Supporting a Semantic Data Model in a Distributed Database System

Arvola Chan
Umeshwar Dayal
Stephen Fox
Daniel Ries

Computer Corporation of America
4 Cambridge Center
Cambridge, Massachusetts 02142

Abstract

Existing distributed database systems are based mostly on the relational model. Furthermore, it has been argued that the relational model is the model best suited for distributed databases. This paper describes an implementation approach for supporting logical pointers between distributed entities. This approach is being employed in a distributed database system that supports a semantically rich data model and that currently is under implementation. The access structures used to support entity-to-entity pointers in this data model facilitate the maintenance of referential integrity across sites. At the same time, they provide efficient access paths for distributed query processing. To make use of these access structures the extensions required of existing query processing techniques are quite straightforward.

1. Introduction

Existing distributed database systems for the most part have been based on the relational data model [SN77] [RBFG80] [WDL82]. Furthermore, it has been argued that the relational model is the model best suited for distribution [CODD82]. There are two properties of the relational model that facilitate its support for distribution. First, all of the relationships between records or types of objects are value-based; there is no need to support logical pointers across sites. Second, a high level language is available for specifying processing on sets of data; there is no need to follow record-at-a-time navigational links across sites.

This paper describes an implementation approach for supporting logical pointers across sites. This approach is currently being used in the implementation of the Distributed Database Manager (DDM) [CDFG83] [CDFG83b], a distributed database system that supports the semantically rich Daplex data model [SHIP81]. Similar approaches could be used for distributed systems that support high level access based on the entity-relationship model [CHEN76] and the network model [MP82], or in relational systems that support physical links between tuples stored at different sites.

In order to extend effectively the relational distributed database technology to these models, three important problems should be addressed. First, it should be possible to control the placement of data that is based on relationships between data objects. Assume, for example, that information about university departments is distributed according to the building in which each department is housed. Now suppose we want to store information about each of the courses with information about the department that is offering the course. In relational systems such as distributed INGRES [SN77], SDD-1 [RBGF80], and R* [WDL82], the course relation must include redundant information about the building that is housing the department offering the course, and the distribution criteria for the course relation would have to be based on the
The second problem is the maintenance of referential integrity across sites. If there is an integrity constraint that requires every course to have an offering department (i.e., the offering function from course to department is total), then even if course and department information are stored at different sites, the DDM will prevent the insertion of information for a new course for which there is no corresponding department. Similarly, the DDM will prevent the deletion of a department that is still referenced by an existing course. Furthermore, the DDM maintains this integrity constraint without requiring cross-site checking on each insertion or deletion. In a relational system, such integrity requirements would have to be specified by general integrity constraints [EC76], [HS78], [BCC81], [KF81]. The efficient support of such constraints for distributed relational databases remains an open problem.

The key to efficient distributed query processing in relational systems is the provision for set-at-a-time operations on distributed data. Thus, the third problem is how to use links that relate data across sites effectively. These links can augment physical access paths as in a single-site relational system like System R, or they can represent logical relationships. The DDM uses pointers across sites[1] to provide faster access to sets of records and to reduce data movement from one site to another during distributed query processing. These intersite pointers identify the logical entities and their residing sites only. Thus, they are unaffected by the relocation of physical records (representing entities) within a site. Furthermore, they are used by high level set-at-a-time operations. Hence, a network message is not incurred for following each such pointer.

Section 2 provides an overview of the semantic data model, Daplex, that is supported by the DDM. Section 3 addresses the first of the three problems stated above by describing the database fragmentation facilities of the DDM. Section 4 addresses the second problem by presenting the auxiliary data structures that are used to enforce referential integrity and to provide faster multi-site query processing. Section 5 addresses the third problem by discussing query processing operations that can make use of the presented structures. Finally, Section 6 summarizes our conclusions about data models and distributed databases.

(1) These pointers identify only the logical entity and the site where the entity is to be found. They do not have to be updated when an entity is relocated within a site.
son, and that age is a function applicable to person. The function inheritance semantics of Daplex automatically guarantee the consistency of age information on student and age information on instructor, since age is a function inherited from the supertype person. At the same time, inherited functions can be applied directly to an entity in data manipulation constructs in Daplex, without the need for tedious explicit joining operations. Figure 2.2 is a graphical representation of the revised database definition. The double-edged arrows represent is-a relationships (e.g., each student is-a person). The entity types person, student, and instructor are said to form a generalization hierarchy. Since every entity in this generalization hierarchy also must be a person, the type person is referred to as the root of the generalization hierarchy.

The implications of the need to efficiently enforce referential integrity and type overlap constraints on storage and access structures, for a centralized environment, have been discussed in [CDDL82]. In this paper, we focus on the implications of these fundamental integrity constraints for distributed database design.

3. Fragmentation and Grouping

The DDM provides complete physical data independence to end users. A separate interface is provided for database administrators (DBAs) for the purpose of specifying physical design parameters. An important design option available to DBAs is the allocation and replication of data to different sites in the network. This section discusses the choice of allocation units in the DDM.

The database fragmentation options supported in the DDM are somewhat different from those found in previous systems like distributed INGRES [SN77], SDD-1 [RBFG80], and R* [WDHL82], which are based on the relational data model. (2) The purpose of distributing and replicating data over different sites is to maximize locality of reference (i.e., to ensure that most transactions can be run using local data) and to provide resiliency against site failures. (3) The typical unit for allocation in these systems is a fragment of a relation (i.e., a logically subset defined by a local predicate).
is necessitated by our support for entity-valued functions in Daplex. Essentially, Daplex allows for "direct" pointers that point from one entity to another. The semantics of Daplex requires that insertion and deletion of entities to and from the database do not result in dangling pointers. In order to simplify the maintenance of "direct" inter-entity pointers, we introduce the notion of a fragment group. A fragment group consists of a collection of fragments, each of which is a logically defined subset of a generalization hierarchy. Each fragment group is a unit for allocation and replication. Fragment groups can be designed to "localize" inter-entity references. This localization simplifies the enforcement of deletion dependencies and improves the efficiency of query processing. We expect that most databases can be designed in such a way that frequently followed inter-entity references (from entities within each fragment group) can be localized. This will eliminate the cumbersome maintenance of nonlocal pointers. It is important to note that we do not require fragment groups to be defined in such a way that all inter-entity references are localized.

Our support for generalization hierarchies in Daplex leads us to forgo vertical partitioning, an option that is provided in SDD-1 [BB:E:1980] and R* [WDBL:82]. Our decision here has been influenced mostly by simplicity and efficiency considerations. Had we supported vertical fragmentation, we would have the additional problem of deciding how inter-entity pointers should be represented. When accessing an entity of a given type, Daplex allows for access to attributes (functions) that are defined from the viewpoint of any overlapping type. In order to provide efficient support for such accesses, we require that all functions/attributes of an entity, regardless of being primitively defined or inherited, be accessible from the same site. Thus, our objects for fragmentation in the DDM are the generalization hierarchies within databases.

The fragmentation of entities within a generalization hierarchy may be defined in terms of function values, subtype memberships, subtype nonmemberships, and one-to-many relationships induced by single-valued entity functions. Since we have ruled out the possibility of vertical fragmentation of entities within a generalization hierarchy across sites, we permit the disjoint partitioning of base entity types (i.e., roots of generalization hierarchies) only. Partitioning of entities within a subtype can be implicitly defined by partitioning the corresponding base entity type. Each fragment of a base entity type is defined by a conjunction of conditions. The DBA assigns a unique identifier to each fragment in order to permit the definition of one fragment to be dependent on that of another fragment. Each defining condition can be in one of the following forms:

1. \( e \) is in "subtype" (where \( e \) is an entity in the base type and "subtype" is contained in the base type).

2. \( e \) is not in "subtype" (where \( e \) is an entity in the base type and "subtype" is contained in the base type).

3. \( f(e) \) is a single-valued scalar function that is applicable to an entity that satisfies the type membership conjunct forming part of the defining predicate in question; and "comparison operator" is one of \( =, /\), \( <, <=, >, >= \).

4. \( f(e) \) is in \( F \) (where \( e \) is an entity in the base type; \( F \) is a previously defined fragment; and \( f \) is a single valued function that ranges over entities in \( F \).

Figure 3.1 illustrates one feasible fragmentation and grouping scheme for the example database from Section 2. Four fragment groups are defined under this particular scheme. The first fragment group stores department entities located on the first-floor, along with the courses offered by these first-floor departments and the instructors who work in these first-floor departments. The second fragment group stores second-floor departments and the associated courses and instructors. The third fragment group stores departments located on other floors, along with the associated courses and instructors. Finally, the fourth fragment group stores only a single fragment that consists of persons who are not instructors.

4. Access Structures

In this section, we discuss auxiliary data structures used in the DDM for supporting the maintenance of referential integrity constraints, and for facilitating the processing of transactions that span fragment group boundaries. Our

(5) We impose the restriction that if fragment A's definition depends on that of fragment B, then fragment B must be assigned to the same fragment group as fragment A.

(6) The predicate "others" in Figure 3.1 is interpreted by the DDM system as a shorthand for the complement of the disjunction of the other fragment defining predicates on the generalization hierarchy in question.
ment groups that contain entities that have specific values of that function.

Since each fragment group entity directory is associated with a single fragment group, it is actually considered part of the representation for entities in that fragment group. When specifying a distribution index for maintenance, a DBA can specify its placement in any previously defined fragment groups or in a new fragment group of its own. The replication of a distribution index is thus governed by the replication of the fragment group in which it is placed. The maintenance of these two types of structures are further explained below.

4.1 Fragment Group Entity Directory

Like the Local Database Manager [CFL81] [UDF82] which supports centralized Daplex databases, we maintain entity directories in the DDM in order to facilitate the resolution of inter-entity references and the enforcement of referential and type overlap constraints. In general, for a given fragment group we maintain a fragment group entity directory for each generalization hierarchy that is either stored in the fragment group or potentially referenced (as specified in the logical schema) by entities stored within the fragment group.

Two kinds of entries may be stored in a fragment group entity directory. A primary entry is one that pertains to an entity that is stored in the fragment group. A secondary entry is for an entity that is referenced but not stored in the fragment group.

A primary entry for an entity contains typing information, local reference count(s), physical record pointer(s), and one or more sets of nonlocal fragment group identifiers. A fragment group’s identifier is in one set if it contains entities that reference the given entity from the viewpoint of a specific entity type in the generalization hierarchy. The local reference counts keep track of local references from within the fragment group, one for each entity type to which the entity belongs. A secondary entry contains typing information and a reference count for each entity type to which the nonlocally stored (but referenced) entity belongs. This reference count indicates the number of times the designated entity is being referenced (from the viewpoint of a specific type) from entities stored within this fragment group.

Figure 4.1 illustrates a fragmentation and grouping scheme for the course and person entity types. Note that the entity to entity references are not completely localized. In this scheme, undergraduate courses are grouped with undergraduate students in \( FG_1 \), and graduate courses are grouped with graduate students in \( FG_2 \). However,
undergraduates can take graduate courses and graduates can take undergraduate courses. Figure 4.2 shows the set of courses taken by each student in the database. Figure 4.3 shows the corresponding fragment group entity directories for the (degenerate) course generalization.

\[ S_1, \ldots, S_4 = \text{undergrad students} \\
S_5, \ldots, S_8 = \text{graduate students} \]

\[ S_1 (C_1, C_2, C_5, C_6) \quad S_5 (C_5, C_6, C_1, C_2) \]
\[ S_2 (C_1, C_2, C_5, C_6) \quad S_6 (C_5, C_6, C_1, C_2) \]
\[ S_3 (C_1, C_2, C_3, C_4) \quad S_7 (C_5, C_6, C_7, C_8) \]
\[ S_4 (C_1, C_2, C_3, C_4) \quad S_8 (C_5, C_6, C_7, C_8) \]

Figure 4.2 Enrollment of Students in Courses

Similarly, the entity directory for courses in FG2 contains primary entries for graduate courses C5 through C8 and secondary entries for C1 and C2. If another graduate student enrolls in C1, only the reference count in FG2 needs to be incremented. If graduate students were no longer enrolled in C7 as a result of updates, its local reference count would become zero. Since C7 is not referenced by entities in any other fragment group, the deletion of C7 would be allowed. The legality of the deletion can always be determined by checking only the appropriate primary entry in the local entity directory.

4.2 Distribution Index

In distributed databases, it is often necessary to access all entities with a specific function value: entities that potentially could be stored in any fragment group. In order to facilitate such accesses, we introduce the notion of a distribution index. Each entry in a distribution index identifies the logical fragment groups that store entities that have a given indexed value. In order to maintain a distribution index efficiently, we require that corresponding local secondary indices be maintained for each fragment group where entities of the indexed entity type...
reside. (A local secondary index contains only entries for entities that are locally stored within a fragment group.) Without such local indices, it would be difficult to determine if an insertion or deletion operation has any effect on the distribution index. With the local indices, we know that the distribution index has to be updated whenever an update operation causes the creation or deletion of an entry in a local secondary index.

Figure 4.4 illustrates a distribution index that could be maintained for the students in the fragment groups identified in Figure 4.1. While this example distribution index is based on a unique function (i.e., each student has a unique student number), distribution indices in general can involve nonunique functions and combinations of functions.

<table>
<thead>
<tr>
<th>Student No.</th>
<th>FG</th>
</tr>
</thead>
<tbody>
<tr>
<td>77446</td>
<td>FG1</td>
</tr>
<tr>
<td>78550</td>
<td>FG2</td>
</tr>
<tr>
<td>79450</td>
<td>FG2</td>
</tr>
</tbody>
</table>

Figure 4.4 A Distribution Index

The distribution index on student number can be used to locate a specific student regardless of whether the student is a graduate or an undergraduate. That index is updated only when students enter or leave the university or change undergraduate/graduate status. No specific physical location information is maintained in the distribution index to shield it from changes of the student record within a fragment group.

In systems such as SDU-1, K*, and Distributed INGRES, both fragments would have to be searched based on student number. Alternately, these systems would allow a fragmentation criterion based on student number. However, the specification of ranges of student numbers as fragmentation criteria violates the important principle of separating logical and physical database designs: student numbers would have to be assigned based on where the records should be stored. On the other hand, the specification of an individual distribution criterion for each student number is cumbersome and would require updates to the distribution criteria (which normally can be performed only by a DBA) for each change in student status. The distribution index seems like a more natural extension to single-site secondary indices for locating data.

4.3 Implications of Replication

The DDM supports data replication [CDFG83a] [CDFG83b], and fragment groups are used as the units for allocation to sites in the system. However, it is important to note that all nonlocal references used in the representation of entity functions, in entity directories, and in distribution indices, refer to logical fragment groups and not to sites that stores copies of those fragment groups. This serves to simplify pointer maintenance significantly. At the same time, it facilitates our compilation approach to the generation of access plans for repetitive transactions. The same plan would be usable, regardless of which copy of a given fragment group is dynamically selected for executing the transaction. At run time, each logical fragment group must be bound to a physically available site for reading purposes. (That is, for each logical fragment group that must be read during the execution of the transaction, a site that stores a copy of the fragment group must be used.)

In fact, our implementation scheme for fragment group entity directories has been influenced strongly by the desire for the capability to add and remove fragment group copies dynamically. The allocation and replication parameters of a fragment group can be changed without requiring update operations on entity directories. For example, a new fragment group copy can be added simply by obtaining an image of an existing copy. Conceivably, we could have chosen a more compact representation for the fragment group entity directories by collapsing the directories for different fragment group copies if they happened to be stored at the same site. However, this would have brought major upheavals when the allocation and replication parameters of fragment groups have to be changed for performance reasons.

5. Distributed Query Processing

An overview of our two-stage approach to optimizing repetitive transactions has been presented in [CDFG83b]. The selection of a processing strategy treats each fragment group as a logical site and is performed at compile time. The result of compilation is a plan that prescribes the local operations to be performed on data stored within individual fragment groups and the movement of data resulting from these local operations. Optimization at compile time
is essentially based on a worst case analysis, since it assumes that different fragment groups are stored at different sites. At run time, the binding of logical fragment groups to physical sites is performed. That is, a set of sites is selected for following the compile time plan. The optimization during site selection for each fragment group takes into consideration which sites are operational (and thus which copies of a needed fragment group are accessible), and which fragment groups are potentially needed for running the transaction.

The presence of distributed access structures discussed in Section 4 affects the set of cross-site operations that are included in the access plan generated at compile time. At run time, they are used to reduce the set of fragment groups that potentially contain the desired data to the set of fragment groups that are actually required. We discuss the uses of fragment group entity directories and distribution indices for query processing below.

5.1 Use of Fragment Group Entity Directories

In addition to providing direct support for referential integrity maintenance, fragment group entity directories support fast access from individual entities to the entities they reference. Consider the fragmentation and grouping scheme illustrated in Figure 4.1. The enrollments function of a given student is represented by a collection of pointers to course entities. As explained in [CDPL82], either logical pointers or hybrid pointers can be used for the representation of entity functions. A logical pointer consists of the entity's unique identifier that was assigned at the time of its creation. A hybrid pointer combines a logical pointer with a physical pointer. The physical pointer portion of a hybrid pointer depends on whether the referenced entity is local or foreign. The physical pointer to a local entity points to a single local record representing the entity from a specific viewpoint (i.e., the range type of the entity-valued function). The physical pointer to a foreign entity is represented as an indirect flag only. It simply indicates that the entity is foreign, and that an appropriate entry in the local fragment group entity directory should be consulted to determine the fragment group that stores the foreign entity. (See [CDPL82] for discussions on hybrid pointer maintenance and validation.) A request can then be passed on to the site chosen during site binding for the identified fragment group in order to access the desired entity. The fragment group entity directory at this foreign fragment group is used to obtain the physical pointer(s) to the record(s) that represent the referenced entity. Thus, fragment group entity directories provide efficient support for "small" transactions that typically access a single entity of one type, along with related entities from other types.

At the same time, it is possible to make use of entity directories to process "large" transactions that perform joining operations on entire entity types. Consider a joining operation between student and course types based on the enrollments function. Assume that students who are not enrolled in any course, and courses that are not taken by any student, do not qualify for output in this transaction.

Now, assume that the database is fragmented as shown in Figure 4.1, that $FG_1$ and $FG_2$ are available at different sites, and that the transaction is originated at a third site. The most straightforward processing strategy is to transfer all student and course information from both $FG_1$ and $FG_2$ to the transaction's home site, reconstruct the student entity type by taking the union of undergraduate students and graduate students, reconstruct the course entity type by taking the union of undergraduate courses and graduate courses, and then compute the desired join between students and courses.

An often used strategy for reducing the amount of data that has to be transferred to the transaction's home site is to make use of semi-join operations [BGWR81]. Typically, this would involve projecting one of the operands on its joining field, and transferring this projection to the site of the second operand in order to reduce the size of the latter. With fragmented operands, the semi-join operation becomes more complicated. (For example, see [YCTB83] for discussions on how semi-joins can be performed on fragmented operands.)

However, with the presence of fragment group entity directories, a faster processing strategy is readily available. Let $SU$ and $CU$ represent undergraduate students and undergraduate courses in $FG_1$. Let $SC$ and $CG$ represent the graduate students and graduate courses in $FG_2$. A possible processing strategy, selected at compile time, is illustrated in Figure 5.1.

At a site that stores $FG_1$, we perform the join based on enrollments between $SU$ and $CU$ and store the result in a temporary $T_1$. By examining entities in $SU$ and the entity directory for courses in $FG_1$, we can also determine locally those undergraduate students who are taking graduate courses ($SU$ semi-joined by $CG$) and those undergraduate courses being taken by graduate students ($CU$ semi-joined by $SC$). These operations result in temporaries $T_4$ and $T_5$, respectively. Similarly, the temporaries $T_4$, $T_5$, and $T_6$ can be formed locally at a site that contains $FG_2$. At step 7, only some of the courses and students need to be joined. $T_7$ represents undergraduates taking graduate courses, $T_8$ represents graduates taking undergraduate courses. The unions at step 3 thus do not need to remove duplicates.
We say that this compile time plan specifies that the join is to be distributed over the union operations and that double semi-join reductions are to be used to reduce the amount of data that has to be transferred. At run time further reductions become apparent automatically (T7 or T8 would be empty if T4 were empty). Note that T2 would be empty if and only if T6 were empty.

With the above processing strategy, we are minimizing the amount of computation cost that will have to be incurred during step 2. If the intermediate results obtained during step 1 must individually be transferred to a different final site before step 2 is carried out, then it may be preferable to perform semi-joins between SU and CU to form temporaries T1a and T1b, and between SG and CG to form temporaries T2a and T2b, and postpone all joining operations until step 2. At the same time, since T1a may overlap with T4, and T1b may overlap with T5, it may be preferable to represent these temporaries with sets of entity identifiers, and to send information concerning entities that belong to multiple overlapping temporaries to the final site only once.\(^{(7)}\)

5.2 Use of Distribution Indices

As previously mentioned, one obvious use for distribution indices is in the processing of valued-based joining operations. Consider a query that joins suppliers and projects based on their respective cities. If we have available distribution indices on both the city function of suppliers and on the city function of projects, we can use the distribution index to identify the fragment groups that actually have matching entities. For each supplier fragment we can identify the cities for which there will be a matching project. We can then use the list of cities to perform a semi-join on that supplier fragment. Similarly, we can collect a list of qualifying cities for each project fragment. At compile time, the optimizer will decide to use neither, one, or both of the semi-join reductions. At run time, it may be determined that entire fragments need not be accessed.

6. Conclusions

The design of the DDM has led us to three important conclusions. The first conclusion is that it is indeed possible to build distributed database systems that support a semantically rich data model like Daplex, the entity-relationship model, and other models that support explicit relationships between different types of objects. The distributed database management system that supports these models must support a high level, set-at-a-time data manipulation facility. With such a facility, the support for distribution should allow for the placement of data that is based on the relationships between the types of data, maintain referential integrity between data stored at different sites, and use distribution access paths along with relational like operations to support query processing.

The second conclusion we have reached is that in addition to the increased functionality of the semantically rich data model, a distributed system like the DDM actually can improve performance over a distributed relational database system. The distributed access structures used to maintain referential integrity also can be used in query processing to locate and limit the number of fragments that need to be accessed and to reduce the amount of data that needs to be shipped between sites. Furthermore, these structures can be designed so that multiple-site checking and updating is not required for individual entity insertions and deletions.

The third conclusion we have reached is that the required extension to relational distributed database technology is straightforward. The extensions described in this paper do not seem significantly more difficult than the implementations...
tions required for a distributed relational system. In fact, the extensions, particularly the query processing extensions, are built on relational query processing techniques. Furthermore, these extensions are orthogonal to other aspects of the DDM implementation such as directory management, (inter-transaction) concurrency control, reliability, and recovery.

7. References

[BCC81]

[BGWR81]

[CDFG83a]

[CDFG83b]

[CDFL82]

[CHEN76]

[CFLR81]

[CODD82]

[DATE81]

[EC75]

[HS78]

[KP81]

[MP82]

[RBFG80]

[SHFP81]

[SN77]

[WDHL82]

[YCTB83]