Data Engineering*, February 1987.

[ED90] A. Elmagarmid and W. Du. A paradigm for concurrency control in heterogeneous distributed database systems. In *Proceedings of the Sixth International Conference on Data Engineering*, Los Angeles, California, February 1990.

[EH88a] A.K. Elmagarmid and A.A. Helal. Supporting updates in heterogeneous distributed database systems. In *Proceedings of the International Conference on Data Engineering*, 1988.

[EN84] R. Elmasri and S. Navathe. Object integration in logical database design. In *Proceedings of the IEEE Data Engineering Conference*, February 1984.

[FS82] A. Ferrier and C. Strangret. Heterogeneity in the distributed database management systems Sirius-Delta. In *Proceeding of the Eight International Conference on Very Large Data Bases*, September 1982.

[GL84] V.D. Gligor and G.L. Luckenbaugh. Interconnecting heterogeneous data base management systems. *IEEE Computer*, 17(1):33–43, January 1984.

[GMS87] H. Garcia-Molina and K. Salem. Sagas. In *Proceedings of the ACM Conference on Management of Data*, pages 249–259, May 1987.

[GPZ85] V.D. Gligor and R. Popescu-Zeletin. Concurrency control issues in distributed heterogeneous database management systems. In F.A. Schreber and W. Litwin, editors, *Distributed Data Sharing Systems*. North-Holland, 1985.

[GPZ86] V.D. Gligor and R. Popescu-Zeletin. Transaction management in distributed heterogeneous database management systems. *Information Systems*, 11(4):287–297, 1986.

[GR89] D. Georgakopoulos and M. Rusinkiewicz. Transaction management in multidatabase systems. Technical Report UH-CS-89-20, Department of Computer Science, University of Houston, September 1989.

[Gra81] J. Gray. The transaction concepts: Virtues and limitations. In *Proceedings of the International Conference on Very Large Data Bases*, pages 144–154, 1981.

[ROE90] M. Rusinkiewicz, S. Ostermann, A. Elmagarmid, and K. Loa. The Distributed Operation Language for Specifying Multisystem Applications. In *Proceedings of the First International Conference on Systems Integration*, IEEE, New Jersey, 1990.

[LA86] W. Litwin and A. Abdellatif. Multidatabase interoperability. *Computer*, 19(12), December 1986.

[LEM88] Y. Leu, A.K. Elmagarmid, and D.N. Mannai. A transaction management facility for InterBase. Technical Report TR-88-064, Computer Engineering Program, Pennsylvania State University, 1988.

[LER89] Y. Leu, A. Elmagarmid, and M. Rusinkiewicz. An extended transaction model for multidatabase systems. Technical Report CSD-TR-925, Department of Computer Science, Purdue University, 1989.

[LR82] T. Landers and R. Rosenberg. An overview of Multibase. In *Proceedings of the International Symposium on Distributed Databases*, 1982. Berlin.

[LS86] T. Logar and A. Sheth. Concurrency control issues in heterogeneous distributed database management systems. Technical report, Honeywell Computer Sciences Center, 1986.

[LT88] W. Litwin and H. Tirri. Flexible concurrency control using value dates. *IEEE Distributed Processing Technical Committee Newsletter*, 10(2):42–49, November 1988.

[Mos81] J.E. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing,*. PhD thesis, Dept. of Electrical Engineering and Computer Science, MIT, April 1981.

[Pu88] C. Pu. Superdatabases for composition of heterogeneous databases. In *IEEE Proceedings of the 4th International Conference on Data Engineering*, 1988.

[RC87] M. Rusinkiewicz and B. Czejdo. An approach to query processing in federated database systems. In *Proceedings of the Twentieth Hawaii International Conference on System Sciences*, 1987.

[Rea88] M. Rusinkiewicz and et. al. Omnibase: design and implementation of a multidatabase system. *Distributed Processing TC Newsletter, IEEE Computer Society*, 10(2):20–28, 1988.

[S+81] J. Smith et al. MULTIBASE: Integrating heterogeneous distributed database systems.

[Sug87] K. Sugihara. Concurrency control based on cycle detection. In *Proceedings of the International Conference On Data Engineering*, pages 267–274, February 1987.

[T+83] M. Templeton et al. An Overview of the Mermaid System - a Frontend to Heterogeneous Databases. In *Proceedings of IEEE EASCON*, September 1983.

[T+87] M. Templeton et al. Mermaid-A Front-End to Distributed Heterogeneous Database. In *Proceedings of IEEE*, May 1987.

[Vid87] K. Vidyasankar. Non-two-phase locking protocols for global concurrency control in distributed heterogeneous database systems. In *CIPS Edmonton*, 1987.

[WQ87] G. Wiederhold and X. Qian. Modeling asynchrony in distributed databases. In *Proc. International Conference On Data Engineering*, 1987.

[BW86] M.R. Barbacci and J.M. Wing. Durra: A task-level description language. Technical Report SEI-86-TR-3, Carnegie Mellon University, 1986.

[Gat87] Bill Gates. Beyond Macro Processing. *BYTE Bonus Edition*, Summer 1987.

[HN87] M.L. Heytens and R.S. Nikhil. Gestalt: an expressive database programming system. *Private Communications*, MIT, December 1987.

[RELL90] M. Rusinkiewicz, A. K. Elmagarmid, Y. Leu and W. Litwin. Extending the Transaction Model to Capture More Meaning. *Sigmod Record*, 19(1):3-7, 1990.

[LITW89] W. Litwin, A. Abdellatif, A. Zeroual, B. Nicolas, and PH. Vigier. MSQL: A Multidatabase Language. *Information Sciences*, 49(1,2,3), 1989 (A. Elmagarmid, Editor).